BLEKINGE INSTITUTE OF TECHNOLOGY
BTH

Code Generation and Interpretation

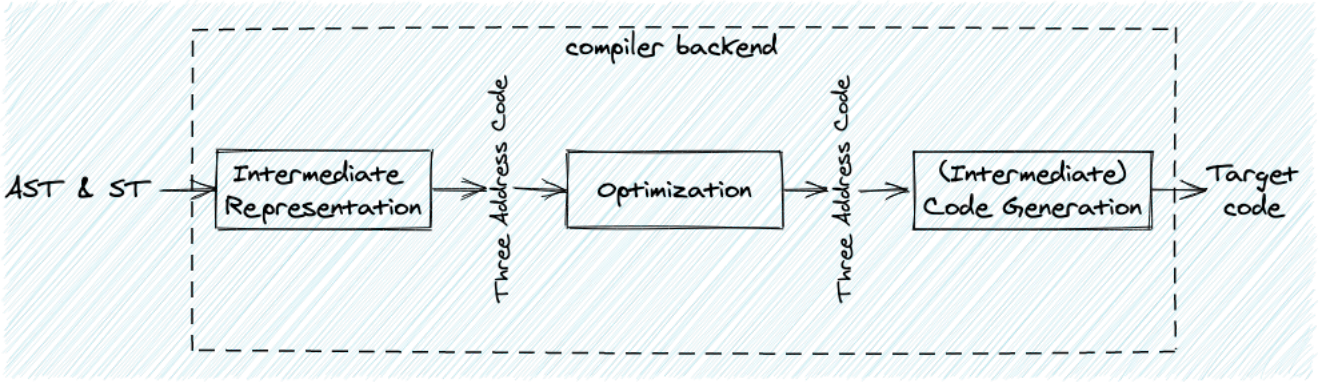
Suejb Memeti

March 2021

Outline

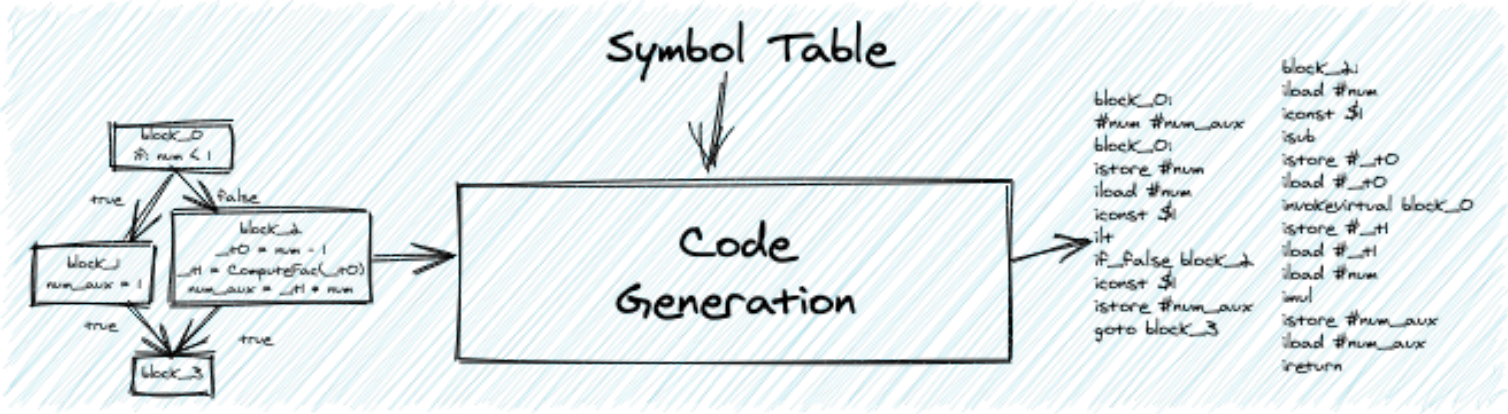
- Backend Overview
- Code Generation
  - Translating CFG into bytecode
- Interpretation
  - Stack machine-based execution of the bytecode

Backend Overview



- Intermediate Representation
  - Transform the AST into a low-level intermediate representation
- Optimization
  - Perform transformations into the IR to improve performance aspects of the program
    - E.g., copy propagation, dead-code elimination, branch-prediction, loop unrolling, …
- (Intermediate) Code generation
  - Generation of the target code
    - E.g., assembly, bytecode, other high-level code (C++, Java, …)

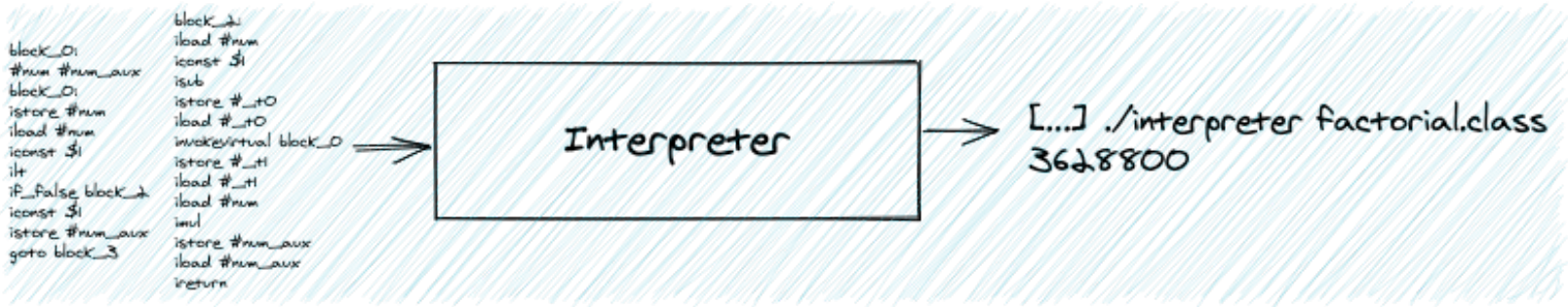
Code Generation



- Input: CFG + Symbol table
- Output: Simplified Java bytecode

- Java bytecode: Low-level intermediate code

Interpreter



- Input: Java bytecode
- Output: program output

- Interpreter: A separate program that reads the generated code and executes it. Like JVM.

Code generation

- Generating java bytecode from Three Address Code is simple

- Steps
  - Traverse the IR (i.e., visit each block and all instructions within a block)
  - Generate bytecode instructions for each TAC instruction
  - For each method declaration
    - Store the variable names (including the temporary generated names during IR construction). Use the symbol table for this
  - Store the result in a file

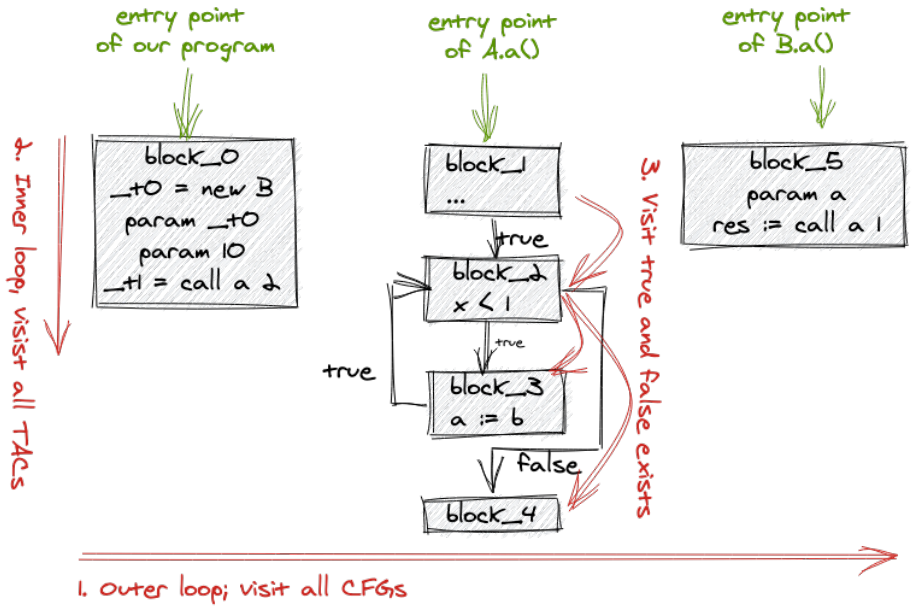
Traversing the CFG

```
list<BBlock> methods;

BBlock::generateBytecode(){
  for(BBlock i: methods)
    i.generateCode();
}

BBlock::generateCode() {
  for(TAC i: instructions)
    i.generateCode();

  if(trueExit && !falseExit)
    genUncondJumpIns(trueExit);
  else if (trueExit && falseExit)
    genCondJumpIns(condition, trueExit, falseExit);
}

TAC::generateCode() {
  //convert each TAC to bytecode isntruction
}
```

1. Outer loop; visit all CFGs

2. Inner loop, visist all TACs

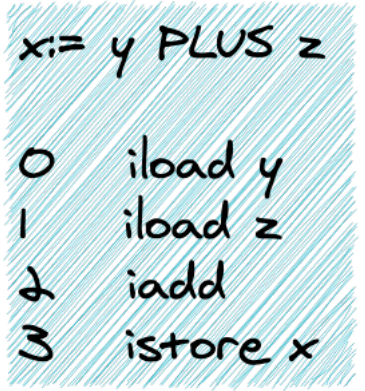
3. Visit true and false exists

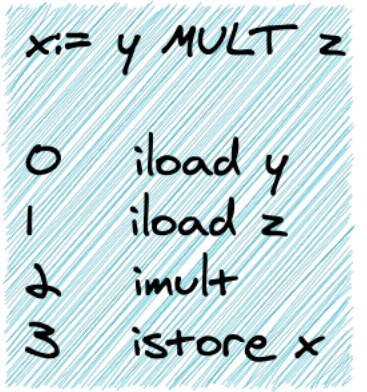


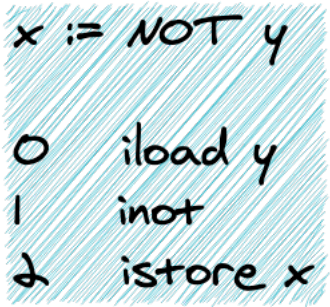
- An outer loop that visits all CFGs
- An inner loop that visits all TAC instructions
- Recursively visit trueExit Block
- Recursively visit falseExit Block
- **Keep track of the visited blocks**

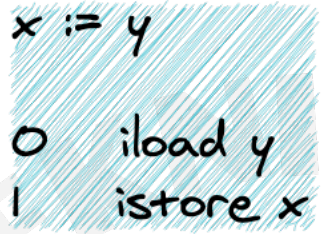
Translating TAC instructions to bytecode

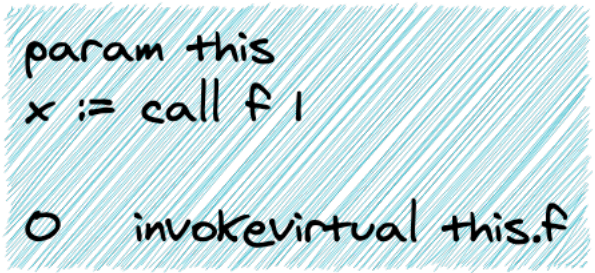
- We use a limited set of bytecode instructions
- iload/istore/iconst
  - to load and store values into the stack
- iadd/isub/imul/ilt/iand/inot
  - Operators of minijava
- Goto/iffalse
  - Conditional and unconditional jumps
- Invokevirtual
  - Method calls
- Ireturn/print/stop
  - Return from method; print; exit

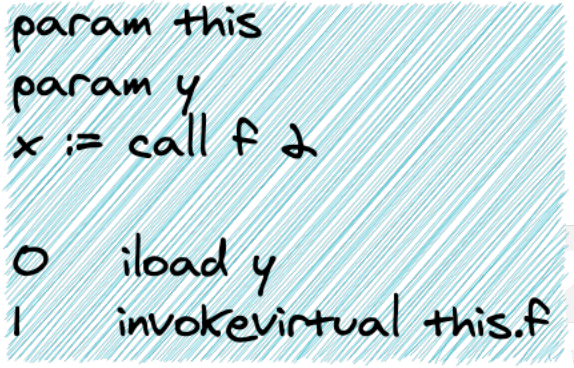


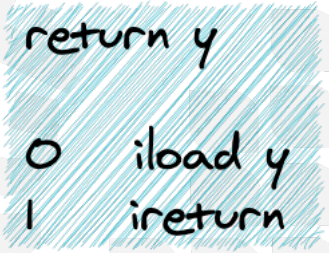










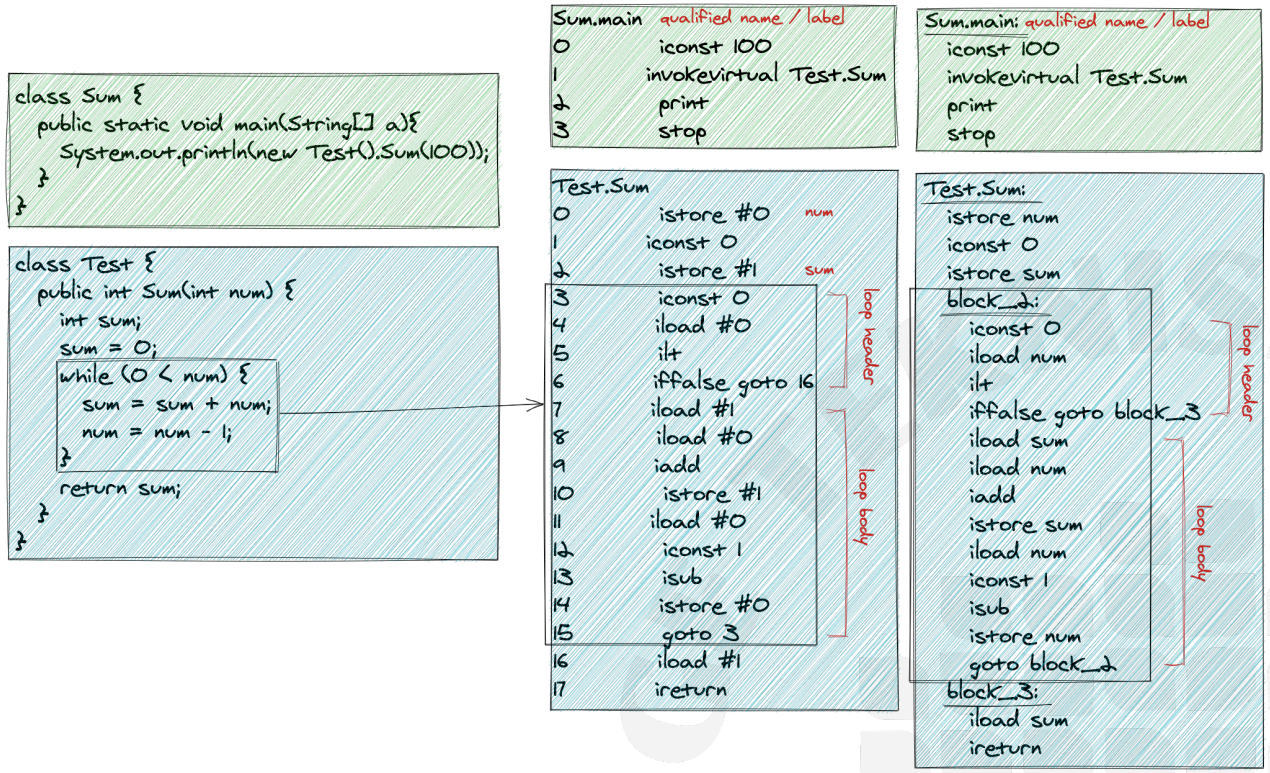


Bytecode instructions

id	Instruction	Description
0	iload n	Push integer value stored in local variable n.
1	iconst v	Push integer value v.
2	istore n	Pop value v and store it in local variable n.
3	iadd	Pop value v1 and v2. Push v2 + v1.
4	isub	Pop value v1 and v2. Push v2 − v1.
4	imul	Pop value v1 and v2. Push v2 * v1.
5	idiv	Pop value v1 and v2. Push v2 / v1.
6	ilt	Pop value v1 and v2. Push 1 if v2 < v1, else push 0.
7	iand	Pop value v1 and v2. Push 0 if v1 * v2 == 0, else push 1.
8	ior	Pop value v1 and v2. Push 0 if v1 + v2 == 0, else push 1.
9	inot	Pop value v. Push 1 if v == 0, else push 0.
10	goto i	Jump to instruction labeled i unconditionally.
11	iffalse goto i	Pop value v from the data stack. If v == 0 jump to instruction labeled i, else continue with the following instruction.
12	invokevirtual m	Push current activation to the activation stack and switch to the method with qualified name m.
13	ireturn	Pop the activation from the activation stack and continue.
14	print	Pop the value from the data stack and print it.
15	stop	Execution completed.

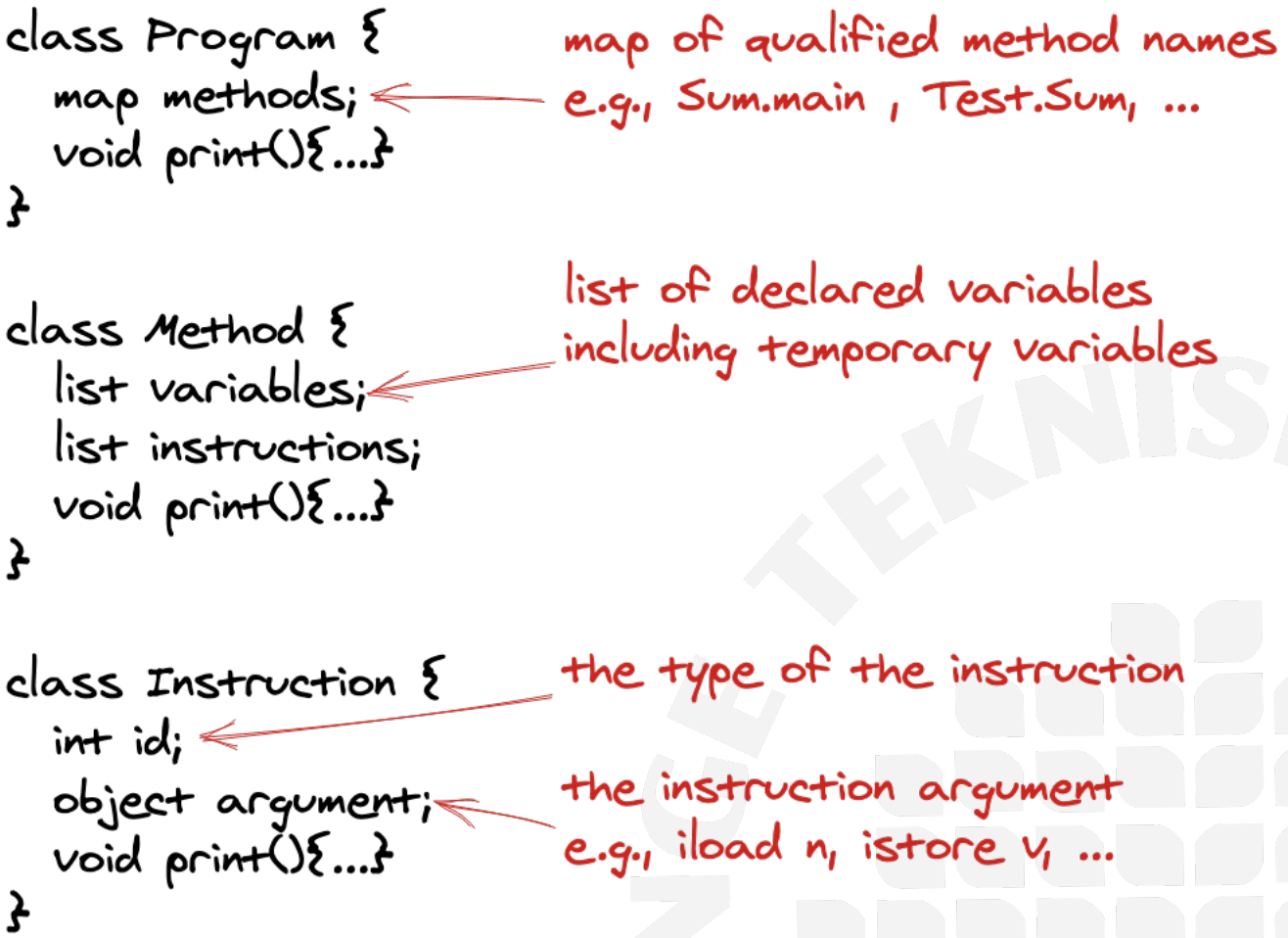
Example

- Simple example
  - Does not generate any temporary variables
- Method labels
  - Qualified method names
    - E.g., *className.methodName*
  - Unique block name
    - E.g., block_0; block_5 …
- Instruction labels
  - Use an incremental number for all instructions
    - E.g., 0, 1, 2, 3
  - Label only instructions used in goto instructions
    - E.g., block_2; block_3
- Access to variables in the stack
  - By index: e.g., #0; #1; …
  - By name: e.g., sum, num



Implementation details

- Solution 1: Build a class hierarchy, for class, method, and Instruction. Serialize it, and store in a file.
  - Requires more effort to design the class hierarchy
  - Easier to interpret
  - Recommended approach
- Solution 2: Traverse the CFG and print each instruction store it in a file
  - Straightforward to generate code
  - More difficult to interpret



Stack machine interpreter

- A separate program to execute/interpret bytecode
- Java and C# use stack machine for execution of source code

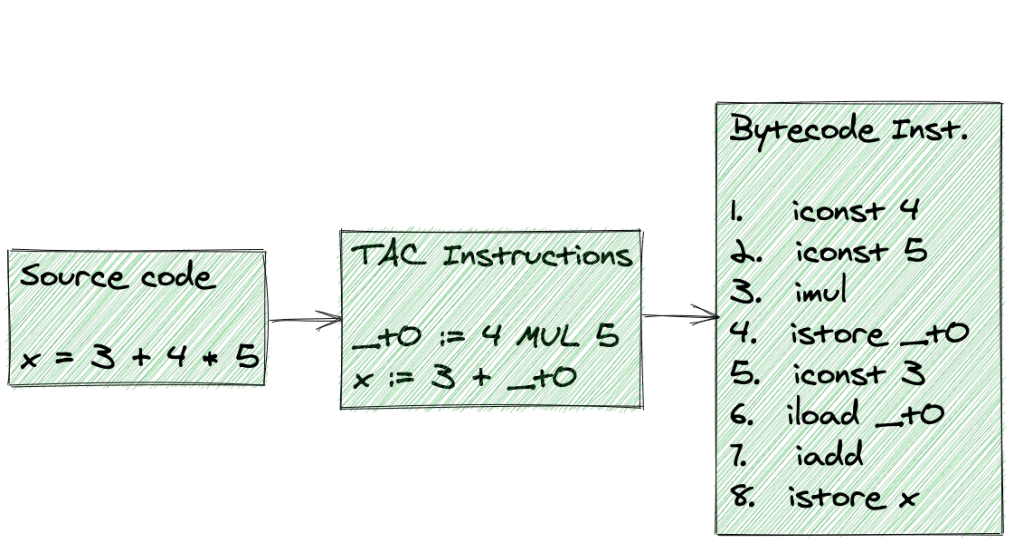
- Advantages
  - The set of instructions is small, i.e. easy to generate code for
  - The user is not exposed to register allocation and management

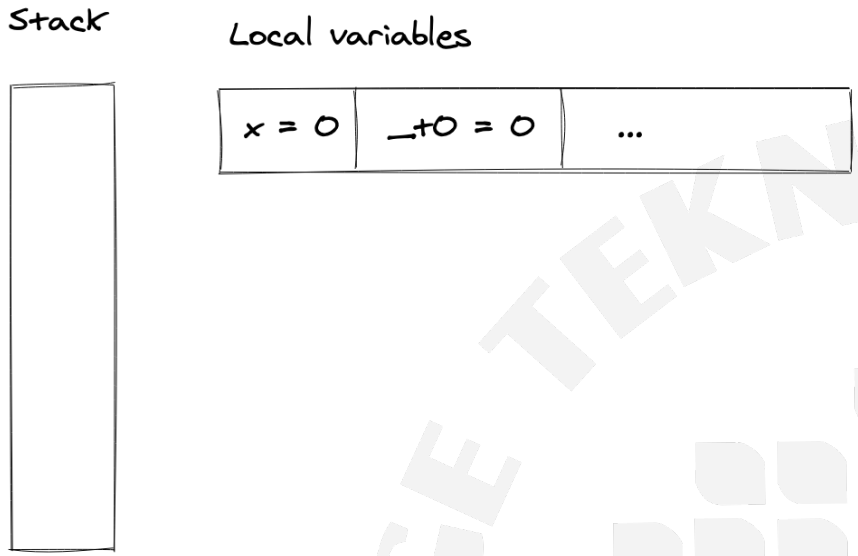
- Disadvantages
  - Machine agnostic - difficult to apply any machine optimization
  - Execution is slightly slower compared to register-based execution

Keeping track of variable values

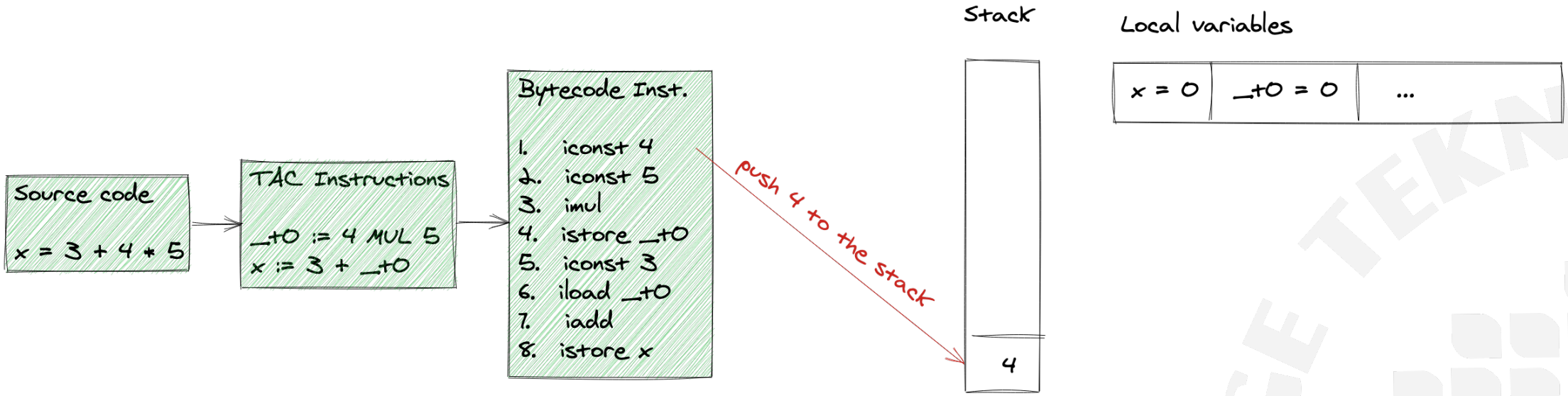
- Bytecode instructions may require access to the current variable values
- We need a data structure to store the current value of all variables
- Solution
  - Use a list or map to store variables
  - You may need a way to access these variables by index or by name
- Bytecode instructions
  - **iload n** – reads a value from the local variable n
  - **istore n** – writes a value to local variable n

Interpreting expressions

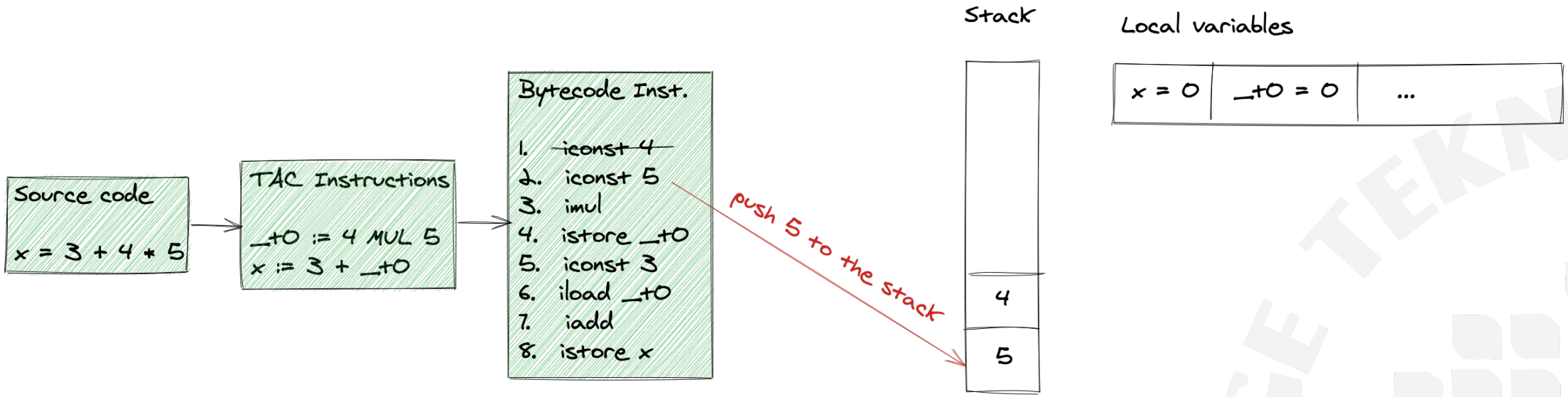




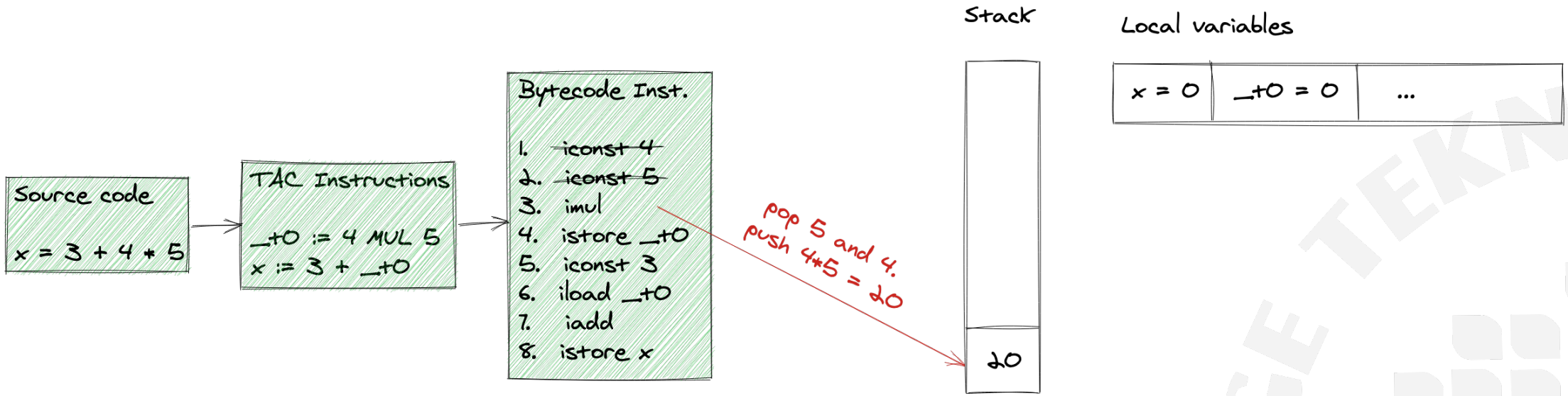
Interpreting expressions



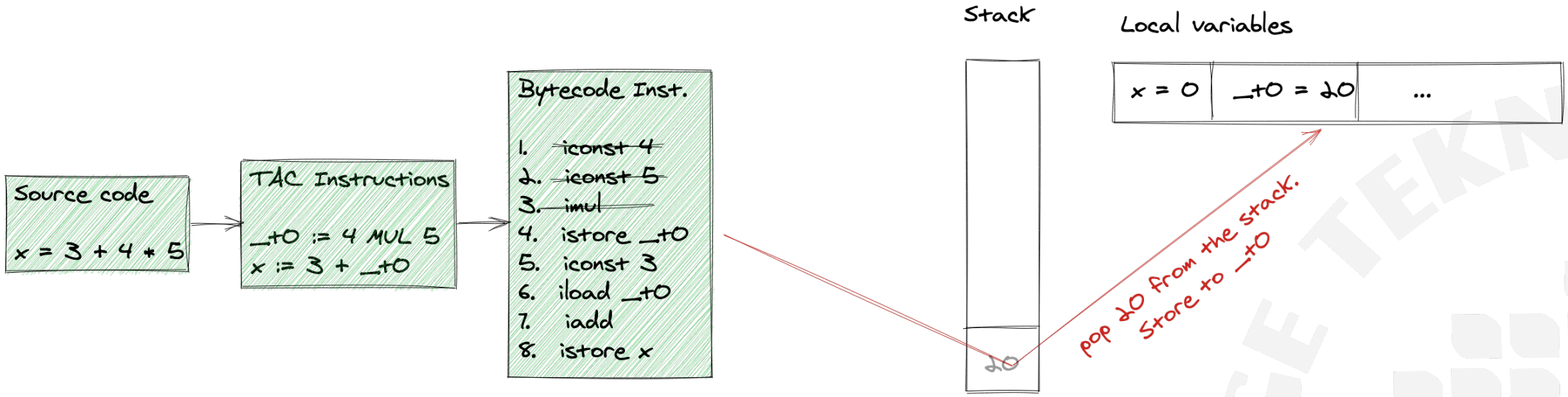
Interpreting expressions



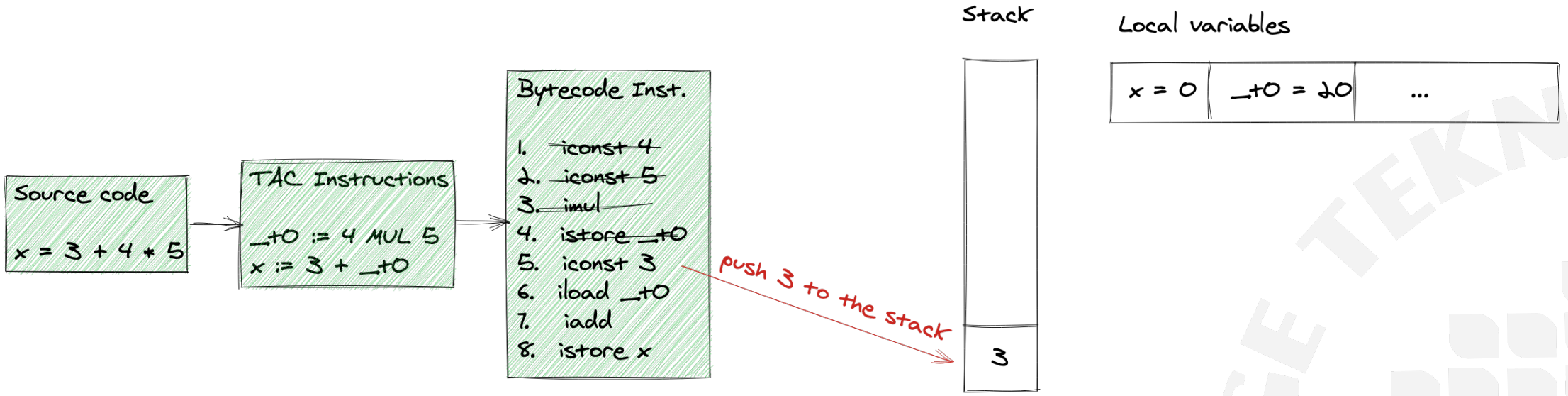
Interpreting expressions



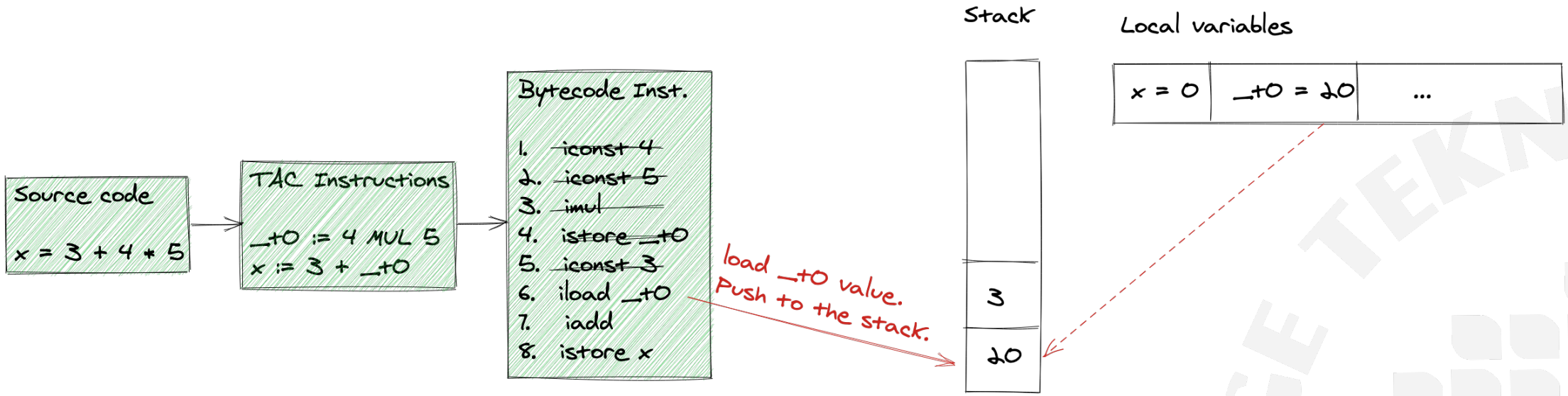
Interpreting expressions



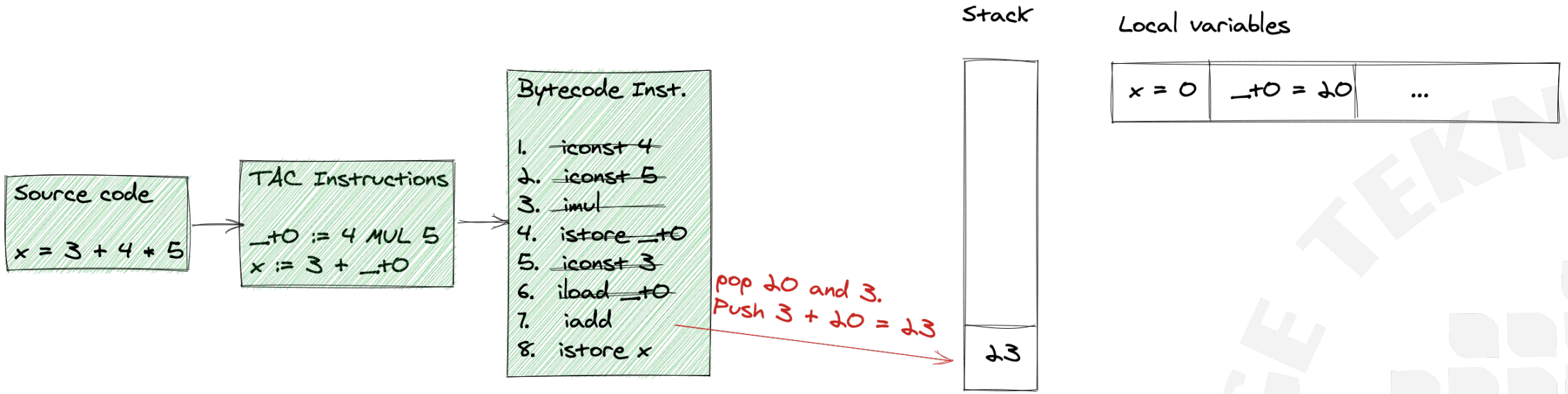
Interpreting expressions



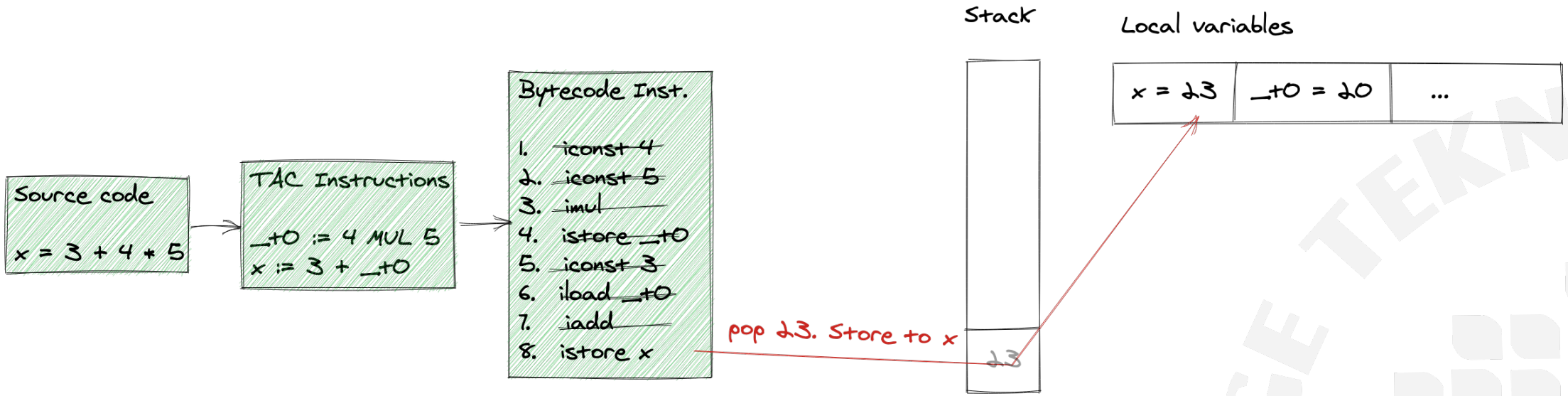
Interpreting expressions



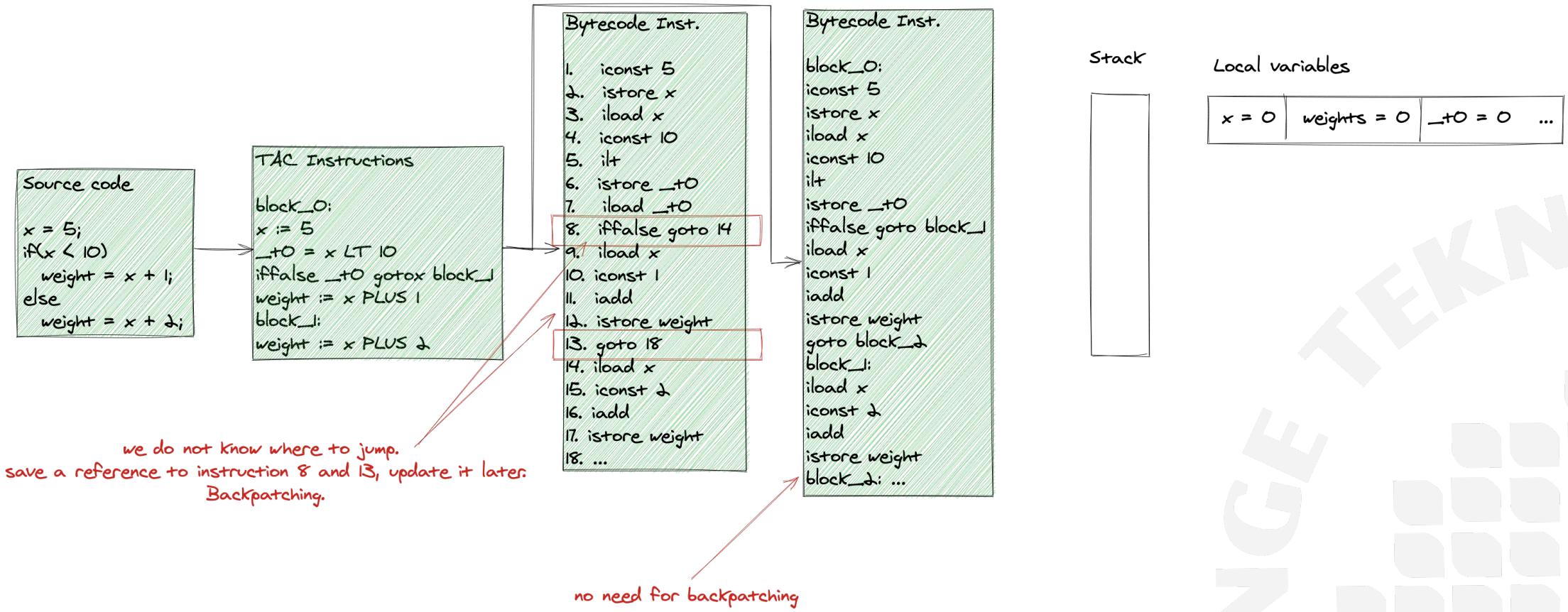
Interpreting expressions



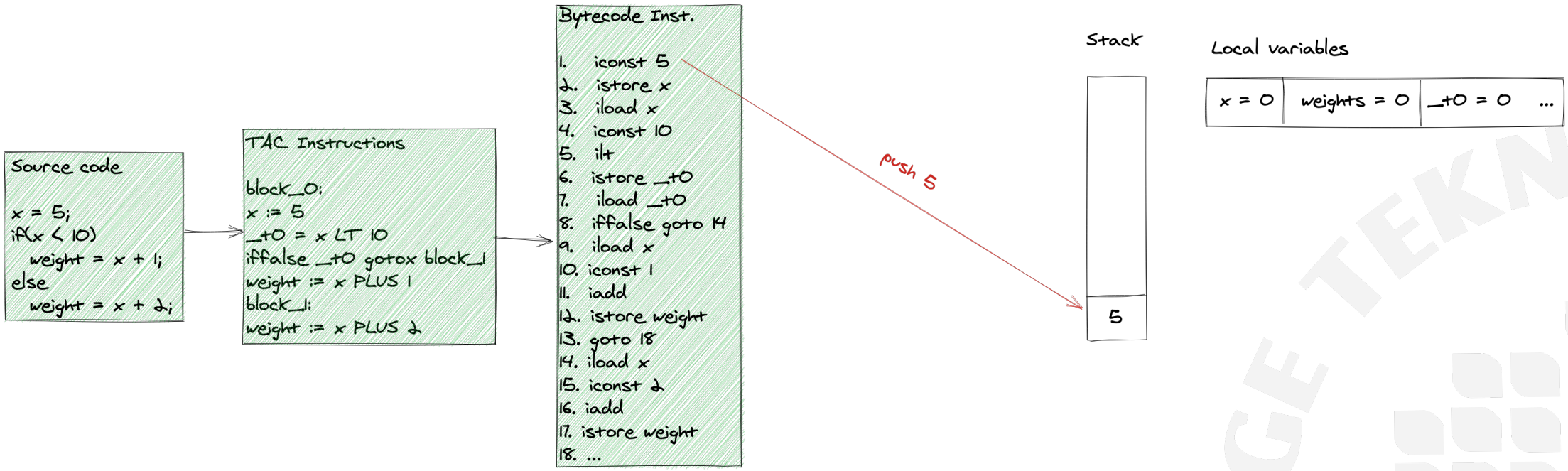
Interpreting expressions



Interpreting if-else statements



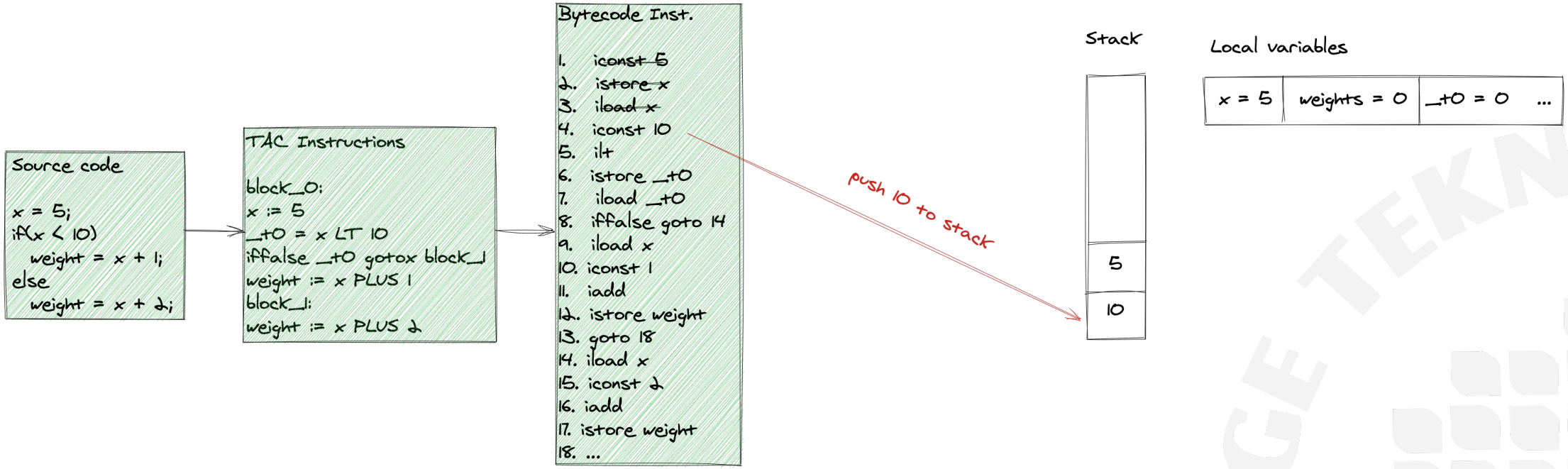
Interpreting if-else statements



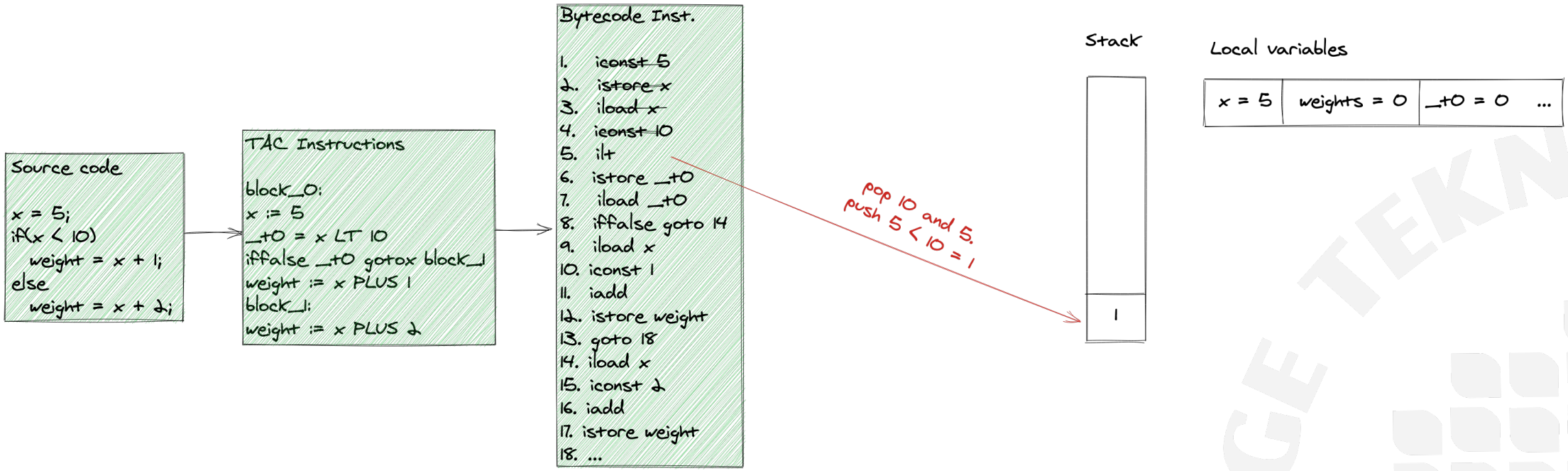
Interpreting if-else statements

Interpreting if-else statements

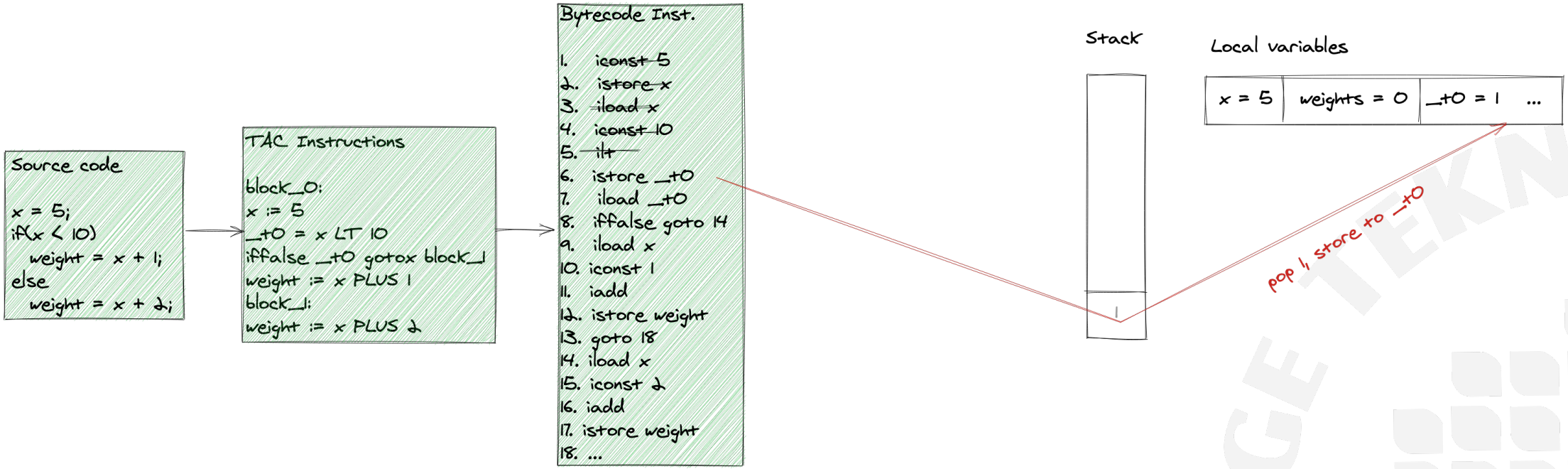
Interpreting if-else statements



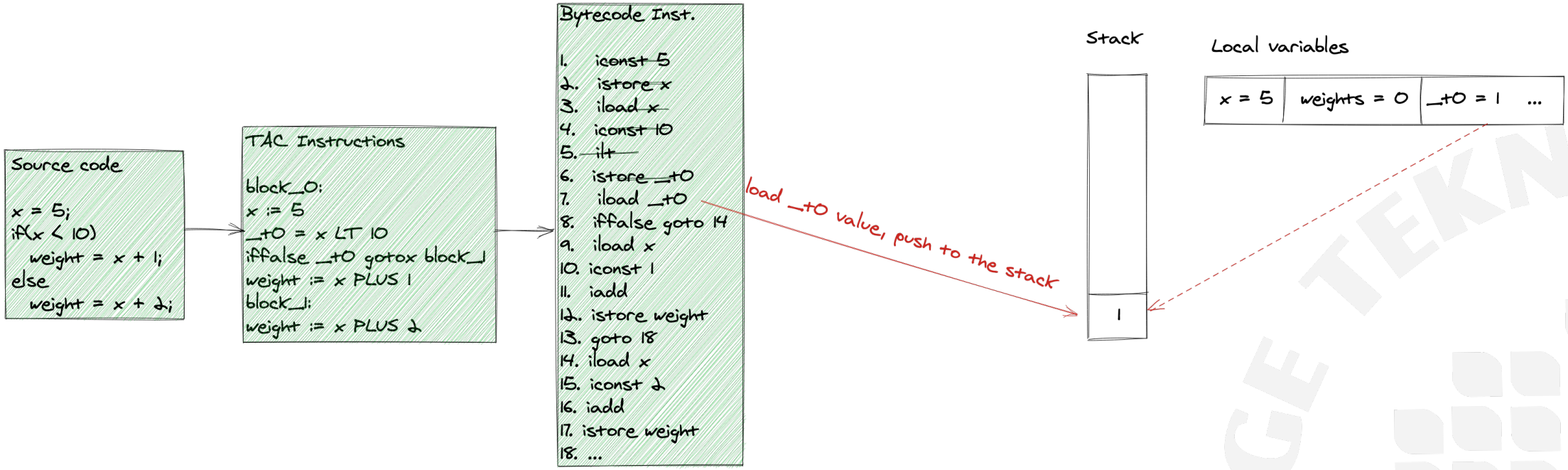
Interpreting if-else statements



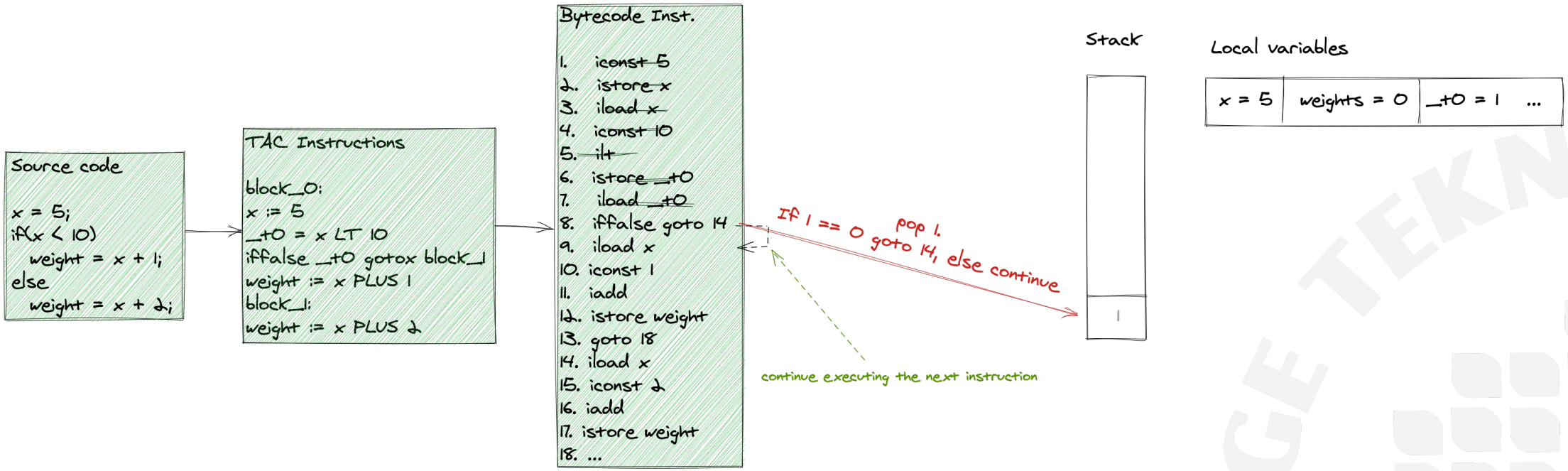
Interpreting if-else statements



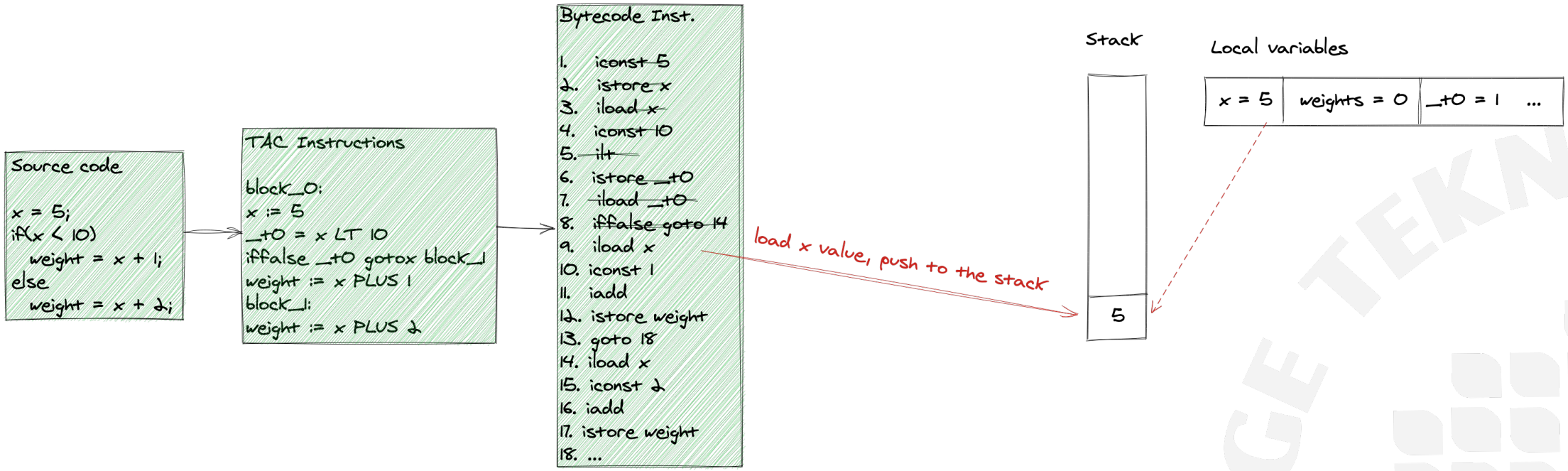
Interpreting if-else statements



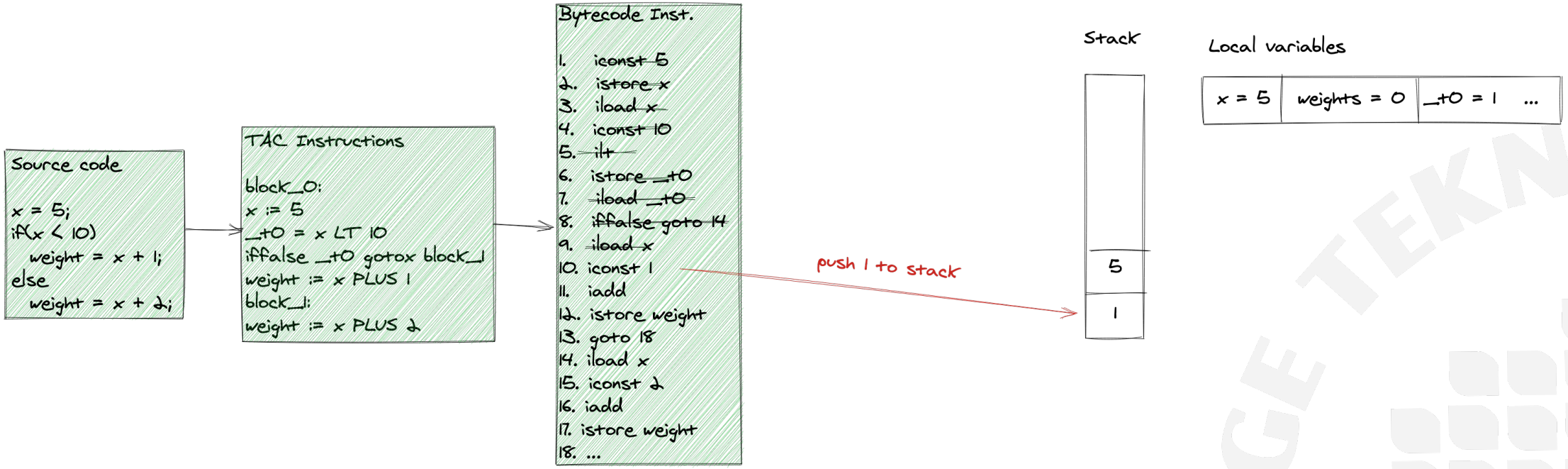
Interpreting if-else statements



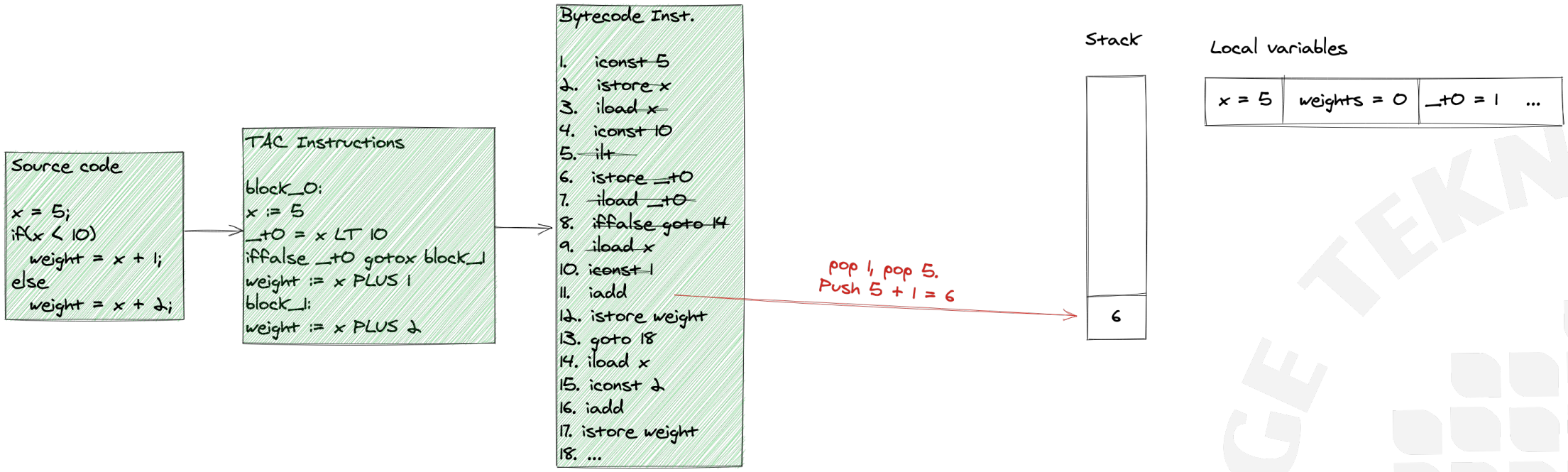
Interpreting if-else statements



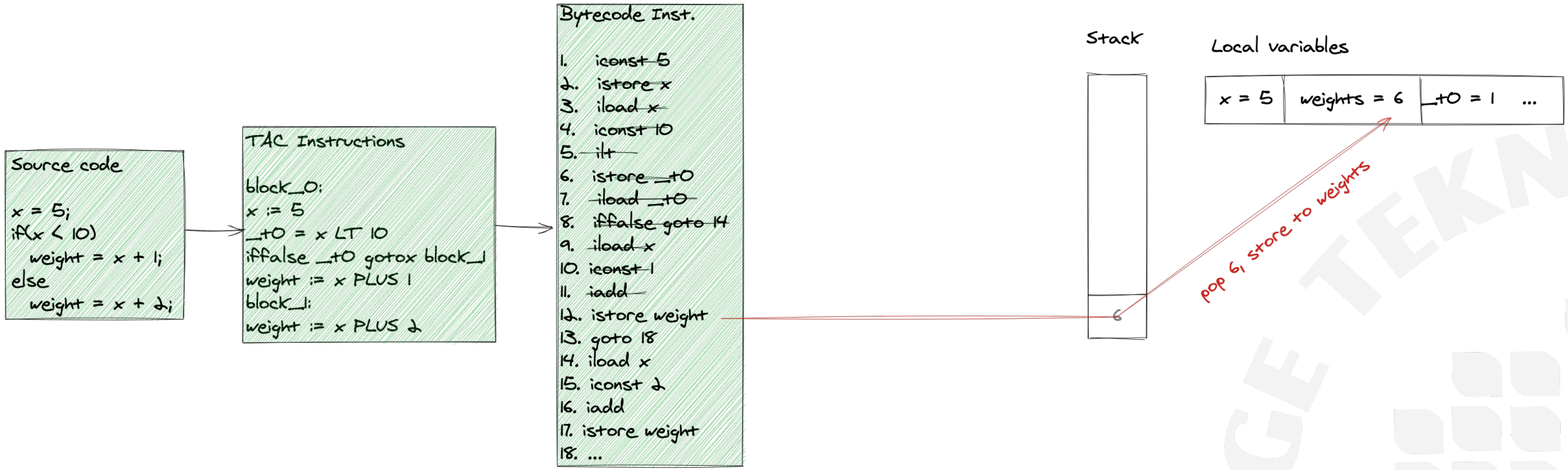
Interpreting if-else statements



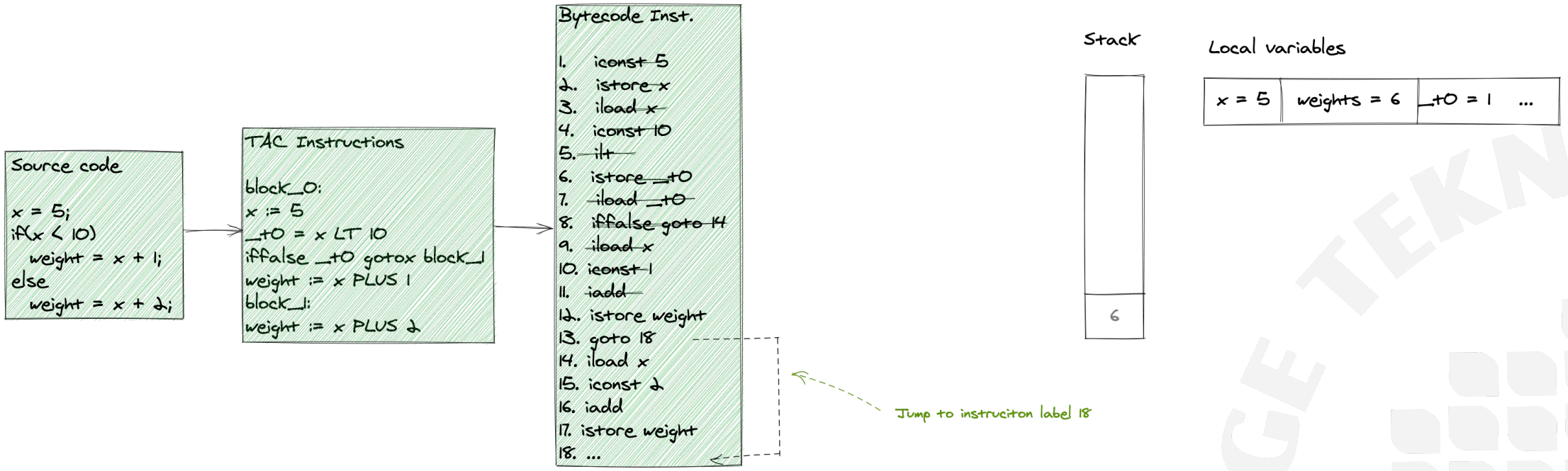
Interpreting if-else statements



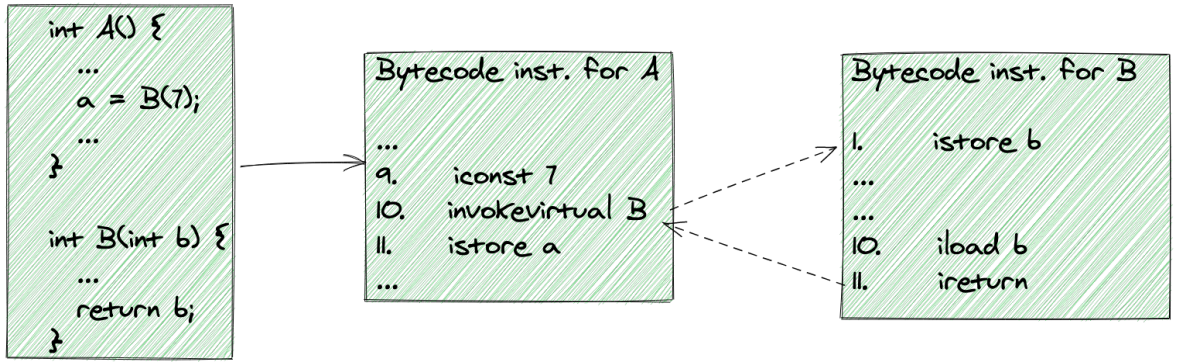
Interpreting if-else statements



Interpreting if-else statements



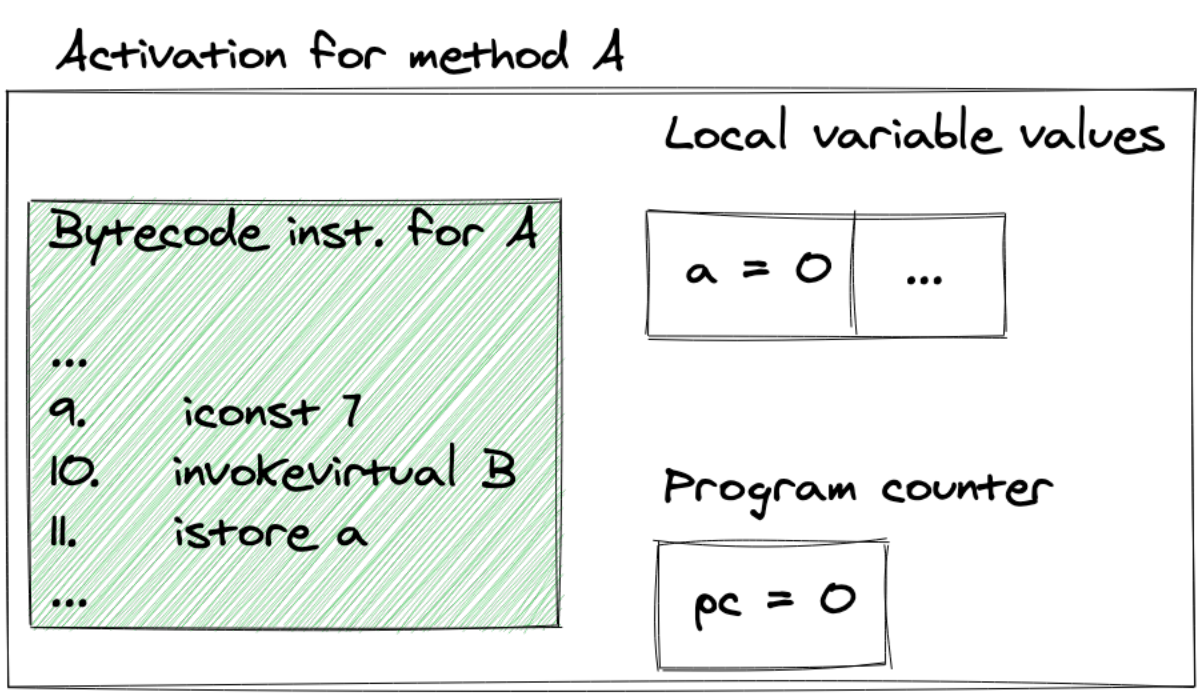
Method calls

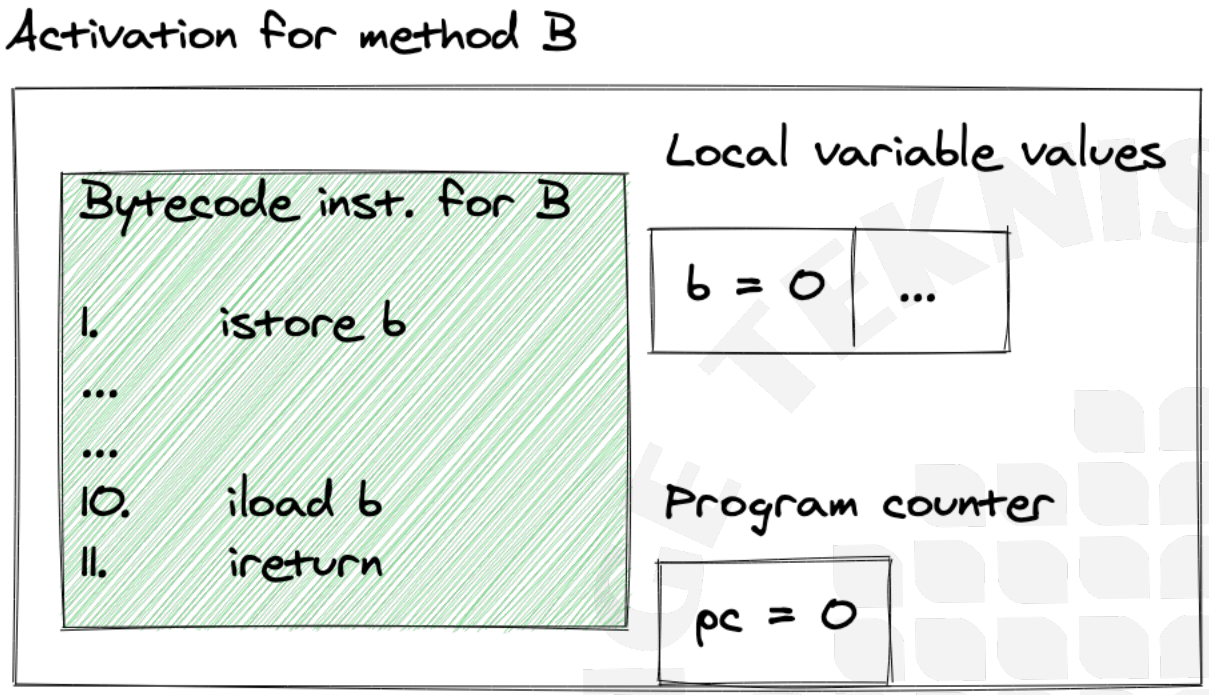


- Each method has a separate set of instructions
- A method call means, we switch the control from one set of instructions to another
- Pass parameters by value
  - Before each method call, all parameters are pushed to the stack
  - On each method, we pop all parameters and assign their values to the method parameters.
  - Same for return values. Push before the return. Pop immediately after a method call.
- We need to keep track of the state and the program counter while switching to another method
  - Implemented as a stack of activations

Activations

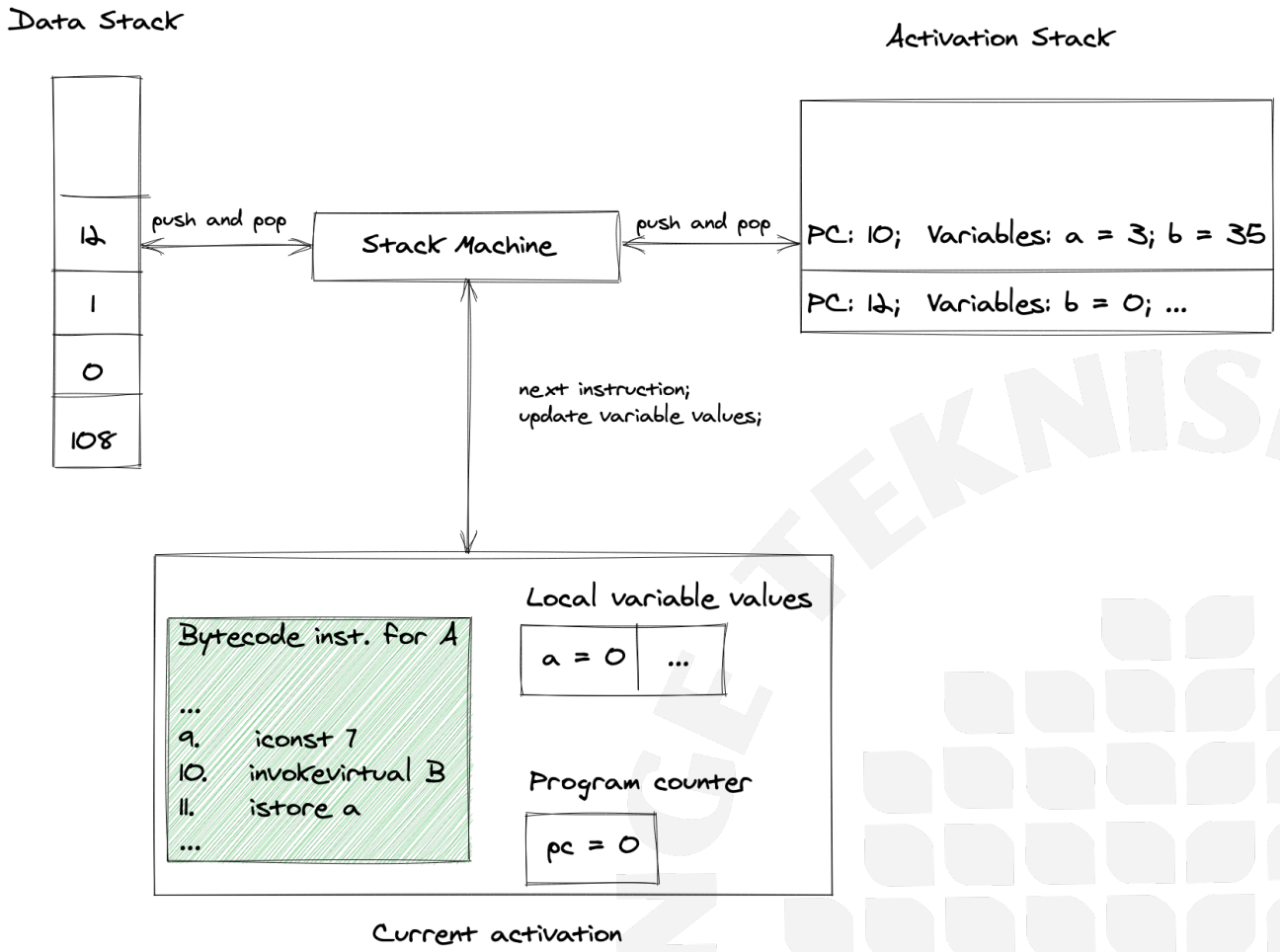
- Each method invocation is associated with a method activation



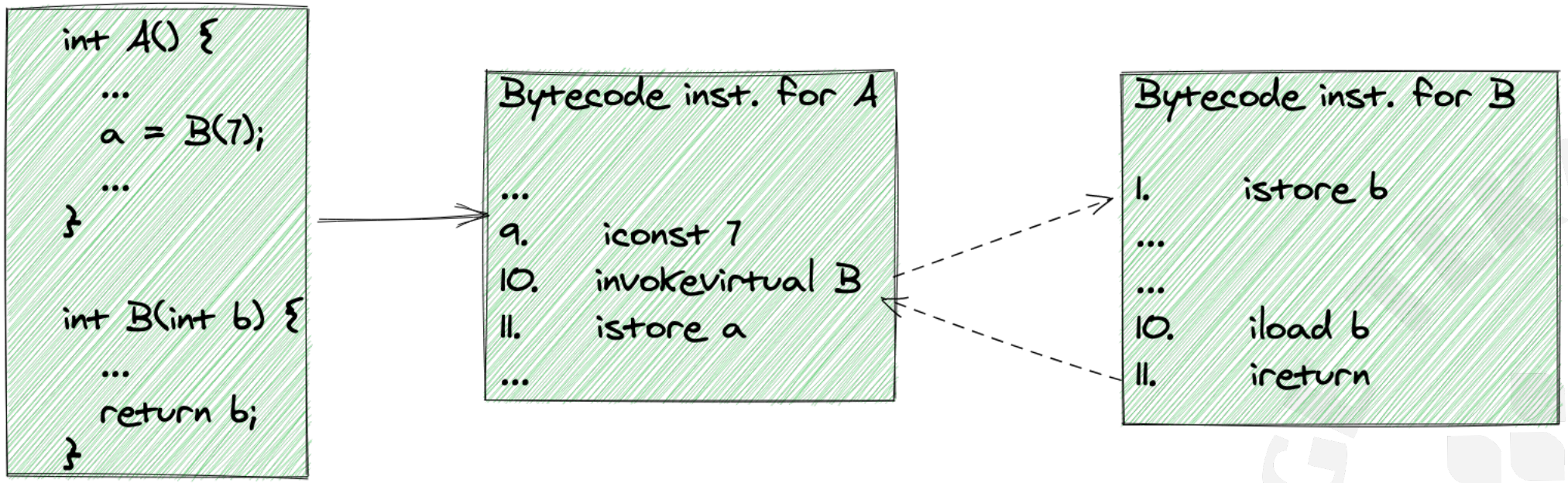


Stack machine

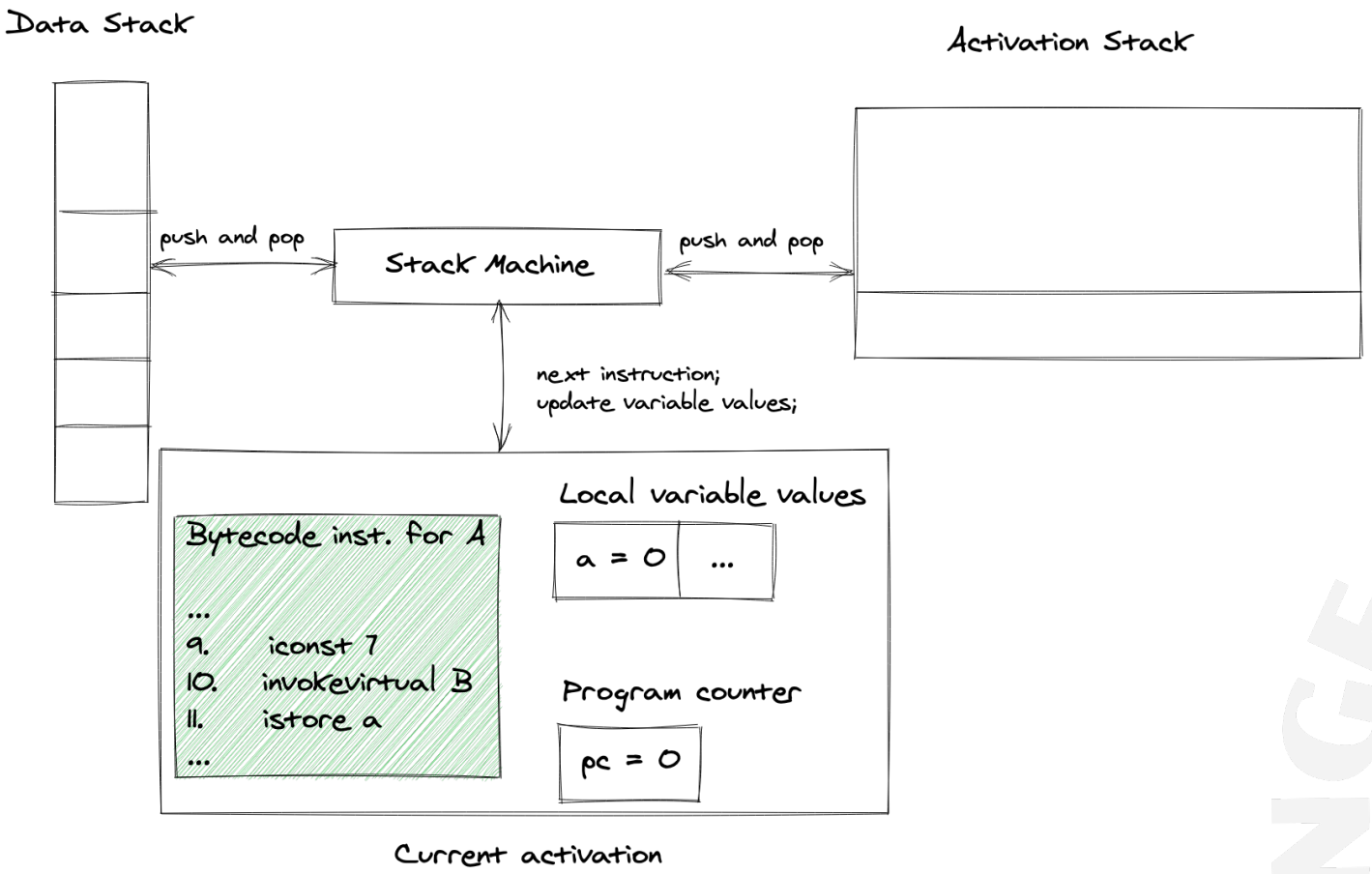
- Each method has a list of instructions and a list of variables
- The data stack is used to store data that might be used between different instructions
- The activation stack is used to store data that represent the current state of a method activation.
  - Keeps track of the previously called but not completed method invokations.
- Recursive method calls result with multiple activations to the same method.



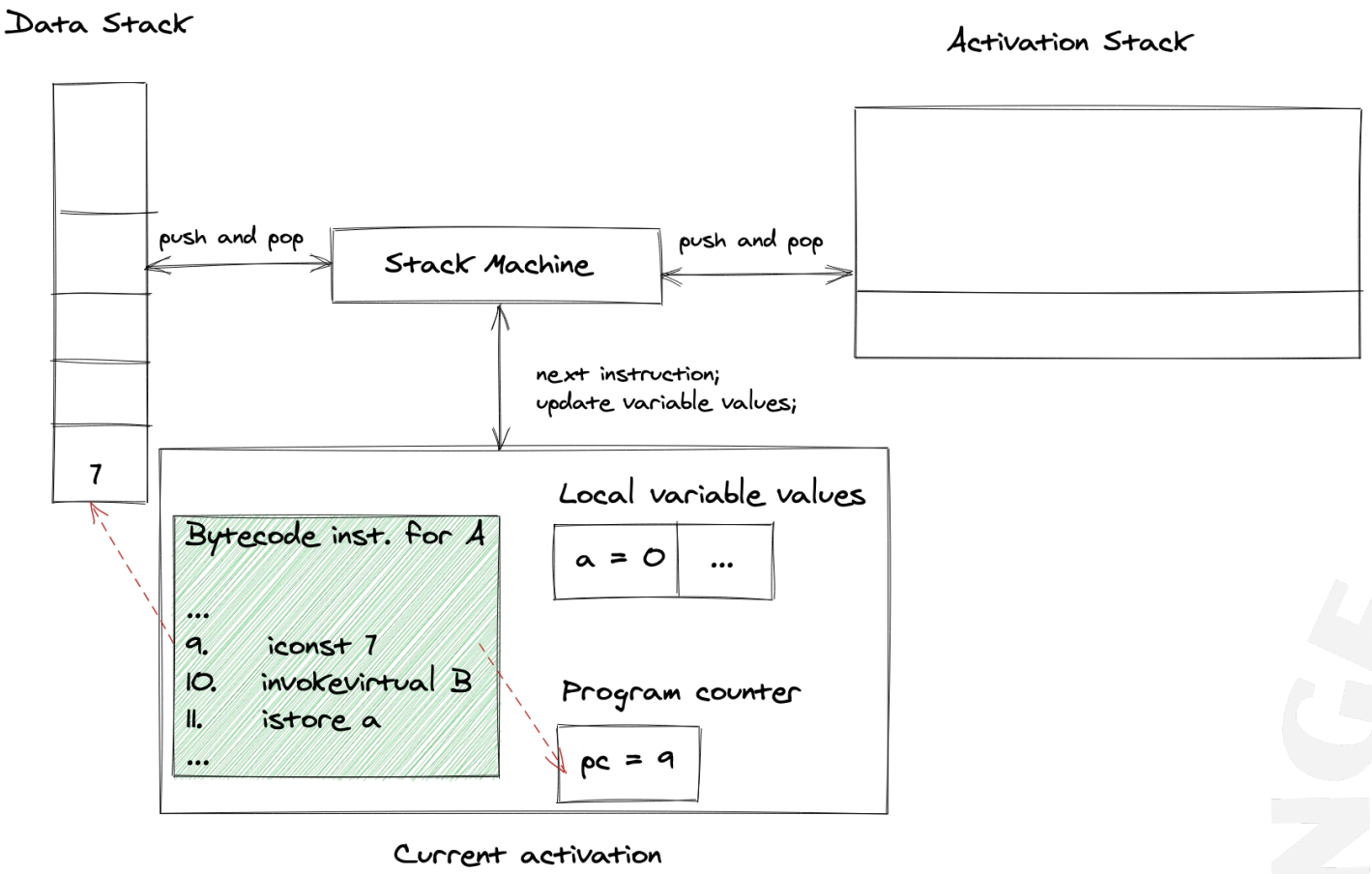
Interpreting method calls



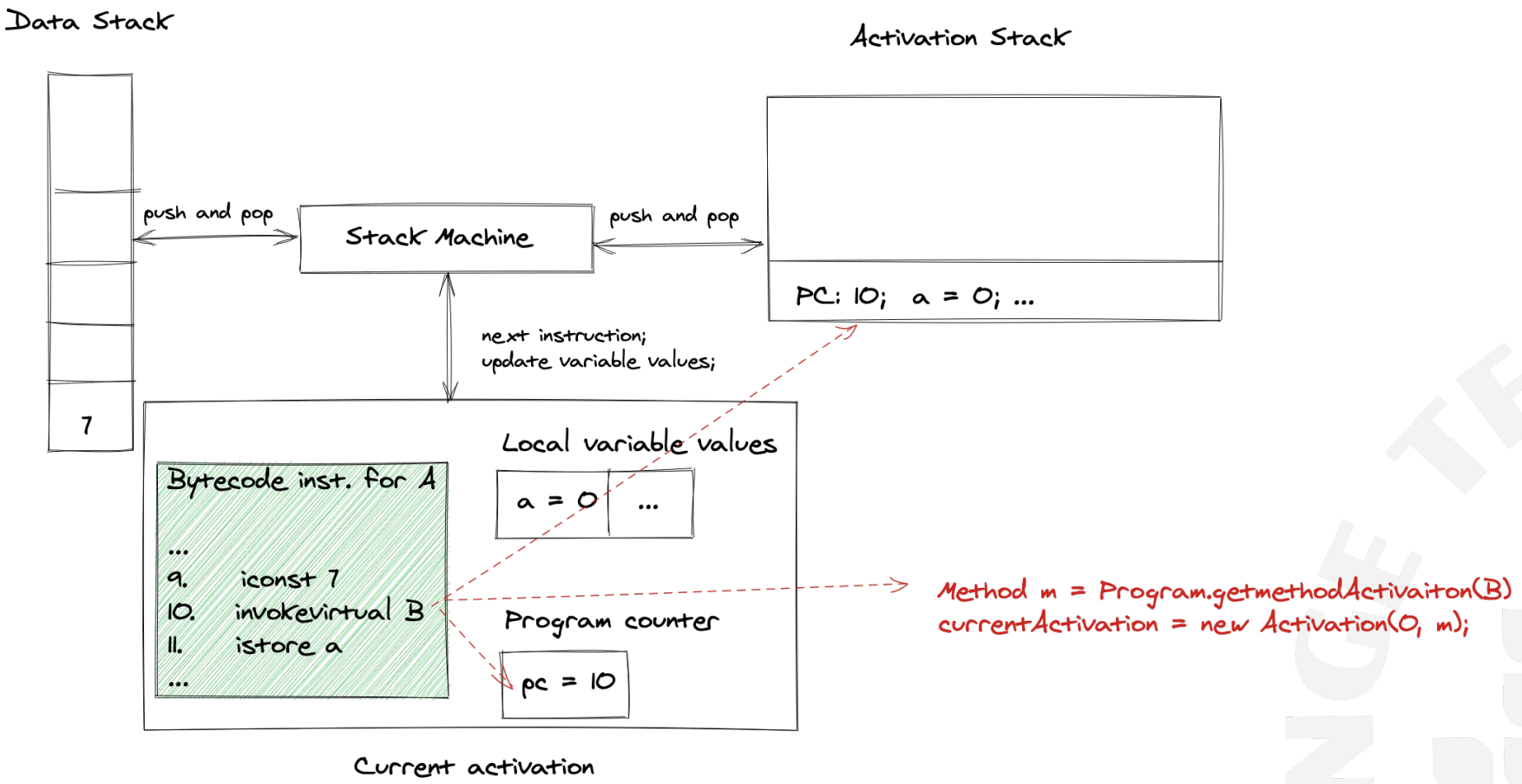
Interpreting method calls



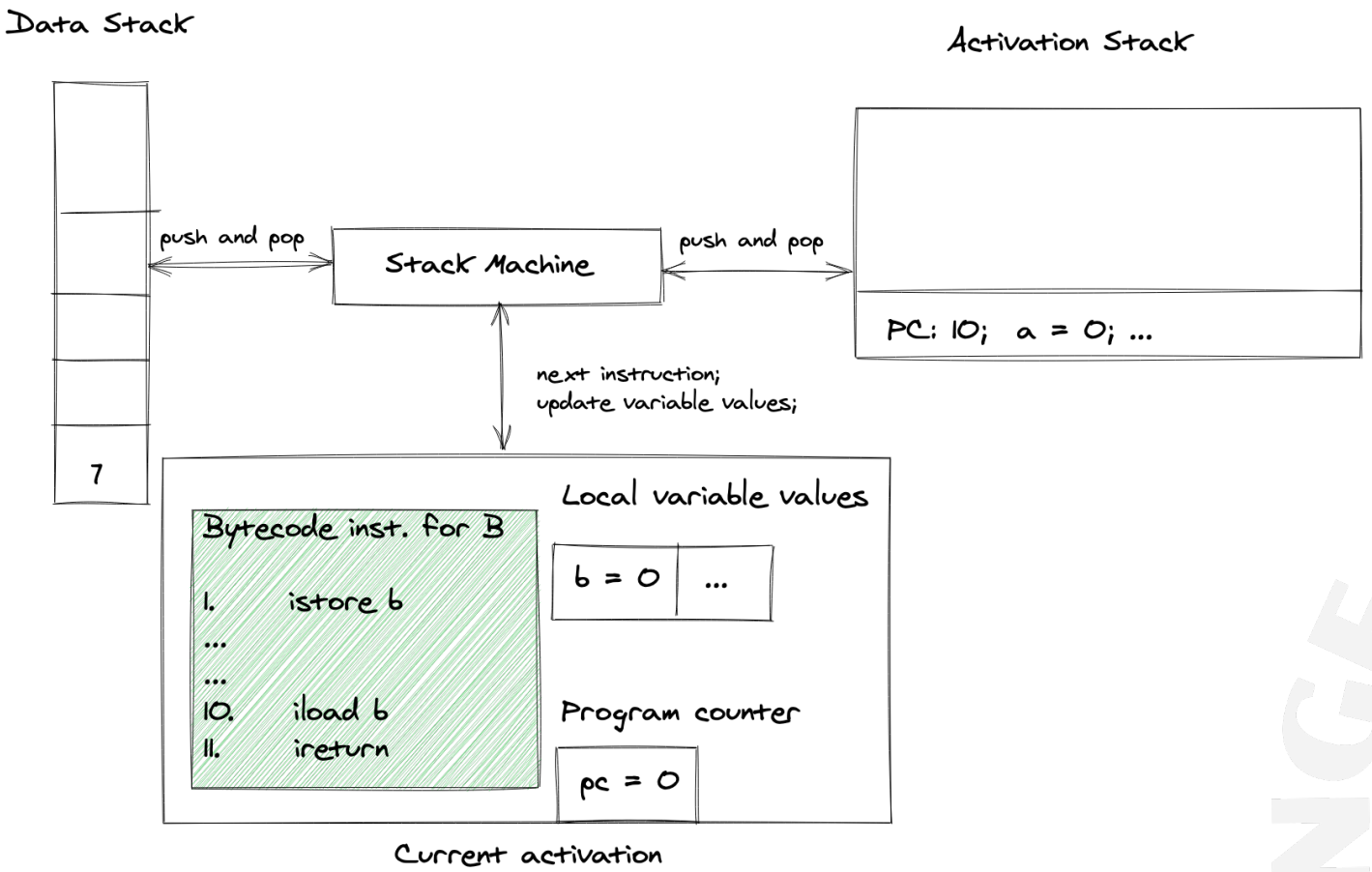
Interpreting method calls



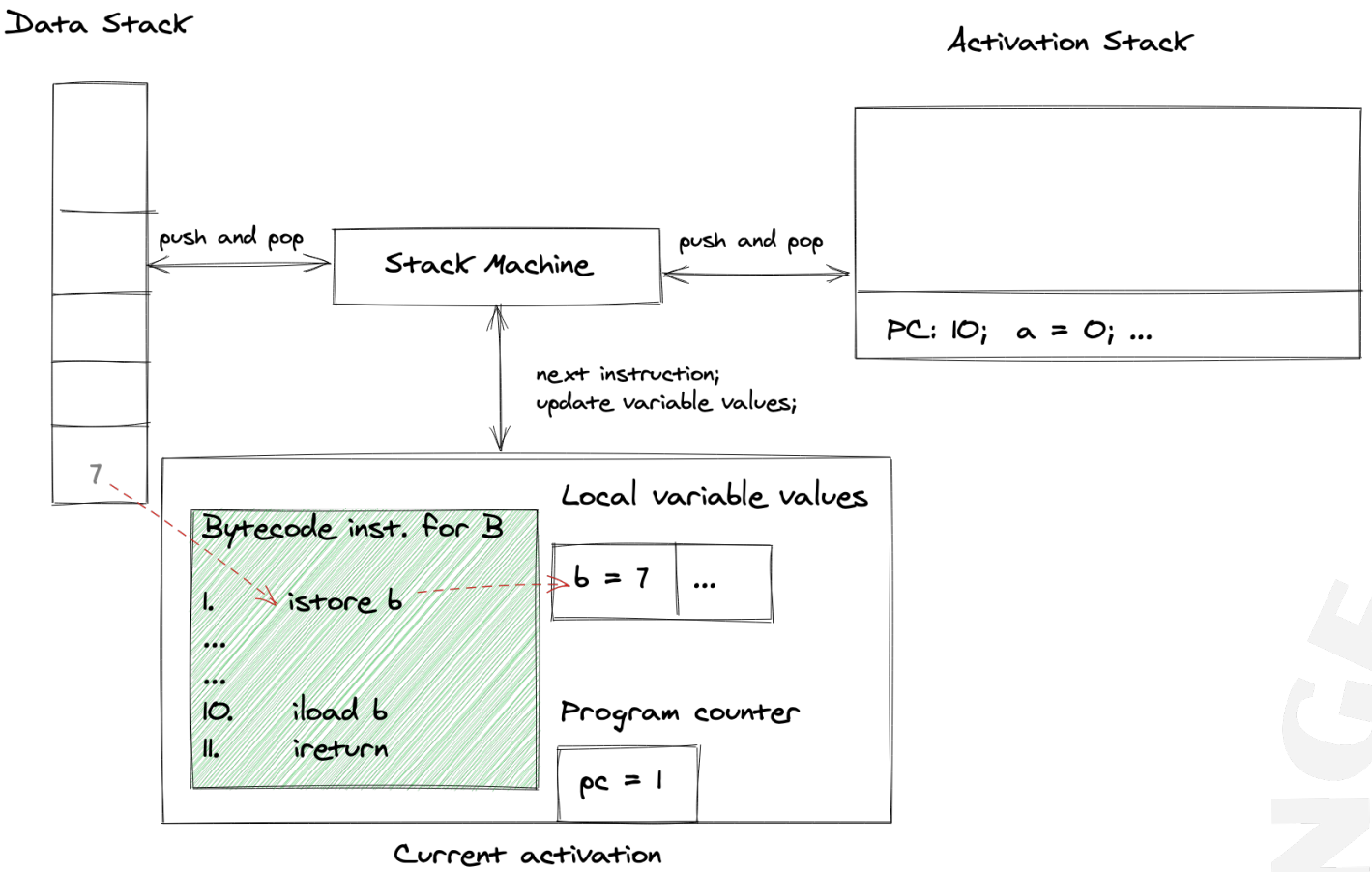
Interpreting method calls



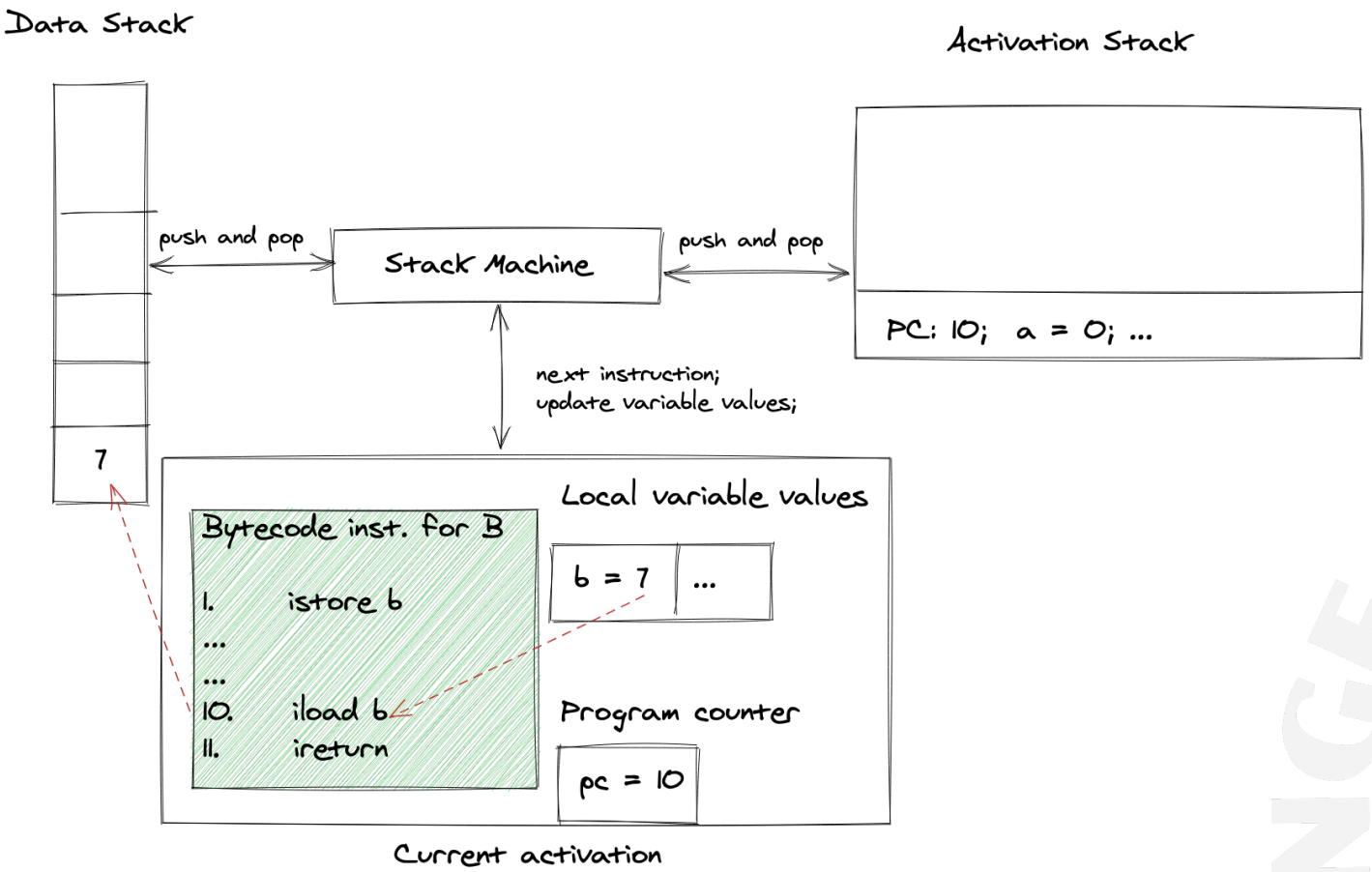
Interpreting method calls



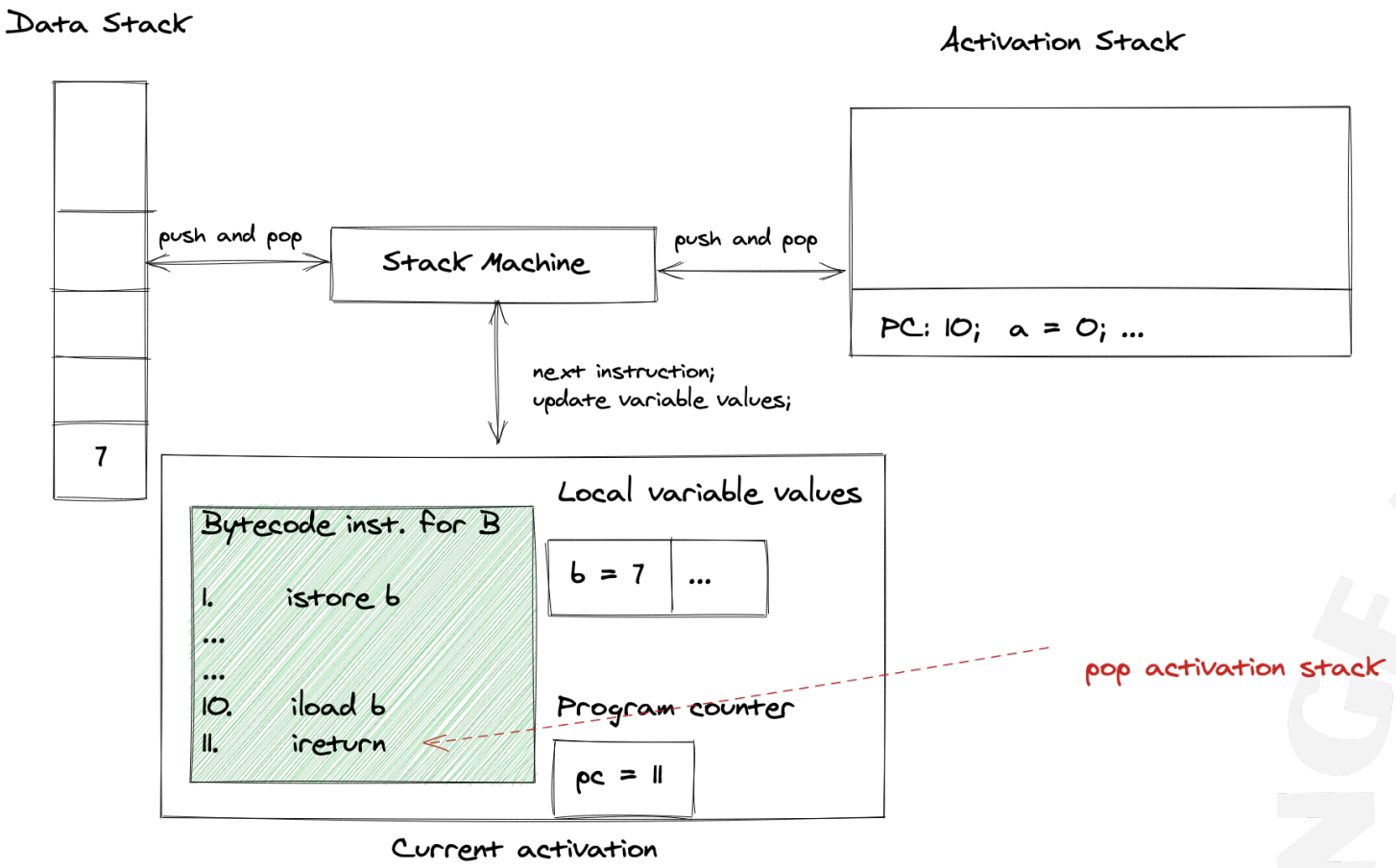
Interpreting method calls



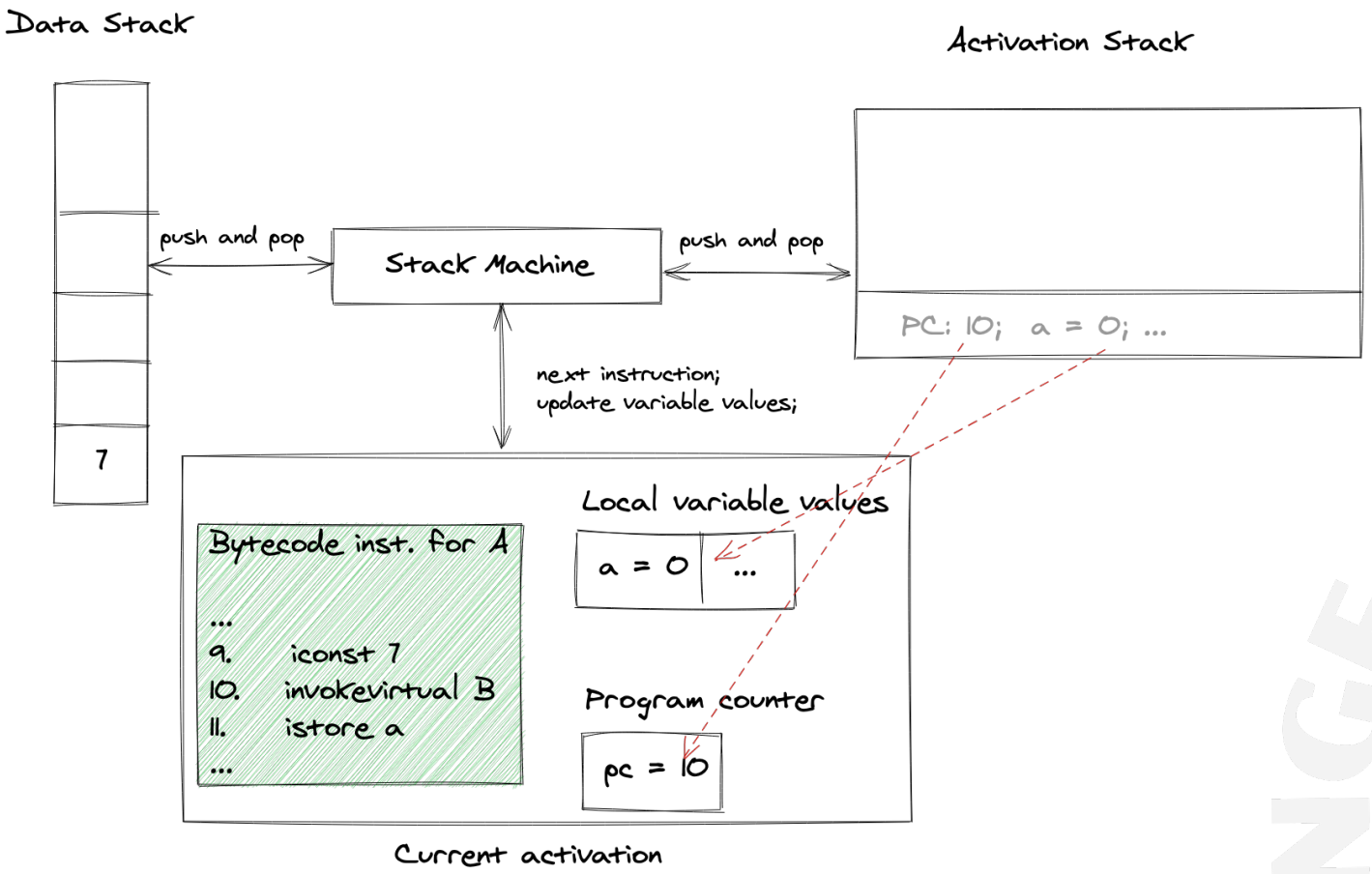
Interpreting method calls



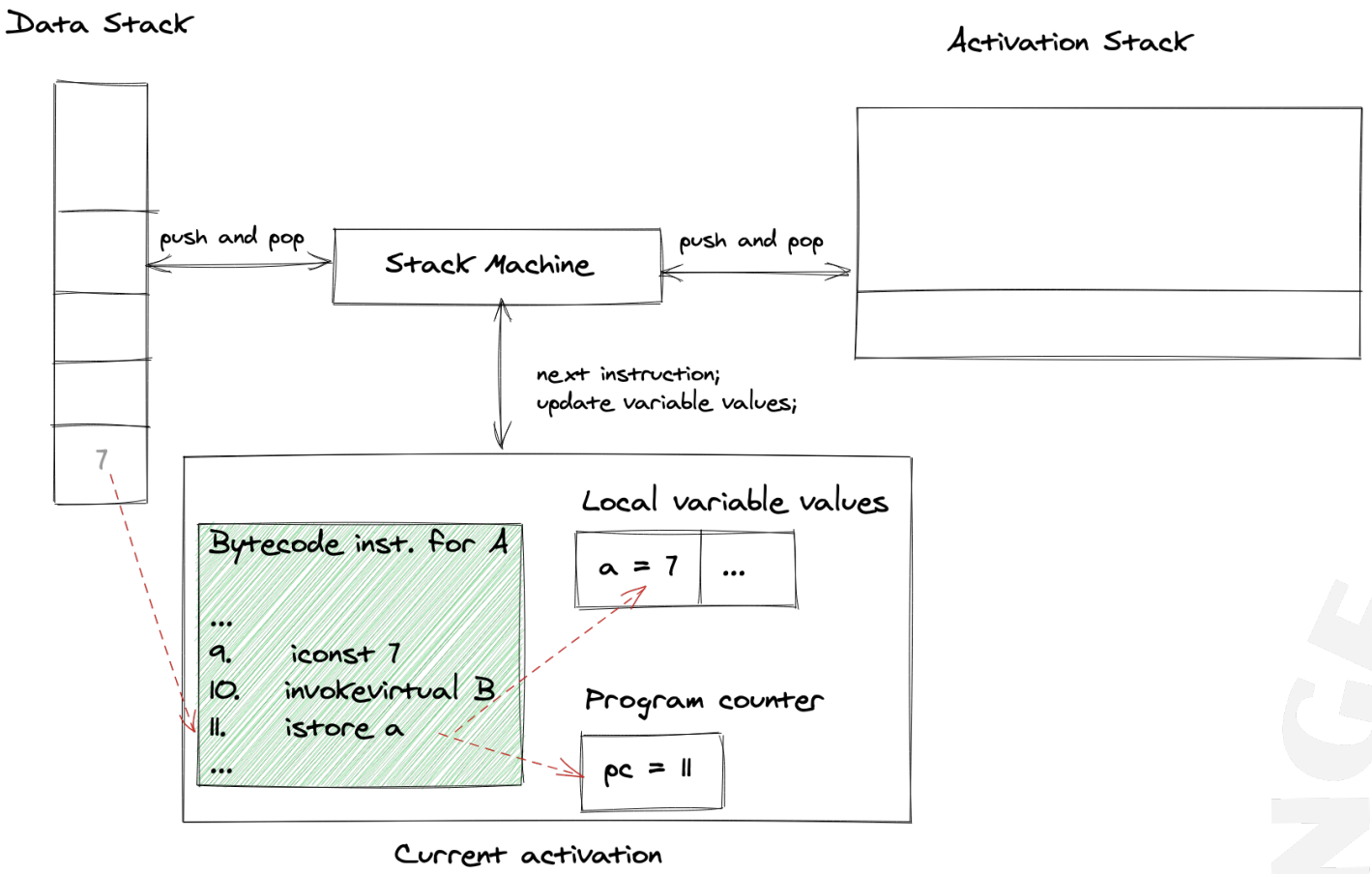
Interpreting method calls



Interpreting method calls



Interpreting method calls



Implementation details

- Two stacks
  - Data stack
  - Activation stack
- One switch statement
- See slide 9 for a reference for all instructions

```
class Activation {
  int pc; //program counter
  map local_variable_values;
  Method method;
...
}

class Interpreter{
  Program program;
  Method main;

  Interpreter(Program p) {
    program = p;
    main = program.getMethod(program.getMainMethod);
  }

  void execute() {
    Method m = main;
    Activation current_activation = new Activation(0, m);
    int instruction_id = -1;
    Stack<Activation> activations_stack = new Stack();
    ...

    while(instruction_id != STOP) {
      instruction = currentActgivation.getNextInstruction();
      switch(instruction.id) {
        case ICONST: { data_stack.push(instruction.argument); }
        case ISTORE: { currentActivation.storeValue(instruction.argument, data_stack.pop()); }
        case IADD: { data_stack.push(data_stack.popo() + data_stack.popo());}
        ...
      }
    }
  }
}
```

```
class Program {
  map methods;
  void print(){...}
}

class Method {
  list variables;
  list instructions;
  void print(){...}
}

class Instruction {
  int id;
  object argument;
  void print(){...}
}
```

Assignment 3

- Step 1
  - Generate Three Address Code by traversing the AST
- Step 2
  - Generate Java byte-code by traversing the CFG
- Step 3
  - Interpret the Java byte-code