CptS355 - Assignment 4 (PostScript Interpreter - Part 2) Fall 2018

An Interpreter for a Simple Postscript-like Language

Assigned: Sunday, November 4, 2018
Due: Wednesday, November 14, 2018

Weight: The entire interpreter project (Part 1 and Part 2 together) will count for 12% of your course grade. Part 2 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-1 submission file as `HW4_part2.py` and continue developing your code in the `HW4_part2.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 dropbox on Blackboard (under AssignmentSubmisions menu).

The file that you upload must be named HW4_part2.py . Be sure to include your name as a comment at the top of the file. You may turn in your assignment up to 4 times. Only the last one submitted will be graded.

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] or [[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, eq, lt, gt`
- built-in operators on boolean values: `and, or, not;` these take boolean operands only. Anything else is an error.
- built-in conditional 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.
- built-in loop operator: `for` (you will implement `for` operator in Part 2).
- built-in operators on array values: `length, get, forall.` (You will implement `length` and `get` in Part 1, and `forall` in Part 2). Check lecture notes for more information on array operators.
- stack operators: `dup, exch, pop, 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 2 - 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 **if** and **ifelse** operators (write the Python methods `psIf` and `psIfelse` implementing these operators)
4. Handling the **array values** (we will assume that arrays only have integer elements)
5. Handling the **for** and **forall** operators (write the Python methods `psFor` and `forAll` implementing these operators)
6. Function calling
7. 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: numbers with optional negative sign, multi-character names (with and without a preceding /), array constants enclosed in square brackets (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, booleans as Python booleans, arrays as Python lists, etc. For code arrays, we will represent things falling between the braces using Python lists.

2.- 6. Handling of code arrays: `if/ifelse, for, forall` operators and 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 top item on the stack when a `def` is executed, it is stored as the value of the name defined by the `def`.
- if it is the body part of an `if/ifelse` operator, it is recursively interpreted as part of the evaluation of the `if/ifelse`. For the if operator, the code array is interpreted only if the "condition" argument for `if` operator is true. For the `ifelse` operator, if the "condition" argument is true, first code array is interpreted, otherwise the second code array is evaluated.
- 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.
- if it is the function argument for the `forall` operator, then it is recursively interpreted for every value in the input array.
- finally, if when a name is looked up you find that its value is a code array, the code array is recursively interpreted.
  (We will get to interpreting momentarily).

3. 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 `if/ifelse, for ,` and `forall` operators recursively.
- When a name lookup produces a code array as its result, recursively interpret it, thus implementing Postscript function calls.

Implementing Your Postscript Interpreter

I. Parsing

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 2 3 4] {square} forall add add add 30 eq stack"
```

will be converted to

```
['/square', '{', 'dup', 'mul', '}', 'def', '[1 2 3 4]', '{', 'square', '}',
'forall', 'add', 'add', 'add', '30', 'eq', 'stack']
```

Use the following code to tokenize your SPS program.

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

Another tokenize example:

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

returns

```
['[1 2 3 4 5]', 'dup', 'length', '/n', 'exch', 'def', '/fact', '{', '0',
'dict', 'begin', '/n', 'exch', 'def', 'n', '2', 'lt', '{', '1', '}', '{',
'n', '1', 'sub', 'fact', 'n', 'mul', '}', 'ifelse', 'end', '}', 'def', 'n',
'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:

```
[[1,2,3,4,5], 'dup', 'length', '/n', 'exch', 'def', '/fact', [0, 'dict',
'begin', '/n', 'exch', 'def', 'n', 2, 'lt', [1],
['n', 1, 'sub', 'fact', 'n', 'mul'], 'ifelse', 'end'], 'def', 'n', '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, the strings that represent booleans to Python boolean values, and strings that represent Postscript 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 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 a string (rather than a list of tokens) using a Python `iter` and we try to find the matching curly braces. This code assumes that the input string includes opening and closing curly braces only (e.g., "{{}{{}}}")

```
# The it argument is an iterator. The sequence of return characters should
# represent a string of properly nested {} parentheses pairs, from which
# the leasing '{' has been removed. If the parentheses 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. Also note how we have already seen the leading
            # '{' of this inner group and consumed it from the iterator.
            res.append(groupMatching(it))
    return False

# Function to parse a string of { and } braces. Properly nested parentheses
# are arranged into a list of properly nested lists.
def group(s):
    res = []
    it = iter(s)
    for c in it:
        if c=='}':  #non matching closing paranthesis; return false
            return False
        else:
            res.append(groupMatching(it))
    return res
```

`So, 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 the tokens between curly braces and include all tokens between 2 matching opening/closing curly braces inside the code arrays .

To illustrate the above point, consider this modified version of `groupMatching` and `group` (now called `parse`) which also handles the tokens before the first curly braces and between matching braces.

```
# The it argument is an iterator.
# The sequence of return characters should represent a list of properly nested
# tokens, where the tokens between '{' and '}' is included as a sublist. If the
# parenteses in the input iterator is not properly nested, returns False.
def groupMatching2(it):
    res = []
    for c in it:
        if c == '}':
            return res
        elif c=='{':
            # Note how we use a recursive call to group the tokens inside the
            # inner matching parenthesis.
            # Once the recursive call returns the code array for the inner
            # paranthesis, it will be appended to the list we are constructing
            # as a whole.
            res.append(groupMatching2(it))
        else:
            res.append(c)
    return False
```

```
# Function to parse a list of tokens and arrange the tokens between { and } braces
# as code-arrays.
# Properly nested parentheses are arranged into a list of properly nested lists.
def group2(L):
    res = []
    it = iter(L)
    for c in it:
        if c=='}':  #non matching closing paranthesis; return false since there is
                    # a syntax error in the Postscript code.
            return False
        elif c=='{':
            res.append(groupMatching2(it))
        else:
            res.append(c)
    return res
```

```
group2(['b', 'c', '{', 'a', '{', 'a', 'b', '}', '{', '{', 'e', '}', 'a', '}', '}'])
```

returns

```
['b', 'c', ['a', ['a', 'b'], [['e'], 'a']]]
```

Your parsing implementation

Start with the `groupMatching2` and `group2` functions above; rename `group2` as `parse` and update the code so that the strings representing numbers/booleans/arrays are converted to Python integers/booleans/lists.

```
# Write your parsing code here; it takes a list of tokens produced by
# tokenize  and returns a code array;
def parse(tokens):
    pass
```

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

returns:

```
[[1, 2, 3, 4, 5], 'dup', 'length', '/n', 'exch', 'def', '/fact', [0, 'dict', 'begin',
'/n', 'exch', 'def', 'n', 2, 'lt', [1], ['n', 1, 'sub', 'fact', 'n', 'mul'], 'ifelse',
'end'], 'def', 'n', 'fact', 'stack']
```

II. 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 `interpretSPS` function needs to be recursive: `interpretSPS` 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 `if`, `ifelse`, `for`, and `forall` operators are interpreted.

III. Interpret the SPS code

```
# 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 interpretSPS(code): # code is a code array
    pass
```

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

```
# Copy this to your HW4_part2.py file>
def interpreter(s): # s is a string
    interpretSPS(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:

1.

```
input1 = """
  /square {dup mul} def
  [1 2 3 4] {square} forall
  add add add 30 eq true
  stack
"""
```

`tokenize(input1)` will return:

```
['/square', '{', 'dup', 'mul', '}', 'def', '[1 2 3 4]', '{', 'square', '}',
'forall', 'add', 'add', 'add', '30', 'eq', 'true', 'stack']
```

`parse(tokenize(input1))` will return:

```
['/square', ['dup', 'mul'], 'def', [1, 2, 3, 4], ['square'], 'forall', 'add',
'add', 'add', 30, 'eq', True, 'stack']
```

2.

```
input2 = """
  [1 2 3 4 5] dup length /n exch def
  /fact {
      0 dict begin
         /n exch def
         n 2 lt
         { 1}
         {n 1 sub fact n mul }
         ifelse
      end
  } def
  n fact stack
"""
```

`tokenize(input2)` will return:

```
['[1 2 3 4 5]', 'dup', 'length', '/n', 'exch', 'def', '/fact', '{', '0', 'dict',
'begin', '/n', 'exch', 'def', 'n', '2', 'lt', '{', '1', '}', '{', 'n', '1', 'sub',
'fact', 'n', 'mul', '}', 'ifelse', 'end', '}', 'def', 'n', 'fact', 'stack']
```

`parse(tokenize(input2))` will return:

```
[[1, 2, 3, 4, 5], 'dup', 'length', '/n', 'exch', 'def', '/fact', [0, 'dict',
'begin', '/n', 'exch', 'def', 'n', 2, 'lt', [1], ['n', 1, 'sub', 'fact', 'n',
'mul'], 'ifelse', 'end'], 'def', 'n', 'fact', 'stack']
```

3.

```
input3 = """
  [9 9 8 4 10] {dup 5 lt {pop} if} forall
  stack
"""
```

`tokenize(input3)` will return:

```
['[9 9 8 4 10]', '{', 'dup', '5', 'lt', '{', 'pop', '}', 'if', '}', 'forall',
'stack']
```

`parse(tokenize(input3))` will return:

```
[[9, 9, 8, 4, 10], ['dup', 5, 'lt', ['pop'], 'if'], 'forall', 'stack']
```

4.

```
input4 = """
  [1 2 3 4 5] dup length exch {dup mul}  forall
  add add add add
  exch 0 exch -1 1 {dup mul add} for
  eq stack
"""
```

`tokenize(input4)` will return:

```
['[1 2 3 4 5]', 'dup', 'length', 'exch', '{', 'dup', 'mul', '}', 'forall', 'add',
'add', 'add', 'add', 'exch', '0', 'exch', '-1', '1', '{', 'dup', 'mul', 'add', '}',
'for', 'eq', 'stack']
```

`parse(tokenize(input4))` will return:

```
[[1, 2, 3, 4, 5], 'dup', 'length', 'exch', ['dup', 'mul'], 'forall', 'add', 'add',
'add', 'add', 'exch', 0, 'exch', -1, 1, ['dup', 'mul', 'add'], 'for', 'eq',
'stack']
```

When you parse:

- Make sure that the integer/real constants are converted to Python integers/floats.
- Make sure that the boolean constants are converted to Python booleans.
- Make sure that the array constants are converted to Python lists.
- Make sure that code arrays are represented as sublists.

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.

When you run your tests make sure to clear the opStack and dictStack.

`interpreter(input1)` should print:

```
True
True
```

`interpreter(input2)` should print:

```
120
[1, 2, 3, 4, 5]
```

`interpreter(input3)` should print:

```
10
8
9
9
```

`interpreter(input4)` should print:

```
True
```