Lecture 23:

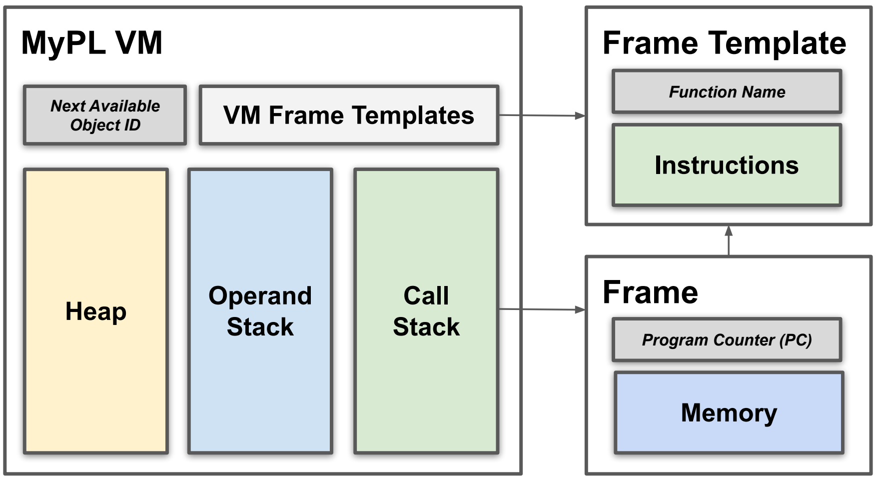
- VM Wrap Up and HW-5

Announcements:

- HW-4 due
- HW-5 out
- Proj Part 2 ...

MyPL VM Architecture

(4) Basic MyPL VM architecture



- 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 (e.g., for instructions)

More on HW-5

For example, for a function: `int f(x: int, y: string) { ... }`

```
// create a new template for f
VMFrameTemplate f = new VMFrameTemplate("f");
// add instructions
f.add(VMInstr.PUSH(10));
// ... etc ...
// create the VM and add the template
VM vm = VM();
vm.add(f);
```

To create a frame instance from within the VM:

```
// create a frame out of a template stored in VM named "f"
VMFrame frame = new VMFrame(templates.get("f"));
// add the instantiated frame to the VM's call stack
callStack.push(frame);
```


More on HW-5

Basic structure of the VM class:

```
public class VM {
  // the next object id:
  private int nextObjectId = 2025;
  // the operand stack (as a Java doubled-ended queue):
  private Deque<Object> operandStack = new ArrayDeque<>();
  // the call stack:
  private Deque<VMFrame> callStack = new ArrayDeque<>();
  // the VM frame templates:
  private Map<String,VMFrameTemplate> templates = new HashMap<>();
  // the heaps (oid -> value)
  private Map<Integer,List<Object>> arrayHeap = new HashMap<>();
  private Map<Integer,Map<String,Object>> structHeap = new HashMap<>();
  // special null value:
  public static final Object NULL = new Object() {...}

  // ... helper functions ...  errors, add templates, etc.

  // execute the program (start the VM)
  public void run() {
    ...
  }
}
```

More on HW-5

Basic layout of a VM instruction:

```java
public class VMInstr {
  // the type of instruction
  public OpCode opcode;
  // for storing the static operand (if needed)
  public Object operand;
  // can add a comment to the instruction (for code gen)
  public String comment = "";

  // ... constructors, toString

  // helpers to create instructions (one per instruction type)
  public static VMInstr PUSH(Object value) { ... }
  public static VMInstr POP() { ... }

  // ... etc.
}
```

Using null values:

```java
frame.add(VMInstr.PUSH(VM.NULL));
```


More on HW-5

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

```java
public void run() {
  // grab the main frame and instantiate it (we check first)
  VMFrame frame = new VMFrame(templates.get("main"));
  callStack.push(frame);

  // run loop until out of call frames or instructions in the frame
  while (!callStack.isEmpty() && frame.pc < frame.template.instructions.size()) {
    // get the next instruction
    VMInstr instr = frame.template.instructions.get(frame.pc);
    // for debugging:
    if (debug) { ... }// ...
    // increment the pc
    ++frame.pc;

    //----------------------------------------------------------------------
    // Literals and Variables
    //----------------------------------------------------------------------
    if (instr.opcode == OpCode.PUSH)
      operandStack.push(instr.operand);
    else if (instr.opcode == OpCode.POP)
      operandStack.pop();

    // ... etc
```

More on HW-5

Note on loading and storing variables in frame instance

- For a `STORE(`i`)` ... If only i variables, then `frame.memory.add(value)`
- Otherwise, `frame.memory.set(i, value)`
- For `LOAD(`i`)` and `STORE(`i`)` don't need error checking

Note on CALL

- create a new frame, add to the call stack, set its pc to 0
- set the current frame (`frame`) to the new frame
- the operand stack will contain the argument values (more later)

Note on RET

- pop the call stack
- set the new frame (if one exists) to top frame (via `peek()`)
- the function's return value is on the operand stack (more later)