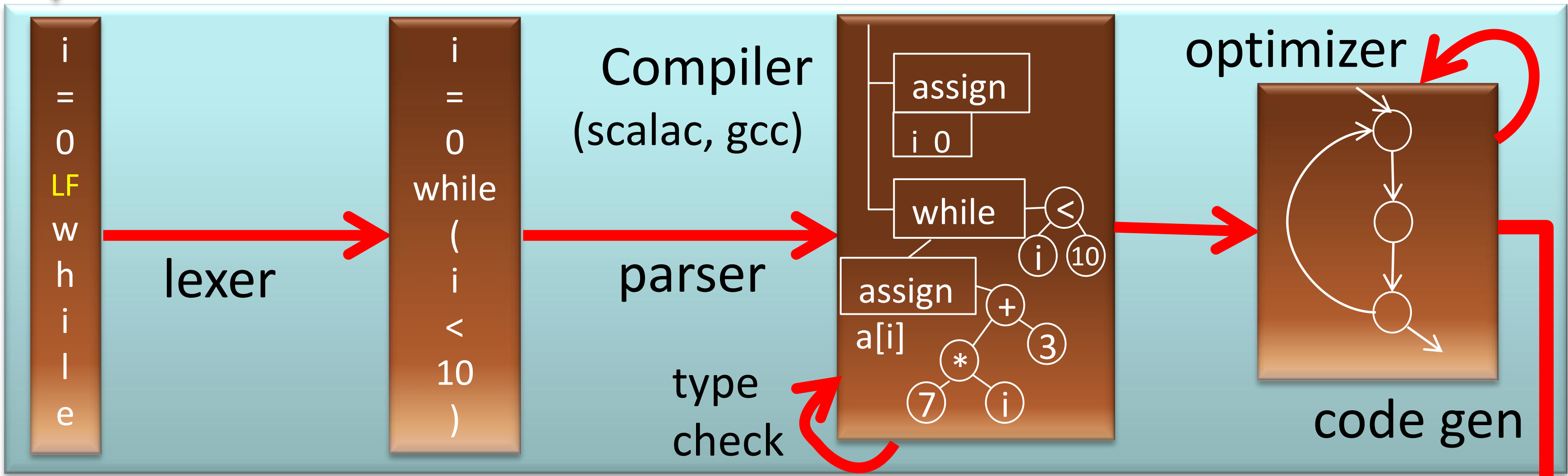
Code Generation Introduction

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java,C)
easy to write

data-flow
graphs



characters

words

trees

```
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
```

machine code
(e.g. x86, arm, JVM)
efficient to execute

Example: gcc

```
#include <stdio.h>
int main() {
  int i = 0;
  int j = 0;
  while (i < 10) {
    printf("%d\n", j);
    i = i + 1;
    j = j + 2*i+1;
  }
}
```

where is it?

gcc test.c -S

```
        jmp .L2
.L3:    movl -8(%ebp), %eax
        movl %eax, 4(%esp)
        movl $.LC0, (%esp)
        call printf
        addl $1, -12(%ebp)
        movl -12(%ebp), %eax
        addl %eax, %eax
        addl -8(%ebp), %eax
        addl $1, %eax
        movl %eax, -8(%ebp)
.L2:
        cmpl $9, -12(%ebp)
        jle .L3
```

what does this do:

gcc -O3 -S test.c

Amusing Question

What does this produce (with GCC 4.2.4)

gcc -O3 test.c

Loop was unrolled, giving a sequence corresponding to

```
printf(“%d\n”, 0)
printf(“%d\n”, 3)
printf(“%d\n”, 8)
printf(“%d\n”, 15)
printf(“%d\n”, 24)
printf(“%d\n”, 35)
printf(“%d\n”, 48)
printf(“%d\n”, 63)
printf(“%d\n”, 80)
printf(“%d\n”, 99)
```

LLVM: Another Interesting Compiler

The **LLVM** Compiler Infrastru

LLVM Overview

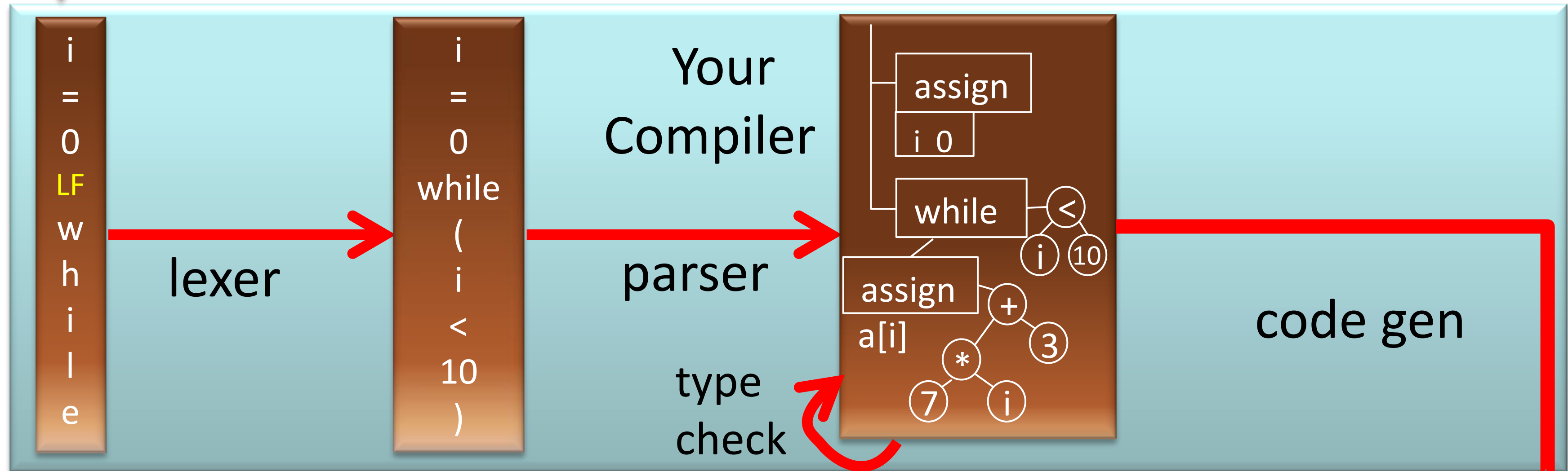
The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them.

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of different subprojects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-Style license.

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
simplified Java-like language

Your Project



characters words trees

```
21: iload_2
22: iconst_2
23: iload_1
24: imul
25: iadd
26: iconst_1
27: iadd
28: istore_2
```

Java Virtual Machine (JVM) Bytecode

javac example

```
while (i < 10) {
  System.out.println(j);
  i = i + 1;
  j = j + 2*i+1;
}
```

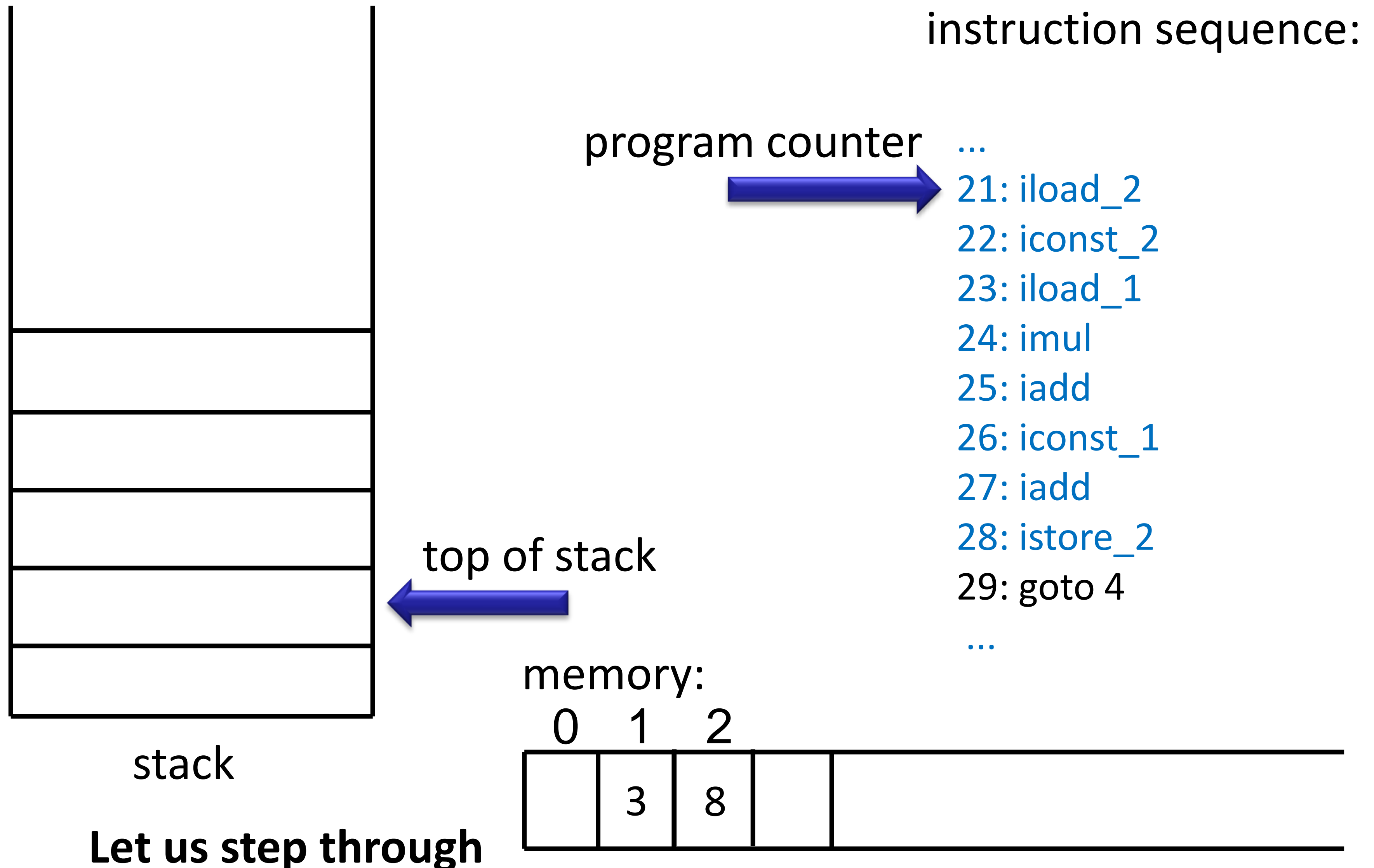
javac Test.java
javap –c Test

```
4: iload_1
5: bipush 10
7: if_icmpge 32
10: getstatic #2; //System.out
13: iload_2
14: invokevirtual #3; //println
17: iload_1
18: iconst_1
19: iadd
20: istore_1
21: iload_2
22: iconst_2
23: iload_1
24: imul
25: iadd
26: iconst_1
27: iadd
28: istore_2
29: goto 4
32: return
```

Next phase: emit such bytecode instructions

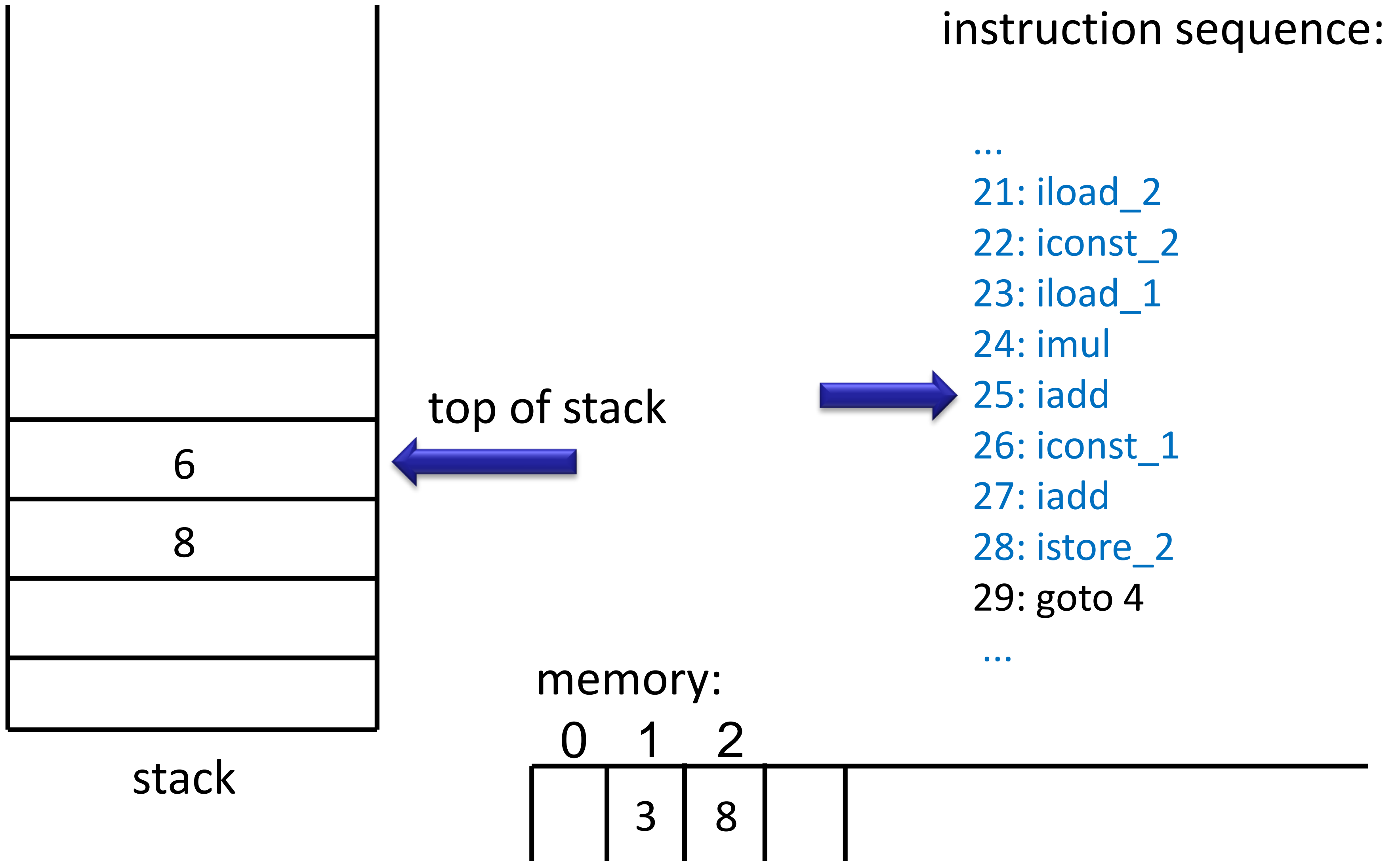
Guess what each JVM instruction for the highlighted expression does.

Stack Machine: High-Level Machine Code

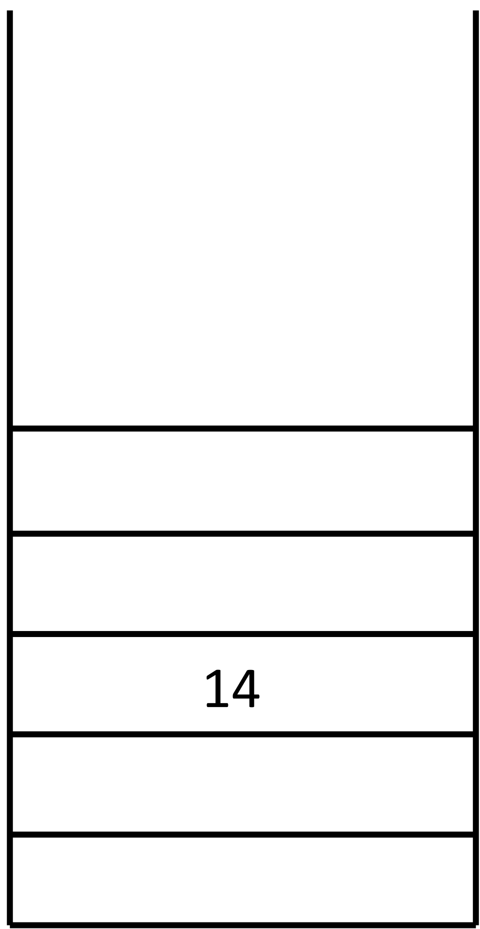


Let us step through

Operands are consumed from stack and put back onto stack



Operands are consumed from stack and put back onto stack



top of stack

instruction sequence:

...
21: iload_2
22: iconst_2
23: iload_1
24: imul
25: iadd

26: iconst_1
27: iadd
28: istore_2
29: goto 4
...

memory:

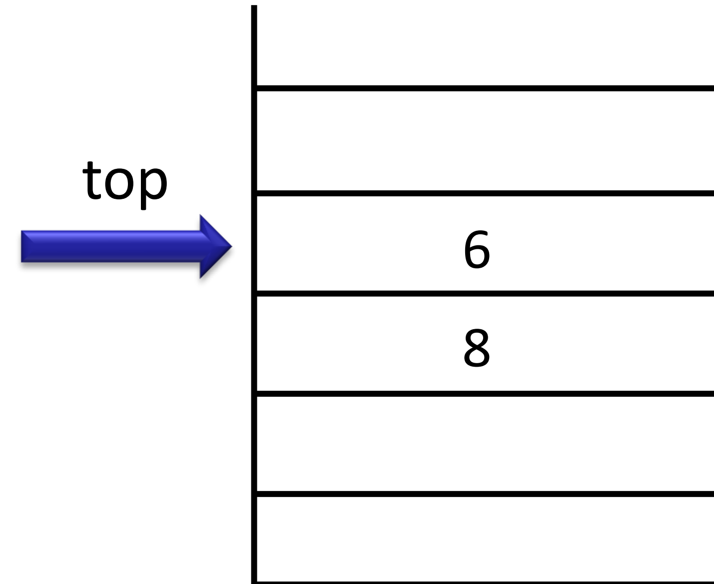
0	1	2		
	3	8		

Instructions in JVM

- Separate for each type, including
  - integer types (iadd, imul, iload, istore, bipush)
  - reference types (aload, astore)
- Why are they separate?
  - Memory safety!
  - Each reference points to a valid allocated object
- Conditionals and jumps
- Further high-level operations
  - array operations
  - object method and field access

Stack Machine Simulator

```
var code : Array[Instruction]
var pc : Int   // program counter
var local : Array[Int]  // for local variables
var operand : Array[Int]  // operand stack
var top : Int
```




```
while (true) step

 def step = code(pc) match {
  case Iadd() =>
   operand(top - 1) = operand(top - 1) +   operand(top)
   top = top - 1  // two consumed, one produced
  case Imul() =>
   operand(top - 1) = operand(top - 1) * operand(top)
   top = top - 1  // two consumed, one produced
```

Stack Machine Simulator: Moving Data

```
case Bipush(c) =>
  operand(top + 1) = c   // put given constant 'c' onto stack
  top = top + 1
case Iload(n) =>
  operand(top + 1) = local(n)  // from memory onto stack
  top = top + 1
case Istore(n) =>
  local(n) = operand(top)  // from stack into memory
  top = top - 1  // consumed
}
if (notJump(code(n)))
  pc = pc + 1   // by default go to next instructions
```

Actual Java Virtual Machine

JVM Instruction Description from JavaTech book

Official documentation:
http://docs.oracle.com/javase/specs/

http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

Use: javac -g *.java   to compile

javap -c -l ClassName   to explore

Example: Twice

```
class Expr1 {
   public static int twice(int x) {
         return x*2;
   }
}
```

javac -g Expr1.java; javap -c -l Expr1

```
public static int twice(int);
 Code:
  0:   iload_0    // load int from var 0 to top of stack
  1:   iconst_2   // push 2 on top of stack
  2:   imul       // replace two topmost elements with their product
  3:   ireturn    // return top of stack
}
```

Example: Area

```
class Expr2 {
   public static int cubeArea(int a, int b, int c) {
          return (a*b + b*c + a*c) * 2;
   }
}
```

javac -g Expr2.java; javap -c -l Expr2

```
LocalVariableTable:
 Start   Length   Slot   Name    Signature
 0       14       0      a        I
 0       14       1      b        I
 0       14       2      c        I
```

```
public static int cubeArea(int, int, int);
 Code:
  0:   iload_0
  1:   iload_1
  2:   imul
  3:   iload_1
  4:   iload_2
  5:   imul
  6:   iadd
  7:   iload_0
  8:   iload_2
  9:   imul
  10:  iadd
  11:  iconst_2
  12:  imul
  13:  ireturn
}
```

What Instructions Operate on

- operands that are part of instruction itself, following their op code
(unique number for instruction - iconst)
- operand stack - used for computation (iadd)
- memory managed by the garbage collector (loading and storing fields)
- constant pool - used to store ‘big’ values instead of in instruction stream
  - e.g. string constants, method and field names
  - mess!

CAFEBABE

Library to make bytecode generation easy and fun!

Named after magic code appearing in .class files when displayed in hexadecimal:

```
0000000  ca fe ba be 00 00 00 32 00 3b 0a 00 12 00 1e 07
0000020  00 1f 0a 00 02 00 1e 0a 00 02 00 20 08 00 21 08
0000040  00 22 09 00 23 00 24 07 00 25 0a 00 08 00 1e 08
0000060  00 26 0a 00 08 00 27 08 00 28 08 00 29 0a 00 02
0000100  00 2a 0a 00 08 00 2b 0a 00 08 00 2c 0a 00 2d 00
```

More on that in the labs!