Writing a Bytecode Compiler for Lisp in C++

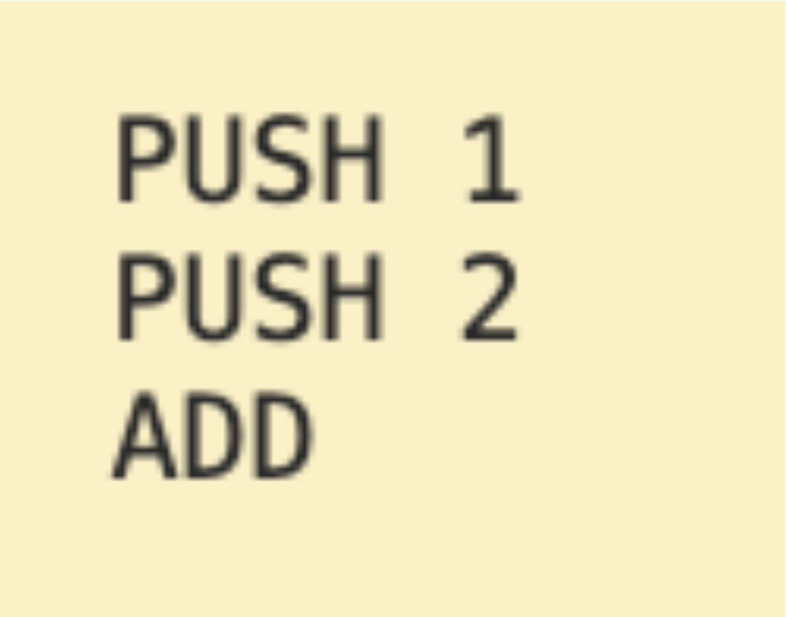
Anvay Grover

What is a Bytecode Compiler?

What is bytecode?

Linearized AST

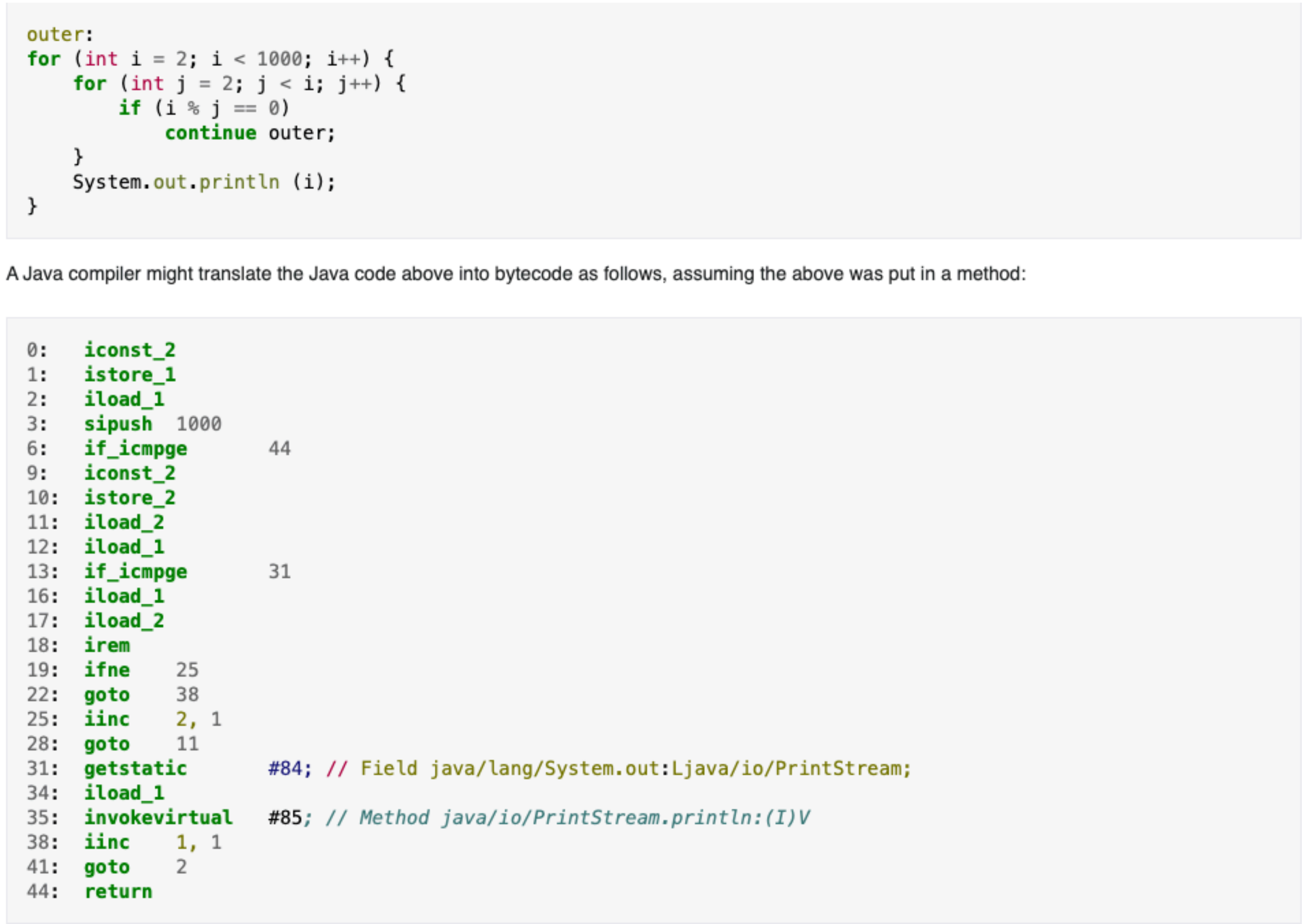
ADD(Lit 1, Lit 2)



```
PUSH 1
PUSH 2
ADD
```

Source —> Bytecode —> do something else (typically evaluate using VM)

What is a Bytecode Compiler?



```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

A Java compiler might translate the Java code above into bytecode as follows, assuming the above was put in a method:

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
18:  irem
19:  ifne    25
22:  goto    38
25:  iinc    2, 1
28:  goto    11
31:  getstatic       #84; // Field java/lang/System.out:Ljava/io/PrintStream;
34:  iload_1
35:  invokevirtual   #85; // Method java/io/PrintStream.println:(I)V
38:  iinc    1, 1
41:  goto    2
44:  return
```

Java Bytecode

compile once

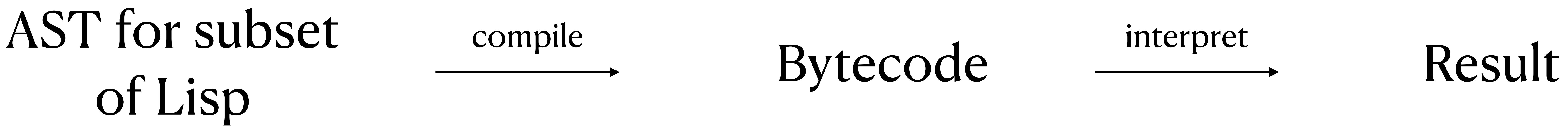
↓

execute on any platform
using the JVM

Why did I choose to write a bytecode compiler?

- To learn more about how bytecode compilation and interpretation works
- To develop greater proficiency with C++
- Adapted this Lisp compiler written in Python: https://bernsteinbear.com/blog/bytecode-interpreters/
    - Stack-based VM
    - CPython (the "Python definition") is also based on a stack machine

Current State of Affairs



AST for subset of Lisp

compile

Bytecode

interpret

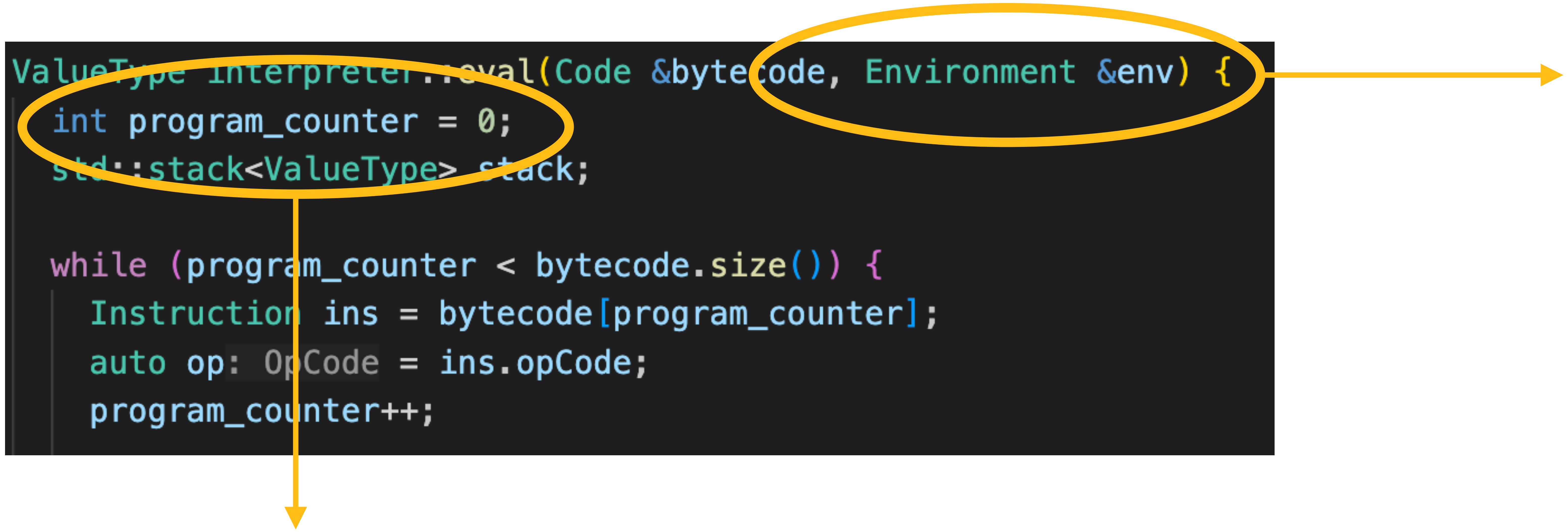
Result

- int and string values and variables
- basic arithmetic
- conditionals
- lambdas and function calls

- LOAD_CONST
- STORE_NAME
- LOAD_NAME
- CALL_FUNCTION
- RELATIVE_JUMP_IF_TRUE
- RELATIVE_JUMP
- MAKE_FUNCTION

Some interesting language constructs

How is bytecode evaluated?



```
ValueType Interpreter::eval(Code &bytecode, Environment &env) {
  int program_counter = 0;
  std::stack<ValueType> stack;

  while (program_counter < bytecode.size()) {
    Instruction ins = bytecode[program_counter];
    auto op: OpCode = ins.opCode;
    program_counter++;
```

Maps names to values

Tracks where in the bytecode we are

Some interesting language constructs

Conditionals

(if a b c)

We can manipulate the program counter!

```
[a BYTECODE]
RELATIVE_JUMP_IF_TRUE b
[c BYTECODE]
RELATIVE_JUMP end
b:
[b BYTECODE]
end:
```

Some interesting language constructs

Compiling Function Calls

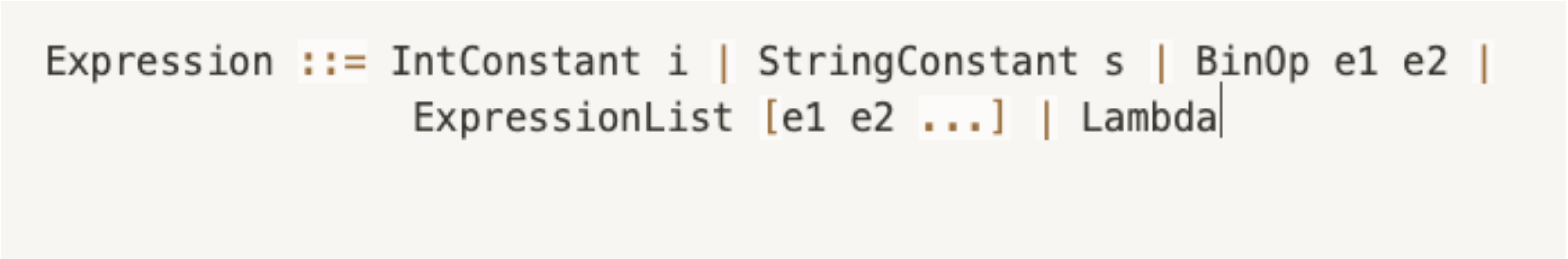
((lambda (x) (x + 1) 5)

How to eval:

1. Pop arguments off the stack
2. Pop function off the stack
3. Construct environment for function
4. Evaluate function in its own stack
5. Push result back on to current stack

Challenges?

The biggest challenge for me was figuring out the AST and recursive datatypes in C++



```
Expression ::= IntConstant i | StringConstant s | BinOp e1 e2 |
                ExpressionList [e1 e2 ...] | Lambda
```

I used multiple inheritance and virtual functions in C++ to define such a datatype

Would like to add more “structure” to the AST (let expressions, values)

What I plan to do (time permitting)

- Writing a frontend for the compiler (lexer, parser)
- The bytecode I'm compiling to is very close to Python bytecode (although I'm using an internal representation of it)
    - Could I compile to a Python VM and have it execute my bytecode?