Memory Diagram Practice!

Environment Diagrams

1. Add columns for Call Stack, Heap, and Output
2. Add a Globals frame to Call Stack

Function Call

1. Verify and prepare for call
   i. Is function name bound in your diagram or built-in?
   ii. Fully evaluate each argument's expression
   iii. Do arguments match function parameters?
2. Establish new frame on call stack
   i. Add name of function
   ii. Add RA (Return Address line #)
   iii. Copy arguments to parameters bound in frame
3. Jump to first line of function definition

Function Return Statement

1. Evaluate returned expression
   - Add RV (Return Value) in current stack frame
2. Jump back to function caller
   i. Line is in RA (Return Address)
   ii. The function call evaluates to last frame's RV

Function Definitions: Enter name in current frame and draw arrow to Function object on heap labeled `Fn: [start_line] - [end_line]`

Current Frame: The most recently added frame that has not returned. (*No RV!*)

Name Resolution: Look for name in the current frame. Not there? Check Globals frame!

Variable Initialization: Enter name and space for variable in current frame.

Variable Assignment: Find variable's location via name resolution, copy assigned value to it.

Variable Access: Find variable via name resolution, use value currently assigned to it.

```python
def main() -> None:
    a: int = 0
    jump(a)
    around(a)
    print(a)
    a = around(a)
    print(a)


def jump(a: int) -> None:
    a += 1
    print(a)


def around(a: int) -> int:
    a += 1
    return a


if __name__ == "__main__":
    main()
```

Diagram 0: Jump Around

- Assume the special dunder variable __name__ is assigned "__main__" in the evaluation of this program.
- Try drawing diagram yourself for 3 minutes, then discuss in breakout rooms for another 3-5 minutes
- Respond on Gradescope to the Diagram 0 questions.

```python
def main() -> None:
    a: int = 0
    jump(a)
    around(a)
    print(a)
    a = around(a)
    print(a)


def jump(a: int) -> None:
    a += 1
    print(a)


def around(a: int) -> int:
    a += 1
    return a


if __name__ == "__main__":
    main()
```

```python
def main() -> None:
    x: int = 1
    x = a(b(x + x))
    print(x)


def a(x: int) -> int:
    print("a")
    y: int = 2 * x
    return y


def b(x: int) -> int:
    print("b")
    y: int = a(2 * x)
    return y


if __name__ == "__main__":
    main()
```

Diagram 1: CPU go brr

- Assume the special dunder variable __name__ is assigned "__main__" in the evaluation of this program.
- Try drawing diagram yourself for 3 minutes, then discuss in breakout rooms for another 3-5 minutes
- Respond on Gradescope to the Diagram 1 questions.

Function Evaluation - Gradescope

What is the result of evaluating the function call expression: **`cute(3)`**

```
1  def cute(force: int) -> str:
2      s: str = ""
3      i: int = 1
4      while (i < force):
5          s += "h"
6          h: int = 0
7          while(h < i):
8              s += "e"
9              h += 1
10         i += 1
11     return s
```

Notes on Nested Loops

- **General Rule:** When the closing curly brace of a loop is encountered, the loop jumps back to the start of **its matching condition**.

- An inner loop will jump back up to the inner loop's condition and an outer loop will jump back up to the outer loop's condition.

- Thus, an inner loop must complete all of its **iterations** for every individual iteration of an outer loop.