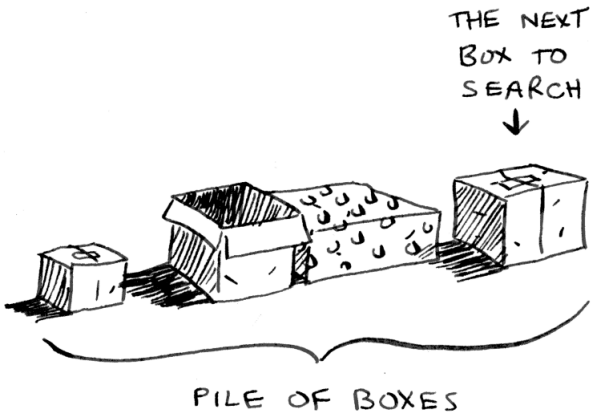
CS326 Project 4:
Programming in Python

Benjamin Cisneros



Description

The purpose of this assignment is to learn more about the implications of using *static* and *dynamic* links. We will put the scoping rules taught in class into practice within a made-up dynamic language called **Jython**. In Jython, you will encounter situations where you need to address references to variable names that might not be declared within the function or method that utilizes those names. For example, let us look at the following piece code:

```
var n int = 10
var i int = 0
while i < n {
  i = i + 1
}
```

Where are the values of n and i stored in memory? We know that they can be resolved during compilation and from the previous assignment, we also know that they can be resolved during execution. The first approach is quite simple: create a global symbol table and insert the identifiers along with their corresponding values. Next, create a local symbol table to represent the while loop's body and link it to the global symbol table. By doing this, you will able to find the declarations of n and i since their scope is global. This is exactly what we did in class. The second approach − resolving their scope during execution − is a bit more complex, as you may recall.

Because **Jython** is a dynamic language, our aim is to ensure it operates within a virtual machine.

In other words, we must utilize **Jython** as the intermediate language before translating the high-level language into the sequences of 0's and 1's that your computer understands. For example, the above piece of code can be translated as follows:

```
# GLOBALS = 2; n, i
# n = 10
ICONST, 10,
GSTORE, 0,
# i = 0
ICONST, 0,
GSTORE, 1,
# while i < n:
# goto (8)
GLOAD, 1,
GLOAD, 0,
ILT,
BRF, 24,
# i = i + 1
GLOAD, 1,
ICONST, 1,
IADD,
GSTORE, 1,
BR, 8,
# done (26)
# print
GLOAD, 1,
PRINT,
HALT
```

The global variables n and i are stored in *main*'s activation record as per our discussion in class. Remember that the activation record for *main* must accommodate room for the two global variables n and i (as well as any local variables). For example, if we start the *offset* count at 0, then the global variable n would be at offset 0, and the global variable i at offset 1. (In reality, addresses such as @0 are relative addresses. That is, n is not located at the absolute address 0, but at the offset from the start of the activation record.) Typically, the pointer that points to the activation record is called a **stack pointer**, and all locals and parameters are addressed relative to it. Here is an illustration of what the activation record for *main* may look like

Activation Record for globals/*main*
@0: n (**int**)
@1: i (**int**)

Notice that we are missing two important pointers, the **static** and **dynamic** links. This is OK since our primary emphasis in this assignment will be on the **dynamic** link.

Collaboration

Students are encouraged to collaborate within the guidelines specified in the syllabus. However, it is important to emphasize that the primary objective of these assignments is to acquire the essential

skills required for success in the field of computer science. Accordingly, students are expected to independently write their own code, ensuring that it is free from any form of plagiarism, whether from peers, past students, or current ones. To maintain academic integrity, all assignments will undergo cross-referencing using the Measure Of Software Similarity (Moss) tool, which automatically assesses program similarities.

Background

For this programming assignment, we will be using Python to write the Virtual Machine (VM) responsible for executing every individual instruction of **Jython**. I have already written a few programs in this language, so all you need to do is execute the instructions of each program. Here is a brief overview of the different components that you will need to use to get the VM running:

- **Instruction:** A class that is used to represent each instruction of **Jython**. An instruction consists of its name and the mnemonic value that it represents (typically a positive integer value).
- **Instructions:** A list of instructions; this list is used to print a string representation of each instruction during execution (in fact, this is use extensively for debugging).
- **ActivationRecord:** The information pushed onto the stack includes metadata that contains details about a function, like the caller (the invoker) and the return address of the caller. This function metadata enables us to reference the function by its index, an integer value that represents the function's location in a the constant pool (a list of functions).
- **FunctionContext:** This is the metadata used by an activation record; it contains information regarding a function, including its name, number of arguments, number of local variables, and its memory location.
- **VirtualMachine:** A simple stack machine that evaluates simple mathematical expressions. The machine holds all temporary values such as operation results on an operand stack.

Remember that, you **CANNOT** modify the codebase. In short, you **CANNOT** and **SHOULD NOT** modify the provided code.

Jython

Jython is a compact bytecode language, so it comprises only a few instructions. Below is the compilation of bytecode instructions that the VM needs to interpret.

Instruction	Description
`iadd`, `isub`, `imul`	Arithmetic operators for integers. Pop two operands, perform operation and push result back on stack.
`ilt`, `ieq`	Equality operators for integers. Pop two operands, perform operation, and push result back on stack.
`call` $f()$	Call function f (via f's constant pool). Push new Activation Record onto call stack, move parameters from operand stack to stack frame, and branch to function start address.

`ret`	Return from function call. Any return value is on the operand stack. Pop the top stack frame off and return to return address stored in the stack frame.
`br` a, `brt` a, `brf` a	Branch to a always, if $code[ip]$ is true or is false. (a is an address.)
`gload` a, `gstore` a	Grab value of a global variable and push it onto the operand stack: $stack[sp\texttt{++}] = globals[a]$. Pop value from operand stack and store it into global variable: $globals[a] = stack[sp\texttt{--}]$. ($a$ is an address.)
`load` i, `store` i	Grab value of a local variable and push it onto the operand stack: $stack[\texttt{++}sp] = A.locals[offset]$. Pop value from the operand stack and store it into the local variable: $A.locals[offset] = stack[sp]$. ($offset$ is the local value index, and A is the activation record.)
`print`	Pop from the operand stack and print to standard output.
`noop`	Do nothing.
`pop`	Throw away the top of the operand stack.
`halt`	Halt program execution.

Virtual Machine

A bytecode interpreter simulates a computer with the following key elements:

- **Global memory:** Holds a fixed number of slots for variables. The memory slots can point at integer values. Unlike in 218, we access variables by their integer addresses (i.e., an index) rather than using actual memory addresses.
- **Code memory:** This byte array contains the program's bytecode instructions. It contains bytecodes and any instruction operands. Remember Addresses are integers.
- **ip register:** The instruction pointer serves as a specialized register that points into the next instruction to be executed within the code memory.
- **sp register:** A dedicated register known as the stack pointer that points to the top of the function call stack.
- **cpu**: To execute instructions, the VM has a simulated CPU that amounts to a loop around a giant **if** on bytecode statement.
- **Constant pool:** This array keeps non-integers operands out of code memory. In this assignment, it is only used to track functions descriptors (i.e., information about functions). Instruction operands can refer to constant pool elements via an integer index.

In addition to these elements, we also have:

- **Operand Stack:** Instead of loading values into registers, the virtual machine pushes temporary values onto the operand stack. All operations of the instructions are either in code memory or the stack.

- **Active context:** This is the current activation record. The VM maintains a stack that stores function addresses. Remember Addresses are integers.
- **Trace:** A boolean flag that is used for debugging.

Finally, we also have

- **Main program address:** The code memory address from which the VM should commence program execution. This address is associated with the *main* function

Here is an illustration of the different components:

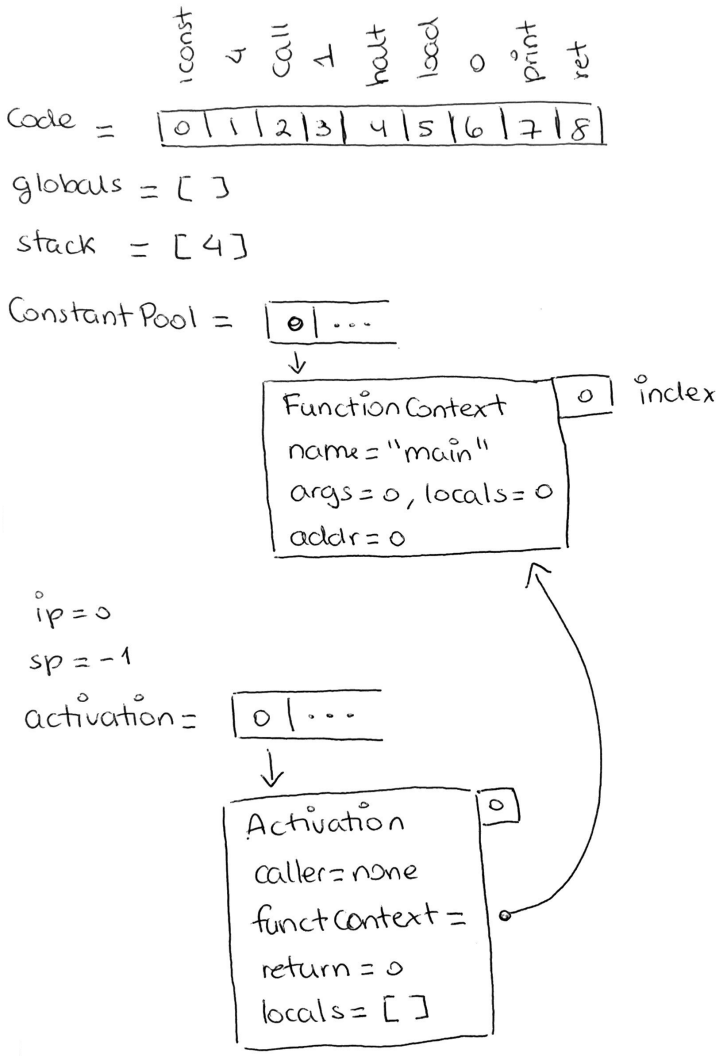


Fig. 1. Components of the VM in action while executing program #5.

Bytecode Processor. The processor serves as the core of the VM and has a straightforward task: it continually cycles through a fetch-decode-execute mechanism. The processor fetches a

bytecode and then moves to a code segment responsible for executing that instruction.

```
Function cpu
    opcode := code[ip]
    while opcode-is-not-halt and ip < code.length do
        ip++
        if opcode-is-iadd then
            execute-bytecode-iadd
        else if opcode-is-isub then
            execute-bytecode-isub
        else if ... then
            ...
        else if opcode-is-ret then
            execute-bytecode-ret
        else
            error-invalid-opcode-found
        end
        opcode := code[ip]
    end
end
```

The CPU stops when it encounters a `halt` instruction or runs out of instructions to execute at the end of the code memory. The conditional expressions that are part of the **if-else** statement execute the code fragment necessary to simulate the instructions. The body of each **if** statement execute the code fragment necessary to simulate the instructions. For example, here is how we execute the `br` instruction (branch to a new code memory address):

```
/* branch to instruction's address operand        */
if opcode-is-br then
    ip := code[ip]
end
```

When the instruction includes an operand (located directly next to the bytecode), the code segment retrieves the operand from the code memory and advances the instruction pointer by four bytes (i.e., we increment *ip* by 1). As another example, here is the code to perform the stack-based `iconst` instruction, which involves pushing an integer constant onto the operand stack:

```
if opcode-is-iconst then
    operand := code[ip]
    stack[sp + 1] := operand
end
```

Keep in mind that certain instructions require data operands, not code addresses. These instructions can access global variables, parameters, and local variables. We will address parameters and locals later. For now, let us examine how instructions interact with global memory.

The `gstore` instruction stores a value into a global memory. To implement that, we can treat the

global memory space as an array of objects.

globals[some-address] := stack[*sp*]

The `gload` instruction does the opposite. It loads a value from global memory into a register and then pushes it onto the operand stack.

stack[$sp + 1$] := globals[some-address]

Now, the `call` instruction must determine the number of parameters and local variables required by a function. It does so by examining the constant pool entry for the function to retrieve this information. In the following example, we can observe how the VM transforms the `call` operand into an index within the constant pool for the program that follows:

```
.def main: ARGS=0, LOCALS=0
  iconst 4  # start by getting the parameter onto operand stack
  call 1    # 1 is the index of foo in the constant pool
  halt
.def foo: ARGS=1, LOCALS=0
  load 0    # get 1st parameter
  print     # print value 4
  ret       # return no value
```

Function calls encompass more than just the constant pool; they also involve the operand stack within the stack-based interpreter. The simplest way to understand the process is to examine the instruction trace, as it illustrates how the stack expands and contracts with each instruction:

```
0000  iconst      4
stack=[ 4 ]             calls=[main]
0002  call        foo
stack=[ ]               calls=[main foo]
0005  load        0
stack=[ 4 ]             calls=[main foo]
0007  print
4
stack=[ ]               calls=[main foo]
0008  ret
stack=[ ]               calls=[main]
0004  halt
stack=[ ]
```

The stack expands to the right, depicting the state before the execution of each instruction. Following each `call` instruction, the call stack expands by one stack frame, and it shrinks by one with each `ret` instruction.

At this point, we know what a bytecode program looks like and how the VM represents the program in a simulated code memory and in a constant pool. Here is a picture of the function call stack:

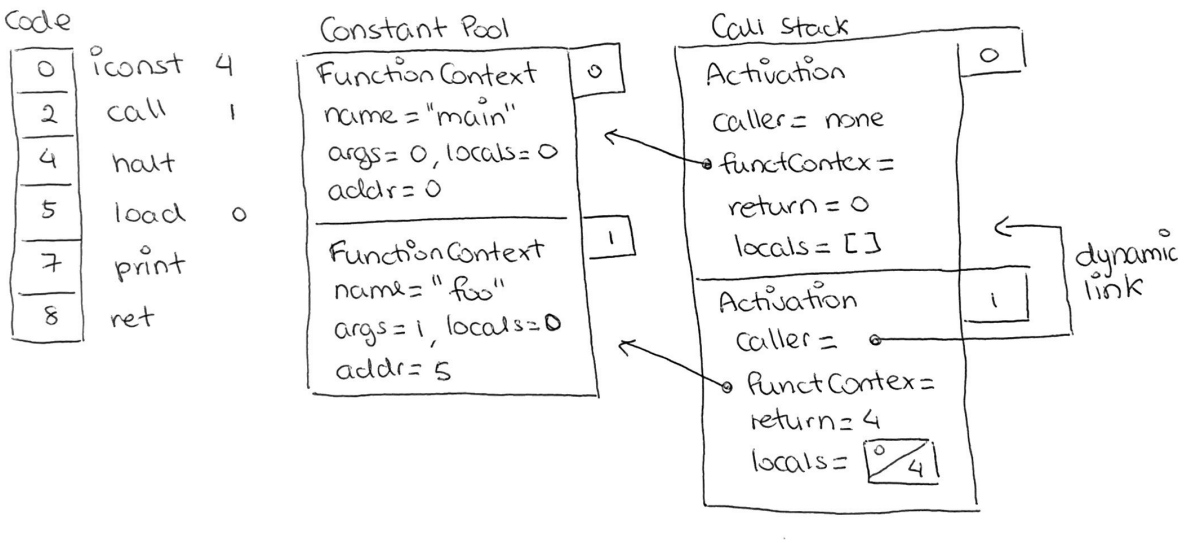


Fig. 2. Function call stack of program #5.

Semantics

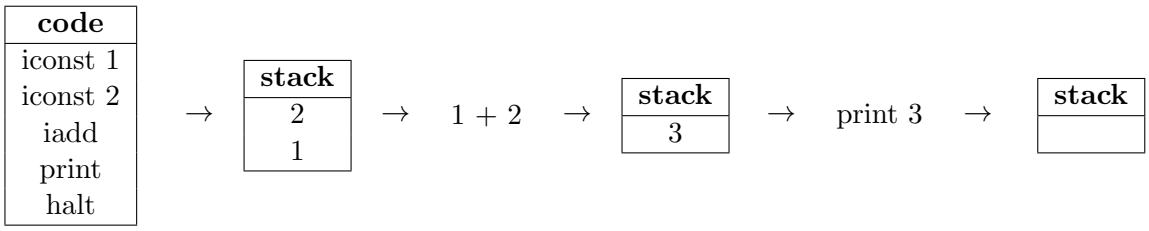
I have included the specification for what the VM commands are supposed to do, but let us look at a simple example. A stack is a sequence of memory that can grow and shrink as needed. The two fundamental operations on the stack are *push* and *pop*. The VM model operates in a stack-based manner, where all operations either draw their operands from the stack or store their results on the stack before transferring them to the activation record.

Here is a simple program that adds two integer values:

```
.def main: ARGS=0, LOCALS=0
  iconst 1
  iconst 2
  iadd
  print
  halt
```

The VM should be able to handle any number of commands (bytecode instructions). Therefore, the VM should loop around a fetch-decode-execute mechanism. Again, *fetch* refers to getting the next instruction to execute from code memory, *decode* means determining which instruction we have just fetched, and finally, *execute* means performing the action that the semantics of the instructions prescribes. The semantics of the above piece of code can be depicted as follows:

- **iadd:** The command `iadd` refers to integer addition. As this is a binary operator, it takes the top two values from the stack, computes the sum, and then pushes the result back onto the stack.

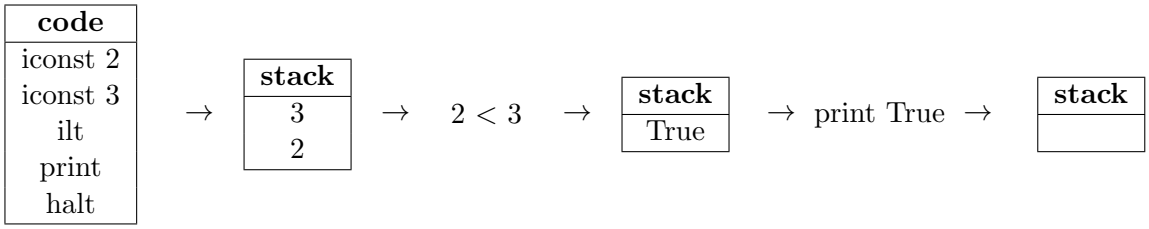


Now, let us examine another program:

```
.def main: ARGS=0, LOCALS=0
  iconst 2
  iconst 3
  ilt
  print
  halt
```

The semantics of the above piece of code can be depicted as follows:

- **ilt:** The `itl` command represents numeric less-than ordering. This operator takes the top two values from the stack and pushes the result, which is a single boolean value, back onto the stack.



Contents of main

Now, let us examine the program's entry point. In the `__main__` function, I have already started writing some of the code, and your task is to complete the remaining portion.

Question 1: Finish writing the implementation for `isub`, `imul`, `ilt`, and `ieq`. They are similar to `iadd`, with the only difference being what you need to push onto the operand stack.

Question 2: Finish writing the implementation for `brt` and `brf`. First, retrieve the address from code memory using the instruction pointer. Next, based on the instruction being used, evaluate whether the top value of the operand stack is true or false. If the value corresponds to true for `brt` or false for `brf`, update the instruction pointer with the address obtained from the code memory. Remember to appropriately increment your instruction pointer and decrement your stack pointer.

Question 3: Finish writing the implementation for `iconst`, `load`, and `store`. For `iconst`, retrieve the value from code memory using the instruction pointer and push it onto the operand stack. Remember to increment your stack pointer and your instruction pointer. In the case of `load`, fetch the offset from the code memory using the instruction pointer. Then, utilize the activation's locals array to access the value stored at the specified offset index and push it onto the operand

stack. Remember to increment your stack pointer and your instruction pointer. When handling `store`, perform the opposite action of what `load` does. Additionally, make sure to decrement your stack pointer and increment your instruction pointer.

Question 4: Finish writing the implementation for `gload` and `gstore`. They are similar to `load` and `store`, but they involve retrieving values from and storing values into the *global* array.

Question 5: Finish writing the implementation for `call`. In the case of the `call` instruction, first, locate the index (i.e., the function's location) in the code memory using the instruction pointer, and then increment the instruction pointer. Next, determine the number of arguments by accessing the constant pool using the function's index. Create a new activation record that includes a reference to the caller, the function's metadata (retrieved from the constant pool), and the instruction pointer. Ensure that the VM's activation record points to this newly-created activation, which corresponds to the invoked function. Calculate the position of the first argument in the operand stack by subtracting the number of arguments from the stack pointer plus one. Then, iterate from zero to the number of arguments to retrieve each argument stored in the operand stack and store them one by one into the activation's locals array. Lastly, do not forget to decrement the stack pointer by the number of arguments and set the instruction pointer to the return address of the function, which can be found in the constant pool using the function's index

Specifications

- Document your code.
- Use variables with meaningful names.
- Make sure your code is error proof.
- Do not modify the provided code (if any).
- All of the user input and output is done in main.
- You may assume valid input.
- No global variables (unless specified)

Submission

Submit the source file to Canvas by the deadline.

Sample Run

- **Output of first program:**

```
0000  iconst      1
stack=[ 1 ]            calls=[main]
0002  iconst      2
stack=[ 1 2 ]          calls=[main]
0004  iadd
stack=[ 3 ]            calls=[main]
0005  print
```

```
3
stack=[ ]              calls=[main]
0006  halt
stack=[ ]
Data memory

None
```

- **Output of second program:**

```
0000  iconst      4
stack=[ 4 ]            calls=[main]
0002  iconst      2
stack=[ 4 2 ]          calls=[main]
0004  isub
stack=[ 2 ]            calls=[main]
0005  print
2
stack=[ ]              calls=[main]
0006  halt
stack=[ ]
Data memory
0: 0
1: 0

None
```

- **Output of third program:**

```
0000  iconst      5
stack=[ 5 ]            calls=[main]
0002  gstore      0
stack=[ ]              calls=[main]
0004  iconst      25
stack=[ 25 ]           calls=[main]
0006  gstore      1
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 25 ]           calls=[main]
0010  gload       0
stack=[ 25 5 ]         calls=[main]
0012  imul
stack=[ 125 ]          calls=[main]
0013  gstore      0
stack=[ ]              calls=[main]
0015  gload       0
stack=[ 125 ]          calls=[main]
0017  print
125
stack=[ ]              calls=[main]
0018  halt
stack=[ ]
Data memory
0: 125
1: 25
```

```
None
```

- **Output of fourth program:**

```
0000  iconst      10
stack=[ 10 ]            calls=[main]
0002  gstore      0
stack=[ ]               calls=[main]
0004  iconst      0
stack=[ 0 ]             calls=[main]
0006  gstore      1
stack=[ ]               calls=[main]
0008  gload       1
stack=[ 0 ]             calls=[main]
0010  gload       0
stack=[ 0 10 ]          calls=[main]
0012  ilt
stack=[ True ]          calls=[main]
0013  brf         24
stack=[ ]               calls=[main]
0015  gload       1
stack=[ 0 ]             calls=[main]
0017  iconst      1
stack=[ 0 1 ]           calls=[main]
0019  iadd
stack=[ 1 ]             calls=[main]
0020  gstore      1
stack=[ ]               calls=[main]
0022  br          8
stack=[ ]               calls=[main]
0008  gload       1
stack=[ 1 ]             calls=[main]
0010  gload       0
stack=[ 1 10 ]          calls=[main]
0012  ilt
stack=[ True ]          calls=[main]
0013  brf         24
stack=[ ]               calls=[main]
0015  gload       1
stack=[ 1 ]             calls=[main]
0017  iconst      1
stack=[ 1 1 ]           calls=[main]
0019  iadd
stack=[ 2 ]             calls=[main]
0020  gstore      1
stack=[ ]               calls=[main]
0022  br          8
stack=[ ]               calls=[main]
0008  gload       1
stack=[ 2 ]             calls=[main]
0010  gload       0
stack=[ 2 10 ]          calls=[main]
0012  ilt
stack=[ True ]          calls=[main]
```

```
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 2 ]            calls=[main]
0017  iconst      1
stack=[ 2 1 ]          calls=[main]
0019  iadd
stack=[ 3 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 3 ]            calls=[main]
0010  gload       0
stack=[ 3 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 3 ]            calls=[main]
0017  iconst      1
stack=[ 3 1 ]          calls=[main]
0019  iadd
stack=[ 4 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 4 ]            calls=[main]
0010  gload       0
stack=[ 4 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 4 ]            calls=[main]
0017  iconst      1
stack=[ 4 1 ]          calls=[main]
0019  iadd
stack=[ 5 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 5 ]            calls=[main]
0010  gload       0
stack=[ 5 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
```

```
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 5 ]            calls=[main]
0017  iconst      1
stack=[ 5 1 ]          calls=[main]
0019  iadd
stack=[ 6 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 6 ]            calls=[main]
0010  gload       0
stack=[ 6 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 6 ]            calls=[main]
0017  iconst      1
stack=[ 6 1 ]          calls=[main]
0019  iadd
stack=[ 7 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 7 ]            calls=[main]
0010  gload       0
stack=[ 7 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 7 ]            calls=[main]
0017  iconst      1
stack=[ 7 1 ]          calls=[main]
0019  iadd
stack=[ 8 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 8 ]            calls=[main]
0010  gload       0
stack=[ 8 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
```

```
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 8 ]            calls=[main]
0017  iconst      1
stack=[ 8 1 ]          calls=[main]
0019  iadd
stack=[ 9 ]            calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 9 ]            calls=[main]
0010  gload       0
stack=[ 9 10 ]         calls=[main]
0012  ilt
stack=[ True ]         calls=[main]
0013  brf         24
stack=[ ]              calls=[main]
0015  gload       1
stack=[ 9 ]            calls=[main]
0017  iconst      1
stack=[ 9 1 ]          calls=[main]
0019  iadd
stack=[ 10 ]           calls=[main]
0020  gstore      1
stack=[ ]              calls=[main]
0022  br          8
stack=[ ]              calls=[main]
0008  gload       1
stack=[ 10 ]           calls=[main]
0010  gload       0
stack=[ 10 10 ]        calls=[main]
0012  ilt
stack=[ False ]        calls=[main]
0013  brf         24
stack=[ ]              calls=[main]
0024  gload       1
stack=[ 10 ]           calls=[main]
0026  print
10
stack=[ ]              calls=[main]
0027  halt
stack=[ ]
Data memory
0: 10
1: 10

None
```

- **Output of fifth program:**

```
0000  iconst      4
stack=[ 4 ]            calls=[main]
```

```
0002  call        foo
stack=[ ]              calls=[main foo]
0005  load        0
stack=[ 4 ]            calls=[main foo]
0007  print
4
stack=[ ]              calls=[main foo]
0008  ret
stack=[ ]              calls=[main]
0004  halt
stack=[ ]
Data memory

None
```

- **Output of sixth program:**

```
0021  iconst      5
stack=[ 5 ]            calls=[main]
0023  call        factorial
stack=[ ]              calls=[main factorial]
0000  load        0
stack=[ 5 ]            calls=[main factorial]
0002  iconst      2
stack=[ 5 2 ]          calls=[main factorial]
0004  ilt
stack=[ False ]        calls=[main factorial]
0005  brf         10
stack=[ ]              calls=[main factorial]
0010  load        0
stack=[ 5 ]            calls=[main factorial]
0012  load        0
stack=[ 5 5 ]          calls=[main factorial]
0014  iconst      1
stack=[ 5 5 1 ]        calls=[main factorial]
0016  isub
stack=[ 5 4 ]          calls=[main factorial]
0017  call        factorial
stack=[ 5 ]            calls=[main factorial factorial]
0000  load        0
stack=[ 5 4 ]          calls=[main factorial factorial]
0002  iconst      2
stack=[ 5 4 2 ]        calls=[main factorial factorial]
0004  ilt
stack=[ 5 False ]      calls=[main factorial factorial]
0005  brf         10
stack=[ 5 ]            calls=[main factorial factorial]
0010  load        0
stack=[ 5 4 ]          calls=[main factorial factorial]
0012  load        0
stack=[ 5 4 4 ]        calls=[main factorial factorial]
0014  iconst      1
stack=[ 5 4 4 1 ]      calls=[main factorial factorial]
0016  isub
stack=[ 5 4 3 ]        calls=[main factorial factorial]
```

```
0017  call        factorial
stack=[ 5 4 ]          calls=[main factorial factorial factorial]
0000  load        0
stack=[ 5 4 3 ]        calls=[main factorial factorial factorial]
0002  iconst      2
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial]
0004  ilt
stack=[ 5 4 False ]    calls=[main factorial factorial factorial]
0005  brf         10
stack=[ 5 4 ]          calls=[main factorial factorial factorial]
0010  load        0
stack=[ 5 4 3 ]        calls=[main factorial factorial factorial]
0012  load        0
stack=[ 5 4 3 3 ]      calls=[main factorial factorial factorial]
0014  iconst      1
stack=[ 5 4 3 3 1 ]    calls=[main factorial factorial factorial]
0016  isub
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial]
0017  call        factorial
stack=[ 5 4 3 ]        calls=[main factorial factorial factorial factorial]
0000  load        0
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial factorial]
0002  iconst      2
stack=[ 5 4 3 2 2 ]    calls=[main factorial factorial factorial factorial]
0004  ilt
stack=[ 5 4 3 False ]  calls=[main factorial factorial factorial factorial]
0005  brf         10
stack=[ 5 4 3 ]        calls=[main factorial factorial factorial factorial]
0010  load        0
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial factorial]
0012  load        0
stack=[ 5 4 3 2 2 ]    calls=[main factorial factorial factorial factorial]
0014  iconst      1
stack=[ 5 4 3 2 2 1 ]  calls=[main factorial factorial factorial factorial]
0016  isub
stack=[ 5 4 3 2 1 ]    calls=[main factorial factorial factorial factorial]
0017  call        factorial
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial factorial factorial]
0000  load        0
stack=[ 5 4 3 2 1 ]    calls=[main factorial factorial factorial factorial factorial]
0002  iconst      2
stack=[ 5 4 3 2 1 2 ]  calls=[main factorial factorial factorial factorial factorial]
0004  ilt
stack=[ 5 4 3 2 True ] calls=[main factorial factorial factorial factorial factorial]
0005  brf         10
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial factorial factorial]
0007  iconst      1
stack=[ 5 4 3 2 1 ]    calls=[main factorial factorial factorial factorial factorial]
0009  ret
stack=[ 5 4 3 2 1 ]    calls=[main factorial factorial factorial factorial]
0019  imul
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial factorial]
0020  ret
stack=[ 5 4 3 2 ]      calls=[main factorial factorial factorial]
```

```
0019  imul
stack=[ 5 4 6 ]        calls=[main factorial factorial factorial]
0020  ret
stack=[ 5 4 6 ]        calls=[main factorial factorial]
0019  imul
stack=[ 5 24 ]         calls=[main factorial factorial]
0020  ret
stack=[ 5 24 ]         calls=[main factorial]
0019  imul
stack=[ 120 ]          calls=[main factorial]
0020  ret
stack=[ 120 ]          calls=[main]
0025  print
120
stack=[ ]              calls=[main]
0026  halt
stack=[ ]
Data memory

None
```