CptS 355 - Programming Language Design

Programming Assignment:	**6**	Title:	**SPSS**
Issued:	April 8, 2015	Due:	Monday, April 27, 2015, by 11:59 pm

Turning in your work

Develop all your code in a directory named "ssps". When you are finished make a zip file of the directory and turn it in using the course turn-in page. Include a README file describing the environment for running your interpreter. The name of the main executable file should be ssps.py or just ssps.

Credit

This assignment will count approximately 15% of your final grade. In addition to correct functioning it will be graded on commenting and appropriate use of Python language constructs.

Policy

This assignment is to be your own, personal, individual work.

Description

This project is intended to enhance understanding of the behavior of, and mechanisms used to implement, static and dynamic scope rules.

You will be modifying your SPS interpreter to handle a slightly different language. Let's call it 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.

The SPSS interpreter will take a command line switch, either `-s` or `-d`, to indicate whether it should behave using static scope rules or dynamic scope rules. The default, if no switch is supplied, is *dynamic scope rules*. You can check for the command line switch by importing the `sys` module and then inspecting the variable `sys.argv` which is a python list of all the command line arguments.

Using dynamic scope rules, SPSS will behave very much like SPS except that there is no need to use `dict`, `begin` or `end` operations in programs (indeed, there will be no implementation of these operations in SSPS). The interpreter for dynamic SSPS should actually be a little smaller than the original SPS interpreter because there is no longer need to detect the `dict`, `begin`, and `end` operations. Each time a postscript function is about to be called push a new dictionary and when a 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, 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 – you search from one end (the warm end) and push and pop at that end as well.

How can you implement the static chain? We suggest making the stack be a stack of tuples instead of just a stack of dictionaries. Each tuple contains an integer and a dictionary. The integer is 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 procedure you create a new dictionary automatically but what you push on the dictionary stack is a pair (dictionary, index-of-definition's stack entry).

Hint 1: In the skeleton implementation this all took only a handful of lines of new code but it was tricky to get all the pieces right. The advice is think more, write less for this part of the project.

Note: If you want to explore Python's object facilities and create a class for the objects on the dictionary stack, feel free to do that. The code will be a little cleaner, but using objects is not required as we have not discussed python objects yet.

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).

Output of the interpreter

Note the following change from SPS: the output of the SPSS interpreter consists of the contents of the operand *and dictionary* stacks whenever the stack operation is executed.

- 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 something that will identify the dictionary that follows and `n` is something that represents the static link for the dictionary that follows (in the case that static scoping (-`s`) is being used. There is an example below. If you use a "system dictionary" at the base of the dictionary stack, whose contents never change, as part of your implementation of the basic interpreter, do not print it.
- 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 may start from the skeleton found at the class web site, but if your code was almost correct it will probably be easier to work with your own code. Note that the skeleton does not meet all the requirements of the sps assignment and you need to first complete it to do so. It is missing at least `false`, `true`, and the boolean operations `and`, `or`, and `not`. Furthermore, *it is your responsibility to make sure that what you submit conforms to the requirements in the SPS assignment regarding command line arguments such as filenames, etc.*

If you use the skeleton:

1. You must clearly state in a comment at that the beginning of your code that you are using the provided code as the basis for your project

2. Your maximum score for the project will be lowered by 10 points

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. Translated to SSPS and simplified a bit that example looks like:

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

which when it ends 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 ----
f [/x 7 def g]
g [x stack]
x 4
==================
```

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 we suggest using [ ] around the values in the array, but we don't care about details as long as it is easy to see which things that you print for dictionaries are names and which are values (be careful that it is possible to make this distinction even when a code array is very large and wraps onto another line).

Remember that testing to the specification is your responsibility. We read your code looking for bugs in addition to running it on some tests. Tests can only show the presence of bugs, never their absence!