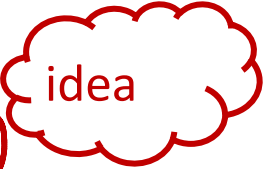


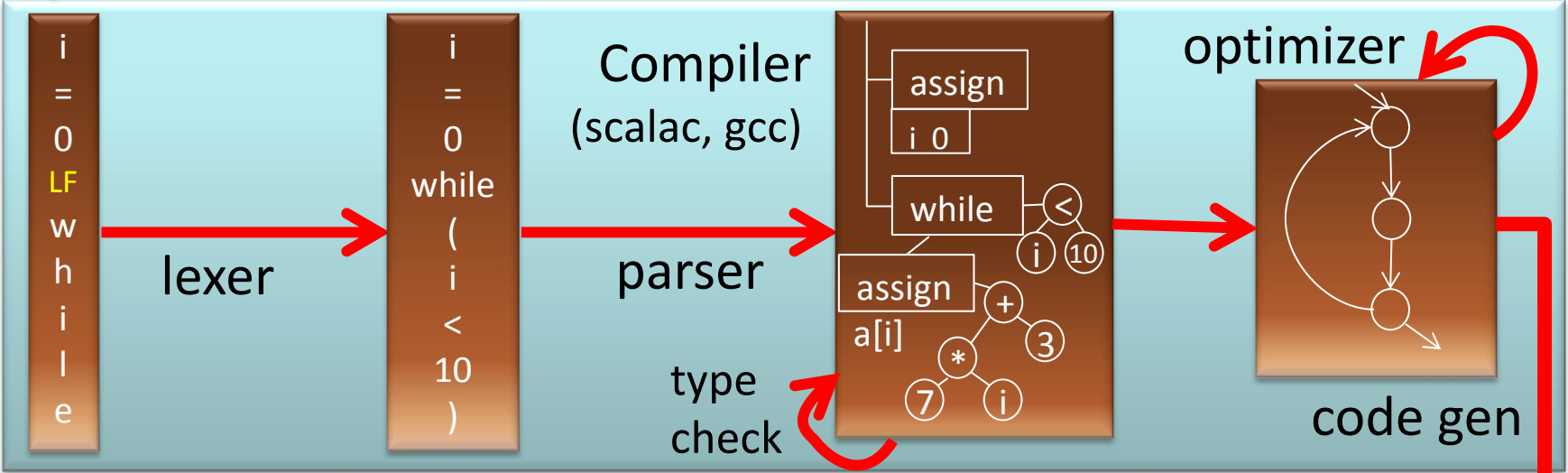
# Code Generation Introduction

```
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

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

```
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
```



# 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
```

# LLVM: Another Interesting C Compiler

## The LLVM Compiler Infrastructure

---

### LLVM Overview

---

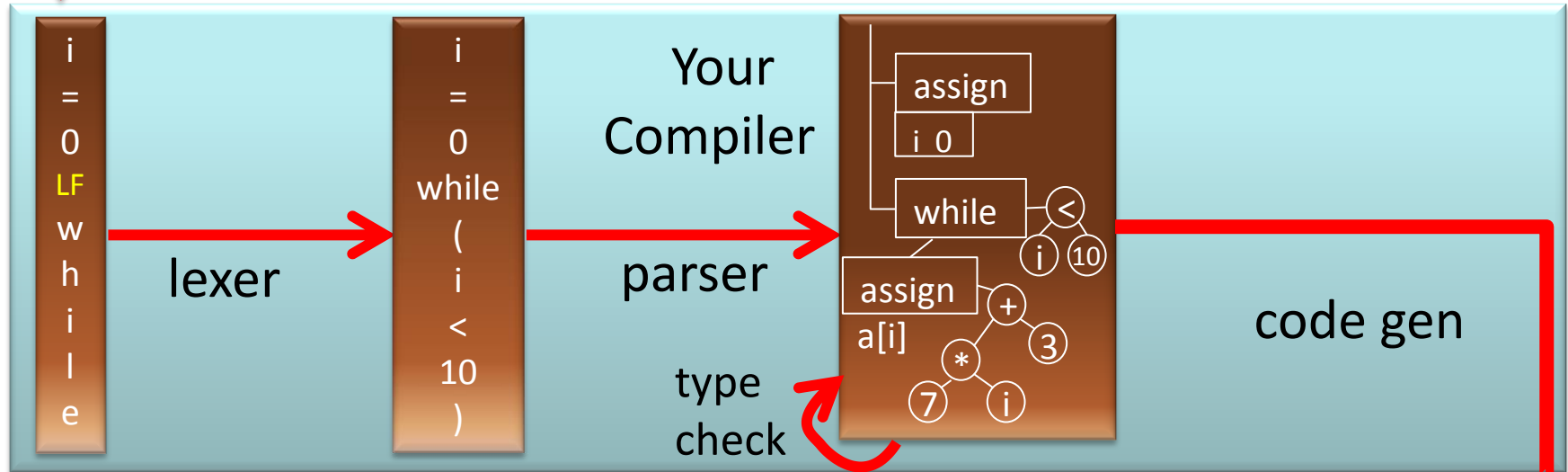
The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be [used to build them](#).

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](#).

# Your Project

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

source code  
simplified Java-like  
language



characters

words

trees

## Java Virtual Machine (JVM) Bytecode

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

# javac example

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

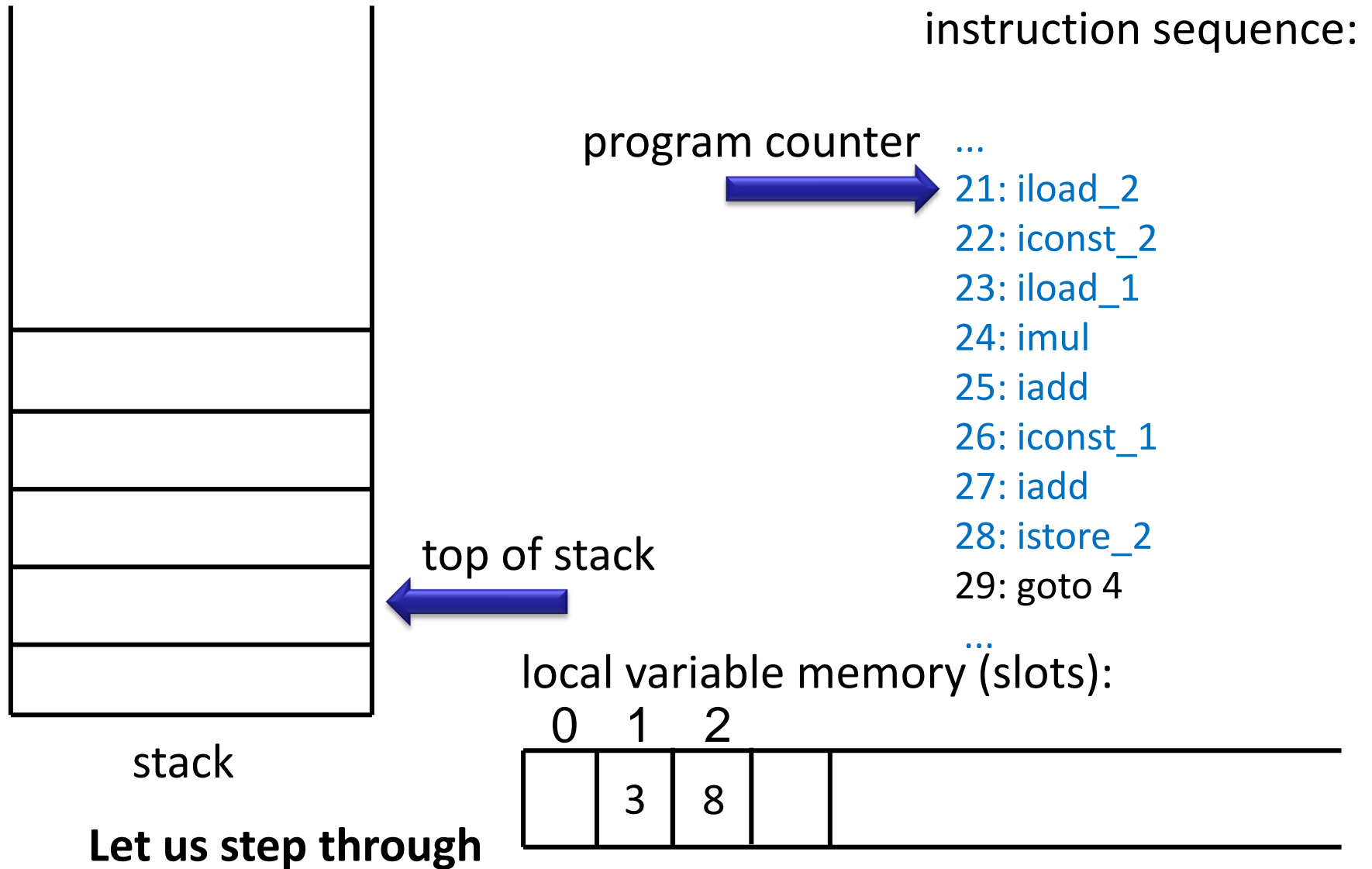
```
javac Test.java  
javap -c Test
```

```
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
```

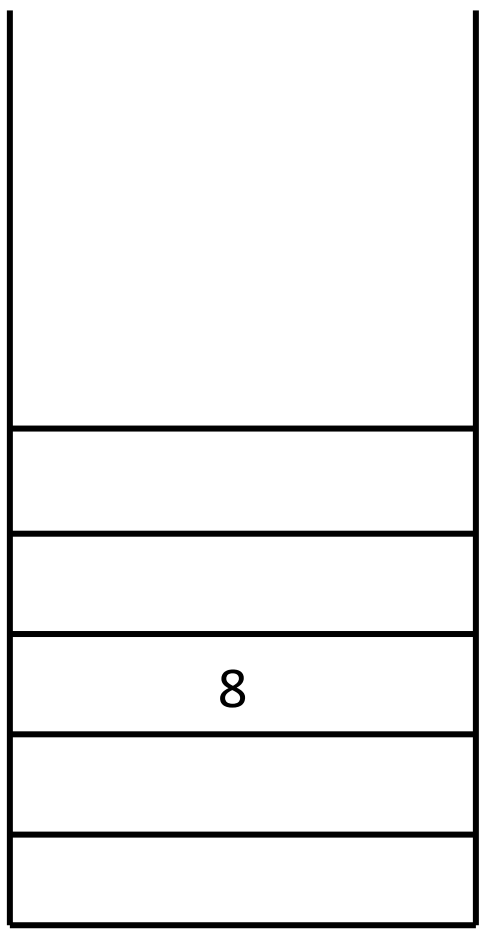
Phase after  
type checking:  
emits such  
bytecode  
instructions

Guess what each JVM instruction for  
the highlighted expression does.

# Stack Machine: High-Level Machine Code



# Operands are consumed from stack and put back onto stack



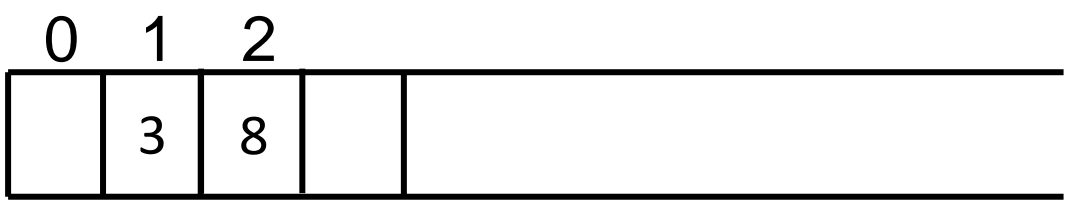
stack

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
- ...

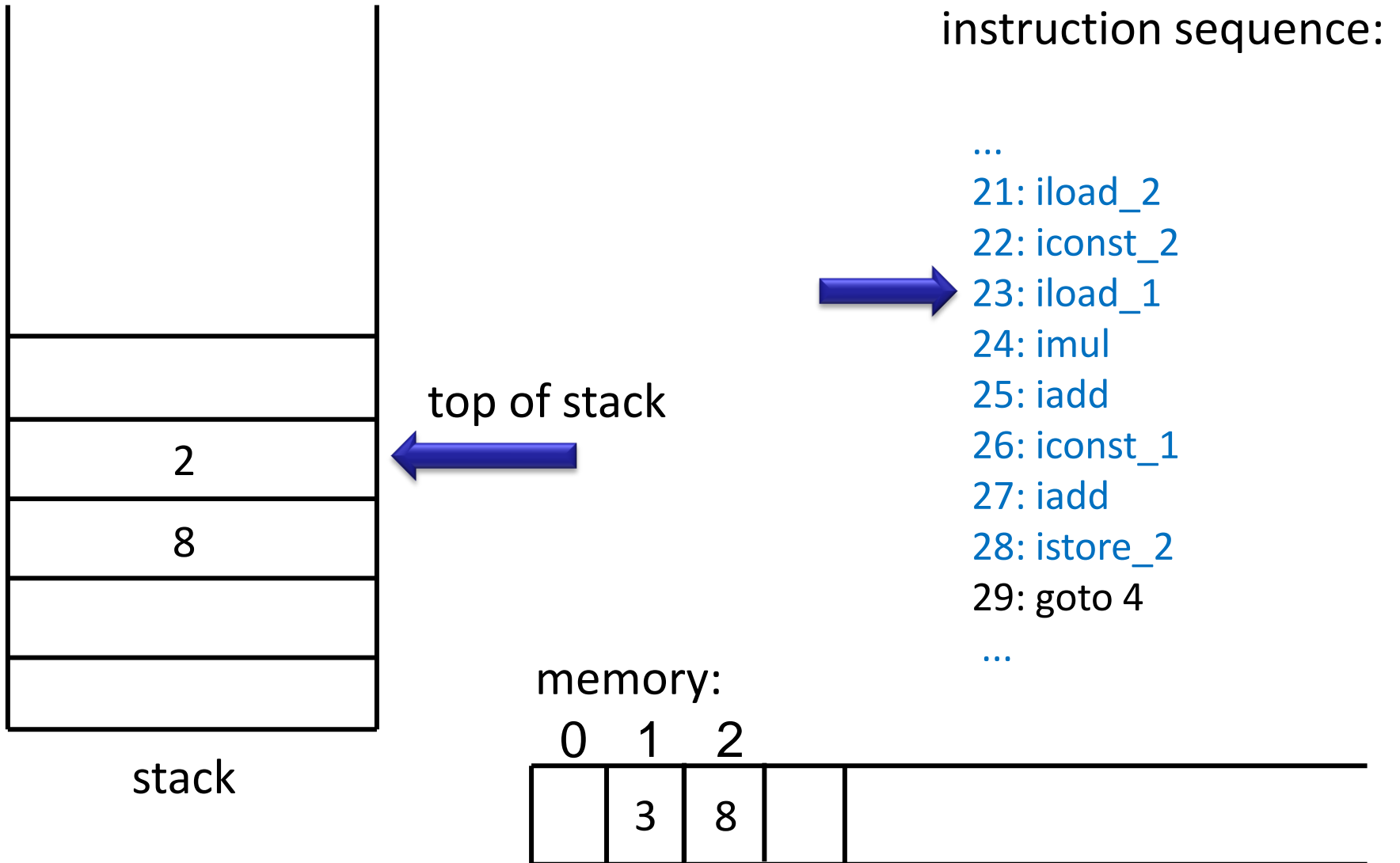


memory:

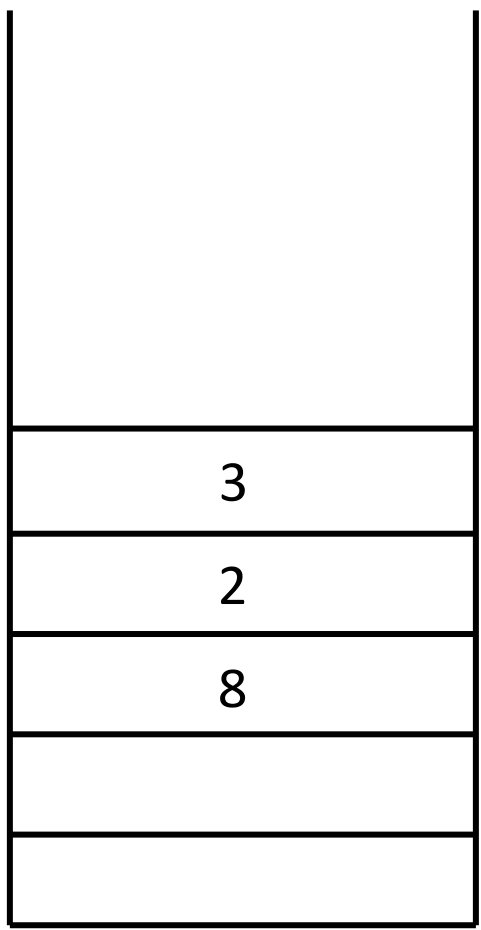




# Operands are consumed from stack and put back onto stack



# Operands are consumed from stack and put back onto stack



top of stack



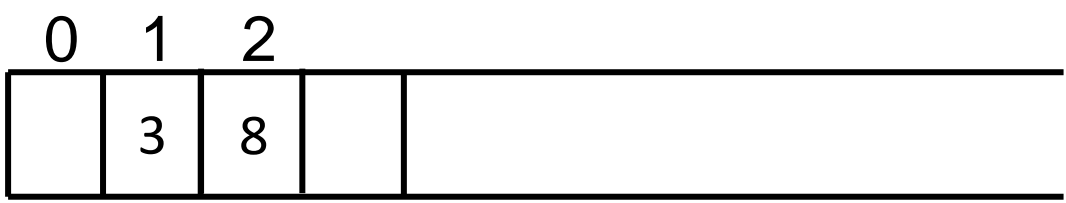
instruction sequence:

- ...
- 21: iload\_2
- 22: iconst\_2
- 23: iload\_1
- 24: imul
- 25: iadd
- 26: iconst\_1
- 27: iadd
- 28: istore\_2
- 29: goto 4
- ...

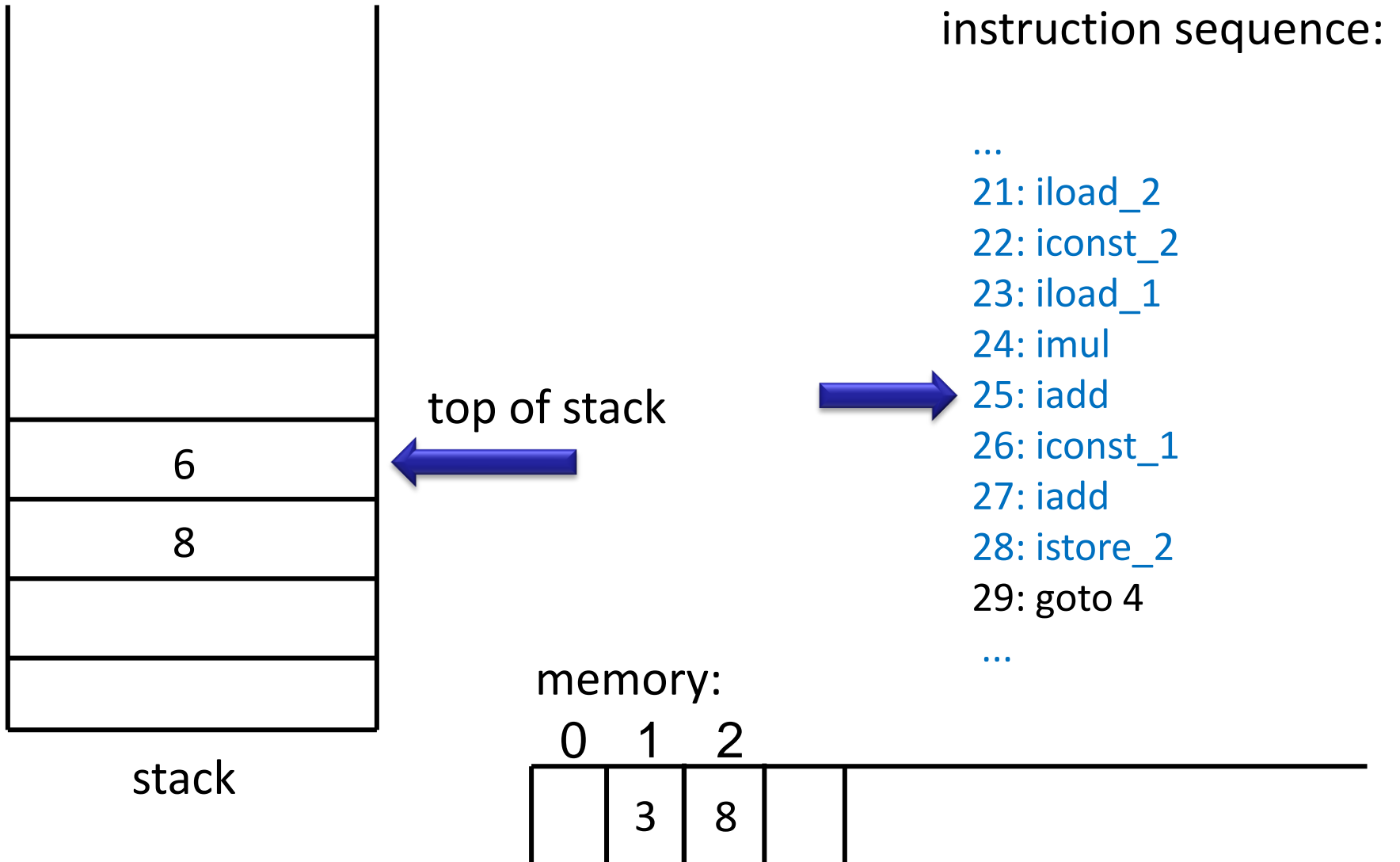


stack

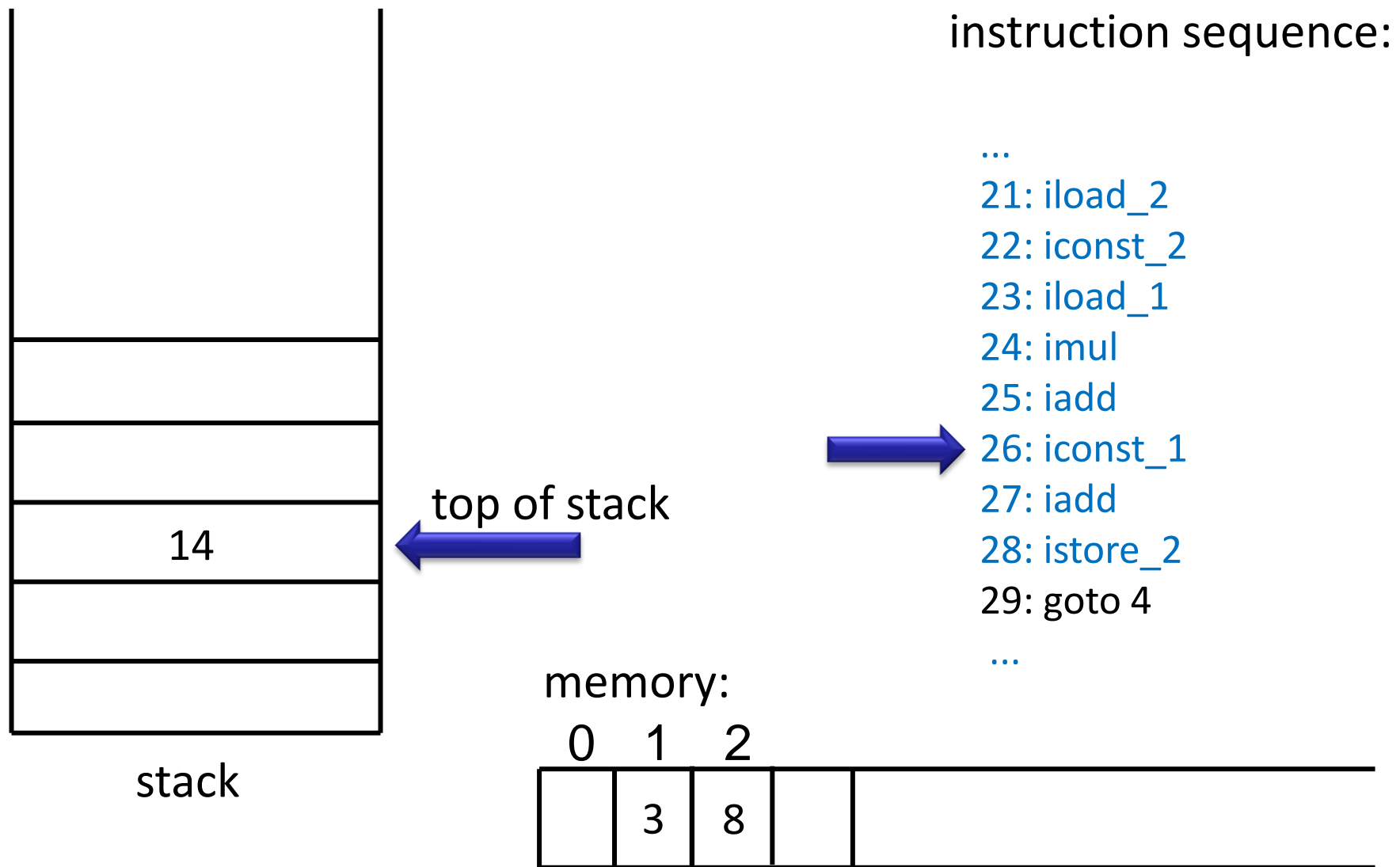
memory:



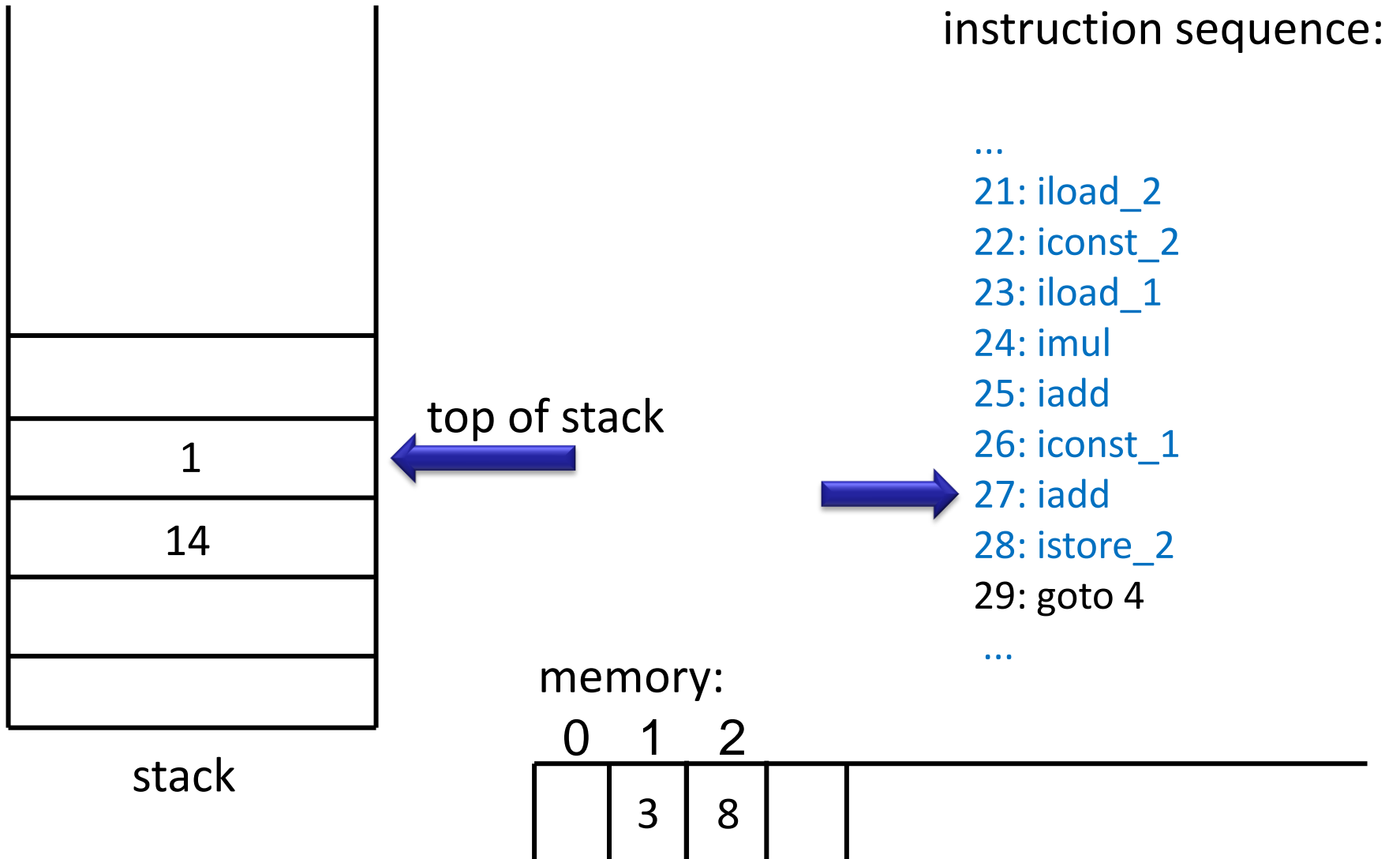
# Operands are consumed from stack and put back onto stack



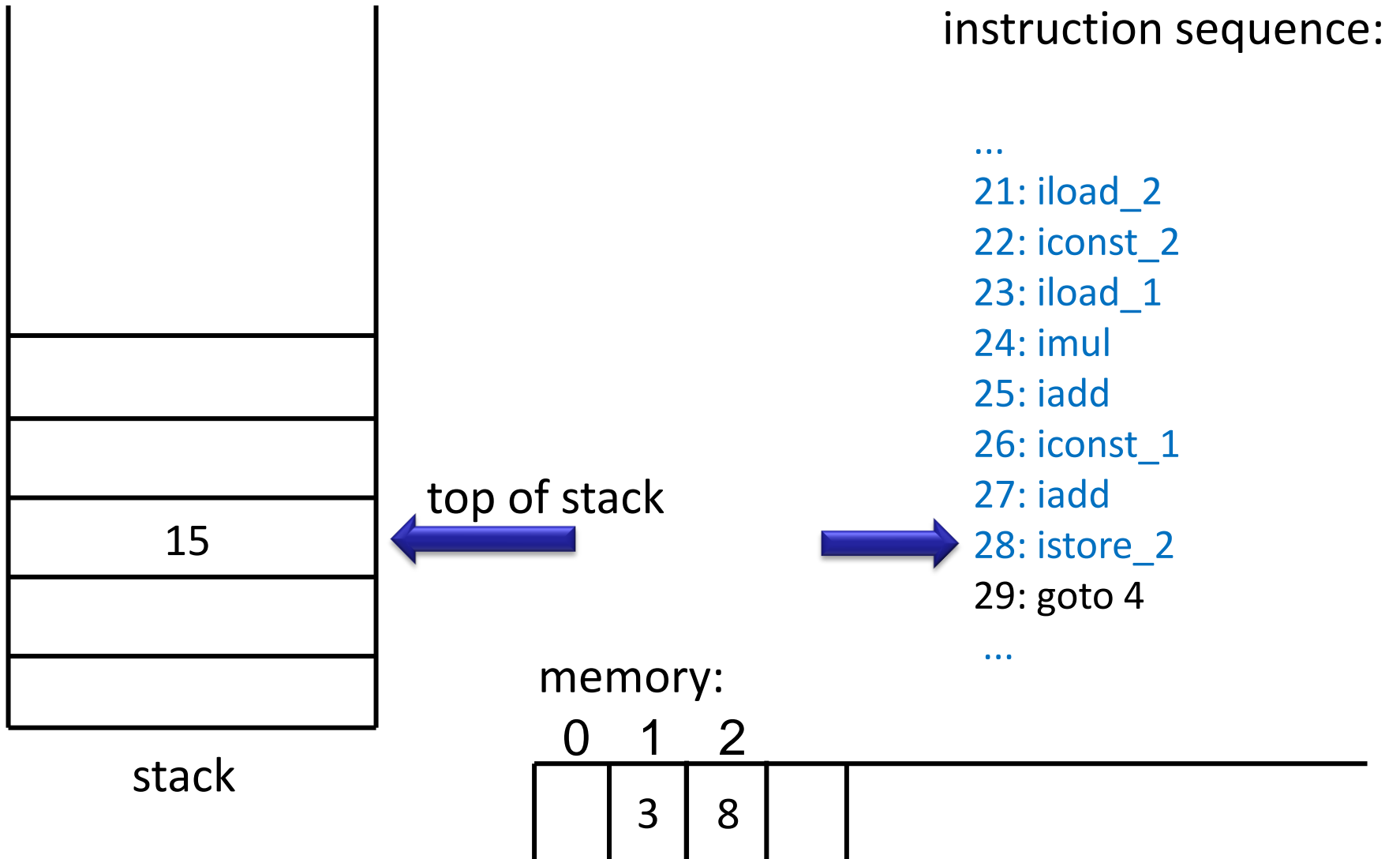
# Operands are consumed from stack and put back onto stack



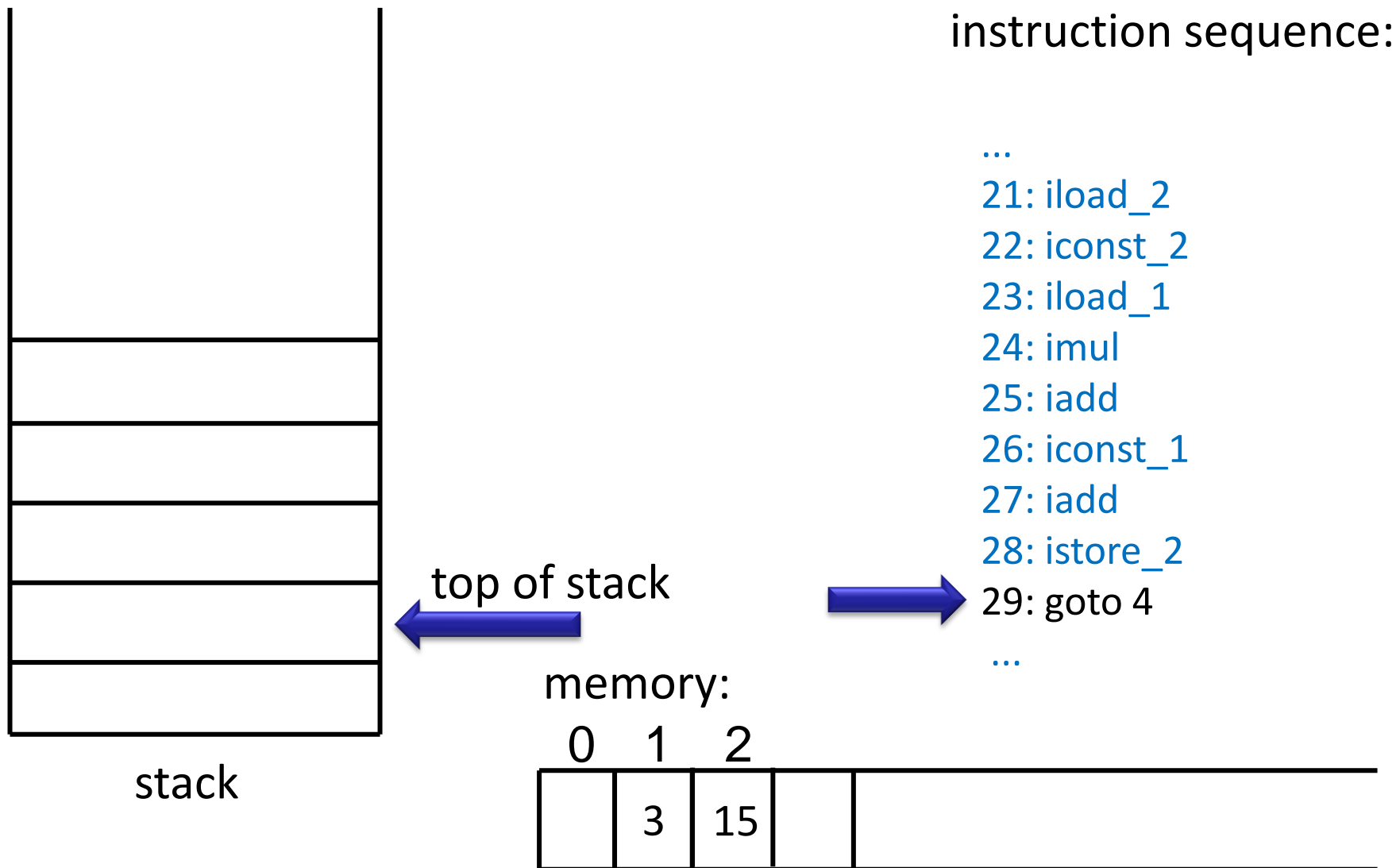
# Operands are consumed from stack and put back onto stack



# Operands are consumed from stack and put back onto stack



# Operands are consumed from stack and put back onto stack



# 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

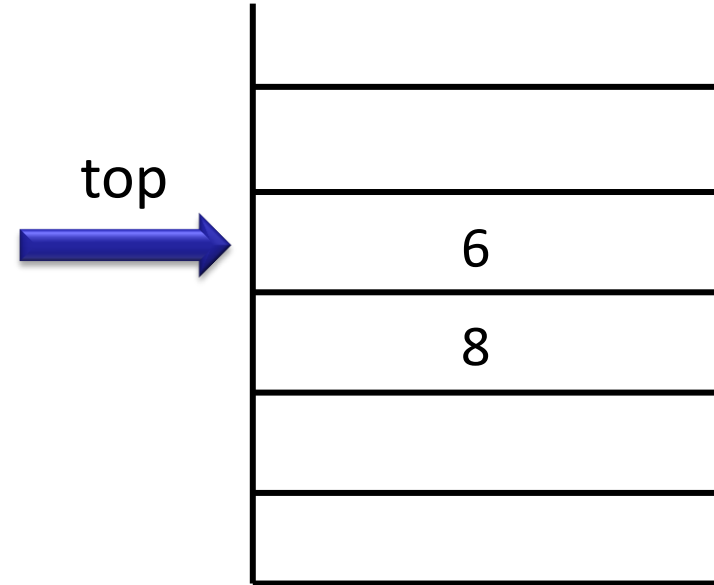


# Stack Machine Simulator

```
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 ladd() =>
    operand(top - 1) = operand(top - 1) + operand(top)
    top = top - 1 // two consumed, one produced
  case lmul() =>
    operand(top - 1) = operand(top - 1) * operand(top)
    top = top - 1 // two consumed, one produced
```



stack

# Stack Machine Simulator: Moving Data

```
case Bipush(c) =>
  operand(top + 1) = c // put given constant 'c' onto stack
  top = top + 1
case lload(n) =>
  operand(top + 1) = local(n) // from memory onto stack
  top = top + 1
case lstore(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

# Selected Instructions

bipush X	Like iconst, but for arbitrarily large X
iload_x	Loads the integer value of the local variable in slot x on the stack. $x \in \{0, 1, 2, 3\}$
iload X	Loads the value of the local variable pointed to by index X on the top of the stack.
iconst_x	Loads the integer constant x on the stack. $x \in \{0, 1, 2, 3, 4, 5\}$ .
istore_x	Stores the current value on top of the stack in the local variable in slot x. $x \in \{0, 1, 2, 3\}$
istore X	Stores the current value on top of the stack in the local variable indexed by X.
ireturn	Method return statement (note that the return value has to have been put on the top of the stack beforehand).
iadd	Pop two (integer) values from the stack, add them and put the result back on the stack.
isub	Pop two (integer) values from the stack, subtract them and put the result back on the stack.
imult	Pop two (integer) values from the stack, multiply them and put the result back on the stack.

idiv	Pop two (integer) values from the stack, divide them and put the result back on the stack.
irem	Pop two (integer) values from the stack, put the result of $x_1 \% x_2$ back on the stack.
ineg	Negate the value on the stack.
iinc x, y	Increment the variable in slot x by amount y.
ior	Logical OR for the two integer values on the stack.
iand	Logical AND for the two integer values on the stack.
ixor	Logical XOR for the two integer values on the stack.
ifXX L	Pop one value from the stack, compare it zero according to the operator XX. If the condition is satisfied, jump to the instruction given by label L. $XX \in \{ eq, lt, le, ne, gt, ge, null, nonnull \}$
if_icmpXX L	Pop two values from the stack and compare against each other. Rest as above.
goto L	Unconditional jump to instruction given by the label L.
pop	Discard word currently on top of the stack.
dup	Duplicate word currently on top of the stack.
swap	Swaps the two top values on the stack.
aload_x	Loads an object reference from slot x.
aload X	Loads an object reference from local variable indexed by X.
iaload	Loads onto the stack an integer from an array. The stack must contain the array reference and the index.
iastore	Stores an integer in an array. The stack must contain the arrayreference, the index and the value, in that order.

# 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!

<https://github.com/psuter/cafebabe/wiki>

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

```
00000000  ca fe ba be 00 00 00 32 00 3b 0a 00 12 00 1e 07
00000020  00 1f 0a 00 02 00 1e 0a 00 02 00 20 08 00 21 08
00000040  00 22 09 00 23 00 24 07 00 25 0a 00 08 00 1e 08
00000060  00 26 0a 00 08 00 27 08 00 28 08 00 29 0a 00 02
00000100  00 2a 0a 00 08 00 2b 0a 00 08 00 2c 0a 00 2d 00
```

More on that in the labs!

# Towards Compiling Expressions: Prefix, Infix, and Postfix Notation

# Overview of Prefix, Infix, Postfix

Let  $f$  be a binary operation,  $e_1 e_2$  two expressions

We can denote application  $f(e_1, e_2)$  as follows

– in **prefix** notation  $f e_1 e_2$

– in **infix** notation  $e_1 f e_2$

– in **postfix** notation  $e_1 e_2 f$

- Suppose that each operator (like  $f$ ) has a known number of arguments. For nested expressions
  - infix requires parentheses in general
  - prefix and postfix do not require any parantheses!

# Expressions in Different Notation

For infix, assume \* binds stronger than +

There is no need for priorities or parens in the other notations

<b>arg.list</b>	$\underline{+}(x,y)$	$\underline{+}(\underline{*}(x,y),z)$	$\underline{+}(x, \underline{*}(y,z))$	$\underline{*}(x, \underline{+}(y,z))$
<b>prefix</b>	$+ x y$	$+ * x y z$	$+ x * y z$	$* x + y z$
<b>infix</b>	$x + y$	$x * y + z$	$x + y * z$	$x * (y + z)$
<b>postfix</b>	$x y +$	$x y * z +$	$x y z * +$	$x y z + *$

Infix is the only problematic notation and leads to ambiguity

Why is it used in math? Ambiguity reminds us of algebraic laws:

$x + y$  looks same from left and from right (commutative)

$x + y + z$  parse trees mathematically equivalent (associative)

# Convert into Prefix and Postfix

**prefix**      $+ + + x y z u$

$+ x + y + z u$

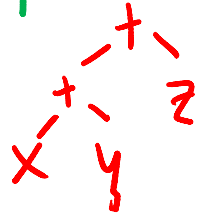
**infix**      $((x + y) + z) + u$

$x + (y + (z + u))$

**postfix**      $x y + z + u +$

$x y \underbrace{z u} + + +$

draw the trees:



Terminology:

prefix = Polish notation

(attributed to Jan Lukasiewicz from Poland)

postfix = Reverse Polish notation (RPN)

Is the sequence of characters in postfix opposite to one in prefix if we have binary operations?

What if we have only unary operations?

# Compare Notation and Trees

**arg.list**

$+(x,y)$

$+(* (x,y),z)$

$+(x,* (y,z))$

$*(x,+(y,z))$

**prefix**

$+ x y$

$+ * x y z$

$+ x * y z$

$* x + y z$

**infix**

$x + y$

$x * y + z$

$x + y * z$

$x * (y + z)$

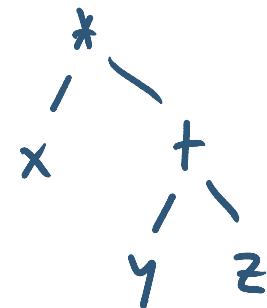
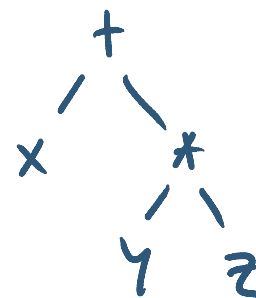
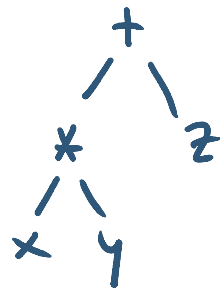
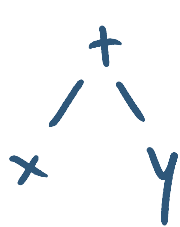
**postfix**

$x y +$

$x y * z +$

$x y z * +$

$x y z + *$



draw ASTs for each expression

How would you pretty print AST into a given form?

# Simple Expressions and Tokens

```
sealed abstract class Expr
```

```
case class Var(varID: String) extends Expr
```

```
case class Plus(lhs: Expr, rhs: Expr) extends Expr
```

```
case class Times(lhs: Expr, rhs: Expr) extends Expr
```

```
sealed abstract class Token
```

```
case class ID(str : String) extends Token
```

```
case class Add extends Token
```

```
case class Mul extends Token
```

```
case class O extends Token // (
```

```
case class C extends Token // )
```

# Printing Trees into Lists of Tokens

```
def prefix(e : Expr) : List[Token] = e match {  
  case Var(id) => List(ID(id))  
  case Plus(e1,e2) => List(Add()) ::: prefix(e1) ::: prefix(e2)  
  case Times(e1,e2) => List(Mul()) ::: prefix(e1) ::: prefix(e2)  
}  
  
def infix(e : Expr) : List[Token] = e match { // should emit parantheses!  
  case Var(id) => List(ID(id))  
  case Plus(e1,e2) => List(O())::: infix(e1) ::: List(Add()) ::: infix(e2) :::List(C())  
  case Times(e1,e2) => List(O())::: infix(e1) ::: List(Mul()) ::: infix(e2) :::List(C())  
}  
  
def postfix(e : Expr) : List[Token] = e match {  
  case Var(id) => List(ID(id))  
  case Plus(e1,e2) => postfix(e1) ::: postfix(e2) ::: List(Add())  
  case Times(e1,e2) => postfix(e1) ::: postfix(e2) ::: List(Mul())  
}
```



# LISP: Language with Prefix Notation

- 1958 – pioneering language
- Syntax was meant to be abstract syntax
- Treats all operators as user-defined ones, so syntax does not assume the number of arguments is known
  - use parantheses in prefix notation:  $f(x,y)$  as  $(f\ x\ y)$

```
(defun factorial (n)
```

```
  (if (<= n 1)
```

```
      1
```

```
      (* n (factorial (- n 1))))))
```

# PostScript: Language using Postfix

- .ps are ASCII files given to PostScript-compliant printers
- Each file is a program whose execution prints the desired pages
- <http://en.wikipedia.org/wiki/PostScript%20programming%20language>

PostScript language tutorial and cookbook

Adobe Systems Incorporated

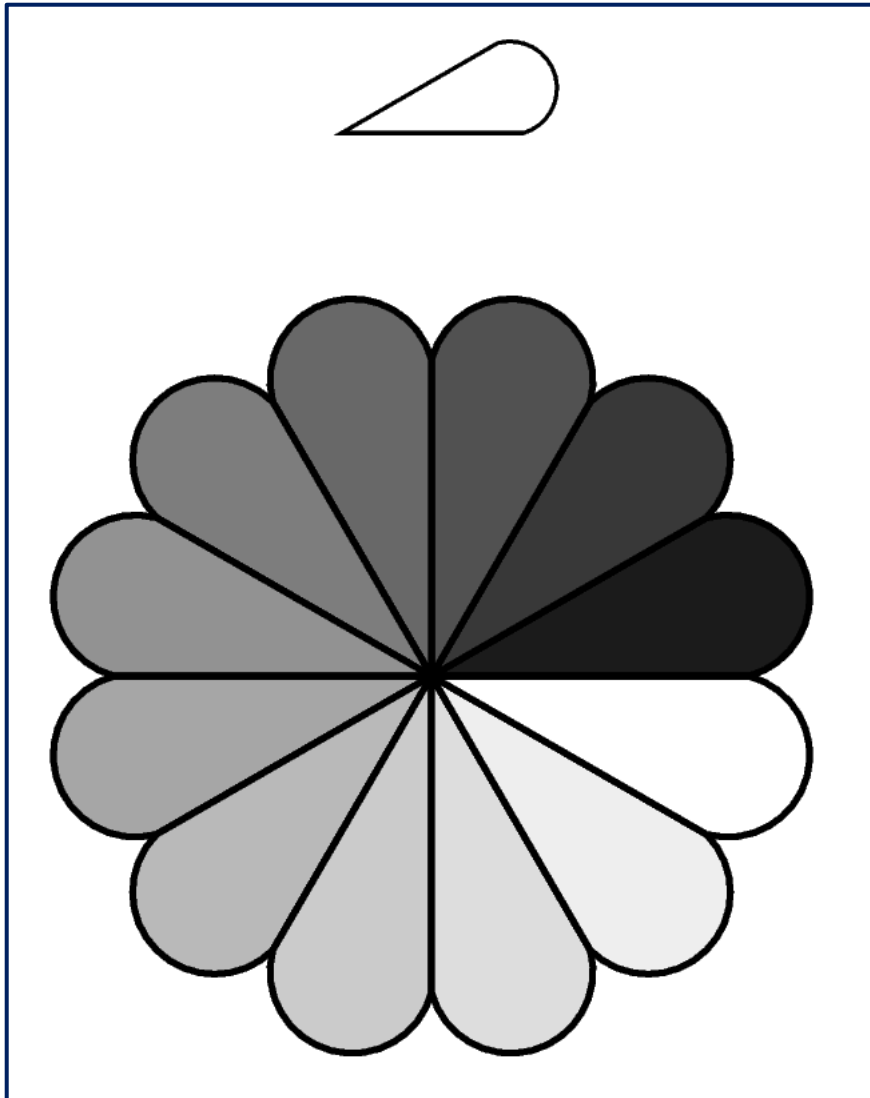
Reading, MA : Addison Wesley, 1985

ISBN 0-201-10179-3 (pbk.)

# A PostScript Program

```
/inch {72 mul} def
/wedge
    { newpath
      0 0 moveto
      1 0 translate
      15 rotate
      0 15 sin translate
      0 0 15 sin -90 90 arc
      closepath
    } def
gsave
  3.75 inch 7.25 inch translate
  1 inch 1 inch scale
  wedge 0.02 setlinewidth stroke
grestore
gsave
  4.25 inch 4.25 inch translate
  1.75 inch 1.75 inch scale
  0.02 setlinewidth
  1 1 12
    { 12 div setgray
      gsave
        wedge
      gsave fill grestore
      0 setgray stroke
    grestore
  30 rotate
    } for
grestore
showpage
```

If we send it to printer  
(or run GhostView viewer gv) we get



```
4.25 inch 4.25 inch translate  
1.75 inch 1.75 inch scale  
0.02 setlinewidth  
1 1 12
```

```
{ 12 div setgray  
  gsave  
    wedge  
    gsave fill grestore  
    0 setgray stroke  
  grestore  
  30 rotate  
} for
```

```
grestore  
showpage
```

# Why postfix? Can evaluate it using stack

```
def postEval(env : Map[String,Int], pexpr : Array[Token]) : Int = { // no recursion!
  var stack : Array[Int] = new Array[Int](512)
  var top : Int = 0; var pos : Int = 0
  while (pos < pexpr.length) {
    pexpr(pos) match {
      case ID(v) => top = top + 1
                    stack(top) = env(v)
      case Add() => stack(top - 1) = stack(top - 1) + stack(top)
                    top = top - 1
      case Mul() => stack(top - 1) = stack(top - 1) * stack(top)
                    top = top - 1
    }
    pos = pos + 1
  }
  stack(top)
}
```

9  
4 9  
3 3 27

$x \rightarrow 3, y \rightarrow 4, z \rightarrow 5$

infix:  $x^*(y+z)$

postfix:  $x y z + *$

Run 'postfix' for this env

# Evaluating Infix Needs Recursion

The recursive interpreter:

```
def infixEval(env : Map[String,Int], expr : Expr) : Int =  
expr match {  
  case Var(id) => env(id)  
  case Plus(e1,e2) => infix(env,e1) + infix(env,e2)  
  case Times(e1,e2) => infix(env,e1) * infix(env,e2)  
}
```

Maximal stack depth in interpreter = expression height

# Compiling Expressions

- Evaluating postfix expressions is like running a stack-based virtual machine on compiled code
- Compiling expressions for stack machine is like translating expressions into postfix form

# Expression, Tree, Postfix, Code

infix:  $x*(y+z)$

postfix:  $x y z + *$

bytecode:

iload 1

$x \rightarrow 1$

iload 2

$y \rightarrow 2$

iload 3

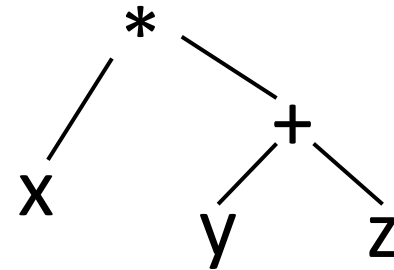
$z \rightarrow 3$

iadd

$+$

imul

$*$



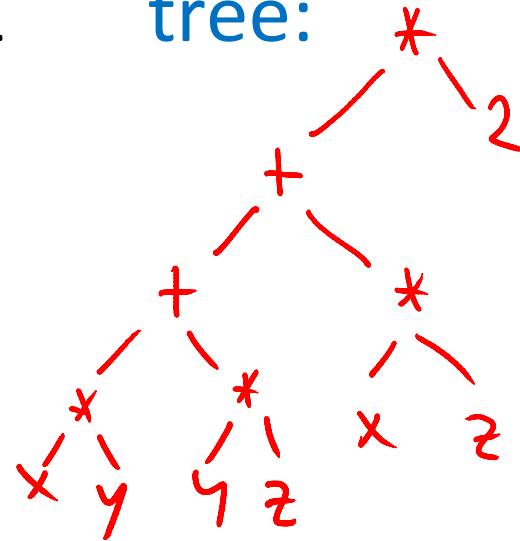


# Show Tree, Postfix, Code

infix:

$$\begin{array}{ccc} \uparrow^0 & \uparrow^1 & \uparrow^2 \\ (x * y + y * z + x * z) * 2 \end{array}$$

tree:



postfix:

bytecode:

x  
y  
\*  
y  
z  
\*  
+  
x  
z  
\*  
+  
2  
x

iload\_0  
iload\_1  
imul  
iload\_1  
iload\_2  
imul  
iadd  
iload\_0  
iload\_2  
imul  
iadd  
iconst\_2  
imul

# “Printing” Trees into Bytecodes

To evaluate  $e_1 * e_2$  interpreter

- evaluates  $e_1$
- evaluates  $e_2$
- combines the result using  $*$

Compiler for  $e_1 * e_2$  emits:

- code for  $e_1$  that leaves result on the stack, followed by
- code for  $e_2$  that leaves result on the stack, followed by
- arithmetic instruction that takes values from the stack and leaves the result on the stack

```
def compile(e : Expr) : List[Bytecode] = e match { // ~ postfix printer
case Var(id) => List(ILoad(slotFor(id)))
case Plus(e1,e2) => compile(e1) ::: compile(e2) ::: List(IAdd())
case Times(e1,e2) => compile(e1) ::: compile(e2) ::: List(IMul())
}
```

# Local Variables

For **int**: instructions **iload** and **istore** from [JVM Spec](#).

- Assigning indices (called *slots*) to local variables using function  $\text{slotOf} : \text{VarSymbol} \rightarrow \{0,1,2,3,\dots\}$
- How to compute the indices?
  - assign them in the order in which they appear in the tree

```
def compile(e : Expr) : List[Bytecode] = e match {  
  case Var(id) => List(ILoad(slotFor(id)))
```

```
  ...
```

```
}
```

↙ id = e

```
def compileStmt(s : Statmt) : List[Bytecode] = s match {  
  case Assign(id,e) => compile(e) ::: List(IStore(slotFor(id)))
```

```
  ...
```

```
}
```

# Global Variables and Fields

## Static Variables

Consult [JVM Spec](#) and see

- **getstatic**
- **putstatic**

as well as the section on **Runtime Constant Pool**.

## Instance Variables (Fields)

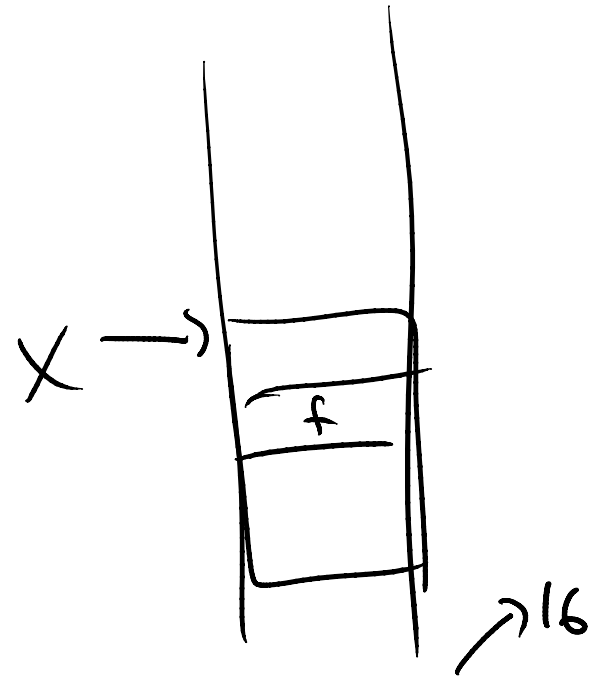
To access these we use

- **getfield**
- **putfield**

In lower-level machines  $x.f$  is evaluated as  $\text{mem}[x + \text{offset}(f)]$

With inheritance,  $\text{offset}(f)$  must be consistent despite the fact that we do not know exactly what the run-time class of  $x$  will be

- this can be very tricky to implement efficiently with multiple inheritance
- general approach: do a run-time test on the dynamic type of an object to decide how to interpret field or method access



# Factorial Example

```
class Factorial {  
    public int fact(int num){  
        int num_aux;   
        if (num < 1)  
            num_aux = 1 ;  
        else  
            num_aux=  
            num*(this.fact(num-1));  
        return num_aux;  
    }  
}
```

*Handwritten annotations:*  
- `this` → 0  
- `fact` → 1  
- `num_aux` → 2  
- `fact` in `num*(this.fact(num-1));` is boxed.

```
public int fact(int);  
Code:  
0: iload_1  
1: iconst_1  
2: if_icmpge 10  
5: iconst_1  
6: istore_2  
7: goto 20  
10: iload_1 → num  
11: aload_0 → this  
12: iload_1 → num  
13: iconst_1 → 1  
14: isub  
15: invokevirtual #2; //Met. fact:(I)I  
18: imul  
19: istore_2  
20: iload_2  
21: ireturn
```

*Handwritten annotations:*  
- A bracket groups instructions 12, 13, and 14, labeled `num-1`.  
- An arrow points from `num` to instruction 10.

`aload_0` refers to receiver object (0<sup>th</sup> argument), since 'fact' is not static

# Correctness

If we execute the compiled code, the result is the same as running the interpreter.

$$\text{exec}(\text{env}, \text{compile}(\text{expr})) == \text{interpret}(\text{env}, \text{expr}) \quad \}$$

**interpret** : Env x Expr -> Int

**compile** : Expr -> List[Bytecode]

**exec** : Env x List[Bytecode] -> Int

Assume 'env' in both cases maps var names to values.

With much more work, can prove correctness of entire compiler:

[CompCert - A C Compiler whose Correctness has been Formally Verified](#)

$\text{exec}(\text{env}, \text{compile}(\text{expr})) ==$   
 $\text{interpret}(\text{env}, \text{expr})$

Attempted proof by induction:

$\text{exec}(\text{env}, \text{compile}(\text{Times}(e1, e2))) ==$   
 $\text{exec}(\text{env}, \text{compile}(e1) :: \text{compile}(e2) :: \text{List}('*')) ==$

We need to know something about behavior of intermediate executions.

`exec` : Env x List[Bytecode] -> Int

`run` : Env x List[Bytecode] x List[Int] -> List[Int]

**// stack as argument and result**

$\text{exec}(\text{env}, \text{bcodes}) == \text{run}(\text{env}, \text{bcodes}, \text{List}()).\text{head}$

# run(env,bcodes,stack) = newStack

Executing sequence of instructions

**run** : Env x List[Bytecode] x List[Int] -> List[Int]

Stack grows to the right, top of the stack is last element

Byte codes are consumed from left

Definition of run is such that

- $\text{run}(\text{env}, \text{`*`} :: L, S ::: \text{List}(x1, x2)) == \text{run}(\text{env}, L, S ::: \text{List}(x1 * x2))$
- $\text{run}(\text{env}, \text{`+`} :: L, S ::: \text{List}(x1, x2)) == \text{run}(\text{env}, L, S ::: \text{List}(x1 + x2))$
- $\text{run}(\text{env}, \text{ILoad}(n) :: L, S) == \text{run}(\text{env}, L, S ::: \text{List}(\text{env}(n)))$

By induction one shows:

- $\text{run}(\text{env}, L1 ::: L2, S) == \text{run}(\text{env}, L2, \text{run}(\text{env}, L1, S))$

execute instructions L1, then execute L2 on the result



# New correctness condition

`exec` : `Env x List[Bytecode] -> Int`

`run` : `Env x List[Bytecode] x List[Int] -> List[Int]`

Old condition:

`exec`(`env, compile(expr)`) == `interpret(env, expr)`

New condition:  $\forall S. \text{run}(\text{env}, \text{compile}(\text{expr}), S) == S :: \text{List}(\text{interpret}(\text{env}, \text{expr}))$

for short (T - table), [x] denotes List(x)

$\text{run}(T, C(e), S) == S :: [I(T, e)]$

induction  
on  $e$

## Proof of one case:

$$\text{run}(T, C(e), S) == S ::: [I(T, e)]$$

$$\begin{aligned} & \text{run}(T, C(\text{Times}(e1, e2)), S) == \\ & \text{run}(T, C(e1) ::: C(e2) ::: [ `*` ], S) == \\ & \text{run}(T, [ `*` ], \text{run}(T, e2, \text{run}(T, e1, S) )) == \\ & \text{run}(T, [ `*` ], \text{run}(T, e2, S ::: [I(T, e1)])) == \\ & \text{run}(T, [ `*` ], S ::: [I(T, e1), I(T, e2)]) == \\ & S ::: [I(T, e1) * I(T, e2)] == \\ & S ::: [I(T, \text{Times}(e1, e2))] \end{aligned}$$