CptS355 - Assignment 4 (PostScript Interpreter - Part 2) Spring 2020

An Interpreter for a Simple Postscript-like Language

Assigned: Monday, March 30, 2020
Due: Wednesday, April 15, 2020

Weight: The entire interpreter project (Part 1 and Part 2 together) will count for 12% of your course grade. Part 2 is worth 9%.

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. (Your `HW4_part2.py` file should include your solution for both part 1 and 2. Please don't include part1 and part2 code in two separate files.) I strongly encourage you to save a copy 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.

The file that you upload must be named HW4_part2.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 Python 3. The TA will run all assignments using Python3 interpreter. You will lose points if your code is incompatible with Python 3.

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. `1, 2, 3, -4, -5`
- boolean constants, e.g. `true, false`
- array constants, e.g. `[1 2 3 4], [-1 2 3 -4], [1 x 3 4 add 2 sub],` `[1 2 x 4]` where `x` is a variable. For simplicity we will assume that SPS arrays are not nested

(can't have subarrays). (If the array includes variables or operands, you need to first evaluate its elements before you push it onto the stack. This will be done in **part2.**)

- 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, eq, lt, gt`
- built-in operators on boolean values: `and, or, not` (we will call these `psAnd, psOr,` and `psNot`)
- built-in operators on array values: `length, get, getinterval, put, putinterval, forall.` See the lecture notes for more information on array functions (you will implement `forall` operator in **Part 2**).
- built-in conditional operators: `if`, `ifelse` (you will implement `if/ifelse` operators in **Part2**)
- built-in loop operator: `repeat` (you will implement `repeat` operator in **Part 2**).
- stack operators: `dup, copy, count, pop, clear, exch, mark, cleartomark, counttomark`
- dictionary creation operator: `dict`; <u>takes one operand from the operand stack, ignores it, and creates a new, empty dictionary on the operand stack</u> (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 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`)
4. Handling the **`repeat`** and **`forall`** operators (write the Python method `psRepeat` and `forall`)
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: numbers with optional negative sign, multi-character names (with and without a preceding /), array constants enclosed in parenthesis (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, array constants as Python lists, etc. **For code-array, we will represent tokens falling between the braces using a Python dictionary** (a dictionary that includes a list of tokens).

2-5. Handling of code-arrays: `if/ifelse`, `repeat`, `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 (i.e. the code array is the body of a function), 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 <u>recursively interpreted</u> 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 `repeat` operator, it is <u>recursively interpreted</u> as part of the evaluation of the repeat loop.
- finally, when a function is called (when a name is looked up its value is a code-array), the function body (i.e., the code-array) is recursively interpreted .

  (We will get to interpreting momentarily).

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 store it in a dictionary.
- Define a Python function "`interpretSPS`" that takes one of these dictionaries (code-arrays) as input and processes the tokens.
- Interpret the body of the `if/ifelse`, `repeat`, 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  0 [-5 -4 3 -2 1]
 {square add} forall 55 eq false and stack"
```

will be converted to

```
['/square', '{', 'dup', 'mul', '}', 'def', '0', '[-5 -4 3 -2 1]',
 '{', 'square', 'add', '}', 'forall', '55', 'eq', 'false', 'and', '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_]*|[\[][a-zA-Z-?0-9_\s!][a-zA-Z-?0-9_\s!]*[\]]|[-]?[0-9]+|[}{]+|%.*|[^ \t\n]", s)
```

2. Convert the token list to a code-array

The output of tokenize is a list of tokens. To differentiate code-arrays from array constants, we will store this token list in a dictionary with key 'codearray'. The nested-code arrays will also be included as a dictionary. We need to convert the above example to:

```
{'codearray': ['/square', {'codearray': ['dup', 'mul']}, 'def',
               0, [-5, -4, 3, -2, 1],
               {'codearray': ['square', 'add']}, 'forall',
               55, 'eq', False, 'and', 'stack']

}
```

Notice how in addition to grouping tokens between curly braces into code-arrays, we've also converted the strings that represent numbers to Python numbers, the strings that represent booleans to Python boolean values, and the strings that represent constant 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 `iterator` 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 parenthesis; 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 `groupMatching` and `group` (now called `groupMatch` and `parse`) which also handle the tokens before the first curly braces and between matching braces.

```
# The it argument is an iterator.
# The tokens between '{' and '}' is included as a sub code-array (dictionary). If the
# parenteses in the input iterator is not properly nested, returns False.
def groupMatch(it):
    res = []
    for c in it:
        if c == '}':
            return {'codearray':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
            # parenthesis, it will be appended to the list we are constructing
            # as a whole.
            res.append(groupMatch(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 dictionaries.
def parse(L):
    res = []
    it = iter(L)
    for c in it:
        if c=='}':  #non matching closing parenthesis; return false since there is
                    # a syntax error in the Postscript code.
            return False
        elif c=='{':
            res.append(groupMatch(it))
        else:
            res.append(c)
    return {'codearray':res}
```

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

returns

```
{'codearray': ['b', 'c', {'codearray': ['a', {'codearray': ['a', 'b']},
               {'codearray': [ {'codearray': ['e']}, 'a' ]} ]} ]}
```

Your parsing implementation
Start with the `groupMatch` and `parse` functions above (also included in the given skeleton code); **update the `parse` code** so that the strings representing numbers/booleans/arrays are converted to Python integers/booleans/lists.

```
['/square', '{', 'dup', 'mul', '}', 'def', '0', '[-5 -4 3 -2 1]',
 '{', 'square', 'add', '}', 'forall', '55', 'eq', 'false', 'and', 'stack']
```

should return:

```
{'codearray': ['/square', {'codearray': ['dup', 'mul']}, 'def',
               0, [-5, -4, 3, -2, 1],
               {'codearray': ['square', 'add']}, 'forall',
               55, 'eq', False, 'and', '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`, `repeat`, and `forall` operators are interpreted.

Interpret the SPS code

```
# 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.
# Write additional auxiliary functions if you need them.
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.

```
def interpreter(s): # s is a string
    interpretSPS(parse(tokenize(s)))
```

Testing

Sample unit tests for the interpreter are attached to the assignment dropbox. **You should provide 5 additional test methods in addition to the provided tests.** Make sure that your tests include several operators. You will loose points if you fail to provide tests or if your tests are too simple.

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 testcases (see pages 7 through 12).

When you parse:

- Make sure that the integer constants are converted to Python integers.
- Make sure that the boolean constants are converted to Python booleans.
- Make sure that constant arrays are represented as Python lists.
- Make sure that code-arrays are represented as dictionaries.

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.

```
input1 = """
          /square {dup mul} def
          0 [-5 -4 3 -2 1]
          {square add} forall
          55 eq false and
     """
```

`tokenize(input1)` will return:

```
['/square', '{', 'dup', 'mul', '}', 'def', '0',
'[-5 -4 3 -2 1]', '{', 'square', 'add', '}', 'forall', '55', 'eq',
'false', 'and']
```

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

```
{'codearray': ['/square', {'codearray': ['dup', 'mul']}, 'def', 0, [-5,-4,
3, -2, 1], {'codearray': ['square', 'add'
]}, 'forall', 55, 'eq', False, 'and']}
```

After `interpreter(input1)` is called the opstack content will be:

```
[False]
```

```
input2 = """
          /x 1 def
          /y 2 def
          1 dict begin
          /x 10 def
          1 dict begin /y 3 def x y end
          /y 20 def
          x y
          end
          x y
     """
```

`tokenize(input2)` will return:

```
['/x', '1', 'def', '/y', '2', 'def', '1', 'dict', 'begin', '/x', '10',
'def', '1', 'dict', 'begin', '/y', '3', 'def', 'x', 'y', 'end', '/y',
'20', 'def', 'x', 'y', 'end', 'x', 'y']
```

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

```
{'codearray': ['/x', 1, 'def', '/y', 2, 'def', 1, 'dict', 'begin', '/x',
10, 'def', 1, 'dict', 'begin', '/y', 3, 'def', 'x', 'y', 'end', '/y', 20,
'def', 'x', 'y', 'end', 'x', 'y']}
```

After `interpreter(input2)` is called the opstack content will be:

```
[10, 3, 10, 20, 1, 2]
```

```
input3 = """
        [3 2 1 3 2 2 3 5 5] dup
        3
        [4 2 1 4 2 3 4 5 1] 6 3 getinterval
        Putinterval
    """
```

`tokenize(input3)` will return:

```
['[3 2 1 3 2 2 3 5 5]', 'dup', '3',
'[4 2 1 4 2 3 4 5 1]', '6', '3', 'getinterval', 'putinterval']
```

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

```
{'codearray': [[3, 2, 1, 3, 2, 2, 3, 5, 5], 'dup', 3, [4, 2, 1, 4, 2, 3,
4, 5, 1], 6, 3, 'getinterval', 'putinterval']}
```

After `interpreter(input3)` is called the opstack content will be:

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

```
input4 = """
        /a [1 2 3 4 5] def
       a {dup mul} forall
    """
```

`tokenize(input4)` will return:

```
['/a', '[1 2 3 4 5]', 'def', 'a', '{', 'dup', 'mul', '}', 'forall']
```

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

```
{'codearray': ['/a', [1, 2, 3, 4, 5], 'def', 'a', {'codearray': ['dup',
'mul']}, 'forall']}
```

After `interpreter(input4)` is called the opstack content will be:

```
[1, 4, 9, 16, 25]
```

```
input5 = """
            a [10 20 30 40 50] def
           [4 2 0] {a exch get} forall
      """
```

`tokenize(input5)` will return:

```
['/a', '[10 20 30 40 50]', 'def', '[4 2 0]', '{', 'a', 'exch', 'get', '}',
'forall']
```

`parse(tokenize(input5))` will return:

```
{'codearray': ['/a', [10, 20, 30, 40, 50], 'def', [4, 2, 0], {'codearray':
['a', 'exch', 'get']}, 'forall']}
```

After `interpreter(input5)` is called the opstack content will be:

```
[50, 30, 10]
```

```
input6 = """
           /N 5 def
            N { N N mul /N N 1 sub def} repeat
      """
```

`tokenize(input6)` will return:

```
['/N', '5', 'def', 'N', '{', 'N', 'N', 'mul', '/N', 'N', '1', 'sub',
'def', '}', 'repeat']
```

`parse(tokenize(input6))` will return:

```
{'codearray': ['/N', 5, 'def', 'N', {'codearray': ['N', 'N', 'mul', '/N',
'N', 1, 'sub', 'def']}, 'repeat']}
```

After `interpreter(input6)` is called the opstack content will be:

```
[25, 16, 9, 4, 1]
```

```
input7 = """
            /n 5 def
            /fact {
                0 dict begin
                /n exch def
                n 2 lt
                { 1}
                {n 1 sub fact n mul }
                ifelse
                end
```

```
        } def
        n fact
    """
```

`tokenize(input7)` will return:

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

`parse(tokenize(input7))` will return:

```
{'codearray': ['/a', [10, 20, 30, 40, 50], 'def', [4, 2, 0], {'codearray':
{'codearray': ['/n', 5, 'def', '/fact', {'codearray': [0, 'dict', 'begin',
'/n', 'exch', 'def', 'n', 2, 'lt', {'codearray': [1]}, {'codearray': ['n',
1, 'sub', 'fact', 'n', 'mul']}, 'ifelse', 'end']}, 'def', 'n', 'fact']}
```

After `interpreter(input7)` is called the opstack content will be:

```
[120]
```

```
input8 = """
      /fact{
            0 dict
            begin
                /n exch def
                1
                n  {n mul /n n 1 sub def} repeat
            end
        } def
        6 fact
    """
```

`tokenize(input8)` will return:

```
['/fact', '{', '0', 'dict', 'begin', '/n', 'exch', 'def', '1', 'n', '{',
'n', 'mul', '/n', 'n', '1', 'sub', 'def', '}', 'repeat', 'end', '}',
'def', '6', 'fact']
```

`parse(tokenize(input8))` will return:

```
{'codearray': ['/fact', {'codearray': [0, 'dict', 'begin', '/n', 'exch',
'def', 1, 'n', {'codearray': ['n', 'mul', '/n', 'n', 1, 'sub', 'def']},
'repeat', 'end']}, 'def', 6, 'fact']}
```

After `interpreter(input8)` is called the opstack content will be:

```
[720]
```

```
input9 = """
            /sumArray { 0 exch {add} forall  } def
            /x 5 def
            /y 10 def
            [1 2 3 add 4 x] sumArray
            [x 7 8 9 y] sumArray
            [y 2 5 mul 1 add 12] sumArray
        """
```

`tokenize(input9)` will return:

```
['/sumArray', '{', '0', 'exch', '{', 'add', '}', 'forall', '}', 'def',
'/x', '5', 'def', '/y', '10', 'def', '[1 2 3 add 4 x]', 'sumArray',
'[x 7 8 9 y]', 'sumArray','[y 2 5 mul 1 add 12]', 'sumArray']
```

`parse(tokenize(input9))` will return:

```
{'codearray': ['/sumArray', {'codearray': [0, 'exch', {'codearray':
['add']}, 'forall']}, 'def', '/x', 5, 'def', '/y', 10, 'def', [1, 2, 3,
'add', 4, 'x'], 'sumArray', ['x', 7, 8, 9, 'y'], 'sumArray', ['y', 2, 5,
'mul', 1, 'add', 12], 'sumArray']}
```

After `interpreter(input9)` is called the opstack content will be:

```
[15, 39, 33]
```

```
input10 = """
            1 2 3 4 5 count copy 15 5 {exch sub} repeat 0 eq
        """
```

`tokenize(input10)` will return:

```
['1', '2', '3', '4', '5', 'count', 'copy', '15', '5', '{', 'exch', 'sub',
'}', 'repeat', '0', 'eq']
```

`parse(tokenize(input10))` will return:

```
{'codearray': [1, 2, 3, 4, 5, 'count', 'copy', 15, 5, {'codearray':
['exch', 'sub']}, 'repeat', 0, 'eq']}
```

After `interpreter(input10)` is called the opstack content will be:

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

```
input11 = """
            /xor {true eq {true eq {false} {true} ifelse } {true eq {true}
{false} ifelse } ifelse } def
           true [true false and false true or false false] {xor} forall
        """
```

`tokenize(input11)` will return:

```
['/xor', '{', 'true', 'eq', '{', 'true', 'eq', '{', 'false', '}', '{',
'true', '}', 'ifelse', '}', '{', 'true', 'eq', '{', 'true', '}', '{',
```

```
'false', '}', 'ifelse', '}', 'ifelse', '}', 'def', 'true', '[true false
and false true or false false]', '{', 'xor', '}', 'forall']
```

`parse(tokenize(input11))` will return:

```
{'codearray': ['/xor', {'codearray': [True, 'eq', {'codearray': [True,
'eq', {'codearray': [False]}, {'codearray': [True]}, 'ifelse']},
{'codearray': [True, 'eq', {'codearray': [True]}, {'codearray': [False]},
'ifelse']}, 'ifelse']}, 'def', True, [True, False, 'and', False, True,
'or', False, False], {'codearray': ['xor']}, 'forall']}
```

After `interpreter(input11)` is called the opstack content will be:

```
[False]
```