# CptS355 - Assignment 5 (PostScript Interpreter - Part 2)
## Summer 2023

## An Interpreter for a Simple Postscript-like Language

**Assigned:** Thursday, 29 June 2023
**Due:** <span style="color:red">**Friday, 14 July 2023 by 10 p.m.**</span>

**Weight:** The entire interpreter project (Part 1 and Part 2 together) will count for 21% of your course grade. Part 2 is worth 15%.

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

## Turning in your assignment

Copy your `HW4.py` submission file to `HW5.py` and continue developing your code in the `HW5.py` file. 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 `HW5.py` file by uploading to Canvas in the normal manner. You can make use of the `HW5_skeleton.py` file for some of the test inputs if you like, but you will want to work from your previous `HW4.py` solution as specified above.

**The file that you upload must be named HW5.py.** Be sure to include your name as a comment at the top of the file. You may turn in your assignment up to 3 times. Only the last one submitted will be graded.

**Implement your code for Python >=3.5. The TA will run all assignments using Python >=3.5 interpreter. You will lose points if your code is incompatible.**

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 do not (1) explain your code with appropriate comments, and (2) 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` (you will revise your implementation for `put` operator in **Part2**).
- 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 **Part2**).
- stack operators: `dup, copy, pop, clear, exch, roll`
- 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. Revising the string **put** operator (some of you may have done this already in Part 1).
2. Parsing "Simple Postscript" code
3. Handling of code-arrays
4. Handling the **if** and **ifelse** operators (write the Python methods `psIf` and `psIfelse`)
5. Handling the **for** operator (write the Python method `psFor`)
6. Function calling
7. Interpreting input strings (code) in the simple Postscript language

### 1. Revise the string **put** operator

Remember that the `put` operator gets a string, an index (integer), and an ASCII character (from the stack), and replaces the character at index with the new character in the string. Revise your string `put` operator implementation from part-1 as follows:

When a string is updated by the "`put`" operator, all copies of the same string (i.e., the strings that have the same `object-id`) in the `opstack` and the top (*current*) dictionary on `dictstack` should be updated. Since Python strings are immutable, rather than changing the value of the string itself, you <u>should update each stack entry</u> in the `opstack` and the `dictstack` that refer to the same string <u>with the updated string value</u>.

<u>Note</u>: In Python, each object has an associated id which can be retrieved using the `id()` method (for example when s='355', `id(s)` will give the unique id for the specified string object.)

The above approach is not exactly replicating Postscript `put`. However, we are simplifying the language to make the implementation easier.

You can unit test your `put` implementation using the following function:

```python
def testPut():
    opPush("(This is a test _)")
    dup()
    opPush("/s")
    exch()
    psDef()
    dup()
    opPush(15)
    opPush(48)
    put()
    if lookup("s") != "(This is a test 0)" or opPop()!= "(This is a test 0)":
        return False
    return True
```

## 2. 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 /), string constants enclosed in parentheses (i.e., ( ) ) and the curly brace characters (i.e., "}" and "{"). We have already decided about how some of these will be represented: numbers as Python numbers, names as Python strings, Booleans as Python Booleans, string constants as Python strings, etc. For code-arrays, we will represent things falling between the braces using Python lists.

## 3 - 6.  Handling of code-arrays: `if/ifelse,  for` 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:

3.  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`
4.  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
5.  if it is the body part of a `for` operator, it is <u>recursively interpreted</u> as part of the evaluation of the for loop. **At each iteration of the `for` loop the loop index is pushed onto the stack**.
6.  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).

### 7. Interpreter

A **key insight** is that a complete SPS program is essentially a code-array. It does not 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, and for` 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 of a string to a program in the form of a code-array. It will work in two stages:

*1.  Convert entire string to a list of tokens.*

Given:
```
"/square {dup mul} def  0 1 1 5 {square add} for 55 eq stack"
```

will be converted to

```
['/square', '{', 'dup', 'mul', '}', 'def', '0', '1', '1', '5', '{', 'square',
'add', '}', 'for', '55', '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_()]*|[-]?[0-9]+|[}{]+|%.*|[^ \t\n]", s)
```

**Important note**: To simplify parsing, we will assume that SPS string constant values do not include any space characters. (The regular expression in the above `tokenize` function will not work with constant strings that include spaces.) Additionally, float numbers will not be parsed correctly either, as the '.' is interpreted as whitespace. For the purposes of the part 2 interpreter, we will only deal with integer numeric input, no floats.

Another tokenize example:
```
print (tokenize("""
    /pow2 {/n exch def
          (pow2_of_n_is) dup 8 n 48 add put
           1 n -1 1 {pop 2 mul} for
          } def
    (Calculating_pow2_of_9) dup 20 get 48 sub pow2
    stack
    """
))
```
returns

```
['/pow2', '{', '/n', 'exch', 'def', '(Pow2_of_n_is)', 'dup', '8', 'n', '48',
'add', 'put', '1', 'n', '-1', '1', '{', 'pop', '2', 'mul', '}', 'for', '}',
'def', '(Calculating_pow2_of_9)', 'dup', '20', 'get', '48', 'sub', 'pow2',
'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:

```
['/pow2', ['/n', 'exch', 'def', '(Pow2_of_n_is)', 'dup', 8, 'n', 48, 'add',
'put', 1, 'n', -1, 1, ['pop', 2, 'mul'], 'for'], 'def',
'(Calculating_pow2_of_9)', 'dup', 20, 'get', 48, 'sub', 'pow2', '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 Booleans to Python Boolean values. We kept the parenthesis delimiters for SPS string constants.

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., "{{}{{}}}")

```python
# The it argument is an iterator. The sequence of return characters should
# represent a string of properly nested {} parentheses pairs, from which
# the leading '{' has been removed. If the parentheses are not properly
# nested, returns False.
def groupMatching1(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(groupMatching1(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(groupMatching1(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.

```python
# 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
            # parenthesis, 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 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(groupMatching2(it))
        else:
            res.append(c)
    return res

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

returns

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

**Your parsing implementation**
Start with the `groupMatching2` and `parse` functions above; update the `parse` code so that the strings representing numbers/Booleans/arrays are converted to Python integers/Booleans/lists.

```
parse(['/pow2', '{', '/n', 'exch', 'def', '(Pow2_of_n_is)', 'dup', '8', 'n',
'48', 'add', 'put', '1', 'n', '-1', '1', '{', 'pop', '2', 'mul', '}', 'for',
'}', 'def', '(Calculating_pow2_of_9)', 'dup', '20', 'get', '48', 'sub',
'pow2', 'stack'])
```
should return:

```
['/pow2', ['/n', 'exch', 'def', '(Pow2_of_n_is)', 'dup', 8, 'n', 48, 'add',
'put', 1, 'n', -1, 1, ['pop', 2, 'mul'], 'for'], 'def',
'(Calculating_pow2_of_9)', 'dup', 20, 'get', 48, 'sub', 'pow2', 'stack']
```

## II.     Interpret code-arrays

We are now ready to write the interpret function. It takes a code-array as an 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,` and `for` 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 have 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 HW5.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
    (square)
    4 square
    dup 16 eq
    {(pass)} {(fail)} ifelse
```

```
        stack
"""
```

tokenize(input1) will return:
['/square', '{', 'dup', 'mul', '}', 'def', '(square)', '4', 'square',
'dup', '16', 'eq', '{', '(pass)', '}', '{', '(fail)', '}', 'ifelse',
'stack']

parse(tokenize(input1)) will return:
['/square', ['dup', 'mul'], 'def', '(square)', 4, 'square', 'dup', 16,
'eq', ['(pass)'], ['(fail)'], 'ifelse', 'stack']

2.
```
input2 ="""
(facto) 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:
['(facto)', '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:
['(facto)', '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 = """
    /fact{
    0 dict
          begin
                /n exch def
                1
                n -1 1 {mul} for
          end
    } def
    6
    fact
    stack
"""
```

tokenize(input3) will return:
['/fact', '{', '0', 'dict', 'begin', '/n', 'exch', 'def', '1', 'n', '-1',
'1', '{', 'mul', '}', 'for', 'end', '}', 'def', '6', 'fact', 'stack']

```
parse(tokenize(input3)) will return:
['/fact', [0, 'dict', 'begin', '/n', 'exch', 'def', 1, 'n', -1, 1,
['mul'], 'for', 'end'], 'def', 6, 'fact', 'stack']
```

4.
```
input4 = """
/lt6 { 6 lt } def
1 2 3 4 5 6 4 -3 roll
dup dup lt6 {mul mul mul} if
stack
clear
"""
```

```
tokenize(input4) will return:
['/lt6', '{', '6', 'lt', '}', 'def', '1', '2', '3', '4', '5', '6', '4', '-
3', 'roll', 'dup', 'dup', 'lt6', '{', 'mul', 'mul', 'mul', '}', 'if',
'stack', 'clear']
```

```
parse(tokenize(input4)) will return:
['/lt6', [6, 'lt'], 'def', 1, 2, 3, 4, 5, 6, 4, -3, 'roll', 'dup', 'dup',
'lt6', ['mul', 'mul', 'mul'], 'if', 'stack', 'clear']
```

5.
```
input5 = """
(CptS355_HW5) 4 3 getinterval
(355) eq
{(You_are_in_CptS355)} if
 stack
 """
```

```
tokenize(input5) will return:
['(CptS355_HW5)', '4', '3', 'getinterval', '(355)', 'eq', '{',
'(You_are_in_CptS355)', '}', 'if', 'stack']
```

```
parse(tokenize(input5)) will return:
['(CptS355_HW5)', 4, 3, 'getinterval', '(355)', 'eq',
['(You_are_in_CptS355)'], 'if', 'stack']
```

6.
```
input6 = """
/pow2 {/n exch def
       (pow2_of_n_is) dup 8 n 48 add put
        1 n -1 1 {pop 2 mul} for
       } def
 (Calculating_pow2_of_9) dup 20 get 48 sub pow2
 stack
 """
```

```
tokenize(input6) will return:
['/pow2', '{', '/n', 'exch', 'def', '(pow2_of_n_is)', 'dup', '8', 'n',
'48', 'add', 'put', '1', 'n', '-1', '1', '{', 'pop', '2', 'mul', '}',
'for', '}', 'def', '(Calculating_pow2_of_9)', 'dup', '20', 'get', '48',
'sub', 'pow2', 'stack']
```

```
parse(tokenize(input6))  will return:
['/pow2', ['/n', 'exch', 'def', '(pow2_of_n_is)', 'dup', 8, 'n', 48,
'add', 'put', 1, 'n', -1, 1, ['pop', 2, 'mul'], 'for'], 'def',
'(Calculating_pow2_of_9)', 'dup', 20, 'get', 48, 'sub', 'pow2', 'stack']
```

**When you parse:**
- Make sure that the integer constants are converted to Python integers/floats.
- Make sure that the Boolean constants are converted to Python Booleans.
- 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:
```
(pass)
16
(square)
```

`interpreter(input2)` should print:
```
120
(facto)
```

`interpreter(input3)` should print:
```
720
```

`interpreter(input4)` should print:
```
300
6
2
1
```
`interpreter(input5)` should print:
```
(You_are_in_CptS355)
```

`interpreter(input6)` should print:
```
512
(Pow2_of_9_is)
(Calculating_pow2_of_9)
```