Implementing an indirect threaded virtual machine in Python
Assignment 2 for CSC 580 243, Spring, 2009, Dr. Dale E. Parson
Assignment 2 is due 11:59 PM on Feb. 23. Use gmake turnitin to turn it in.

Perform the following steps to copy and inspect my initial code handout.

```
    cp -pr ~parson/ThryLang/vm_in_python ~/ThryLang
```

The vm.py file in that directory is a bare bones Java implementation of an emulated direct-threaded virtual machine in class `DirectThreadedVM`, and an emulated indirected-threaded vm in class `IndirectThreadedVM`.

Your assignment is to implement within class `DirectThreadedVM` all of the direct-threaded primitive operations of the STEP (Stack-based Threaded Emulated Processor) in the document “STEP: A Stack-based Controller for HDM Tap Managers,” which I will hand out in class, and to implement within class `IndirectThreadedVM` replacement “incomplete_” operations that are counterparts to the operations of class `DirectThreadedVM` that have in-line arguments in the code dictionary. Code dictionaries for `IndirectThreadedVM` consist strictly of functions pointers, with no in-line data. Data that is in-lined for `DirectThreadedVM` is captured via closures in `IndirectThreadedVM` as I will describe in class.

You must also reimplement method `IndirectThreadedVM.intern` so that it creates only 1, unique partialFunction-positionalParams object for each unique value of positionalParams. For example, note the following interactive test run.

```
$ python -i vm.py
TOP OF STACK:  8
TOP OF STACK:  9
>>> copya = ivm.intern(ivm.incomplete_STEP_CONST, (6,))
>>> copyb = ivm.intern(ivm.incomplete_STEP_CONST, (6,))
>>> copya == copyb
False
>>> copya is copyb
False
```

The new `intern` method must keep track of partialFunction-positionalParams bindings using a Python dictionary within each `IndirectThreadedVM` object. The constructor initializes this empty dictionary. Note the following example.

```
>>> binding = (ivm.incomplete_STEP_CONST, (6,))
>>> interns = {}
>>> if (interns.has_key(binding)):
...     copya = interns[binding]
```
... else:
...     copya = ivm.intern(binding[0], binding[1])
...     interns[binding] = copya
...
>>> if (interns.has_key(binding)):
...     copyb = interns[binding]
... else:
...     copyb = ivm.intern(binding[0], binding[1])
...     interns[binding] = copyb
...
>>> copya == copyb
True
>>> copya is copyb
True

Note also that attempting to store a non-constant binding as a dictionary key fails.

>>> binding = [ivm.incomplete_STEP_CONST, (6,)]
>>> copyc = ivm.intern(binding[0], binding[1])
>>> interns[binding] = copyc
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    TypeError: list objects are unhashable

Your intern method should allow this TypeError exception to pass out to the caller. Your intern function must document the possibility of this exception if either of the parameters to intern (partialFunction or positionalParams) is not a constant value.

You must implement all primitive operations in the STEP VM document, Tables 1 through 6, except the inp, outp, inpw and outpw primitives of Table 7. Operation STEP_PAUSE should raise a VMPauseException as defined in vm.py, and STEP_HALT should raise a VMHaltException with an str()-cast copy of the popped top of data stack as documented for Table 1. All of the primitive operations must use names as defined in the “C++ name” of the tables, e.g., STEP_PAUSE. All of these primitives go into class DirectThreadedVM. In addition, EVERY method in class DirectThreadedVM that requires an in-line argument in the code stream MUST have an incomplete_ counterpart for interning available in class IndirectThreadedVM.

I will write an assembler for the postfix Forth language for this VM, with a target delivery date of February 9. That gives you a week to start coding before testing. Use gmake turnitin by February 26 11:59 PM to turn it in.
import exceptions

class VMException(exceptions.Exception):
    
    VMException is any Exception from a VM that is not an error.
    
    def __init__(self, args=None):
        self.args = args

class VMError(exceptions.Exception):
    
    VMError is any Exception from a VM that is an error.
    
    def __init__(self, args=None):
        self.args = args

class VMPauseException(VMException):
    pass

class VMHaltException(VMException):
    pass

class DirectThreadedVM(object):
    default_stacksize = 100  # class static value
    def __init__(self, dsz=default_stacksize, tsz=default_stacksize):
        self.ip = 0
        self.ds = 0  # one above top of data stack
        self.ts = 0  # one above top of thread stack
        self.data_stack = [None for index in xrange(0, dsz)]
        self.thread_stack = [None for index in xrange(0, tsz)]
        self.code_dictionary = []
        self.data_dictionary = []

def STEP_CALL_SECONDARY(self):
    
    The secondary address follows the instruction in-line.
    
    self.thread_stack[self.ts] = self.ip + 2  # save return address
    self.ts += 1
    self.ip = self.code_dictionary[self.ip + 1]  # make the call

def STEP_RETURN(self):
    self.ts -= 1
self.ip = self.thread_stack[self.ts]
def STEP_GOTO(self):
    self.ip = self.code_dictionary[self.ip+1]
def STEP_GOTO0(self):
    if (self.data_stack[self.ds-1] == 0):
        self.ip = self.code_dictionary[self.ip+1]
    else:
        self.ip += 2
        self.ds -= 1
def STEP_NOOP(self):
    self.ip += 1
def STEP_CONST(self):
    self.ds = self.ds + 1
    self.ip += 2 # get an instruction beyond
def STEP_FETCH(self):
    self.ip += 1
def STEP_ADD(self): # add top 2 params, leave sum on stack
    self.ds -= 1
    self.ip += 1
def STEP_PRINT(self):
    print "TOP OF STACK: ", self.data_stack[self.ds-1]
    self.ds -= 1
    self.ip += 1
def run(self, codedict, datadict):
    """ Start the machine from the beginning """
    self.code_dictionary = codedict
    self.data_dictionary = datadict
    self.reset()
    self.resume()
    def resume(self):
        """ Resume the machine after a VMPauseException or VMHaltException """
        while (self.ip < len(self.code_dictionary)):
            apply(self.code_dictionary[self.ip],())
    def reset(self):
        """ Reset IP and DS and TS to 0 -- empty the stacks. """
        self.ip = 0
        self.ds = 0
        self.ts = 0
# Now do the same thing, but eliminate the in-line parameters by using
# closures to bind distinct in-line arguments to functions.
We are now emulating indirect threaded code.

class IndirectThreadedVM(DirectThreadedVM):
    def __init__(self, dsz=DirectThreadedVM.default_stacksize, tsz=DirectThreadedVM.default_stacksize):
        super(IndirectThreadedVM, self).__init__(dsz, tsz)
    def intern(self, partialFunction, positionalParams):
        def function_args_closure():
            return apply(partialFunction, positionalParams)
        return function_args_closure
    def incomplete_STEP_CALL_SECONDARY(self, address):
        """ The secondary address follows the instruction in-line. """
        self.thread_stack[self.ts] = self.ip + 1 # save return address
        self.ts += 1
        self.ip = address
    def incomplete_STEP_GOTO(self, address):
        self.ip = address
    def incomplete_STEP_CONST(self, constant):
        self.data_stack[self.ds] = constant
        self.ds = self.ds + 1
        self.ip += 1 # get an instruction beyond
import exceptions

from vm import *

vm = DirectThreadedVM()

code = []   # Values for code dictionary.

vm.STEP_CALL_SECONDARY, 4,  # subroutine address in this dictionary
vm.STEP_GOTO, 12,         # jump to last instruction (STEP_NOOP)
vm.STEP_CONST, 5,          # start of a subroutine
vm.STEP_CONST, 0,          # offset into data dictionary
vm.STEP_FETCH,
vm.STEP_ADD,
vm.STEP_PRINT,
vm.STEP_RETURN,
vm.STEP_NOOP  # last instruction at address 12

vm.run(code, data)

ivm = IndirectThreadedVM()

icode = []   # Values for code dictionary.

ivm.intern(ivm.incomplete_STEP_CALL_SECONDARY, (2,)),  # 2 is the subroutine
ivm.intern(ivm.incomplete_STEP_GOTO, (8,)),
# jump to 8, final STEP_NOOP instruction
ivm.intern(ivm.incomplete_STEP_CONST, (6,)),  # start of a subroutine
ivm.intern(ivm.incomplete_STEP_CONST, (0,)),  # offset into data dictionary
ivm.STEP_FETCH,
ivm.STEP_ADD,
ivm.STEP_PRINT,
ivm.STEP_RETURN,
ivm.STEP_NOOP  # last instruction at address 12

ivm.run(icode, idata)