CptS355 - Assignment 2 (PostScript Interpreter - Part B) Spring 2017

An Interpreter for a Simple Postscript-like Language

Assigned: Tuesday February 14, 2017

Due: Monday February 27th, 2017

Weight: The entire interpreter project (Part A and Part B together) will count for 10% of your course grade. Part B is worth 8%.

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

Turning in your assignment

Rename your Part-A submission file as `HW2_partB.py` and continue developing your code in the `HW2_partB.py` file. I strongly encourage you to save a copy of periodically so you can go back in time if you really mess something up. To submit your assignment, turn in your file by uploading on the Assignment2(Interpreter-PartB) DROPBOX on Blackboard (under AssignmentSubmisions menu).

The file that you upload must be named HW2_partB.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. You may turn in your assignment up to 5 times. Only the last one submitted will be graded. Please let the instructor know if you need to resubmit it a 6th time.

Implement your code for Python 3. The TA will run all assignments using Python3 interpreter. You will lose points if your code is incompatible with Python 3.

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 / 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
- array constants, e.g. `[1 2 3 4] , [[1 2] 3 4]`

- 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`
- built-in operators on array values: `length, get, forall.` (You will implement `length` and `get` in PartA, and `forall` in Part B). Check lecture notes for more information on array operators.
- built-in loop operator: `for`; make sure that you understand the order of the operands on the stack. Try `for` loop examples on Ghostscript if necessary to help understand what is happening.
- stack operators: `dup, exch, pop, roll,copy, clear`
- dictionary creation operator: `dict`; takes one operand from the operand stack, ignores it, and creates a new, empty dictionary on the operand stack
- 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`) and calling functions
- stack printing operator (prints contents of stack without changing it): `stack`

Part B - Requirements

In Part 2 you will continue building the interpreter, making use of everything you built in Part 1. The pieces needed to complete the interpreter are:

1. Parsing Simple Postscript code
2. Handling of code arrays
3. Handling the **for loop** operator
4. Handling the **forall** operator (we will assume that arrays only have integer elements)
5. Function calling
6. Interpreting input strings (code) in the simple Postscript language.

1. Parsing

Parsing is the process by which a program is converted to a data structure that can be further processed by an interpreter or compiler. To parse the SPS programs, we will convert the continuous input text to a list of tokens and convert each token to our chosen representation for it. In SPS the tokens are: multidigit numbers with optional negative sign, multi-character names (with and without a preceding /), array constants enclosed in square brakets (i.e.,[ ]) and the curly brace characters (i.e., “}” and “{“)
We've already decided about how some of these will be represented: numbers as Python numbers, names as Python strings, etc. For code arrays, we will represent things falling between the braces using Python lists.

2., 3., 4., 5. Handling of code arrays, `for` loop operator, `forall` operator, function calling

Recall that a code array is pushed on the stack as a single unit when it is read from the input. Once a code array is on the stack several things can happen:

- if it is the body part of a for loop, it is recursively interpreted as part of the evaluation of the for loop. At each iteration of the `for` loop the loop index is pushed onto the stack. (We will get to interpreting momentarily).
- if it is the body part of `forall` operation, it is recursively interpreted for each element in the input constant array (i.e., an element of the input array is pushed onto the stack; the body of the `forall` is interpreted and the result is pushed onto the stack. This is repeated for every element in the input array. Fro simplicity we will assume all array elements are integers.)
- if it is the top item on the stack when a `def` is executed, it is stored as the value of the name defined by the `def`. Finally, if when a name is looked up you find that its value is a code array, the code array is recursively interpreted.

6. Interpreter

A key insight is that a complete SPS program is essentially a code array. It doesn't have curly braces around it but it is a chunk of code that needs to be interpreted. This suggests how to proceed:

- Convert the SPS program (a string of text) into a list of tokens and code arrays.
- Define a Python function interpret that takes one of these lists as input and processes it.
- Interpret the body of the for loop operator recursively, for each value of the loop index.
- Interpret the body of the forall operator recursively, for each value in the input array.
- When a name lookup produce a code array as its result, recursively interpret it, thus implementing Postscript function calls.

Parsing revisited

Parsing converts an SPS program in the form a string to a program in the form of a code array. It will work in two stages:

1. Convert all the string to a list of tokens. Given:

   "`/square {dup mul} def 1 square [1 2 3 4] length"))`"

   will be converted to

```
['/square', '{', 'dup', 'mul', '}', 'def', '1', 'square', '[1 2 3 4]', 'length']
```

2. And then convert the token list to a code array.

```
['/square', ['dup', 'mul'], 'def', 1, 'square', [1, 2, 3, 4], 'length']
```

   The difference between the two is that, in the second step (code array), everything between matching curly braces will be represented as a single element (which itself is a code array). In the process of converting from token list to code array the curly braces will disappear, and the string representations of numbers and arrays will be converted to Python integers and lists, respectively. For simplicity, we will assume that arrays only have integer elements.

1. Convert all the string to a list of tokens.

Use the following code to tokenize your SPS program.

```
# For tokenizing we'll use the “re” package for Regular Expressions.

import re

def tokenize(s):
    retValue = re.findall("/?[a-zA-Z][a-zA-Z0-9_]*|[[][a-zA-Z0-9_\s!][a-zA-Z0-9_\s!]*[]]|[-]?[0-9]+|[}{]+|%.*|[^ \t\n]", s)
```

Tokenize example:

```
# See what tokenize does
print (tokenize(
"""
    /fact{
    0 dict
            begin
                    /n exch def
                    1
                    n -1 1 {mul} for
            end
    }def
    [1 2 3 4 5] dup 4 get pop
    length
    fact
    stack
    """))

['/fact', '{', '0', 'dict', 'begin', '/n', 'exch', 'def', '1', 'n', '−1', '1', '{', 'mul', '}', 'for', 'end', '}', 'def', '[1 2 3 4 5]', 'dup', '4', 'get', 'pop', 'length', 'fact', 'stack']
```

2. Convert the token list to a code array

The output of tokenize isn't fully suitable because things between matching curly braces are not themselves grouped into a code array. We need to convert the output for the above example to:

```
['/fact', [0, 'dict', 'begin', '/n', 'exch', 'def', 1, 'n', −1, 1, ['mul'], 'for', 'end'], 'def', [1, 2, 3, 4, 5], 'dup', 4, 'get', 'pop', 'length', 'fact', 'stack']
```

Notice how in addition to grouping tokens between curly braces into lists, we've also converted the strings that represent numbers to Python numbers and the strings that represent arrays to Python lists.

The main issue in how to convert to a code array is how to group things that fall in between matching curly braces. There are several ways to do this. One possible way is to find the matching opening and closing parenthesis (“{“ and “}”) recursively, and including all tokens between them in a Python list.

Here is some starting code to find the matching parenthesis using an iterator. Here we iterate over the characters of an array (rather than a list of tokens) using a Python `iter` and we try to find the matching parenthesis (rather than curly braces). This code assumes that the input string includes opening and closing parenthesis characters only (e.g., “`(()(()))`”)

```
# The it argument is an iterator that returns left or right parenthesis characters.
```

```
# The sequence of characters returned by the iterator should represent a string of properly nested
# parentheses pairs, from which the leading '(' has already been removed. If the
# parenteses are not properly nested, returns False.

def groupMatching(it):
    res = ['(']
    for c in it:
        if c==')':
            res.append(')')
            return res
        else:
            # note how we use a recursive call to group the inner
            # matching parenthesis string and append it as a whole
            # to the list we are constructing.
            # Also note how we've already seen the leading '(' of this
            # inner group and consumed it from the iterator.
            res.append(groupMatching(it))
    return False

# function to turn a string of properly nested parentheses
# into a list of properly nested lists.

def group(s):
    if s[0]=='(':
        return groupMatching(iter(s[1:]))
    else: return False               # If it starts with ')' it is not properly nested

group('((()(())))')    # will return ['(', ['(', ')'], ['(', ['(', ')'], ')'], ')']
```

Here we use an iterator constructed from a string, but the `iter` function will equally well create an iterator from a list. Of course, your code has to deal with curly braces instead of parentheses and it must also deal with the tokens between curly braces and include all tokens between 2 matching opening/closing curly braces inside the code arrays . And don't forget that strings representing numbers/arrays should be converted to Python integers/lists at this stage as well.

The above code keeps the parenthesis characters in the resulting code array. In your parse algorithm, you should not include the curly brace characters. The structure of the code array itself is sufficient for what we will do with it. To illustrate the above point, consider the below modified version of `groupMatching` and `group` which doesn't put the '(' and ')' characters into its result. Just the structure of the result is sufficient to show the structure of the orginal string.

```
# The it argument is an iterator that returns left or right parenthesis characters.
# The sequence of return characters should represent a string of properly nested
# parentheses pairs, from which the leading '(' has already been removed. If the
# parenteses are not properly nested, returns False.

def groupMatching(it):
    res = []
    for c in it:
        if c==')':
            return res
        else:
            # note how we use a recursive call to group the inner
```

```
            # matching parenthesis string and append it as a whole
            # to the list we are constructing.
            res.append(groupMatching(it))
    return False

# function to turn a string of properly nested parentheses
# into a list of properly nested lists.
def group(s):
    if s[0]=='(':
        return groupMatching(iter(s[1:]))
    else: return False              # If it starts with ')' it is not properly nested

group('((()()))')  # will return:  [[], [[]]]
```

Your parsing implementation

Start with the group() function above; rename it as parse() and update the code so that it takes a list of string tokens, looks for matching curly braces in the token list, recursively converts the tokens between matching curly braces into code arrays (sub-lists of tokens).

```
# Write your parsing code here; it takes a list of tokens produced by tokenize
# and returns a code array;  Of course you may create additional functions to help you write parse()
#
def parse(tokens):
    pass
```

```
parse(['/fact', '{', '0', 'dict', 'begin', '/n', 'exch', 'def', '1', 'n', '-1', '1', '{', 'mul', '}', 'for', 'end', '}', 'def', '[1 2 3 4 5]', 'dup', '4', 'get', 'pop', 'length', 'fact', 'stack'])
```

returns :

```
['/fact', [0, 'dict', 'begin', '/n', 'exch', 'def', 1, 'n', -1, 1, ['mul'], 'for', 'end'], 'def', [1, 2, 3, 4, 5], 'dup', 4, 'get', 'pop', 'length', 'fact', 'stack']
```

3. Interpret code arrays

We're now ready to write the interpret function. It takes a code array as argument, and changes the state of the operand and dictionary stacks according to what it finds there, doing any output indicated by the SPS program (using the stack operator) along the way. Note that your `interpret` function needs to be recursive: `interpret` will be called recursively when a name is looked up and its value is a code array (i.e., function call), or when the body of the `for` loop and `forall` operators are interpreted.

(**Important note:** You may assume that SPS doesn't allow variables of array type and therefore we will not have test cases which binds an array value to a variable using the "def" operator. In SPS, code arrays are represented as Python lists. When a function call is made, the code array for that function will be looked up at the dictionary stack and will be interpreted. If variables of array type were allowed, this would have caused ambiguity between code arrays and array constants.)

```
# Write the necessary code here; again write
# auxiliary functions if you need them. This will probably be the largest
# function of the whole project, but it will have a very regular and obvious
# structure if you've followed the plan of the assignment.
#
def interpret(code): # code is a code array
    pass
```

4. Interpret the SPS code

Finally, we can write the `interpreter` function that treats a string as an SPS program and interprets it.

```
# Copy this to your HW2_partB.py file>
def interpreter(s): # s is a string
    interpret(parse(tokenize(s)))
```

Testing

First test the parsing

Before even attempting to run your full interpreter, make sure that your parsing is working correctly. Make sure you get the correct parsed output for the following.

- Make sure that the integer constants are converted to Python integers.
- Make sure that the array constants are converted to Python lists (remember that SPS arrays only include integer values; therefore you can directly convert all array elements to integers).
- Make sure that code arrays are represented as sublists.

```
parse(tokenize(
"""
/square {dup mul} def 1 square 2 square 3 square add add
"""
))
```

```
['/square', ['dup', 'mul'], 'def', 1, 'square', 2, 'square', 3, 'square', 'add', 'add']
```

```
parse(tokenize(
"""
/n 5 def 1 n -1 1 {mul} for
"""
))
```

```
['/n', 5, 'def', 1, 'n', -1, 1, ['mul'], 'for']
```

```
parse(tokenize(
"""
    /sum { -1 0 {add} for} def
    0
    [1 2 3 4] length
    sum
    2 mul
    [1 2 3 4] {2 mul} forall
    add add add
    stack
"""))
```

```
['/sum', [-1, 0, ['add'], 'for'], 'def', 0, [1, 2, 3, 4], 'length', 'sum', 2, 'mul', [1, 2
, 3, 4], [2, 'mul'], 'forall', 'add', 'add', 'add', 'stack']
```

```
print (tokenize(
"""
    /fact{
    0 dict
            begin
                    /n exch def
                    1
                    n -1 1 {mul} for
            end
    }def
    [1 2 3 4 5] dup 4 get pop
    length
    fact
    stack
    """))
```

```
  ['/fact', [0, 'dict', 'begin', '/n', 'exch', 'def', 1, 'n', -1, 1, ['mul'], 'for', 'end'],
'def', [1, 2, 3, 4, 5], 'dup', 4, 'get', 'pop', 'length', 'fact', 'stack']
```

Finally, test the full interpreter. Run the test cases on the Ghostscript shell to check for the correct output and compare with the output from your interpreter.

```
interpreter (
"""
    /fact{
    0 dict
            begin
                    /n exch def
                    1
                    n -1 1 {mul} for
            end
    }def
    [1 2 3 4 5] dup 4 get pop
    length
    fact
    stack
 """)
```

The above test case should print [120].