Lecture 25:

- MyPL Virtual Machine (wrap up)

Announcements:

- HW-4 out (soft deadline)
- HW-5 out
- Quiz 6 this Wed
- Exam 2 next Wed

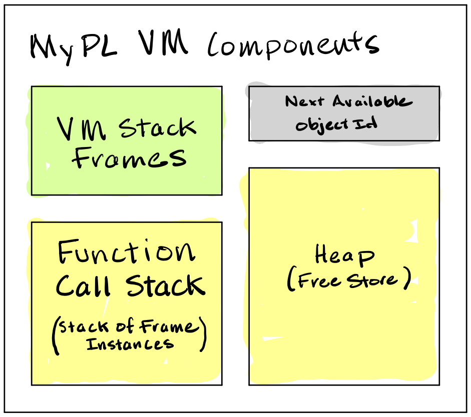
Note on `WRITE()` and `READ()` instructions:

- `WRITE()` pops value x and calls Python `print(x)`
- `READ()` calls Python `input()` and pushes result on the stack

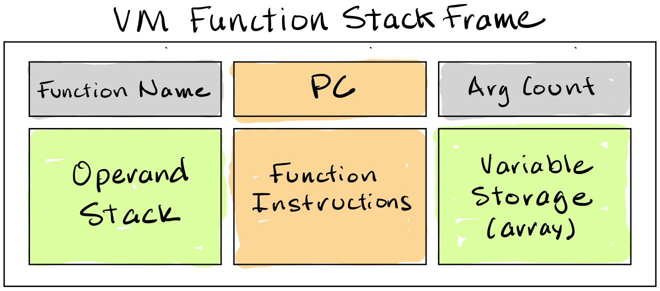
The `WRITE()` instruction is used throughout the unit tests for HW-5

The `INPUT()` instruction isn't tested in the unit tests ...

(4) Basic MyPL VM architecture (`mypl_vm.py` and `mypl_frame.py`)



(a) Function Frames (`myple_frame.py`)



- we separate into **`VMFrameTemplate`** and **`VMFrame`**
- each **`VMFrame`** can be thought of as an "instance" of the template
- a **`VMFrame`** holds a reference to its template (for name, arg count, instructions)

For example, for a function: **`int f(int x, string y)`**

```python
f = VMFrameTemplate('f', 2)
f.instructions.append(STORE(0))
# ... etc ...
vm = VM()
vm.add_frame_template(f)
```

To create a frame instance from within the VM:

```python
# create a frame out of a template stored in VM named 'f'
frame = VMFrame(self.frame_templates['f'])
# add the instantiated frame to the VM's call stack
self.call_stack.append(frame)
```

Basic structure of the VM class:

```python
class VM:

    def __init__(self):
        self.struct_heap = {}       # id -> dict
        self.array_heap = {}        # id -> list
        self.next_obj_id = 2024     # next available object id
        self.frame_templates = {}   # fun name -> VMFrameTemplate
        self.call_stack = []        # fun call stack

    def __repr__(self):             # for debugging

    def add_frame_template(self, template):

    def error(self, msg, frame=None):

    def run(self, debug=False):     # where all the work happens!
```

Basic layout of a VM instruction (in **mypl_frame.py**):

```
@dataclass
class VMInstr:
    opcode: OpCode
    operand: Any = None         # default is None
    comment: str = ''           # default is empty

    def __repr__(self):         # for debugging / errors
        # ...

# Helper functions for creating specific instruction types
def PUSH(value):
    return VMInstr(OpCode.PUSH, value)

def POP():
    return VMInstr(OpCode.POP)

# ...
```

Basic layout of the VM's **run()** function:

```
def run(self, debug=False):
    # grab the "main" function frame and instantiate it
    if not 'main' in self.frame_templates:
        self.error('No "main" functrion')
    frame = VMFrame(self.frame_templates['main'])
    self.call_stack.append(frame)

    # run loop (continue until run out of call frames or instructions)
    while self.call_stack and frame.pc < len(frame.template.instructions):
        # get the next instruction
        instr = frame.template.instructions[frame.pc]
        # increment the program count (pc)
        frame.pc += 1
        # for debugging:
        if debug:
            # ... print out helpful stuff ...

        #------------------------------------------------------------
        # Literals and Variables
        #------------------------------------------------------------

        if instr.opcode == OpCode.PUSH:
            frame.operand_stack.append(instr.operand)

        elif instr.opcode == OpCode.POP:
            frame.operand_stack.pop()

        # ... and so on for each opcode ...
```

Note on loading and storing variables in frame instance

- Given a `STORE(`i`)` instruction ...
- If only i variables, then `frame.variables.append(value`)
- Otherwise, set `frame.variables[i]` to value
- For `LOAD(`i`)` and `STORE(`i`)` don't need error checking

(b) Implementing CALL instructions (in `run()`)

1. Get the stack frame using function name in CALL instruction
2. Instantitate the frame and push it onto frame stack
3. Pop argument values off current frame's operand stack
4. Push each argument value onto new frame's operand stack
5. Set the instantiated frame as the current frame

(c) Implementing RET instructions (in `run()`)

1. Pop the return value off the current frame's operand stack
2. Pop the frame off the frame stack
3. Get (peek) the frame at top of frame stack (caller), set as current frame
4. If a calling frame exists, push the return value onto it's operand stack