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”

\[
\begin{array}{c}
\text{Initial} \\
\text{stack}
\end{array}
\begin{array}{c}
\downarrow \text{PUSH 3} \\
3
\end{array}
\begin{array}{c}
\downarrow \text{PUSH 4} \\
4
\end{array}
\begin{array}{c}
\downarrow \text{ADD} \\
7
\end{array}
\]

The VM components include:
- operand stack (see above)
- memory for storing local variables
- struct heap storage
- array heap storage
- function-call stack (stack of call “frames”)
- ... more later
- ... list of values/objects
- ... oid → {field:value}
- ... oid → [value]
(3) MyPL VM Instruction Set (high level) ... see mypl_opcode.py

Note: $\text{OP}(A)$ 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 \textit{statically} versus \textit{dynamically} to instruction

(a) Literals and variables

\begin{align*}
\text{PUSH}(A) & \quad \text{push argument } A \text{ onto the operand stack} \\
\text{POP()} & \quad \text{pop value off of the stack} \\
\text{STORE}(A) & \quad \text{pop } x, \text{ store } x \text{ at memory address } A \text{ (a list index)} \\
\text{LOAD}(A) & \quad \text{get } x \text{ at memory address } A, \text{ push } x \text{ on to stack}
\end{align*}

(b) Arithmetic operations

\begin{align*}
\text{ADD()} & \quad \text{pop } x, \text{ pop } y, \text{ push } (y + x) \text{ on to stack} \\
\text{SUB()} & \quad \text{pop } x, \text{ pop } y, \text{ push } (y - x) \text{ on to stack} \\
\text{MUL()} & \quad \text{pop } x, \text{ pop } y, \text{ push } (y \times x) \text{ on to stack} \\
\text{DIV()} & \quad \text{pop } x, \text{ pop } y, \text{ push } (y \div x) \text{ on to stack}
\end{align*}

(c) Logical operators

\begin{align*}
\text{AND()} & \quad \text{pop bool } x, \text{ pop bool } y, \text{ push } (y \text{ and } x) \\
\text{OR()} & \quad \text{pop bool } x, \text{ pop bool } y, \text{ push } (y \text{ or } x) \\
\text{NOT()} & \quad \text{pop bool } x, \text{ push } (\text{not } y)
\end{align*}
(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: \[\text{while } j < 3 \{ j = j + 1 \} \ldots\]

- 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