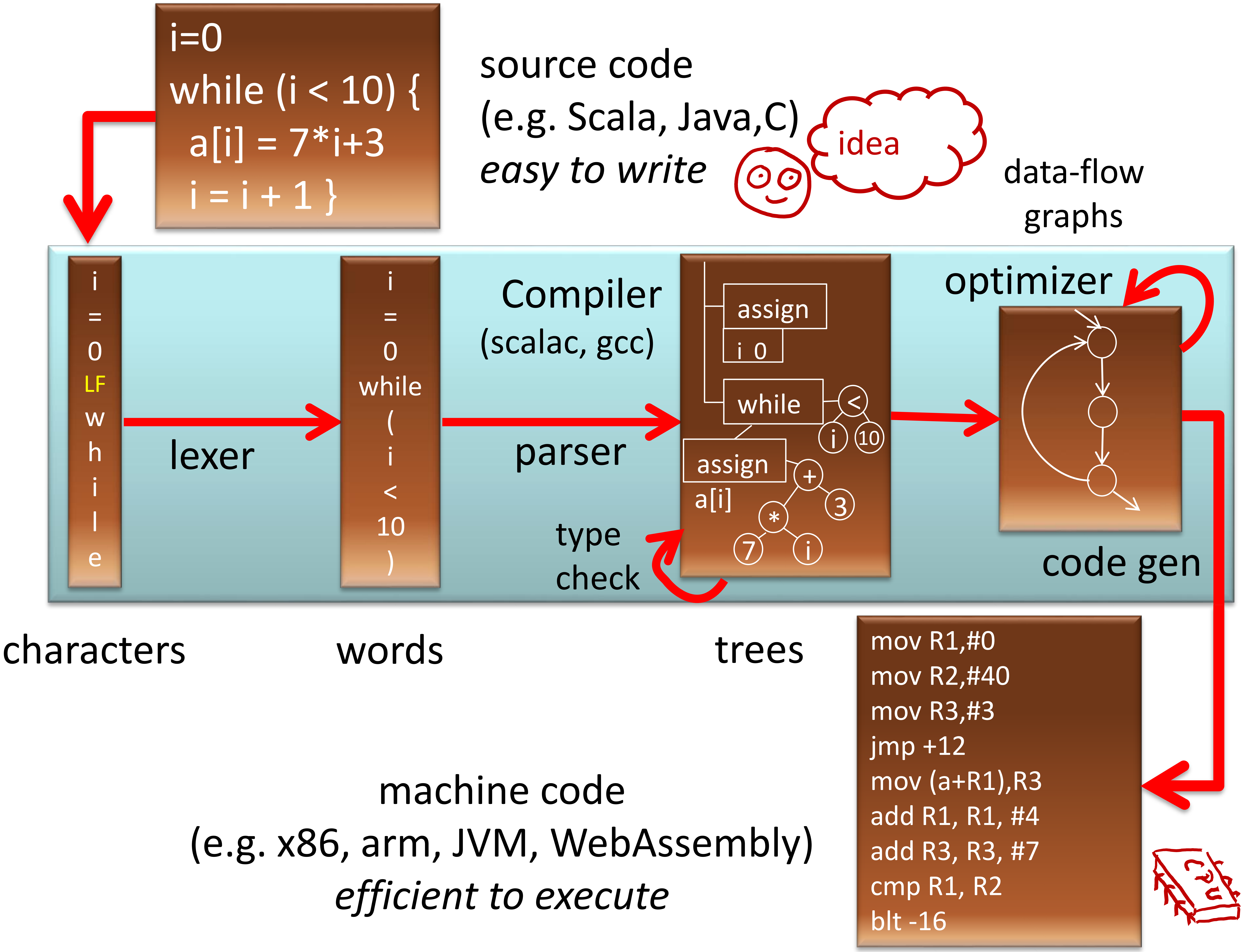
Code Generation: Introduction



```
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

Compiler
(scalac, gcc)

lexer

parser

type
check

optimizer

code gen

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, WebAssembly)
efficient to execute

Example: gcc

test.c

```
#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;
  }
}
```

gcc test.c -S

test.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 did (i<10) compile to?

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
```

Guess what each JVM instruction for the highlighted expression does.

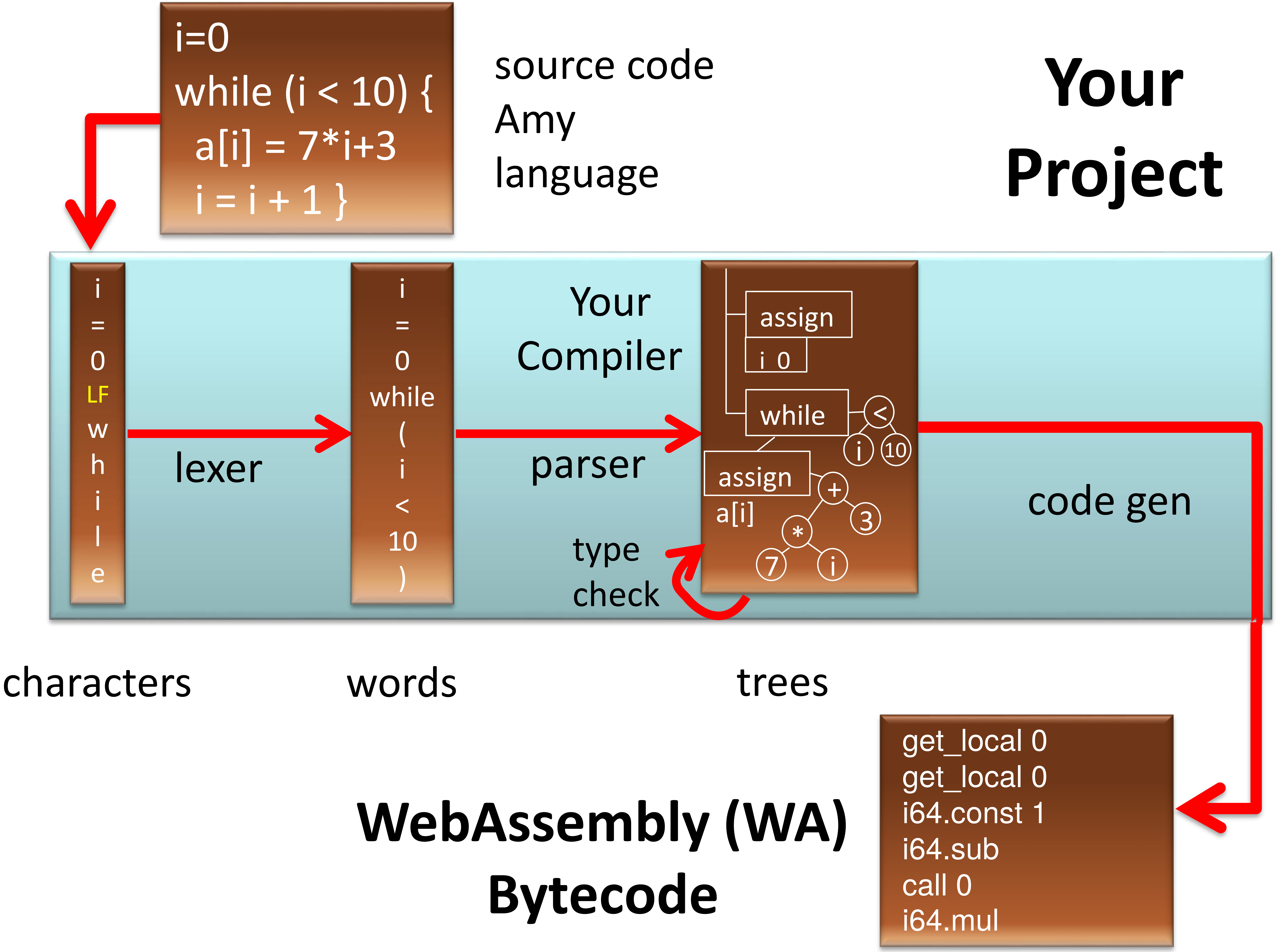
Java Virtual Machine

Use: **javac -g *.java** to compile

javap -c -l ClassName to explore

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.11

Your Project



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

source code
Amy
language

Your Compiler

characters

words

trees

WebAssembly (WA) Bytecode

```
get_local 0
get_local 0
i64.const 1
i64.sub
call 0
i64.mul
```

WebAssembly

- Overview of bytecodes:
  http://webassembly.org/docs/semantics/
- Compiling from C:
  http://webassembly.org/getting-started/developers-guide/
  https://hacks.mozilla.org/2017/03/previewing-the-webassembly-explorer/
- Research paper and the talk:
  Bringing the Web up to Speed with WebAssembly
  by Andreas Haas, Andreas Rossberg, Derek Schuff, Ben L. Titzer, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, Michael Holman.
  ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI), 2017.

WebAssembly example

C++

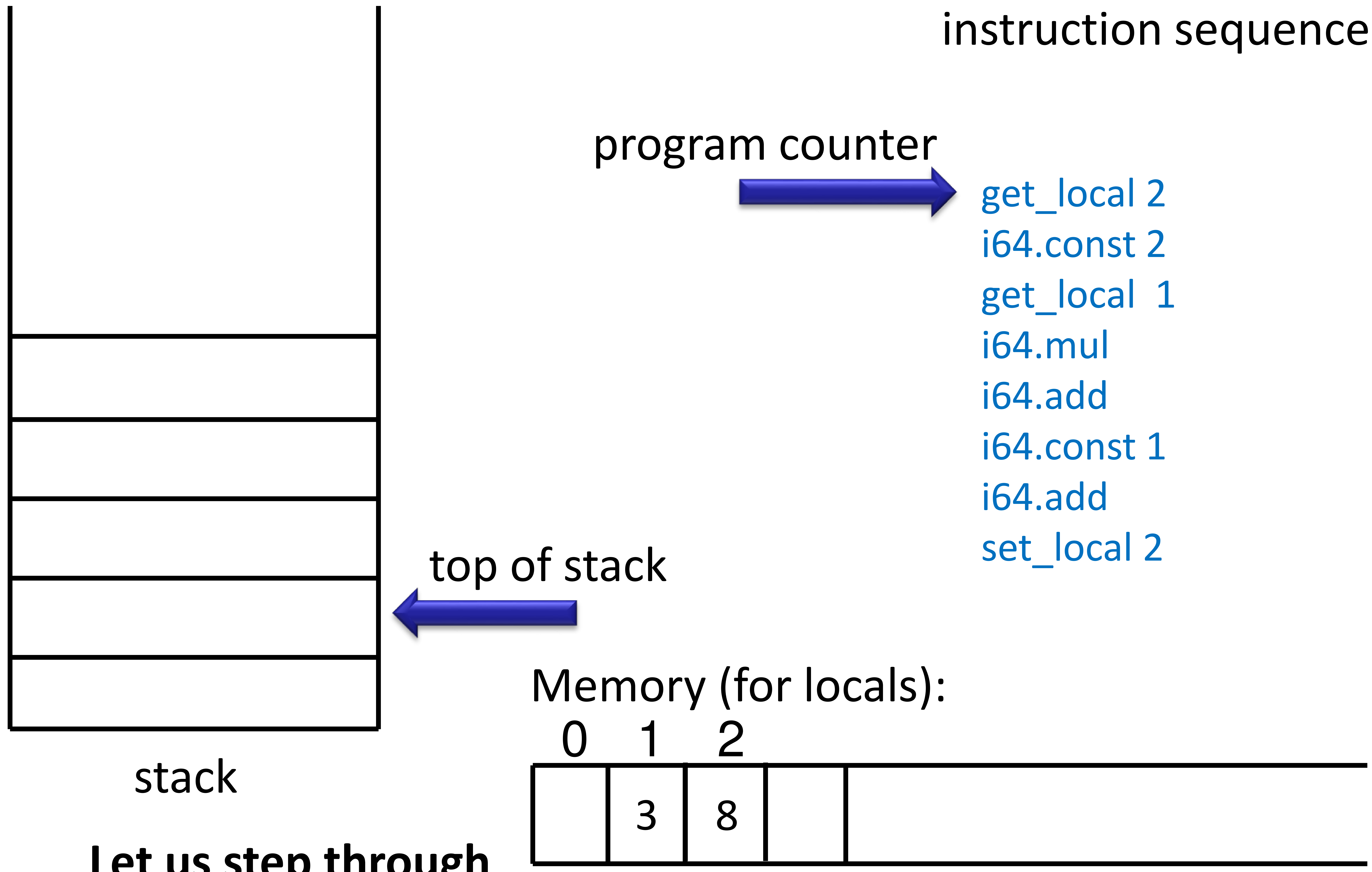
```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

WebAssembly

```
get_local 0          // n
i64.const 0          // 0
i64.eq               // n==0 ?
if i64
    i64.const 1      // 1
else
    get_local 0      //  n
    get_local 0      //  n
    i64.const 1      //  1
    i64.sub          // n-1
    call 0           // f(n-1)
    i64.mul          // n*f(n-1)
end
```

More at: https://mbebenita.github.io/WasmExplorer/

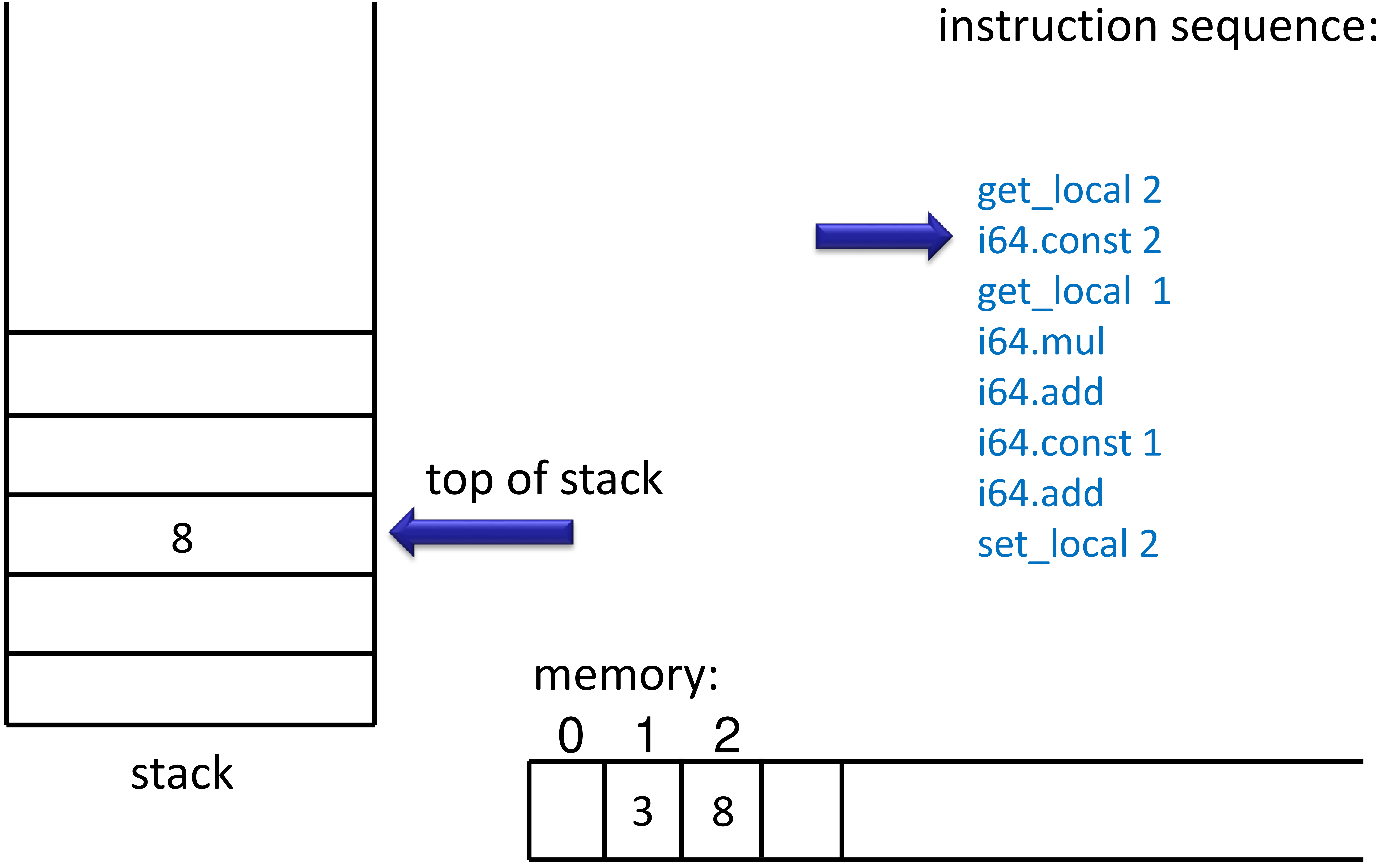
Stack Machine: High-Level Machine Code



```
get_local 2
i64.const 2
get_local  1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2
```

Let us step through

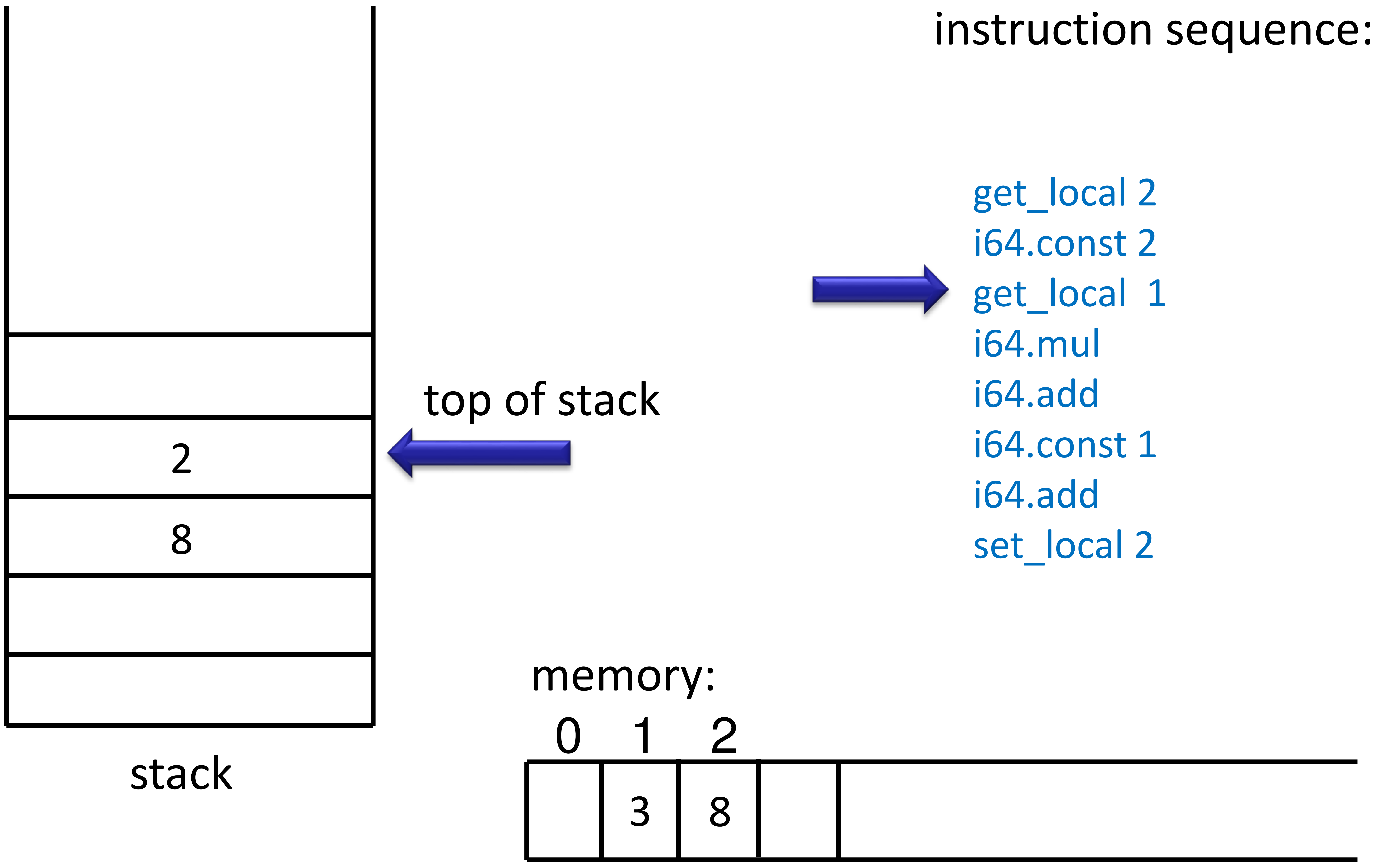
Operands are consumed from stack and put back onto stack



instruction sequence:

```
get_local 2
i64.const 2
get_local  1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2
```

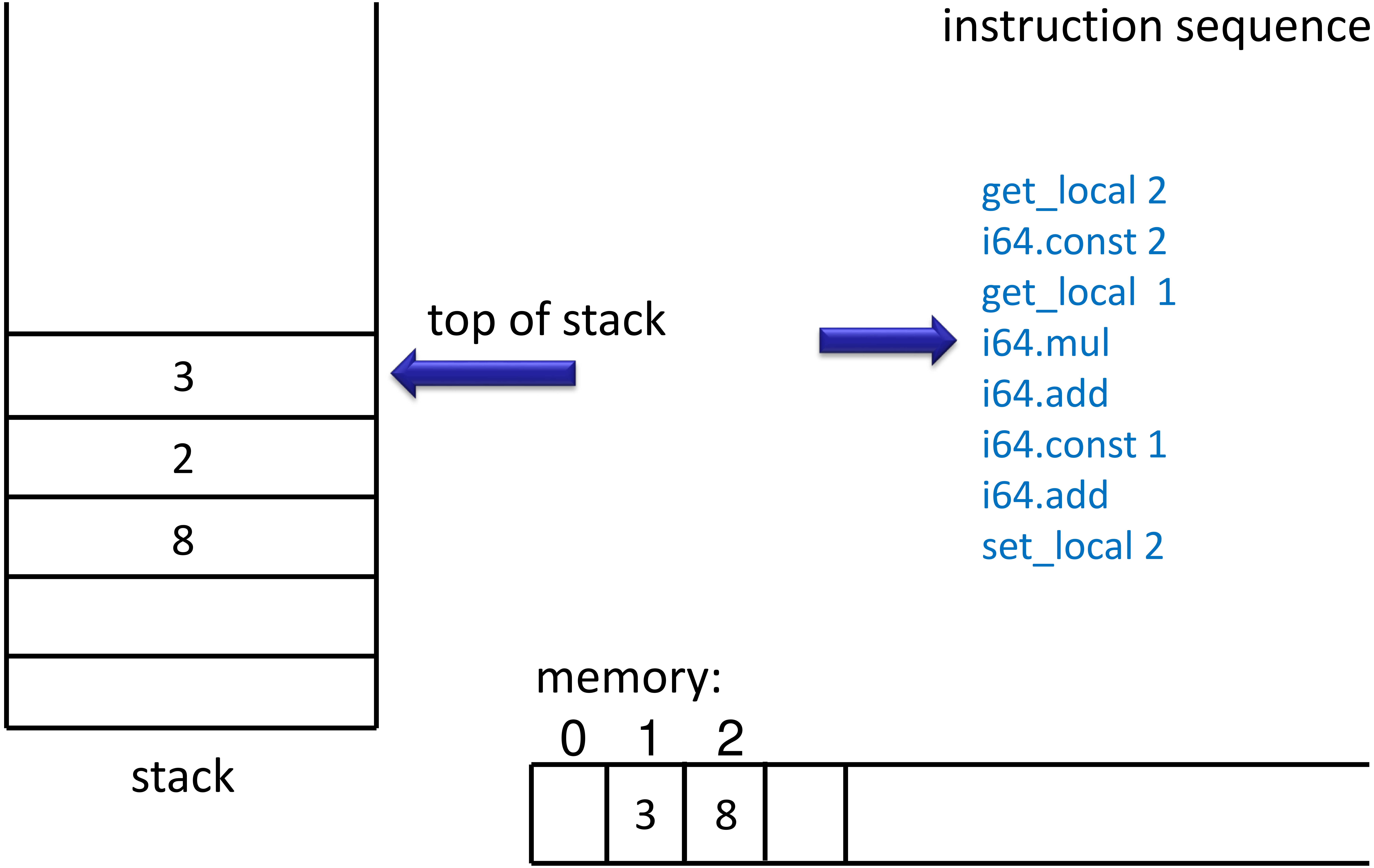
Operands are consumed from stack and put back onto stack



instruction sequence:

```
get_local 2
i64.const 2
get_local  1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2
```

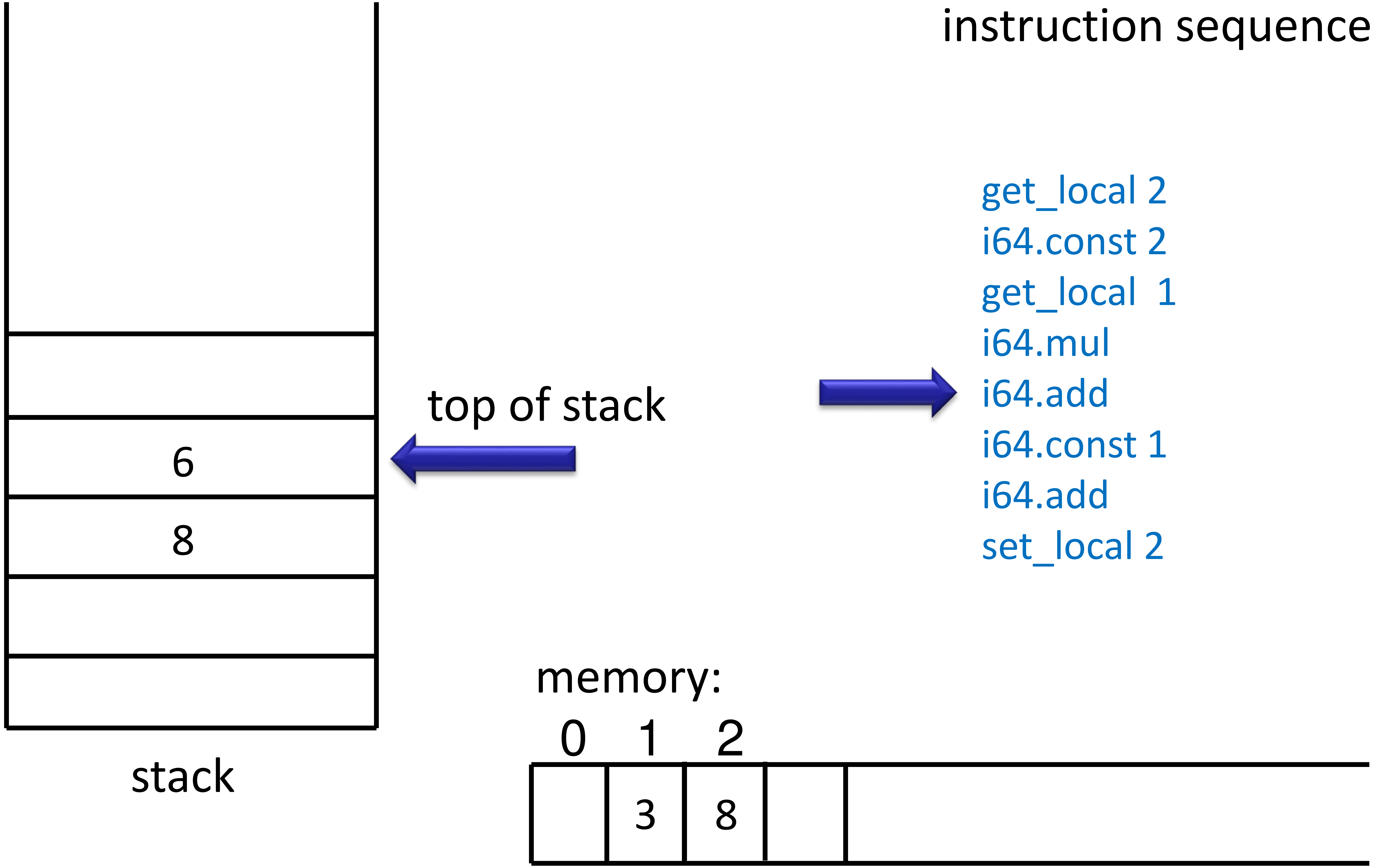
Operands are consumed from stack and put back onto stack



instruction sequence:

get_local 2
i64.const 2
get_local 1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2

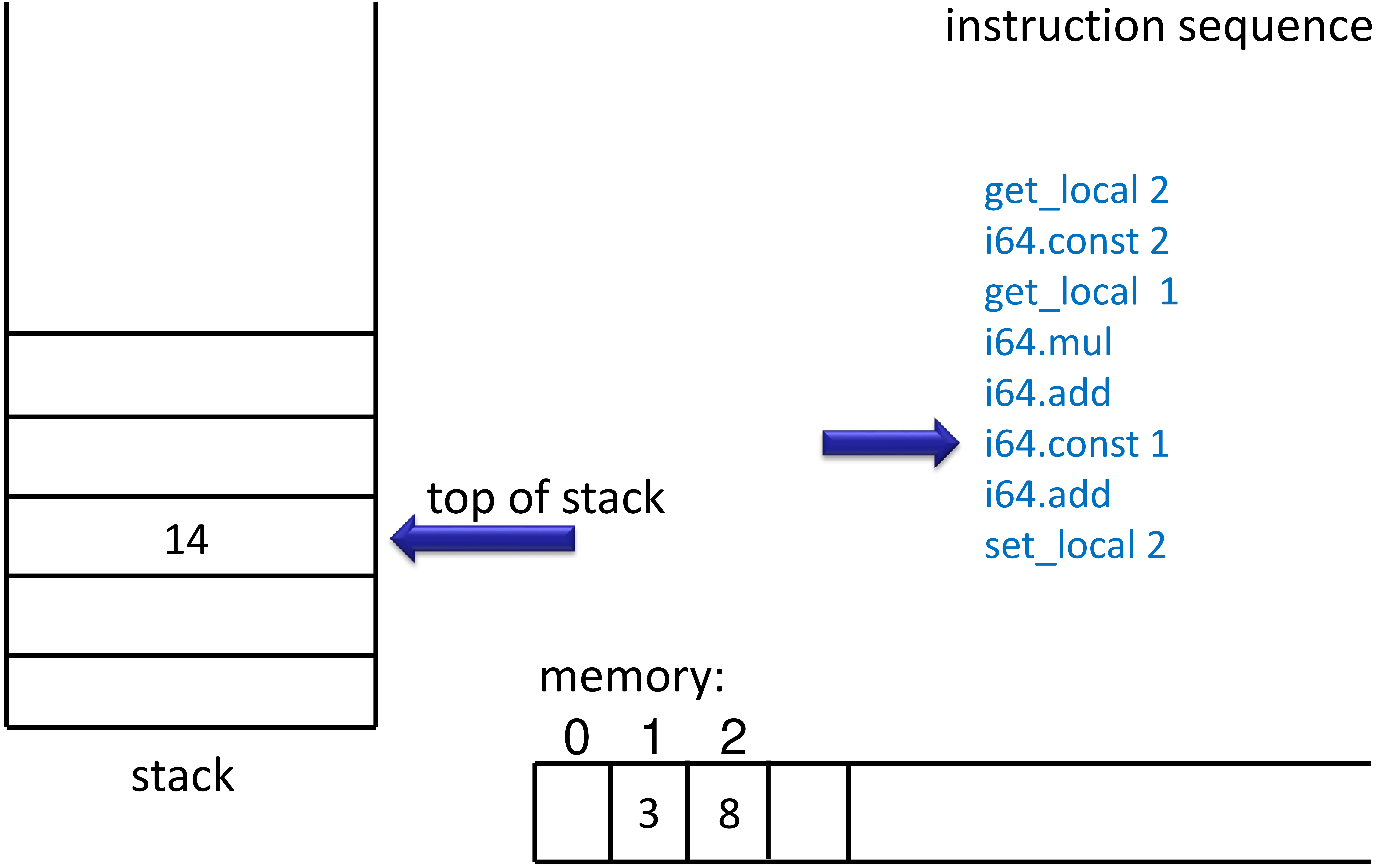
Operands are consumed from stack and put back onto stack



instruction sequence:

get_local 2
i64.const 2
get_local 1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2

Operands are consumed from stack and put back onto stack



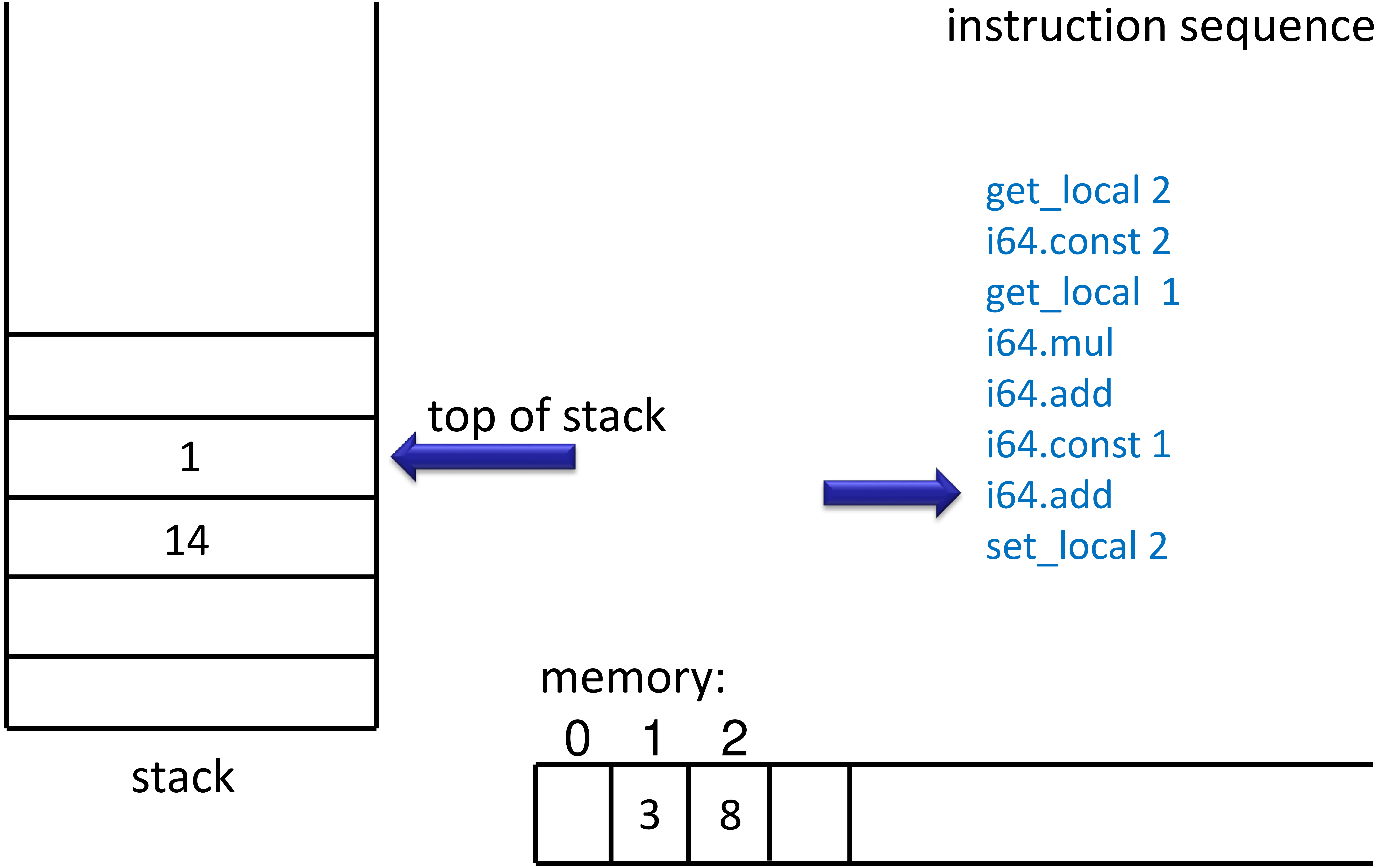
instruction sequence:

get_local 2
i64.const 2
get_local 1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2

memory:

0	1	2		
	3	8		

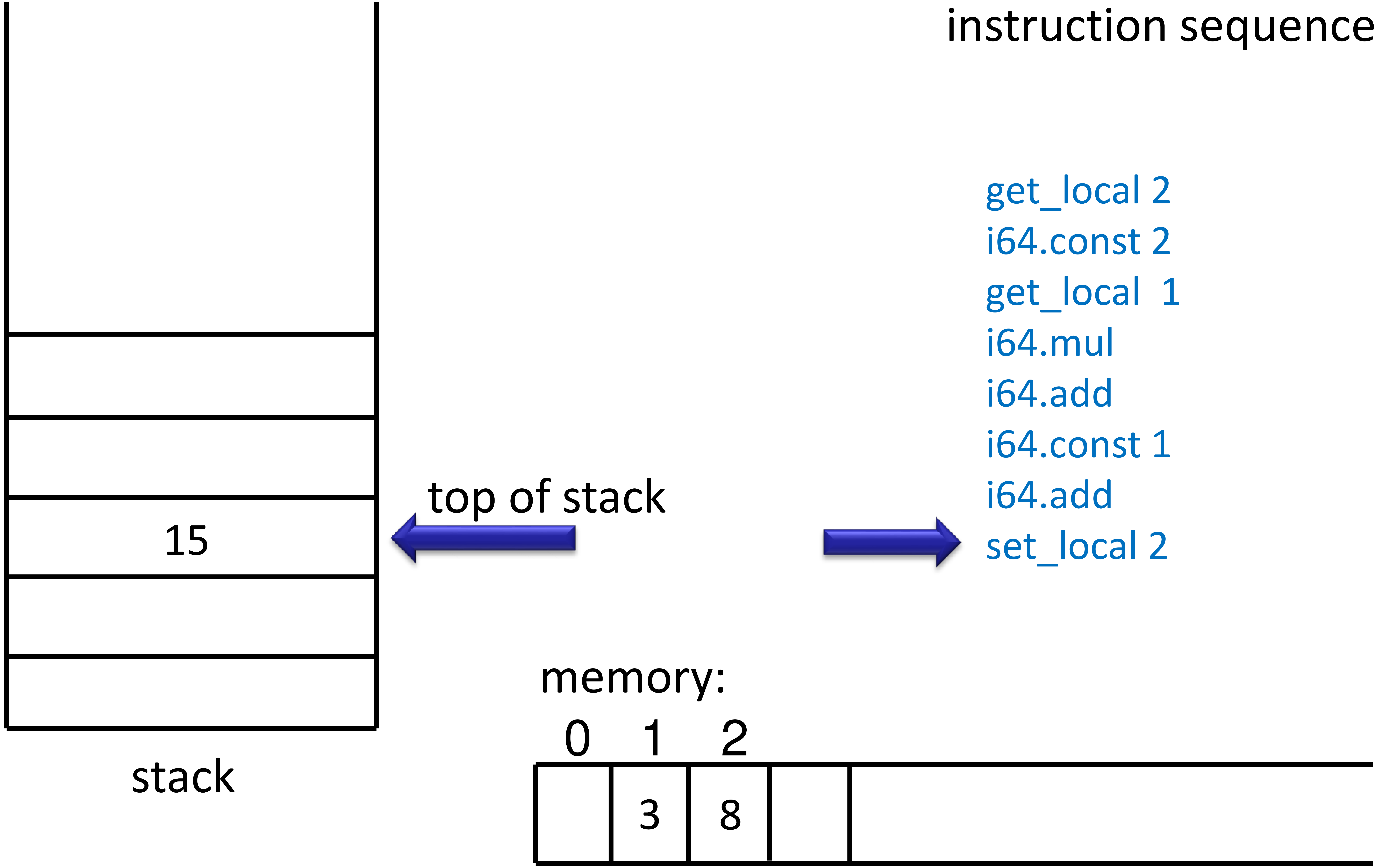
Operands are consumed from stack and put back onto stack



instruction sequence:

```
get_local 2
i64.const 2
get_local  1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2
```

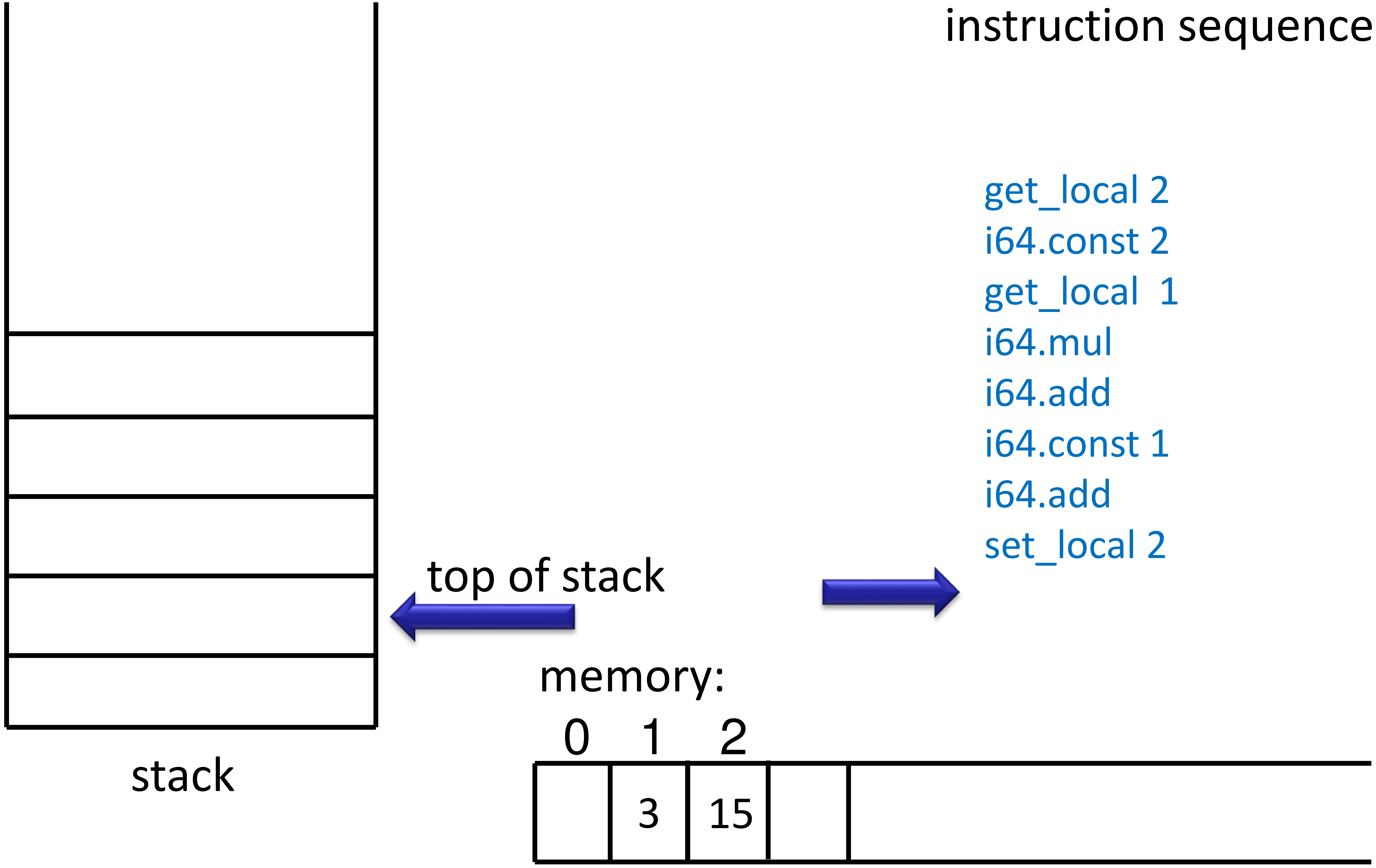
Operands are consumed from stack and put back onto stack



instruction sequence:

get_local 2
i64.const 2
get_local 1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2

Operands are consumed from stack and put back onto stack



instruction sequence:

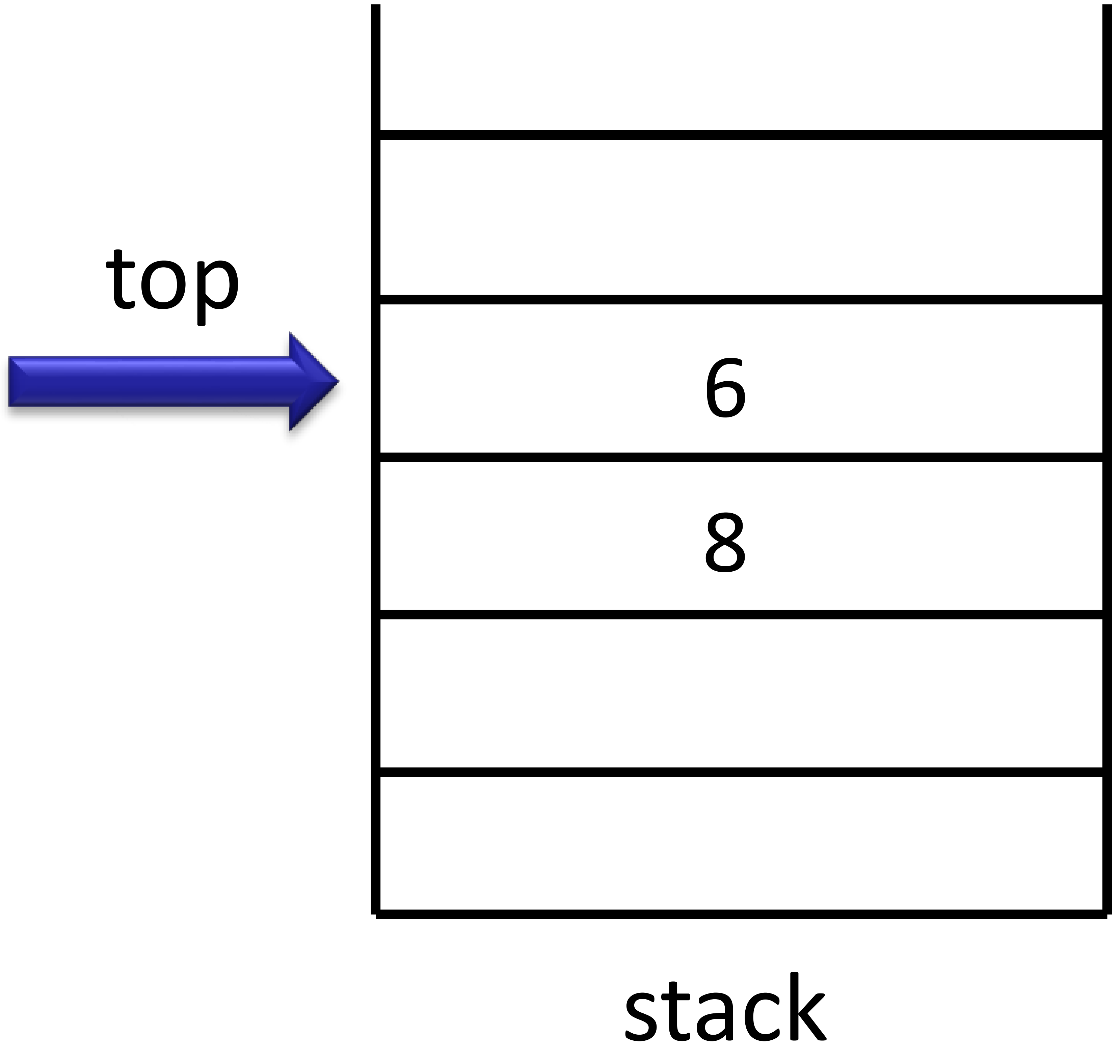
```
get_local 2
i64.const 2
get_local  1
i64.mul
i64.add
i64.const 1
i64.add
set_local 2
```

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 iconst(c) =>
  operand(top + 1) = c   // put given constant 'c' onto stack
  top = top + 1
case Igetlocal(n) =>
  operand(top + 1) = local(n)  // from memory onto stack
  top = top + 1
case Isetlocal(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
```

WebAssembly reference interpreter in ocaml:

https://github.com/WebAssembly/spec/tree/master/interpreter

Selected Instructions

Reading and writing locals (and parameters):

- **get_local**: read the current value of a local variable
- **set_local**: set the current value of a local variable
- **tee_local**: like set_local, but also returns the set value

Arithmetic operations (take args from stack, put result on stack):

i32.add: sign-agnostic addition

i32.sub: sign-agnostic subtraction

i32.mul: sign-agnostic multiplication (lower 32-bits)

i32.div_s: signed division (result is truncated toward zero)

i32.rem_s: signed remainder (result has the sign of the dividend x in x%y)

i32.and: sign-agnostic bitwise and

i32.or: sign-agnostic bitwise inclusive or

i32.xor: sign-agnostic bitwise exclusive or

Comparisons, stack, memory

i32.eq: sign-agnostic compare equal

i32.ne: sign-agnostic compare unequal

i32.lt_s: signed less than

i32.le_s: signed less than or equal

i32.gt_s: signed greater than

i32.ge_s: signed greater than or equal

i32.eqz: compare equal to zero (return 1 if operand is zero, 0 otherwise)

There are also: 64 bit integer operations i64._ and floating point f32._ , f64._

drop: drop top of the stack

i32.const C: put a given constant **C** on the stack

Access to memory (given as one big array):

i32.load: get memory index from stack, load 4 bytes (little endian), put on stack

i32.store: get memory address and value, store value in memory as 4 bytes

Can also load/store small numbers by reading/writing fewer bytes, see http://webassembly.org/docs/semantics/

Example: Area

```
int fact(int a, int b, int c) {
  return ((c+a)*b + c*a) * 2;
}
```

```
(module  (type $type0 (func (param i32 i32 i32)
                                    (result i32)))
  (table 0 anyfunc)  (memory 1)
  (export "memory" memory)
  (export "fact" $func0)
  (func $func0 (param $var0 i32)
                 (param $var1 i32)
                 (param $var2 i32)    (result i32)
  get_local $var2
  get_local $var0
  i32.add
  get_local $var1
  i32.mul
  get_local $var2
  get_local $var0
  i32.mul
  i32.add
  i32.const 1
  i32.shl                 // shift left, i.e.  *2
))
```

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

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 + *

Infix is the only problematic notation and leads to ambiguity

Why is it used in math? Amgiuity 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		
infix	((x + y) + z) + u	x + (y + (z + u))
postfix		

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 { // needs to 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: write 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

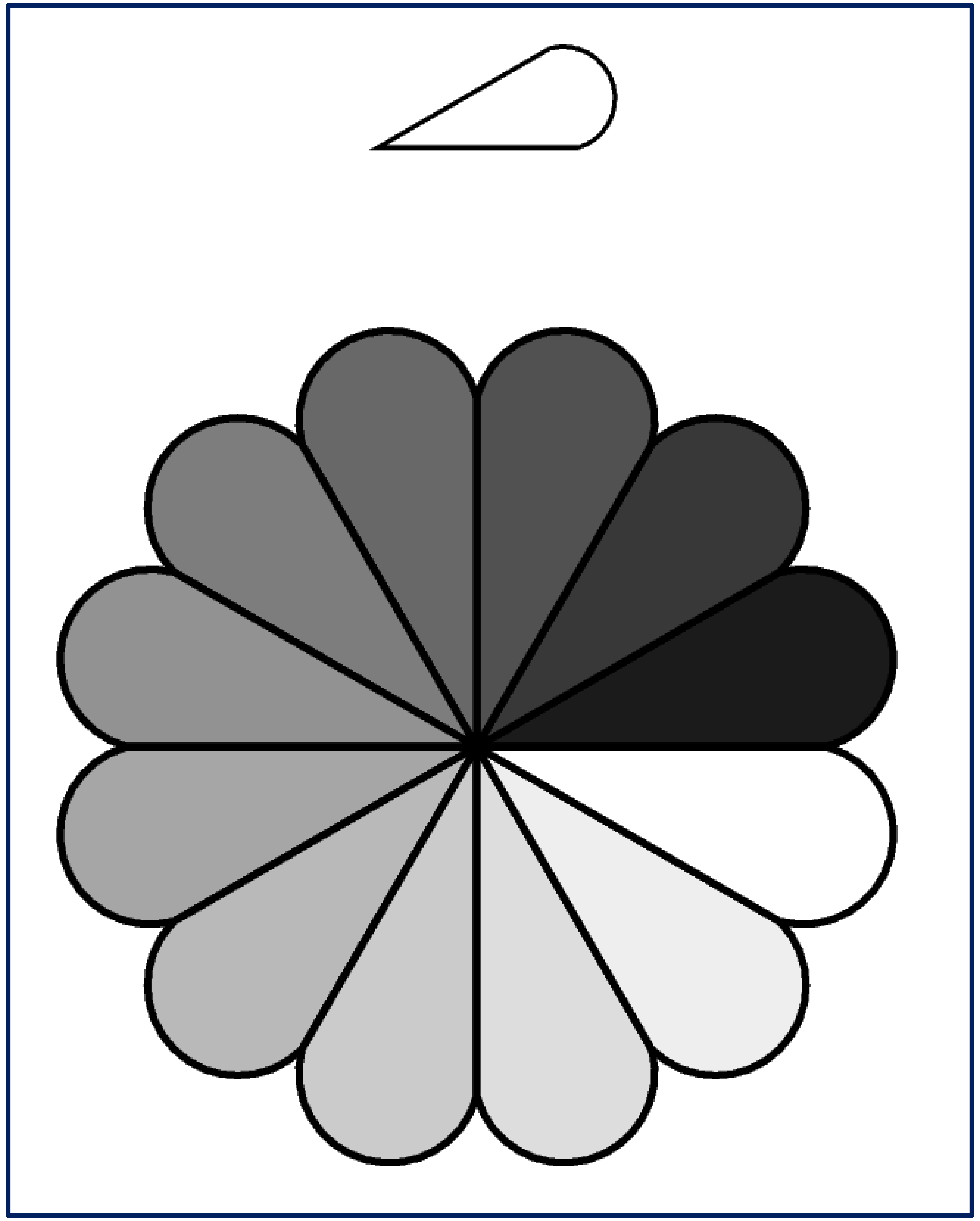
PostScript language tutorial and cookbook
Adobe Systems Incorporated
Reading, MA : Addison Wesley, 1985
ISBN 0-201-10179-3 (pbk.)

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
```

Related: https://en.wikipedia.org/wiki/Concatenative_programming_language

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)
}
```

x -> 3, y -> 4, z -> 5

infix: x*(y+z)

postfix: x y z + *

Run ‘postfix’ for this env

Evaluating Infix Needs Recursion

The recursive interpreter:

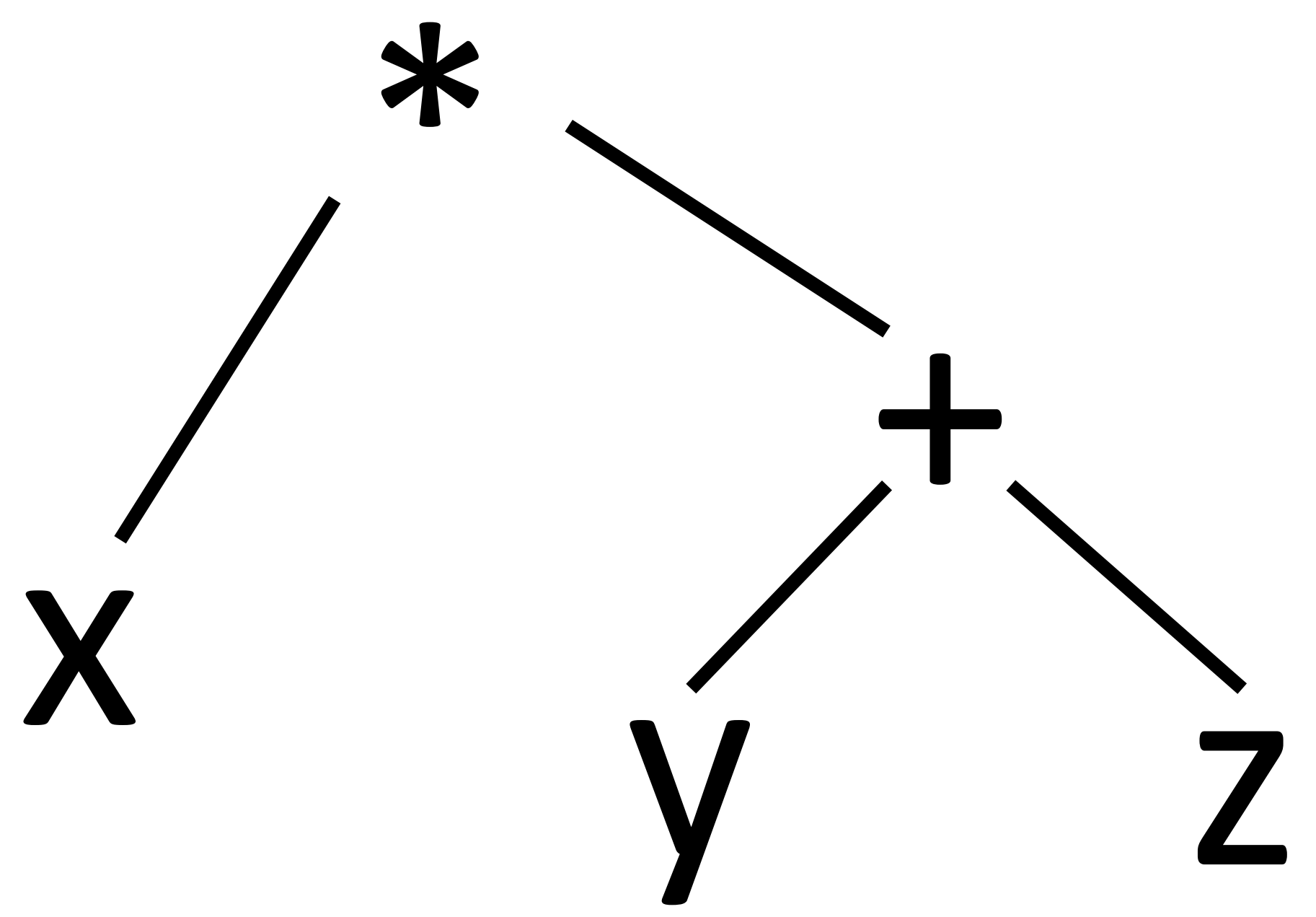
```
def infixEval(env : Map[String,Int], expr : Expr) : Int =
expr match {
   case Var(id) => env(id)
   case Plus(e1,e2) => infix(env,e1) + infix(env,e2)
   case Times(e1,e2) => infix(env,e1) * infix(env,e2)
}
```

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:

get_local 1	x
get_local 2	y
get_local 3	z
i32.add	+
i32.mul	*

Show Tree, Postfix, Code

infix: $(x*y + y*z + x*z)*2$ tree:

postfix: bytecode:

“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(Igetlocal(slotFor(id)))
  case Plus(e1,e2)  => compile(e1) ::: compile(e2) ::: List(Iadd())
  case Times(e1,e2) => compile(e1) ::: compile(e2) ::: List(Imul())
}
```

Local Variables

- Assigning indices (called *slots*) to local variables using function

  slotOf : VarSymbol → {0,1,2,3,…}
- 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(Igetlocal(slotFor(id)))
   …
}
def compileStmt(s : Statmt) : List[Bytecode] = s match {
  // id=e
 case Assign(id,e) => compile(e) ::: List(Iset_local(slotFor(id)))
   …
}
```

Compiler Correctness

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

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

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.

Can prove correctness of entire compiler:

CompCert - A C Compiler whose Correctness has been Formally Verified

CakeML project: https://cakeml.org/

A simple proof with two quantifiers

A simple case of proof for (non-negative int y,x)

$$\forall y \; \forall x \quad P(x,y)$$

is: *let y be arbitrary*, and then fix y throughout the proof.

Suppose that we prove

$$\forall x \; P(x,y)$$

by induction. We end up proving

$P(0, y)$ for some arbitrary y

$P(x,y)$ implies $P(x+1,y)$ for arbitrary x,y

Induction with Quantified Hypothesis

Prove P holds for all non-negative integers x,y:

$\forall x\ \underbrace{\forall y\ \ P(x,y)}_{Q(x)}$ i.e. $\forall x\ Q(x)$

where $Q(x)$ denotes $\forall y\ P(x,y)$

Induction on x means we need to prove:

1. $Q(0)$ that is, $\forall y\ P(0,y)$
2. $Q(x)$ implies $Q(x+1)$
   If $\forall y_1\ P(x,y_1)$ then $\forall y_2\ P(x+1,y_2)$ x,y_2 arbit.

We can instantiate $\forall y_1\ P(x,y_1)$ multiple times when proving that, for any y_2, $P(x,y_2)$ holds

One can instantiate y_1 with y_2 but not only

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

Attempted proof by induction:

exec(env,compile(Times(e1,e2))) ==
exec(env,compile(e1) ::: compile(e2) ::: 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

exec(env,bcodes) == **run**(env,bcodes,List()).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

- run (env,`*` :: L, S ::: List(x1, x2)) == run(env,L, S:::List(x1*x2))
- run (env,`+` :: L, S ::: List(x1, x2)) == run(env,L, S:::List(x1+x2))
- run(env,ILoad(n) :: L, S) == run(env,L, S:::List(env(n)))

By induction one shows:

- run (env,L1 ::: L2,S) == run(env,L2, run(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:

run(env,compile(expr),S) == S:::List(interpret(env,expr))

shorthands:

env – T, compile – C, interpret – I, List(x) - [x]

∀e ∀S run(T,C(e),S) == S:::[I(T,e)]

By induction on e,

```
∀S   run(T,C(e),S) == S:::[I(T,e)]
```

One case (multiplication):

```
run(T,C(Times(e1,e2)),S) ==
run(T,C(e1):::C(e2):::[`*`],S) ==
run(T,[`*`], run(T,C(e2), run(T,C(e1),S) )) ==
run(T,[`*`], run(T,C(e2), S:::[I(T,e1)]) ) ==    (∀S !)
run(T,[`*`], S:::[I(T,e1)]:::[I(T,e2)]) ==
S:::[I(T,e1) * I(T,e2)] ==
S:::[I(T,Times(e1,e2))]
```

Shorthand Notation for Translation

[$e_1 + e_2$ **]** =

[e_1 **]**

[e_2 **]**

add

[$e_1 * e_2$ **]** =

[e_1 **]**

[e_2 **]**

mul

Code Generation for Control Structures

Sequential Composition

How to compile statement sequence?

s1; s2; … ; sN

- Concatenate byte codes for each statement!

```
def compileStmt(e : Stmt) : List[Bytecode] = e match {
  …
  case Sequence(sts) =>
    for { st <- sts;  bcode <- compileStmt(st) }
      yield bcode
}
```

i.e. sts **flatMap** compileStmt

that is: (sts **map** compileStmt) **flatten**

Compiling Control: Example

```
int count(int counter,
          int to,
          int step) {
  int sum = 0;
  do {
    counter = counter + step;
    sum = sum + counter;
  } while (counter < to);
  return sum; }
```

We need to see how to:

- translate boolean expressions
- generate jumps for control

```
(func $func0
 (param $var0 i32) (param $var1 i32)
 (param $var2 i32) (result i32)
 (local $var3 i32)
  i32.const 0
  set_local $var3
  loop $label0
    get_local $var3
    get_local $var0
    get_local $var2
    i32.add
    tee_local $var0
    i32.add
    set_local $var3
    get_local $var0
    get_local $var1
    i32.lt_s
    br_if $label0
  end $label0
  get_local $var3  )
```

Representing Booleans

“All comparison operators yield 32-bit integer results with 1 representing true and 0 representing false.” – WebAssembly spec

Our generated code uses 32 bit int to represent boolean values in: **local variables**, **parameters**, and intermediate **stack values**.

1, representing true

0, representing false

i32.eq: sign-agnostic compare equal

i32.ne: sign-agnostic compare unequal

i32.lt_s: signed less than

i32.le_s: signed less than or equal

i32.gt_s: signed greater than

i32.ge_s: signed greater than or equal

i32.eqz: compare equal to zero (return 1 if operand is zero, 0 otherwise) // not

Truth Values for Relations: Example

```
int test(int x, int y){
  return (x < y);
}
```

```
(func $func0
  (param $var0 i32)
  (param $var1 i32)
  (result i32)

  get_local $var0
  get_local $var1
  i32.lt_s
)
```

Comparisons, Conditionals, Scoped Labels

```
int fun(int x, int y){
  int res = 0;
  if (x < y) {
    res = (y / x);
  } else res = (x / y);
  return res+x+y;
}
```

```
(local $var2 i32)
block $label1 block $label0
  get_local $var0
  get_local $var1
  i32.ge_s
  br_if $label0       // to else branch
  get_local $var1
  get_local $var0
  i32.div_s
  set_local $var2
  br $label1          // done with if
end $label0           // else branch
  get_local $var0
  get_local $var1
  i32.div_s
  set_local $var2
end $label1           // end of if
get_local $var1
get_local $var0
i32.add
get_local $var2
i32.add
```

Main Instructions for Labels

- **block**: the beginning of a block construct, a sequence of instructions with a **label at the end**
- **loop**: a block with a label at the **beginning** which may be used to form loops
- **br**: branch to a given label in an enclosing construct
- → **br_if**: conditionally branch to a given label in an enclosing construct
- **return**: return zero or more values from this function
- **end**: an instruction that marks the end of a block, loop, if, or function

Compiling If Statement

Notation for compilation:

[if (cond**)** tStmt **else** eStmt **]** =
block $nAfter **block** $nElse
[!cond **]**
bf_if $nElse
[tStmt **]**
br $nAfter
end $nElse:
[eStmt **]**
end $nAfter:

```
block $label1 block $label0
 (negated condition code)
 br_if $label0        // to else branch
     (true case code)
 br $label1           // done with if
end $label0           // else branch
     (false case code)
end $label1           // end of if
```

Is there alternative without negating condition?

How to introduce labels

- For forward jumps to $label: use
  block $label
  …
  end $label

- For backward jumps to $label: use
  loop $label
  …
  end $label