CptS355 - Assignment 5
Scoped Simple PostScript Interpreter (SSPS) - Spring 2022

Due: Wednesday, April 27, 2022

Weight: Assignment 5 will count for 5% 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 PostScript Interpreter) to handle a slightly different language which supports static scoping. We will call this language Scoped Simple PostScript - SSPS. **SSPS has no `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.

Getting Started

Make a copy of your `HW4_part2` folder and rename it as `HW5`. Download and copy the provided `repl.py` and `load.py` files from Canvas and copy them to your `HW5` folder. These two files are already updated to support static scoping. Below is a summary of changes on these files. We will explain these further during class.

1. `load.py`: interprets each test input first using static scoping , then using dynamic scoping.
2. repl.py: expects an optional argument for defining the scoping rule.

In this assignment, you will start working with your HW4-part2 code and make changes in the following files:

- `elements.py`: You need to change the way function calls are applied. When `Name` object represents a function call (i.e., its value is a `CodeArray`), before the `CodeArray` is applied, a new *Activation Record* (AR) with an empty dictionary should be pushed onto the `dictstack` and when function execution is done, the AR should be popped from the `dictstack`. We will talk about how to represent the AR in the following sections.
- `psbuiltins.py`: You need to make changes to the following methods: `lookup`, `define`, `stack`, `dictPush` ,`psIf` ,`psIfelse` , and,`for` . Also, you will remove the following operator methods from `psbuiltins.py`: `begin`, and `end`.

Turning in your assignment

All code should be developed in the directory **HW5**. To submit your assignment, zip all the files in `HW5` folder as `HW5.zip` and turn in your zip file by uploading on the dropbox on Canvas. Please zip only the 8 Python files; not the `HW5` directory. Also, exclude any binary python bytecode files.

The file that you upload must be named HW5.zip . At the top of the `psbuiltins.py` 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.

Project Description

You will be modifying your SPS interpreter (HW4_part2) to handle a slightly different language which we will call Scoped Simple PostScript - SSPS. As explained above, SSPS has no `begin`, and `end` operations. Instead, each time a function is called, a new AR is automatically pushed on the `dictstack` and when the function execution is complete, this AR will be popped out of the `dictstack`.

Compared to the `Stacks` object in HW4, the HW5 `Stacks` object will be initialized with a `scope` argument whose value will either be the string "static" or the string "dynamic", to indicate whether it should behave using static scope rules or dynamic scope rules. For example:

```
class Stacks:
    def __init__(self,scoperule):
     ……
```

- In load.py, we create the `psstacks_s` and `psstacks_d` objects that are initialized with 'static' and 'dynamic' string arguments, respectively.

  ```
  psstacks_s = Stacks("static")
  psstacks_d = Stacks("dynamic")
  ```

  To evaluate a list of expressions, i.e., `expr_list`, using static scoping, we pass `psstacks_s` to the `evaluate` method of the expressions:

  ```
  for expr in expr_list:
      expr.evaluate(psstacks_s)
  ```

  To evaluate `expr_list`, using dynamic scoping, we pass `psstacks_d` to the `evaluate` method of the expressions:

  ```
  for expr in expr_list:
      expr.evaluate(psstacks_d)
  ```

3. To run the REPL tool using stating scoping, you should execute `repl.py` using `'--static'` command line argument, i.e.,

   ```
   python repl.py --static
   ```

   If '`--static`' option is provided, the REPL interpreter will initialize the `Stacks` object with `'static'` argument.

   If '`--static`' option is not provided, by default, the interpreter will use dynamic scoping rule and will initialize the `Stacks` object with `'dynamic'` argument.

Static vs Dynamic Scoping

1. To implement **static scope rules,** you need a static chain which is the set of dictionaries visited by following the static links we discussed in class.
2. ***How can you implement the static links?*** You should change your `dictstack` and make it a stack of tuples (instead of just a stack of dictionaries) where each tuple contains an integer index and a dictionary, i.e., (static-link-index, dictionary). The integer index represents the static link that tells you the position of the parent scope in the list.
3. ***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 (*Activation Record* (AR)) 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 function, in the beginning of the `` `apply` `` method of the `` `CodeArray` ``, you will push a tuple including the static link and an empty dictionary onto the `` `dictstack` `` (see below). This tuple represents the AR for that function call.

     ```
     (index-of-definition's stack entry, {})
     ```

   - And when the execution of the function is complete (i.e., at the end of the `` `apply` `` method) this tuple will be popped from the `` `dictstack` ``.
   - In addition, when the bodies of `` `if` ``, `` `ifelse` ``, and `` `for` `` operators are evaluated, a tuple representing the AR for the operator's block will be pushed onto the stack. The static-link of the tuple should point to the top of the stack. (Note that these operators will be evaluated in the most recent referencing environment)

     ```
     (index-of-top-of-the-stack, {})
     ```

   - And when the execution of the operator is complete, this tuple will be popped from the `` `dictstack` ``.

4. 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 Lab3, 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 lookup will behave very much like SPS lookup. Of course, you should change your lookup code for the new dictionary structure.
   - The `` `dictstack` `` structure will be the same 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*).
     - If the scoping rule is static, it should look up using *static links*.

Summary of the changes you need to make:

- The `Stacks` object is initialized with the scoping rule that will be used in interpreting SPS code. You should store the scope argument value in a local attribute in the `Stacks` object (for example `scope`) . The `lookup`, and `define` methods will check this attribute value to identify the scoping rule (i.e., `self.scope`).
- Your interpreter should store tuples in `dictstack` where first value in the tuple is the *static-link-index* and second value is the *dictionary*.
- When a function body is applied (in `Name`s `evaluate` method) a new tuple (AR) will be pushed onto the `dictstack`. *"Static-link-index"* in that tuple is the index of the dictionary (in the `dictstack`) where the function is defined. We will discuss the algorithm for finding this in class. When a function execution is done, remember to pop the tuple for that function call from the `dictstack`.
- When an `if`, `ifelse`, or `for` body is applied (in `psIf`, `psIfelse`, `for` methods) a new tuple (AR) will be pushed onto the `dictstack`. *"Static-link-index"* in that tuple is the index of the top of the stack. When the execution of the body is done, remember to pop that tuple from the `dictstack`.
- Change your `lookup` function for static scoping. If the scoping rule is dynamic, perform lookup by searching the AR's (tuples) top to down. If it is static, use static-links for the search. You should also change the `define` function and make it work with the new `dictstack` structure.
- Change your `stack` operator implementation as explained below.
- You may need to change your `dictPush` to make it work with the new `dictstack` structure.

Output of the Interpreter

In our new SSPS Interpreter, whenever the `stack` operation is executed, the contents of the operand and dictionary stacks are printed.

- Print a line containing `"===**opstack**==="` to mark the beginning of the `opstack` content.
- Print the operand stack one value per line; print the top-of-stack element first.
- Print a line containing `"===**dictstack**==="` 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. Please see next section for examples.
- Print a line containing `"=================="` to separate the dictionary stack from any subsequent output.

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

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

Your own tests:

Below we provide several tests that you can use for testing your interpreter. **Please create at least 3 additional test inputs and include them at the end of your solution file.** When we grade your assignment, we will compare the outputs of your interpreter prints.

Given tests:

1)
```
testinput1 = """
    /x 4 def
    /g { x stack } def
    /f { /x 7 def g } def
    f
    """
```
The above SPS code will have **7 on the stack using dynamic scoping and 4 using static scoping when stack operator is called in "g"**. The output from the **`stack`** operator in function **`g`** would look like this when using static scoping:

```
STATIC
STATIC
===**opstack**===
4
===**dictstack**===
----2----0----
----1----0----
/x   7
----0----0----
/x   4
/g   CodeArray([Name(x), Name(stack)])
/f   CodeArray([Name(/x), Literal(7), Name(def), Name(g)])
================
```
And using dynamic scoping, the output from the **`stack`** operator will look like the following:

```
DYNAMIC
===**opstack**===
7
===**dictstack**===
----2----0----
----1----0----
/x   7
----0----0----
/x   4
/g   CodeArray([Name(x), Name(stack)])
/f   CodeArray([Name(/x), Literal(7), Name(def), Name(g)])
================
```

2)

```
testinput2 = """
    /x 4 def
    (static_y) dup 7 120 put /x exch def
    /g { x stack } def
    /f { /x (dynamic_x) def g } def
    f
    """
```

Expected Output

```
STATIC
===**opstack**===
StrConstant('(static_x)')
===**dictstack**===
----2----0----
----1----0----
/x   StrConstant('(dynamic_x)')
----0----0----
/x   StrConstant('(static_x)')
/g   CodeArray([Name(x), Name(stack)])
/f   CodeArray([Name(/x), StringExpr((dynamic_x)), Name(def), Name(g)])
=================

DYNAMIC
===**opstack**===
StrConstant('(dynamic_x)')
===**dictstack**===
----2----0----
----1----0----
/x   StrConstant('(dynamic_x)')
----0----0----
/x   StrConstant('(static_x)')
/g   CodeArray([Name(x), Name(stack)])
/f   CodeArray([Name(/x), StringExpr((dynamic_x)), Name(def), Name(g)])
=================
```

3)
```
testinput3 = """
    /m 50 def
    /n 100 def
    /egg1 {/m 25 def n} def
    /chic
        { /n 1 def
          /egg2 { (egg2) n stack} def
          n m
          egg1
          m
          egg2
        } def
    n
    chic
"""
```

Expected Output

```
STATIC
===**opstack**===
1
StrConstant('(egg2)')
50
100
50
1
100
===**dictstack**===
----2----1----
----1----0----
/n   1
/egg2   CodeArray([StringExpr((egg2)), Name(n), Name(stack)])
----0----0----
/m   50
/n   100
/egg1   CodeArray([Name(/m), Literal(25), Name(def), Name(n)])
/chic   CodeArray([Name(/n), Literal(1), Name(def), Name(/egg2),
Block([StringExpr((egg2)), Name(n), Name(stack)]), Name(def), Name(n), Name(m),
Name(egg1), Name(m), Name(egg2)])
================

DYNAMIC
===**opstack**===
1
StrConstant('(egg2)')
50
1
50
1
100
===**dictstack**===
----2----1----
----1----0----
/n   1
```

```
/egg2   CodeArray([StringExpr((egg2)), Name(n), Name(stack)])
----0----0----
/m   50
/n   100
/egg1   CodeArray([Name(/m), Literal(25), Name(def), Name(n)])
/chic   CodeArray([Name(/n), Literal(1), Name(def), Name(/egg2),
Block([StringExpr((egg2)), Name(n), Name(stack)]), Name(def), Name(n), Name(m),
Name(egg1), Name(m), Name(egg2)])
=================
```

4)

```
testinput4 = """
    /x 10 def
    /A { x } def
    /C { /x 40 def A stack } def
    /B { /x 30 def /A { x 2 mul } def C } def
    B
    """
```

Expected Output

```
STATIC
===**opstack**===
10
===**dictstack**===
----2----0----
/x   40
----1----0----
/x   30
/A   CodeArray([Name(x), Literal(2), Name(mul)])
----0----0----
/x   10
/A   CodeArray([Name(x)])
/C   CodeArray([Name(/x), Literal(40), Name(def), Name(A), Name(stack)])
/B   CodeArray([Name(/x), Literal(30), Name(def), Name(/A), Block([Name(x),
Literal(2), Name(mul)]), Name(def), Name(C)])
=================

DYNAMIC
===**opstack**===
80
===**dictstack**===
----2----0----
/x   40
----1----0----
/x   30
/A   CodeArray([Name(x), Literal(2), Name(mul)])
----0----0----
/x   10
/A   CodeArray([Name(x)])
/C   CodeArray([Name(/x), Literal(40), Name(def), Name(A), Name(stack)])
/B   CodeArray([Name(/x), Literal(30), Name(def), Name(/A), Block([Name(x),
Literal(2), Name(mul)]), Name(def), Name(C)])
=================
```

5)

```
testinput5 = """
    /x 2 def
    /n 5  def
    /A { 1  n -1 1 {pop x mul} for} def
    /C { /n 3 def /x 40 def A stack } def
    /B { /x 30 def /A { x } def C } def
    B
    """
```

Expected Output

```
STATIC
===**opstack**===
32
===**dictstack**===
----2----0----
/n   3
/x   40
----1----0----
/x   30
/A   CodeArray([Name(x)])
----0----0----
/x   2
/n   5
/A   CodeArray([Literal(1), Name(n), Literal(-1), Literal(1), Block([Name(pop),
Name(x), Name(mul)]), Name(for)])
/C   CodeArray([Name(/n), Literal(3), Name(def), Name(/x), Literal(40),
Name(def), Name(A), Name(stack)])
/B   CodeArray([Name(/x), Literal(30), Name(def), Name(/A), Block([Name(x)]),
Name(def), Name(C)])
=================

DYNAMIC
===**opstack**===
40
===**dictstack**===
----2----0----
/n   3
/x   40
----1----0----
/x   30
/A   CodeArray([Name(x)])
----0----0----
/x   2
/n   5
/A   CodeArray([Literal(1), Name(n), Literal(-1), Literal(1), Block([Name(pop),
Name(x), Name(mul)]), Name(for)])
/C   CodeArray([Name(/n), Literal(3), Name(def), Name(/x), Literal(40),
Name(def), Name(A), Name(stack)])
/B   CodeArray([Name(/x), Literal(30), Name(def), Name(/A), Block([Name(x)]),
Name(def), Name(C)])
=================
```

6)

```
testinput6 = """
    /out true def
    /xand { true eq {pop false} {pop true} 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

```
STATIC
===**opstack**===
False
===**dictstack**===
----3----0----
/x   False
----2----0----
/x   True
----1----0----
/out   False
----0----0----
/out   True
/xand   CodeArray([Literal(True), Name(eq), Block([Name(pop), Literal(False)]),
Block([Name(pop), Literal(True)]), Name(ifelse), Name(dup), Name(/x), Name(exch),
Name(def), Name(stack)])
/myput   CodeArray([Name(out), Name(dup), Name(/x), Name(exch), Name(def),
Name(xand)])
/f   CodeArray([Name(/out), Literal(False), Name(def), Name(myput)])
================

DYNAMIC
===**opstack**===
True
===**dictstack**===
----3----0----
/x   True
----2----0----
/x   False
----1----0----
/out   False
----0----0----
/out   True
/xand   CodeArray([Literal(True), Name(eq), Block([Name(pop), Literal(False)]),
Block([Name(pop), Literal(True)]), Name(ifelse), Name(dup), Name(/x), Name(exch),
Name(def), Name(stack)])
/myput   CodeArray([Name(out), Name(dup), Name(/x), Name(exch), Name(def),
Name(xand)])
/f   CodeArray([Name(/out), Literal(False), Name(def), Name(myput)])
================
```

7)

```
testinput7 = """
    /x  1 dict def
    x /i 22 put
    /A { (global)  x /i get } def
    /C { /x 1 dict def x /i 33 put A stack } def
    /B { /x 1 dict def x /i 11 put /A { (function B) x /i get  } def C } def
    B
    """
```

Expected Output

```
STATIC
===**opstack**===
22
StrConstant('(global)')
===**dictstack**===
----2----0----
/x   DictConstant({'/i': 33})
----1----0----
/x   DictConstant({'/i': 11})
/A   CodeArray([StringExpr((function B)), Name(x), Name(/i), Name(get)])
----0----0----
/x   DictConstant({'/i': 22})
/A   CodeArray([StringExpr((global)), Name(x), Name(/i), Name(get)])
/C   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(x), Name(/i),
Literal(33), Name(put), Name(A), Name(stack)])
/B   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(x), Name(/i),
Literal(11), Name(put), Name(/A), Block([StringExpr((function B)), Name(x),
Name(/i), Name(get)]), Name(def), Name(C)])
=================

DYNAMIC
===**opstack**===
33
StrConstant('(function B)')
===**dictstack**===
----2----0----
/x   DictConstant({'/i': 33})
----1----0----
/x   DictConstant({'/i': 11})
/A   CodeArray([StringExpr((function B)), Name(x), Name(/i), Name(get)])
----0----0----
/x   DictConstant({'/i': 22})
/A   CodeArray([StringExpr((global)), Name(x), Name(/i), Name(get)])
/C   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(x), Name(/i),
Literal(33), Name(put), Name(A), Name(stack)])
/B   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(x), Name(/i),
Literal(11), Name(put), Name(/A), Block([StringExpr((function B)), Name(x),
Name(/i), Name(get)]), Name(def), Name(C)])
=================
```

8)

```
testinput8 = """
    /x 1 dict def
    /a 10 def
    /A { x /m 0 put } def
    /C { /x 1 dict def  x /m 9 put A  a x /m get stack } def
    /B { /x 1 dict def /A { x /m 99 put } def /a 5 def C } def
    B
    """
```

Expected Output

```
STATIC
===**opstack**===
9
10
===**dictstack**===
----2----0----
/x   DictConstant({'/m': 9})
----1----0----
/x   DictConstant({})
/A   CodeArray([Name(x), Name(/m), Literal(99), Name(put)])
/a   5
----0----0----
/x   DictConstant({'/m': 0})
/a   10
/A   CodeArray([Name(x), Name(/m), Literal(0), Name(put)])
/C   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(x), Name(/m),
Literal(9), Name(put), Name(A), Name(a), Name(x), Name(/m), Name(get),
Name(stack)])
/B   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(/A),
Block([Name(x), Name(/m), Literal(99), Name(put)]), Name(def), Name(/a),
Literal(5), Name(def), Name(C)])
=================

DYNAMIC
===**opstack**===
99
5
===**dictstack**===
----2----0----
/x   DictConstant({'/m': 99})
----1----0----
/x   DictConstant({})
/A   CodeArray([Name(x), Name(/m), Literal(99), Name(put)])
/a   5
----0----0----
/x   DictConstant({})
/a   10
/A   CodeArray([Name(x), Name(/m), Literal(0), Name(put)])
/C   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(x), Name(/m),
Literal(9), Name(put), Name(A), Name(a), Name(x), Name(/m), Name(get),
Name(stack)])
```

```
/B   CodeArray([Name(/x), Literal(1), Name(dict), Name(def), Name(/A),
Block([Name(x), Name(/m), Literal(99), Name(put)]), Name(def), Name(/a),
Literal(5), Name(def), Name(C)])
=================
```