Lecture 4: ANTLR and SVM

Programming Languages (H)

Simon Fowler & Michele Sevegnani

Semester 2, 2024/2025

sli.do 1238123

University of Glasgow

sli.do 1238123

Overview

- **Last lecture:**
  - Compilers vs. interpreters, big-step operational semantics, L_{If}
- **This lecture:** An introduction to some of the tools you'll be using in the rest of the course
  - The ANTLR parser generator
  - The SVM virtual machine

sli.do 1238123

Disclaimer

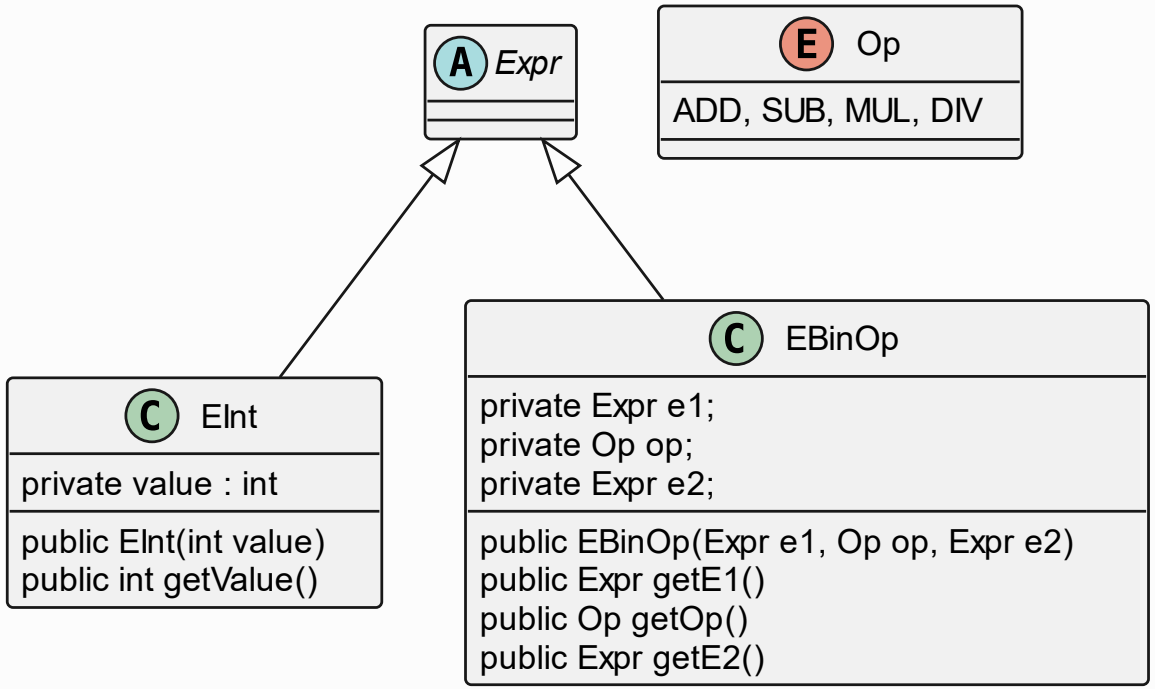
- I believe that functional programming languages (like Haskell and OCaml) are the best tools for implementing compilers and interpreters, because of their support for algebraic datatypes and focus on immutability
  - However – the functional programming course is in fourth year, so we're all stuck with Java ☺
  - I haven't written Java seriously since 2014, so bear with me…
- Nevertheless, I think ANTLR is a pretty good parser generator, and you can still get quite far with Java (even if parts are a bit more awkward to write)
- The examples will still be written in quite a functional style (less mutation), as that will be closer to the formal specifications

sli.do 1238123

Representing an AST in Java

```
Integers n
Operators ⊙    ::= + | - | * | /
Exprs. L, M, N ::= n
                 | L ⊙ M
```

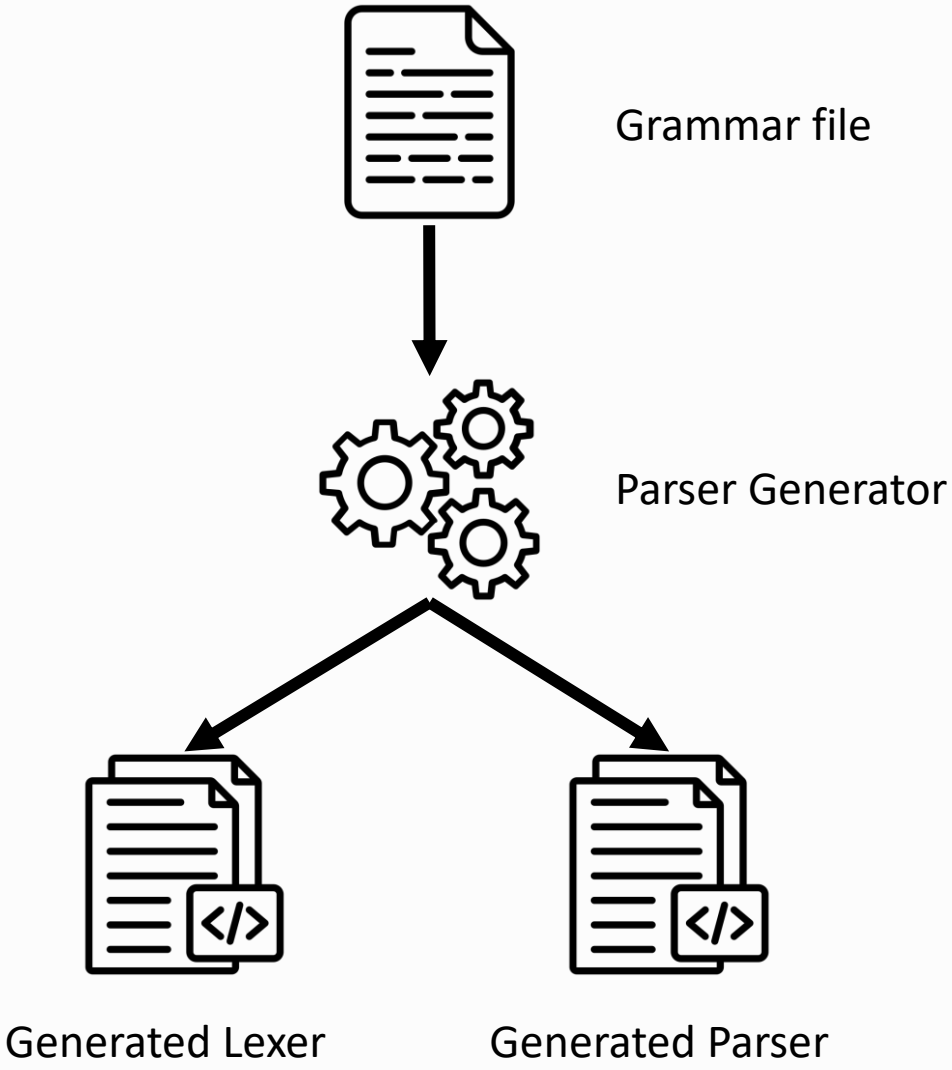
- We need an abstract class `Expr` to represent expressions
- We need separate classes for integer literals (`EInt`) and binary operators (`EBinOp`), which extend `Expr`
- `EInt` contains the integer value, and `EBinOp` contains the two subexpressions and an operator
- We can represent operators as an `enum`



ANTLR

sli.do 1238123

Parser Generators



- A **parser generator** is a program that takes a grammar and produces a lexer and parser
- This means that a language implementer can concentrate on the **design** of the grammar, rather than writing hand-crafted logic each time
- Many different parser generators exist for different languages (e.g. Yacc / Bison for C, Happy / Alex for Haskell, Menhir for OCaml)

sli.do 1238123

ANTLR

- ANTLR is an open-source parser generator for Java, originally written by Terence Parr, and widely used in industry
- The idea is that you can write a grammar using EBNF syntax, and it will generate a lexer, parser, and **visitor** code to traverse the parse tree
- We will use ANTLR in this course mainly for its parser generation component, as opposed to using visitors directly to implement interpretation / typechecking

sli.do 1238123

L_{Arith} ANTLR Grammar

```
expr: expr DIV expr # Div
    | expr MUL expr # Mul
    | expr ADD expr # Add
    | expr SUB expr # Sub
    | INT # Int
    | '(' expr ')' # Parens
    ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
MUL: '*' ;
DIV: '/' ;
SUB: '-' ;
ADD: '+' ;
```

- The ANTLR grammar for $_{LArith}$ is very close to its EBNF concrete syntax
- Operator precedence is **handled by ANTLR**: operator parses are tried in order (meaning **we don't need to restrict the grammar unnaturally**)
- Labels, e.g. `# Div`, are hints to ANTLR as to what to call the generated parse tree node
- It's generally good to define terminals separately, in capitals

Background: The Visitor Design Pattern

sli.do 1238123

Visitor Definition

```
public interface TreeVisitor {
  public void visitNode(Node n);
  public void visitLeaf(Leaf l);
}

public class BaseTreeVisitor
    implements TreeVisitor {
  public void visitNode(Node n) {
    n.getLeft().accept(this);
    n.getRight().accept(this);
  }
  public void visitLeaf(Leaf l) { }
}
```

Visitor Implementations

```
public class PrintLeafVisitor extends BaseTreeVisitor {
  @Override
  public void visitLeaf(Leaf l) {
    System.out.println(l.getValue());
  }
}

public class IncrementLeafVisitor extends BaseTreeVisitor {
  @Override
  public void visitLeaf(Leaf l) {
    l.setValue(l.getValue() + 1);
  }
}
```

- The **visitor pattern** separates the **operation** on each element of a structure from **the act of traversing it**.
- The code above shows two visitors: one that increments the leaves of a binary tree, the other that prints their values.
- We **do not** need to manually inspect the left and right subtrees, only writing the logic for the leaves. The traversal logic is contained in `BaseTreeVisitor`.
- Note that you **don't need to understand how to implement a visitor yourself**, just that it allows us to write the logic for each type of element independently.

sli.do 1238123

ANTLR Visitors

- ANTLR generates a base Visitor class from the supplied grammar.
- We can either use this to **create instances of our AST nodes** (and then write our own code to traverse them), or **work directly with the generated visitors** (without needing to write our own AST)
- The labs and assignment (and code in my lectures) will use a custom AST.
  - We will only use ANTLR visitors to create instances of our hand-written ASTs.
  - This is because hand-written ASTs have more specific structure and types, are easier to conceptualise, and writing explicit recursive functions is closer to the formal definitions.

sli.do 1238123

General pointers for using ANTLR Visitors

- Parameterise your visitor with the type you wish to return.
- Override the `visit` method for each generated expression, and implement the desired behaviour.
- The generated AST classes have accessor methods for subexpressions, based on the production names in the grammar. If there are multiple, you can supply the index.

sli.do 1238123

Generating an AST using an ANTLR Visitor

```
public class LArithASTGenerator
  extends LArithBaseVisitor<Expr> {

  @Override
  public Expr visitAdd(LArithParser.AddContext ctx) {
    Expr e1 = visit(ctx.expr(0));
    Expr e2 = visit(ctx.expr(1));
    return new EBinOp(e1, EBinOp.Op.ADD, e2);
  }

  @Override
  public Expr visitInt(LArithParser.IntContext ctx) {
    int val = Integer.valueOf(ctx.INT().getText());
    return new EInt(val);
  }
  …
}
```

We can **parameterise** the visitor with the type we want each method to return, in this case an Expr

Each rule in the grammar will result in a method in the visitor that we can override (in this case, Add)

We visit both subexpressions (accessed using the `ctx.expr()` method) to get two `Exprs`, and we can use these to construct a new `EBinOp` expression

Terminals are always represented as strings, so we need to use the `getText()` method and convert to an integer

sli.do 1238123

L_{Arith} starter code

- `antlr/`
  - `LArith.g4`: Larith ANTLR grammar definition
- `ast/`
  - `Expr.java`: Abstract Expression superclass
  - `EInt.java / EBinOp.java`: Expression AST nodes
  - `Value.java / VInt.java`: Value AST nodes
- `LArithASTGenerator.java`: ANTLR visitor to generate AST nodes
- `LArithInterp.java`: Interpreter
- `Main.java`: Entrypoint to read file, then call parser and interpreter

- This week's lab has two parts:
  - The first is exploring the (completed) interpreter for L_{Arith}
  - The second is extending this code to implement the additional features from L_{if}
- There are also scripts to help you compile and run, and some example programs

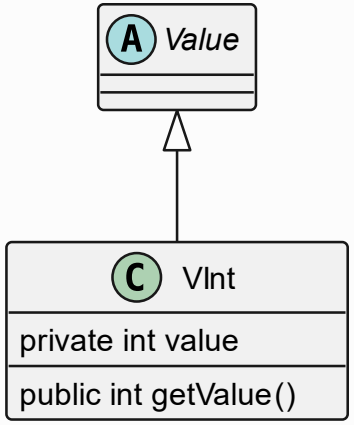
sli.do 1238123

Implementing an Interpreter (Overview)

- Remember the form of our operational semantics judgement, which says that an expression M will evaluate to a value V.

$$M \Downarrow V$$

- It helps to write a class hierarchy representing Values (in L_{Arith}, this will only contain `VInt`). Values are **only introduced at runtime and are never parsed in**.



- Our interpreter is therefore a function from an Expr to a Value.

```
public Value interpExpr(Expr e) { … }
```

sli.do 1238123

Implementing an Interpreter (Values)

- To write an interpreter, we need different cases based on each rule in the operational semantics
- The value rule just states that a number is already a value, so we just need to create a `VInt` with the same value.

Note: e is of type Expr. We need to do **instance tests** to check which kind of expression e is, and then **cast** it to an EInt

$$\frac{}{n \Downarrow n}$$

```
if (e instanceof EInt) {
  EInt exprInt = (EInt) e;
  return new VInt(exprInt.getValue());
}
```

sli.do 1238123

Implementing an Interpreter (Binary Operators)

$$\frac{L \Downarrow V \qquad M \Downarrow W}{L \odot M \Downarrow V \mathbin{\widehat{\odot}} W}$$

Key idea: recursively evaluate subexpressions, switch on the operator, and apply the concrete operation

Note: `unwrapInt` is a function that extracts an integer from a Value

```
  …
} else if (e instanceof EBinOp) {
  EBinOp exprBinOp = (EBinOp) e;
  Value v1 = interpExpr(exprBinOp.getE1());
  Value v2 = interpExpr(exprBinOp.getE2());
  int i1 = unwrapInt(v1);
  int i2 = unwrapInt(v2);
  switch (exprBinOp.getOp()) {
    case ADD: return new VInt(i1 + i2);
    case SUB: return new VInt(i1 - i2);
    case MUL: return new VInt(i1 * i2);
    case DIV: return new VInt(i1 / i2);
    default:
      throw new RuntimeException("Invalid op");
}
```

SVM

sli.do 1238123

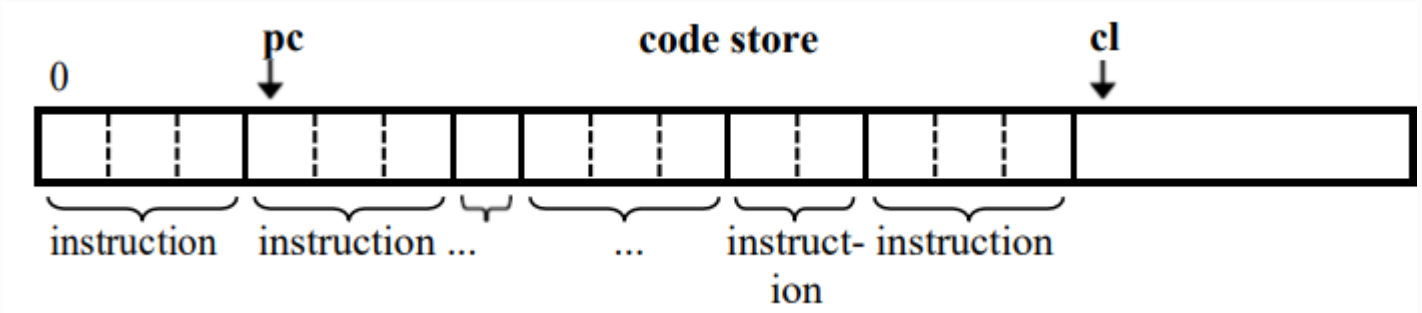
SVM Overview

- Last lecture, we touched on **virtual machines**: instruction sets + virtual architecture for running programs in a platform-independent way.
- SVM is a simple(ish) VM used for teaching, and suitable for implementing simple imperative languages. It'll be used in Michele's part of the course.
- Much like both the JVM and WebAssembly, SVM is **stack-based**: instructions use an operand stack rather than having explicit parameters.

sli.do 1238123

Machine State (Code Store)

The first part of the SVM state is a **code store** containing the machine's bytecode.



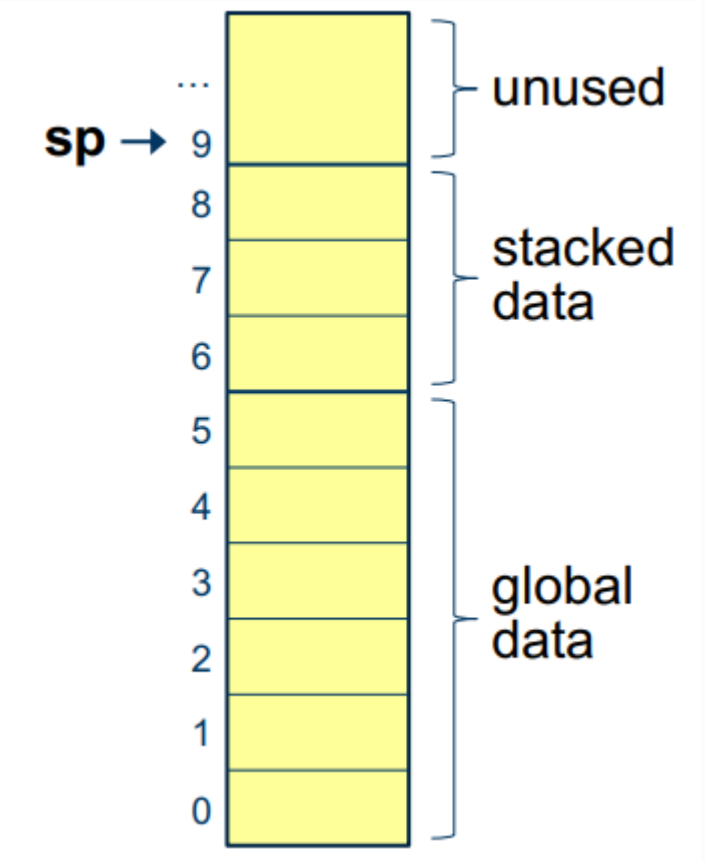
There are two registers associated with the code store:

- A **program counter** (pc) that tracks the current instruction
- A **code limit** (cl) that marks the end of the code store

sli.do 1238123

Machine State (Data Store)

- The second part of the SVM state is a **data store** containing the program's data (simplified diagram on the right).
- Roughly speaking, there is a **stack pointer** (sp) that denotes the top of the stack, some data that occurs on the stack, and then global data.
- The full definition includes additional information to remember return addresses, etc.
- Finally, the machine contains a **status** register that indicates whether the program is running, halted, or failed.



sli.do 1238123

Simplified SVM Instruction Set (1)

Opcode	Mnemonic	Description
6	ADD	pop w_2; pop w_1; push (w_1 + w_2)
7	SUB	pop w_2; pop w_1; push (w_1 – w_2)
8	MUL	pop w_2; pop w_1; push (w_1 × w_2)
9	DIV	pop w_2; pop w_1; push (w_1 / w_2)
10	CMPEQ	pop w_2; pop w_1; push (if w_1 = w_2 then 1 else 0)
11	CMPLT	pop w_2; pop w_1; push (if w_1 < w_2 then 1 else 0)
14	INV	pop w; push (if w = 0 then 1 else 0)

sli.do 1238123

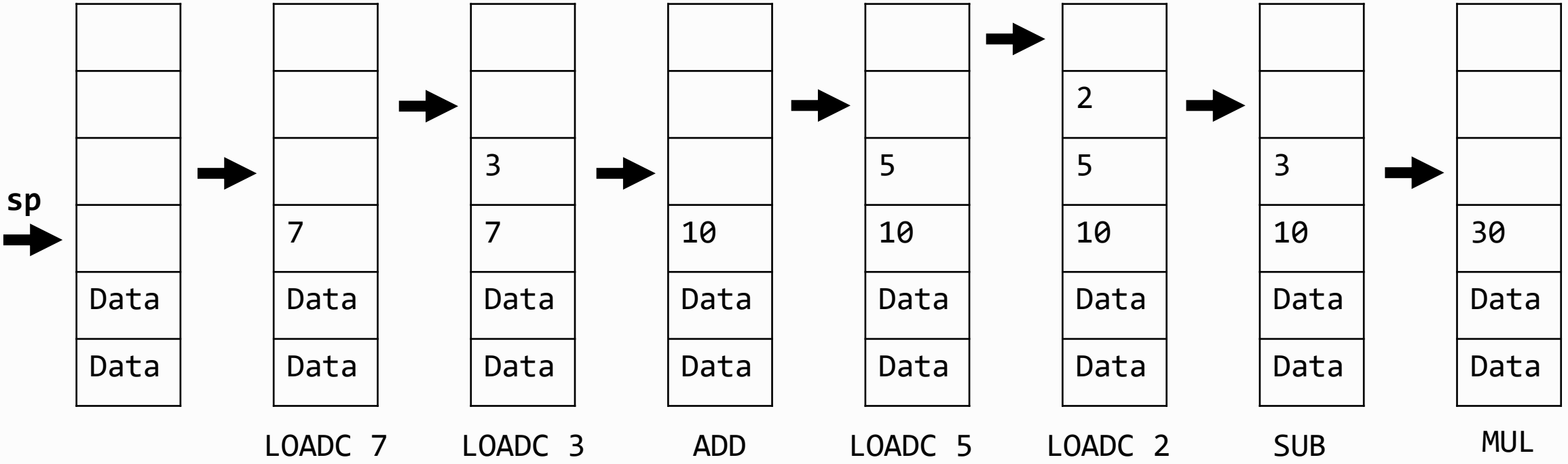
Simplified SVM Instruction Set (2)

Opcode	Mnemonic	Description
0	LOADG d	w <- word at address d; push w
1	STOREG d	pop w; word at address d <- w
4	LOADC v	push v
…		
16	HALT	status <- halted
17	JUMP c	pc <- c
18	JUMPF c	pop w; if w = 0 then pc <- c
19	JUMPT c	pop w; if w =/= 0 then pc <- c

sli.do 1238123

Example SVM Evaluation

Suppose we want to evaluate `(7 + 3) * (5 - 2)` in SVM



sli.do 1238123

SVM Implementation

```
public void interpret () {
  // Initialise state
  status = RUNNING;
  sp = 0; fp = 0; pc = 0;
  do {
    // Fetch the next instruction:
    byte opcode = code[pc++];
    // Inspect opcode and execute:
    switch (opcode) {
      case ADD:
        int w2 = data[--sp];
        int w1 = data[--sp];
        data[sp++] = w1 + w2;
        break;
      …
    }
  } while (status == RUNNING);
}
```

- There's a full specification of SVM on Moodle, along with an interpreter – take a look if you're interested!
- Virtual machines don't have to be complicated. It's essentially one big loop that continually fetches and executes each instruction until the machine is halted.

sli.do 1238123

Conclusion

- In this lecture we've seen:
  - How to represent an AST using Java classes
  - How to use ANTLR to generate a lexer, parser, and visitor code
  - How to write an interpreter from an operational semantics
  - An overview of the SVM virtual machine
- Next time, we'll be looking at variables and binding, and first-class functions: important lecture!
- Please come to the labs to get some hands-on PL implementation experience with help from myself and our wonderful GTA team