Lecture #25

Run-time System & Code Generation

Procedure Activations

```
class main {
    int g() : { return1; };
    int f(int x) { if (x == 0) then return g(); else
                                        return 1+ f(x - 1); };

    int main()  { f(3); };
}
```

Nesting of activations

- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Activation Records

- Information needed to manage one procedure activation is called an *activation record (AR) or frame*
- If procedure **F** calls **G**, then **G**'s activation record contains a mix of info about **F** and **G**.
- **F** is "suspended" until **G** completes, at which point **F** resumes
- **G**'s AR contains information needed to
  - Complete execution of **G**
  - Resume execution of **F**

Activation Records

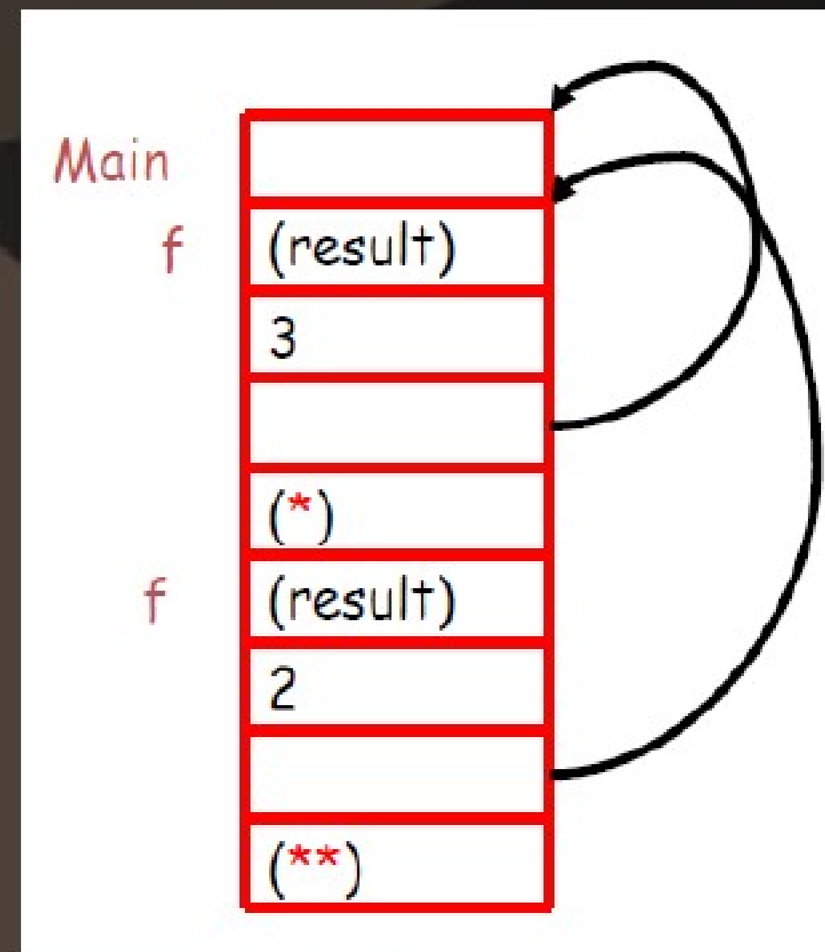
- Space for **G**'s return value
- Actual parameters
- Pointer to the previous activation record
  - The *control link; points to AR of caller of* **G**
- Machine status prior to calling **G**
  - Contents of registers & program counter
  - Local variables
- Other temporary values

Activation Records

```
class main {
    int g() : { return1; };
    int f(int x) { if (x == 0) then return g();
                   else return 1+ f(x - 1);(**) };
    int main()  {f(3); (*)};
}
```

- AR:

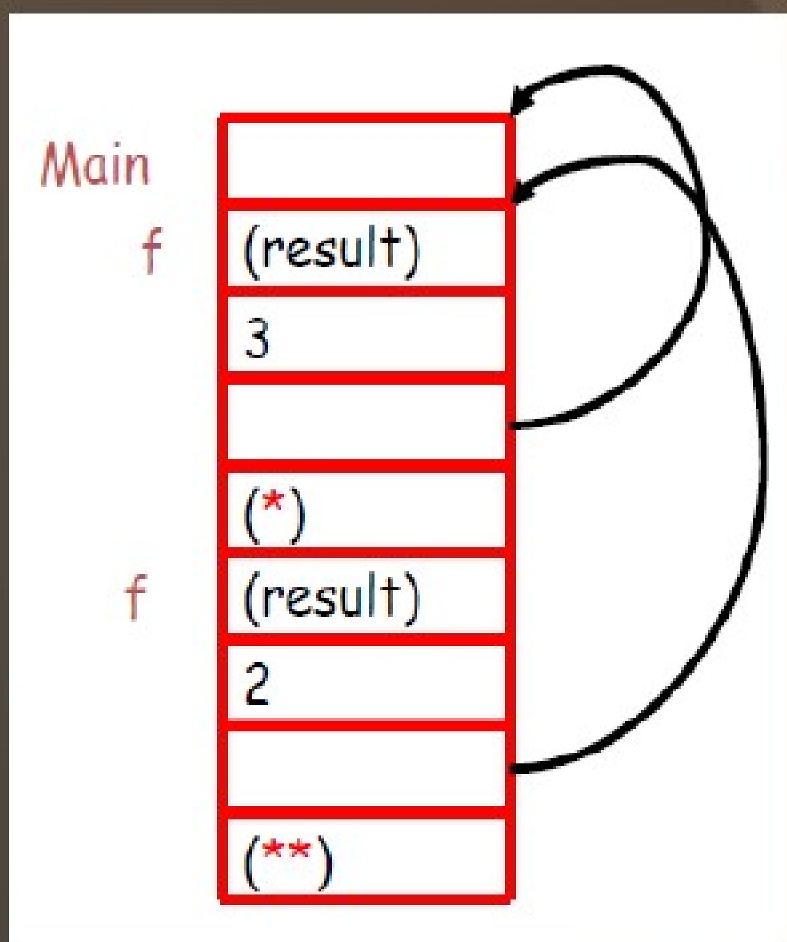
result
argument
control link
return address



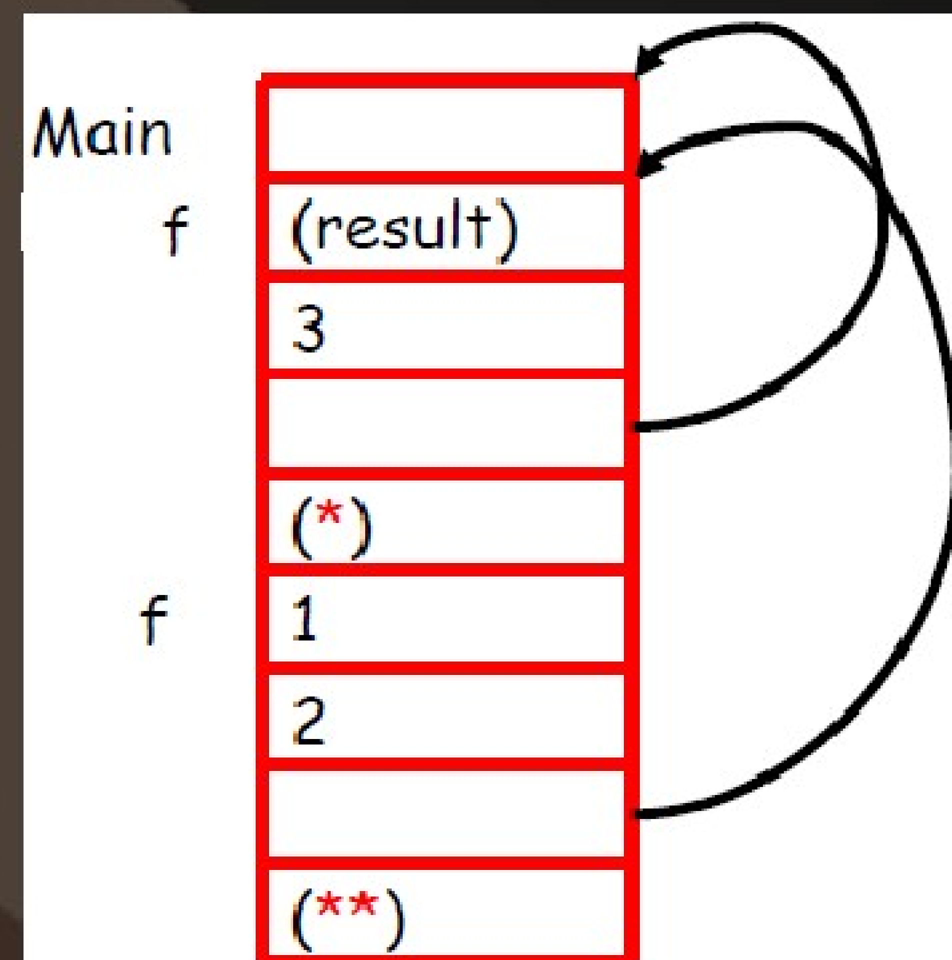
- **Main** has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of **f**

Activation Records

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame

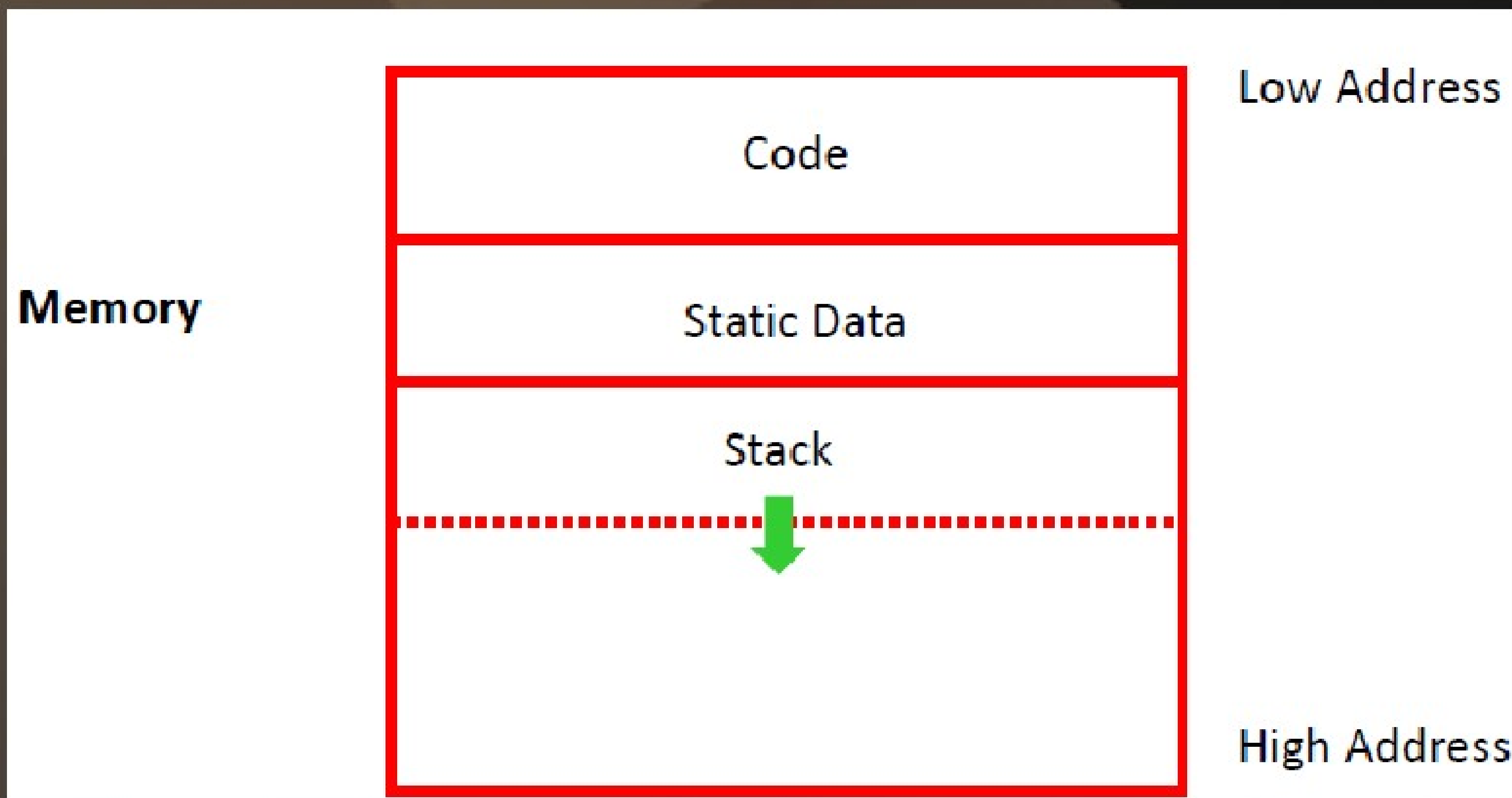


Stack state after the call to the 2nd invocation of **f** returns



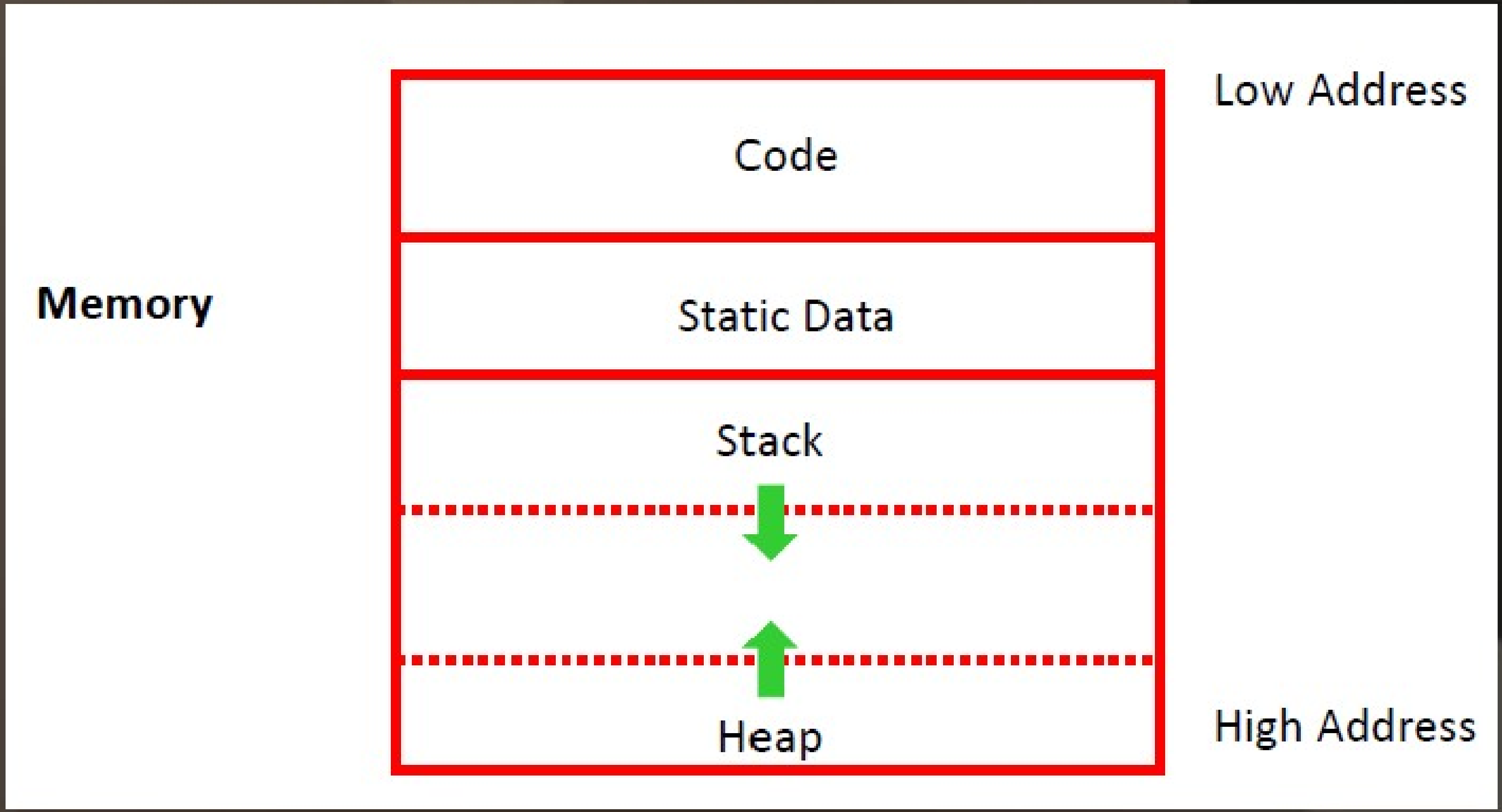
Global Data

- All references to a global variable point to the same object
  - Can't store a global in an activation record
- Globals are assigned a fixed address statically



Heap

- A value that outlives the procedure that creates it cannot be kept in AR
  - method foo() { new Bar }
  - The Bar value must survive deallocation of foo's AR
- A *heap* is generally used to store dynamically allocated data



Alignment

- Most modern machines are 32 or 64 bit
- Machines are either byte or word addressable
- Data is *word aligned if it begins at a word boundary*
- Machines generally have alignment restrictions
  - Accessing mis-aligned data incurs significant overhead – say 5x slower
- *Padding* is used to word align next data object in memory
  - Most frequently used with strings
  - Say we have the string "Hello" – requires 2 "padding" characters

Stack Machines

- Only storage is a stack
- An instruction $r = F(a_1, \ldots a_n)$:
  - Pops **n** operands from the stack
  - Computes the operation **F** using the operands
  - Pushes the result **r** on the stack

- Consider two instructions:
  - push i - push integer i on the stack
  - add - add two integers

Stack Machines

- A program:

```
push 7
push 5
Add
```

- Stack machines provide a simple machine model
  - Simple compiler
  - Inefficient
- Location of the operands/result is not explicitly stated
  - Always the top of the stack

Stack Machines

- Stack machine Vs. register machine
  - **add** instead of **add r_1, r_2, r_3**
  - More compact programs
- One reason why Java bytecode uses stack evaluation
- There is an intermediate point between a pure stack machine and a pure register machine
- An *n-register stack machine*
  - Conceptually, keep the top *n locations of the pure stack machine's stack in registers*
- 1-register stack machine
  - The register is called the *accumulator*

Stack Machines

- In a pure stack machine
  - An **add** does **3** memory operations: Two reads and one write
- In a 1-register stack machine the **add** does
  - acc ← acc + top_of_stack
- In general, for an operation $\mathbf{op(e_1,\ldots,e_n)}$
  - $\mathbf{e_1,\ldots,e_n}$ are subexpressions
- For each $\mathbf{e_i}$ $\mathbf{(0 < i < n)}$
  - Compute $\mathbf{e_i}$
  - Push result on the stack
- Pop **n-1** values from the stack, compute **op**
- Store result in the accumulator

Stack Machines

***Operations for the stack machine with accumulator*: 3 + (7 + 5)**

Code	**Acc**	**Stack**
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>