Programming Assignment 1
CptS 355 - Fall 2011
An Interpreter for a PostScript-like Language

September 12, 2011

Overview

Assigned: September 9, 2011

Due: Monday, September 23, 2011 at 11:59:59PM. Develop all your code in a directory named “one”. When you are finished, make a zip file of the directory and turn it in using the turn-in web page as for the first assignment. It must be possible to run your program on an input file 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 11% of your final grade.

Policy: This assignment is to be your own work. Do not work with other students on it or copy code that you find on the web (or any other source for that matter): if you have a question or are stumped see the instructor or the TA.

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`
- 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 /
- built-in operators: `add`, `sub`, `mul`, `div`, `eq`, `lt`, `gt`, `and`, `or`, `not`; these take boolean operands only. Anything else is an error.
- built-in sequencing operators: `if`, `ifelse`
- stack operators: `dup`, `exch`, `pop`
- code constants: code between matched curly braces { ... }
- dictionary creation operator: `dict`; this takes one integer operand
- dictionary stack manipulation operators: `begin`, `end`. `begin` requires one dictionary operand; `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

```
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.

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

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:

- 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.) (Hint: you may have to type this in rather than copying it from the PDF file – there seems to be an issue with disagreements about character encodings between PDF and Python.)
- the operand stack
- SPS dictionaries
- the dictionary stack
- the execution stack: this can be implicit in the use of Python recursion by recursively calling your interpreter whenever a code array needs to be executed. We will discuss this aspect more in class.

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.

Grading

Assignments will be graded for

- Correct functioning on SPS programs - 70%. *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 `dict`, `begin` and `end` as *separate* operators.** Although the pattern `n dict begin` is very common in PS programs, it is not an operation in and of itself. Furthermore, it is *much easier* to implement separate `dict` and `begin` operators than some weird hybrid of the two.
  - Make sure that you implement nested { } parsing 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.

- Use re.findall for parsing input - 10%

- 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. If you do not fully understand PostScript execution you will spend a lot of time writing code that won't be worth much.