Programming Assignment 2
CptS 355 - Fall 2014
An Interpreter for a PostScript-like Language

September 16, 2014

Overview

Assigned: September 10, 2014 (Revised Sept. 16, 2014 to correct dates in the "Credit" paragraph below.)

Due: Monday, September 22, 2014 at 11:59:59PM. Preliminary code. Develop all your code in a directory named "sps". When you are finished, make a **zip** file of the directory and turn it in using the course Turnin Page. *Please clean irrelevant files from the directory before making the zip file!* For this first due date, you need to turn in preliminary code containing implementations, as Python functions, of all of the operators listed onlines that start with * in the "The Problem". The Python functions' names should be suggestive of the Postscript operator names but some function names cannot be identical to the Postscript operator names because the operator names conflict with Python keywords. For this first part the TAs will be inspecting your function implementations but there is not necessarily a run-able program at this point.

Due: Monday, October 6, 2014 at 11:59:59PM. The complete solution. Develop all your code in a directory named "sps". When you are finished, make a zip file of the directory and turn it in using the course Turnin Page. *Please clean irrelevant files from the directory before making the zip file! Note you must turn in a complete implementation at this stage even if you happen to have completed the entire assignment for the Sept. 22 deadline.*

It must be possible to run your program on an input file containing postscript code using the command

```
python sps.py input-filename
```

or

```
python3 sps.py input-filename
```

depnding on how python3 is installed on your system. Note, regardless of how it is invoked, the program must be written using the **python3** variant of the python language.

Credit: This assignment will count approximately 10% of your final grade - 2% for the preliminary work due on Sept 22, and 8% for the final product due on Oct. 6.

Policy: This assignment is to be your own work. Refer to the course academic integrity statement in the syllabus.

The problem

In this assignment you will write an interpreter in Python for a small PostScript-like language, concentrating on key computational features of the abstract machine, omitting all PS features related to graphics, and using a somewhat-simplified syntax.

The simplified language, SPS, has the following features of PS

- integer constants, e.g. `123`: in python3 there is no practical limit on the size of integers
- boolean constants, `true` and `false` (Note that the boolean constants in python are `True` and `False`)
- name constants, e.g. `/fact`: start with a / and letter followed by an arbitrary sequence of letters and numbers

- names to be looked up in the dictionary stack, e.g. `fact`: as for name constants, without the /
- code constants: code between matched curly braces `{ ... }`
- * built-in operators on numbers: `add`, `sub`, `mul`, `div`, `eq`, `lt`, `gt`

```
Examples:
def add():
   spush(spop()+spop())
def sub():
   secondOperand = spop()
   firstOperand = spop()
   spush(firstOperand-secondOperand)
Notice how you have to pay attention to the order of the operands on the stack for op
```

- * built-in operators on boolean values: `and`, `or`, `not`; these take boolean operands only. Anything else is an error.
- * built-in sequencing operators: `if`, `ifelse`; make sure that you understand the order of the operands on the stack. Play with ghostscript if necessary to help understand what is happening.
- * stack operators: `dup`, `exch`, `pop`
- * dictionary creation operator: `dictz`; takes no operands[1]
- * dictionary stack manipulation operators: `begin`, `end`. `begin` requires one dictionary operand on the operand stack; `end` has no operands.
- * name definition operator: `def`. This requires two operands, a name and a value
- * stack printing operator (prints contents of stack without changing it): `stack`
- * top-of-stack printing operator (pops the top element of the stack and prints it): =

An SPS program is a sequence of numbers, names, operators, and braces.

Input to the interpreter is an SPS program read from the file named on the command line. You can read the entire input in one shot with

```
import sys
open(sys.argv[1]).readlines()
```

which reads the entire input into a list of lines. (Don't forget to import sys). Your program does not have to be interactive – assume that all of the input is available when it starts executing.

The assignment

Write an interpreter for SPS in Python. Make use of Python datatypes like dictionaries and lists to implement key data structures of the interpreter, for example dictionaries and stacks. For incorrect programs your interpreter should print a helpful error message, but extensive error messages are not required.

For correct SPS programs your interpreter should produce the same output (stack contents) as produced by a PostScript interpreter. The code for this assignment will be used as a starting place for a future assignment. Therefore, it is important to do a good job of organizing and documenting your code so it can be easily understood and modified several weeks later. In particular you must modularize the parts of your code that implement the following components of the SPS abstract machine (for the Sept. 22 deadline):

- the operand stack; modularlizing this means having operators to push and pop values so that elsewhere in your code you don't have to worry about the details of the stack implementation, or checking that the stack contains at least one value when you want to do a pop.

[1] A note about `dictz`: normally Postscript uses a `dict` operator requiring one operand. Our SPS has the `dictz` operator with zero operands. You can run regular postscript code in your interpreter if you prefix the Postscript code by `/dict {pop dictz} def`, to provide a sensible definition for the missing `dict` operation.

```
Example
def spop():
   if len(stack) < 1:
      print( "Error in spop: empty stack" )
      sys.exit()
   return stack.pop()
```

- SPS dictionaries; modularizing the dictionaries means implementing functions that operate on a dictionary to determine whether it contains an entry for a particular name, to retrieve the value associated with a particular name, and to enter a new new (name, value) pair.
- the dictionary stack: as with the operand stack, implement functions to push and pop values on the dictionary stack, check whether it is empty, etc.
- **IMPORTANT NOTE: implementing the functions required for the preliminary Sept. 22 due date requires that you also design and implement the operand stack, dictionaries, and the dictionary stack.**

After completing the requirements for Sept. 22, begin to work on the following aspects of the interpreter, completing a fully working interpreter by Oct. 6.

- the *reader,* which is responsible for reading in the program and breaking it into individual tokens (numbers, names, braces, etc.) Make good use of Python string handling here. The `re` module is particularly helpful. After importing `re`, use `re.findall( "/?[a-zA-Z][a-zA-Z0-9_]*|[-]?[0-9]+|[}{]+|%.*|[^ \t\n]", line)` to obtain a list of all the input tokens on an input line. (More about regular expressions later in class, but try this in an interactive interpreter on a line of PS code to see what it does.) Notice that programs may extend over multiple lines, and even blocks of code between braces may extend over multiple lines. In fact line boundaries in this language are irrelevant and should be completely ignored.
- A function loops over program tokens, pushing and popping values, computing results, and recursively calling itself when a function defined in the SPS program is encountered.
- Code to look up names in the dictionary stack and either push their value on the operand stack or call the code bound to the name as a function.

Output from the interpreter: In addition to output produced by the "stack" and "=" operators (described above), print the contents of the operand stack, one value per line, when the SPS program terminates. Then print the contents of the dictionary stack. Print the contents of the top dictionary first. Following each dictionary (not each dictionary entry) print a separator line like "==========".

Observations on the assignment

Unlike the ghostscript interpreter, the SPS interpreter is not interactive. It is not required to produce any output until all of its input has been read and processed.

Stacks in Python: built-in Python list methods can easily be used to implement stacks. If variable `L` refers to a list, then `L.append(v)` pushes `v` as a new element at the end of the list, and `L.pop()` returns the last value on the list and also removes it from the list.

Grading

Assignments will be graded for

- Correct functioning on SPS programs - 80%. *You* are responsible for creating test cases and examining the specification for ambiguity or unclarity. I will answer questions with email to the class. I will not answer questions of the form "what is this SPS program supposed to print?" unless you point out specifically why it is not clear what the correct answer is supposed to be.
  - **Make sure that your code treats `dictz`, `begin` and `end` as *separate* operators.** Although the pattern `dictz begin` is very common in PS programs, it is not an operation in and of itself. Furthermore, it is *much easier* to implement separate `dictz` and `begin` operators than some weird hybrid of the two.

  - Make sure that you handle nested braces ({ }) correctly
  - Make sure that you handle entering and returning from execution of code arrays correctly.

- Appropriate commenting - 10%
  - Each function should at least have comments describing its inputs and outputs.
  - Key abstractions such as the operand stack and dictionary stack should have comments describing the representation conventions you use, for example, how is the top of the stack represented, and in the case of the operand stack how are values of different types represented.

- Appropriate use of Python language features - 10%. Examples:
  - for loops when possible
  - functions rather than repeated code
  - clearly designed and implemented abstractions for the operand stack and dictionary stack, used consistently

- Deductions will be made if the program produces unspecified output (for example because debugging output is being produced).

Experience in previous versions of CptS 355 shows that almost all students are able to achieve grades greater than 90% on programming projects. If you are aspiring to or receiving grades less than 90% you should be worried.

This project will probably require 300-500 lines of code. If you fully understand the PostScript execution model it will be straightforward, since much of the required code is fairly repetitive – there's not that much difference between code for `add` and that for `sub`. If you do not fully understand PostScript execution you will spend a lot of time writing code that won't be worth much.

Example

While you are expected to become familiar enough with Postscript to write your own test cases, here is one program that your interpreter should handle. It illustrates a few issues that might be problematic.

```
/fact {
  dictz
  begin
  /n exch def
    n 2 lt
    {1}
    {n -1 add fact n mul }
  ifelse
  end
}def
5 fact =
```