Lecture 23:

- MyPL Virtual Machine

Announcements:

- HW-4 out

MyPL VM for HW-5 and HW-6

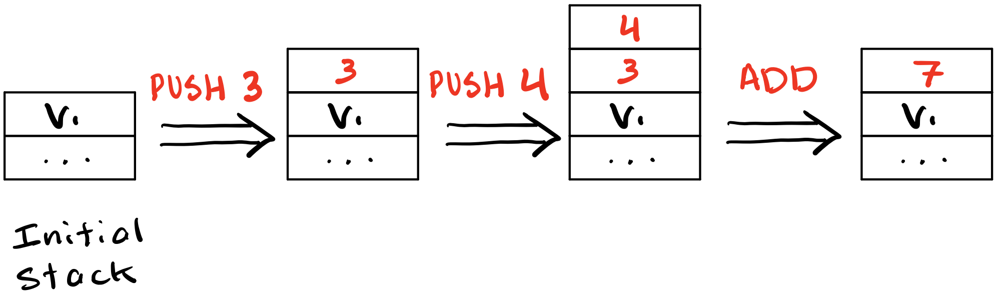
- Based loosely on the JVM architecture (stack machine, stack frames)
- Via API calls instead of using bytecode encoding/decoding
- Takes some short cuts, tailored to MyPL
- Performs minimal error checking (except for runtime program errors)

(1) Data Types/Values

- Uses Python types to represent values and assumes programs are well typed
- Uses Python `None` value for representing MyPL `null` values

(2) Abstract Stack Machine

- instead of registers, uses an "operand stack"



The VM components include: ... more later

- operand stack (see above)
- memory for storing local variables ... list of values/objects
- struct heap storage ... oid → {field:value}
- array heap storage ... oid → [value]
- function-call stack (stack of call "frames")

(3) MyPL VM Instruction Set (high level) … see `mypl_opcode.py`

Note: OP(A) says A is supplied directly to the OP instruction

- instructions take inputs directly and/or from the operand stack
- difference is what can be provided *statically* versus *dynamically* to instruction

(a) Literals and variables

PUSH(A) push argument A onto the operand stack

POP() pop value off of the stack

STORE(A) pop x, store x at memory address A (a list index)

LOAD(A) get x at memory address A, push x on to stack

(b) Arithmetic operations

ADD() pop x, pop y, push $(y + x)$ on to stack

SUB() pop x, pop y, push $(y - x)$ on to stack

MUL() pop x, pop y, push $(y \times x)$ on to stack

DIV() pop x, pop y, push $(y \div x)$ on to stack

(c) Logical operators

AND() pop bool x, pop bool y, push (y **and** x)

OR() pop bool x, pop bool y, push (y **or** x)

NOT() pop bool x, push (**not** y)

(d) Relational (comparison) operators

`CMPLT()`	pop x, pop y, push $(y < x)$
`CMPLE()`	pop x, pop y, push $(y \leq x)$
`CMPEQ()`	pop x, pop y, push (y `==` x)
`CMPNE()`	pop x, pop y, push (y `!=` x)

(e) Jumps

`JMP(`A`)`	jump to instruction A (int index into instruction list)
`JMPF(`A`)`	pop x, if x is false jump to instruction A (int index)

Simple example: `while j < 3 { j = j + 1 } ...`

```
0:  LOAD(0)       # assume j stored in variables[0]
1:  PUSH(3)       # literal value for the comparison
2:  CMPLT()       # true if j < 3
3:  JMPF(9)       # if j >= 3, jump to instruction 9
4:  LOAD(0)       # get j again
5:  PUSH(1)       # for the literal value 1
6:  ADD()         # compute j + 1
7:  STORE(0)      # store result back into j
8:  JMP(0)        # go back to start of while
9:  ...           # continue on after while loop
```