CptS355 - Assignment 4 (PostScript Interpreter - Part 1)

Summer 2023

An Interpreter for a Simple Postscript-like Language

Assigned: Tuesday, 20 June 2023

Due: Tuesday, 4 July 2023 by 10 p.m.

Weight: The entire interpreter project (Part 1 and Part 2 together) will count for **21%** of your course grade. This first part is worth 6% and second part is 15% - the intention is to make sure that you are on the right track and have a chance for mid-course correction before completing Part 2. However, note that the work and amount of code involved in Part 1 is a large fraction of the total project, so you need to get going on this part right away.

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

Turning in your assignment

All the problem solutions should be placed in a **single file named `HW4.py`** and all the tests should be placed **in `HW4tests.py`**. When you are done and certain that everything is working correctly, turn in your files by uploading for Assignment4 on Canvas.

The solution file that you upload **must** be named HW4.py. At the top of the 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.

Implement your code for Python3 (≥ 3.5). The TA will run all assignments using Python3 (≥ 3.5) interpreter. You will lose points if your code is incompatible with Python3 (≥ 3.5).

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 / additional test cases, (2) explain your code with appropriate comments, and (3) follow a good programming style.

The Problem

In this assignment you will write an interpreter in Python for a **simplified** 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
- string constants, e.g. `(CptS355)`: string delimited in parenthesis (Make sure to keep the parenthesis delimiters when you store the string constants in the `opstack` and the `dictstack`.)
- 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, mod, eq, lt, gt`
- built-in operators on string values: `length, get, getinterval, put.` See the lecture notes for more information on string functions.
- built-in conditional operators: `if`, `ifelse` (you will implement `if/ifelse` operators in Part2)
- built-in loop operator: `for` (you will implement `for` operator in Part 2).
- stack operators: `dup, copy, pop, clear, exch, roll`
- dictionary creation operator: `dict`; takes one operand from the operand stack, ignores it, and creates a new, *empty* dictionary on the operand stack (we will call this `psDict`)
- dictionary stack manipulation operators: `begin, end`. `begin` requires one dictionary operand on the operand stack; `end` has no operands.
- name definition operator: `def`.
- defining (using `def` we will call this `psDef`) and calling functions
- stack printing operator (prints contents of stack without changing it): `stack`

Part 1 - Requirements

In Part 1 you will build some essential pieces of the interpreter but not yet the full interpreter. The pieces you build will be driven by Python test code rather than actual Postscript programs. The pieces you are going to build first are:

1. The operand stack
2. The dictionary stack
3. Defining variables (with **`def`**) and looking up names
4. The operators that do not involve code arrays: all the operators **except `for` loop operator, `if/ifelse` operators, and calling functions.** (You will complete these in Part 2)

1. The Operand Stack - `opstack`

The operand stack should be implemented as a Python list. The list will contain Python integers, strings, and, later in Part 2, code arrays. Python integers and lists on the stack represent Postscript integer constants and array constants. Python strings which start with a slash / on the stack represent names of

Postscript variables. When using a list as a stack, assume that the top of the stack is the end of the list (i.e., the pushing and popping happens at the end of the list).

2. The Dictionary Stack - `dictstack`

The dictionary stack is also implemented as a Python list. It will contain Python dictionaries which will be the implementation for Postscript dictionaries. The dictionary stack needs to support adding and removing dictionaries at the top of the stack (i.e., end of the list), as well as defining and looking up names.

3. define and lookup

You will write two helper functions, `define` and `lookup`, to define a variable and to lookup the value of a variable, respectively.

The `define` function adds the "`name:value`" pair to the top dictionary in the dictionary stack. Your `psDef` function (i.e., your implementation of the Postscript `def` operator) should pop the name and value from operand stack and call the "`define`" function.

You should keep the `'/'` in the name constant when you store it in the `dictStack`.

```
def define(name, value):
    pass
    #add name:value pair to the top dictionary in the dictionary stack.
```

The `lookup` function should look-up the value of a given variable in the dictionary stack. In Part 2, when you interpret simple Postscript expressions, you will call this function for variable lookups and function calls.

```
def lookup(name):
    pass
    # return the value associated with name
    # What is your design decision about what to do when there is no
definition for "name"? If "name" is not defined, your program should not
break, but should give an appropriate error message.
```

4. Operators

Operators will be implemented as **zero-argument Python functions** that manipulate the operand and dictionary stacks. For example, the `div` operator could be implemented as the below Python function (with comments replaced by actual implementations)

```
def div():
    op1 = # pop the top value off the operand stack
    op2 = # pop the top value off the operand stack
    # push (op2 / op1) onto the operand stack
```

The `begin` and `end` operators are a little different in that they manipulate the dictionary stack in addition to or instead of the operand stack. Remember that the `psDict` operator affects only the operand stack.

(Note about `dict`: Remember that the `dict` operator takes an integer operand from the operand stack and pushes an empty dictionary to the operand stack (affects only the operand stack). The initial

size argument is ignored – Postscript requires it for backward compatibility of the `dict` operator with the early Postscript versions).

The `def` operator takes two operands from the operand stack: a string (recall that strings that start with "/" in the operand stack represent names of postscript variables) and a value. It changes the dictionary at the top of the dictionary stack so that the string is mapped to the value by that dictionary. Notice that `def` does not change the number of dictionaries on the dictionary stack!

SKELETON CODE:

You may start your implementation using the below skeleton code (also found in the file `HW4_skeleton.py`. Please make sure to use the function names provided.

```
#------------------------ 10% -------------------------------------
# The operand stack: define the operand stack and its operations
opstack = []  #assuming top of the stack is the end of the list

# Now define the helper functions to push and pop values on the opstack
# (i.e, add/remove elements to/from the end of the Python list)
# Remember that there is a Postscript operator called "pop" so we choose
# different names for these functions.
# Recall that `pass` in python is a no-op: replace it with your code.

def opPop():
    pass
    # opPop should return the popped value.
    # The pop() function should call opPop to pop the top value from the
opstack, but it will ignore the popped value.

def opPush(value):
    pass

#------------------------- 20% -------------------------------------
# The dictionary stack: define the dictionary stack and its operations
dictstack = []  #assuming top of the stack is the end of the list

# now define functions to push and pop dictionaries on the dictstack, to
# define name, and to lookup a name

def dictPop():
    pass
    # dictPop pops the top dictionary from the dictionary stack.

def dictPush(d):
    pass
    #dictPush pushes the dictionary 'd' to the dictstack.
    #Note that, your interpreter will call dictPush only when Postscript
    #"begin" operator is called. "begin" should pop the empty dictionary from
    #the opstack and push it onto the dictstack by calling dictPush.

def define(name, value):
    pass
    #add name:value pair to the top dictionary in the dictionary stack.
    #Keep the '/' in the name constant.
    #Your psDef function should pop the name and value from operand stack and
```

```
    #call the "define" function.

def lookup(name):
    pass
    # return the value associated with name
    # What is your design decision about what to do when there is no
definition for "name"? If "name" is not defined, your program should not
break, but should give an appropriate error message.


#--------------------------- 10% -------------------------------------
# Arithmetic and comparison operators: add, sub, mul, div, mod, eq, lt, gt
# Make sure to check the operand stack has the correct number of parameters
# and types of the parameters are correct.
def add():
    pass

def sub():
    pass

def mul():
    pass

def div():
    pass

def mod():
    pass

def eq():
    pass

def lt():
    pass

def gt():
    pass

#--------------------------- 15% -------------------------------------
# String operators: define the string operators length, get, getinterval, put
def length():
    pass

def get():
    pass

def getinterval():
    pass

def put():
    pass

#--------------------------- 25% -------------------------------------
# Define the stack manipulation and print operators: dup, copy, pop, clear,
exch, roll, stack
def dup():
    pass
```

```
def copy():
    pass

def pop():
    pass

def clear():
    pass

def exch():
    pass

def roll():
    pass

def stack():
    pass

#-------------------------- 20% -------------------------------------
# Define the dictionary manipulation operators: psDict, begin, end, psDef
# name the function for the def operator psDef because def is reserved in Python.
Similarly, call the function for dict operator as psDict.
# Note: The psDef operator will pop the value and name from the opstack and call your
own "define" operator (pass those values as parameters).
# Note that psDef()won't have any parameters.

def psDict():
    pass

def begin():
    pass

def end():
    pass

def psDef():
    pass
```

Important Note:

For all operators you need to implement basic checks, i.e., check whether there are sufficient numbers of values in the operand stack and check whether those values have correct types.

Examples:

`def` operator: the operands stack should have 2 values where the second value from top of the stack is a string starting with '/'

`get` operator : the operand stack should have 2 values; the top value on the stack should be an integer and the second value should be an string value.

Test your code:

```
def testAdd():
    opPush(1)
    opPush(2)
    add()
```

```
    if opPop() != 3:
        return False
    return True

def testLookup():
    opPush("/n1")
    opPush(3)
    psDef()
    if lookup("n1") != 3:
        return False
    return True

# go on writing test code for ALL of your code here; think about edge cases,
# and other points where you are likely to make a mistake. You can make use
# of unittest if you like, or continue with the above approach
```

Main Program

To run all the tests, your main may look like:

```
def main_part1():
    testCases = [('define',testDefine),('lookup',testLookup),('add',
testAdd), ('sub', testSub),('mul', testMul),('div', testDiv)]

# add the rest of the test functions to this list along with suitable names
    failedTests =
    [testName for (testName, testProc) in testCases if not testProc()]

    if failedTests:
        return ('Some tests failed', failedTests)
    else:
        return ('All part-1 tests OK')

if __name__ == '__main__':
    print(main_part1())
```