# CptS355 - Assignment 5
# Static Scoping PostScript Interpreter (SSPS)
# Spring 2018

**Assigned:** Wednesday, April 4, 2018

**Due:** Friday, April 13, 2018

**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. The dictionary must be able to hold an arbitrary number of names.

If you had major issues and problems in your SPS Interpreter assignment (Assignment 4), please let the instructor know as soon as possible.

## Turning in your assignment

All code should be developed in the file called HW5.py. Be sure to include your name as a comment at the top of the file. Also in a comment, indicate whether your code is intended for Unix/Linux or Windows. Implement your code for Python 3. The TA will run all assignments using Python3 interpreter. To submit your assignment, turn in your file by uploading on the Assignment5 (SSPS) DROPBOX on Blackboard (under AssignmentSubmisions menu).

The work you turn in is to be your own personal work. 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.

## 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. The dictionary must be able to hold an arbitrary number of names (a Python dictionary is just what is needed!).

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 argumentto 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)
```

To interpret code “s” 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.
- 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).
- 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, probably implemented as a list, so you don’t have explicit control links (also called 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 (and push and pop at that end as well).
- ***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) 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 (dictionary, index-of-definition’s stack entry).

    - 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 all the dictionaries on the stack in turn, which gives dynamic scope rules).
        - Note: In Assignment 3 problem 3(b), 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).

## Additional Notes:

You should not have tow separate interpret functions to for the static and dynamic scopincg rules. Please include the scoping rule as an argument as suggested above and customize the interretation if static scoping is specified.  scoping rule specified. If the scoping rule is static, you should

- Store  tuples into the opstack and whenver a function call is made
- You need to find the index of the dictionary where the function is defined. I will discuss the algorithm to find this in class.
- Define a new lookup function for static scoping. You already implemented a similar function in Assignment 3.

## 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 to test your interpreter.

```
/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']
==============
```

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. Remember that testing to the specification is your responsibility. We read your code looking for bugs in addition to running it on some tests.

**Additional Test Cases:**

```
1)
/m 50 def
/n 100 def
/egg1 {/m 25 def n} def
/chic {
    /n 1 def
    /egg2 { n } def
    m  n
    egg1
    egg2
    stack  } def
n
chic
```

**Expected Output**
*Using static scoping*

```
Static
==============
1
100
1
```

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

*Using dynamic scoping*

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

2)

```
/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   ['x']
----0----0----
/x   10
/A   ['x']
```

```
/C   ['/x', 40, 'def', 'A', 'stack']
/B   ['/x', 30, 'def', '/A', ['x'], 'def', 'C']
==============
```

*Using dynamic scoping*

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