CS448h: Lua and Terra

Zach DeVito

Last Time

Do an Image Blur

```
local r =   (a + a:shift(-1,0)
               + a:shift(0,1)
               + a:shift(0,-1)
               + a:shift(1,0)) / 5.0
```

Our Lua implementation: 0.27 MP/s

Naive C loop doing the same thing: 48.2 MP/s

Why?

Last Time

Do an Image Blur

```
local r =   (a + a:shift(-1,0)
               + a:shift(0,1)
               + a:shift(0,-1)
               + a:shift(1,0)) / 5.0
```

Our Lua implementation: 0.27 MP/s

Naive C loop doing the same thing: 48.2 MP/s

Why?

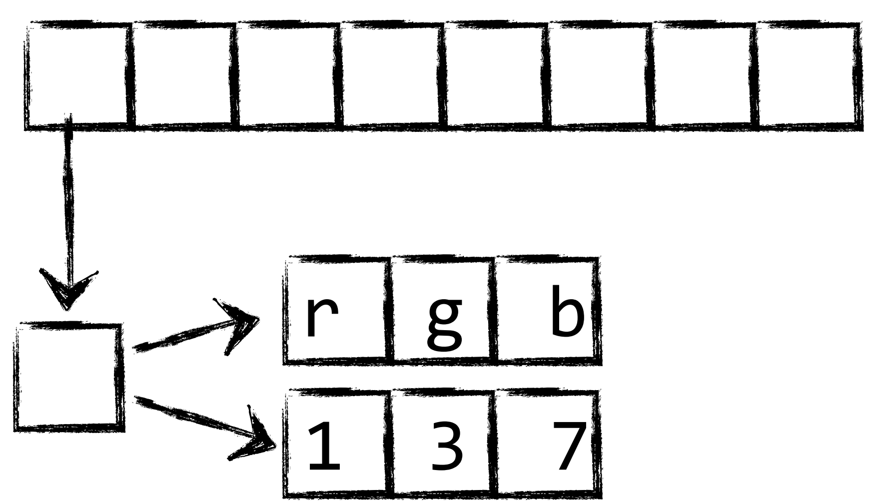
- Our storage of the image is inefficient Lua data structures and operations
- We are doing individual operations on the entire image, the C code just does it in one pass

Inefficient Data Structures and Operators

```
for i = 0, self.width * a - 1 do
    local l,r = self.data[i],rhs.data[i]
    result.data[i] = { r = l.r + r.r, g = l.g + r.g, b = l.b + r.b }
end
```

All hash-table lookups.

Data-layout:



Order of Image operations

```
local r =   (a - a:shift(-1,0)
               - a:shift(0,1)
               - a:shift(0,-1)
               - a:shift(1,0)) / 5.0
```

```
For each pixel:
  shift by -1,0
For each pixel:
  subtract
For each pixel:
  shift by 0,1
For each pixel:
  subtract
For each pixel:
  shift by 0,-1
For each pixel:
  subtract
For each pixel:
  shift by 1,0
For each pixel:
  subtract
```

```
For each pixel:
  set to 5.0
For each pixel:
  divide
```

High level specification is nice but the order of the operations is a really bad idea.

Order of Image operations

```
local r =   (a - a:shift(-1,0)
               - a:shift(0,1)
               - a:shift(0,-1)
               - a:shift(1,0)) / 5.0
```

```
For each pixel:
  shift by -1,0
For each pixel:
  subtract
For each pixel:
  shift by 0,1
For each pixel:
  subtract
For each pixel:
  shift by 0,-1
For each pixel:
  subtract
For each pixel:
  shift by 1,0
For each pixel:
  subtract
```

```
For each pixel:
  set to 5.0
For each pixel:
  divide
```

High level specification is nice but the order of the operations is a really bad idea.

How bad is it?

Estimating Performance

Physical limits of your computer:

- Bandwidth to main memory (~20--30GB/s)
- FLOPs (~30--60 GFLOPS double precision per core)

Each shift:
  2 passes (read,write) x 4
Each math op:
  3 passes (read,read,write) x 5
Each constant:
  1 pass (write) x 1

24 passes

Single Loop: 2 passes (read, write)

Some of these inefficiencies are fixable in Lua itself:

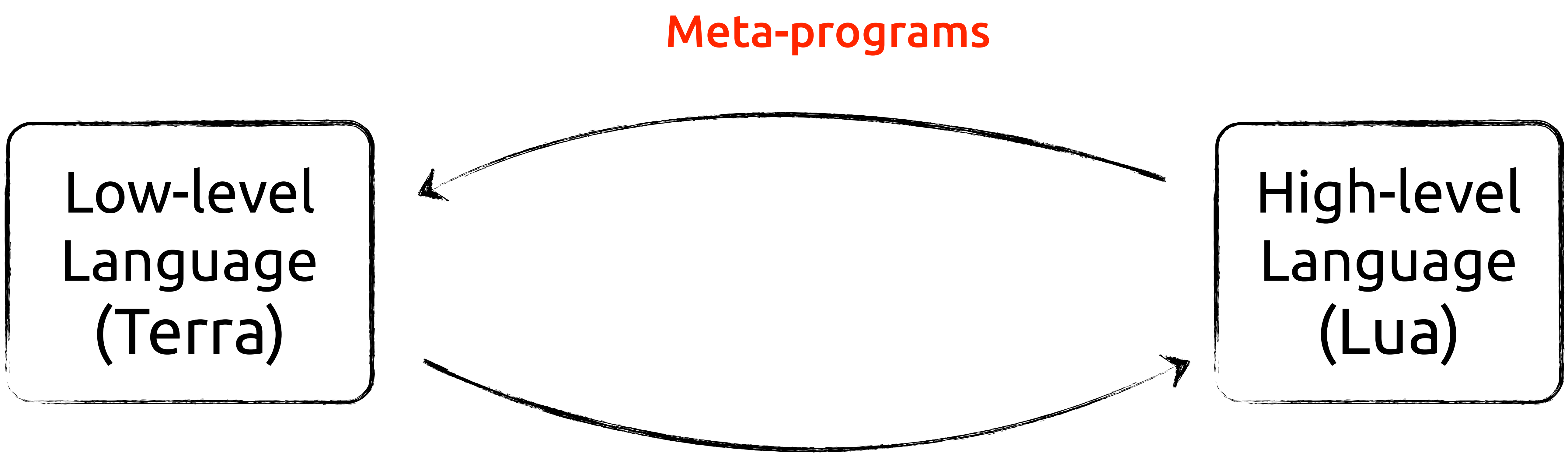
- Use a flat RGB array for data.

Others would be difficult to fix:

- Get code into a single loop, but still keep the high-level representation.
- Use only three bytes for each pixel

Specify the operation in a **high-level language**,
then transform it into code in a **low-level language**.

Our approach: A Two-language design



Combining High- and Low-level Languages

Web Server Development

Database Language (C/C++),

ORM layer, Business Logic (Ruby)

Scientific Computing

MATLAB, C++/FORTRAN

Game Programming

Shading Language (OpenGL), Scripting Language (Lua),

Engine Language (C++)

Integrating existing languages is problematic

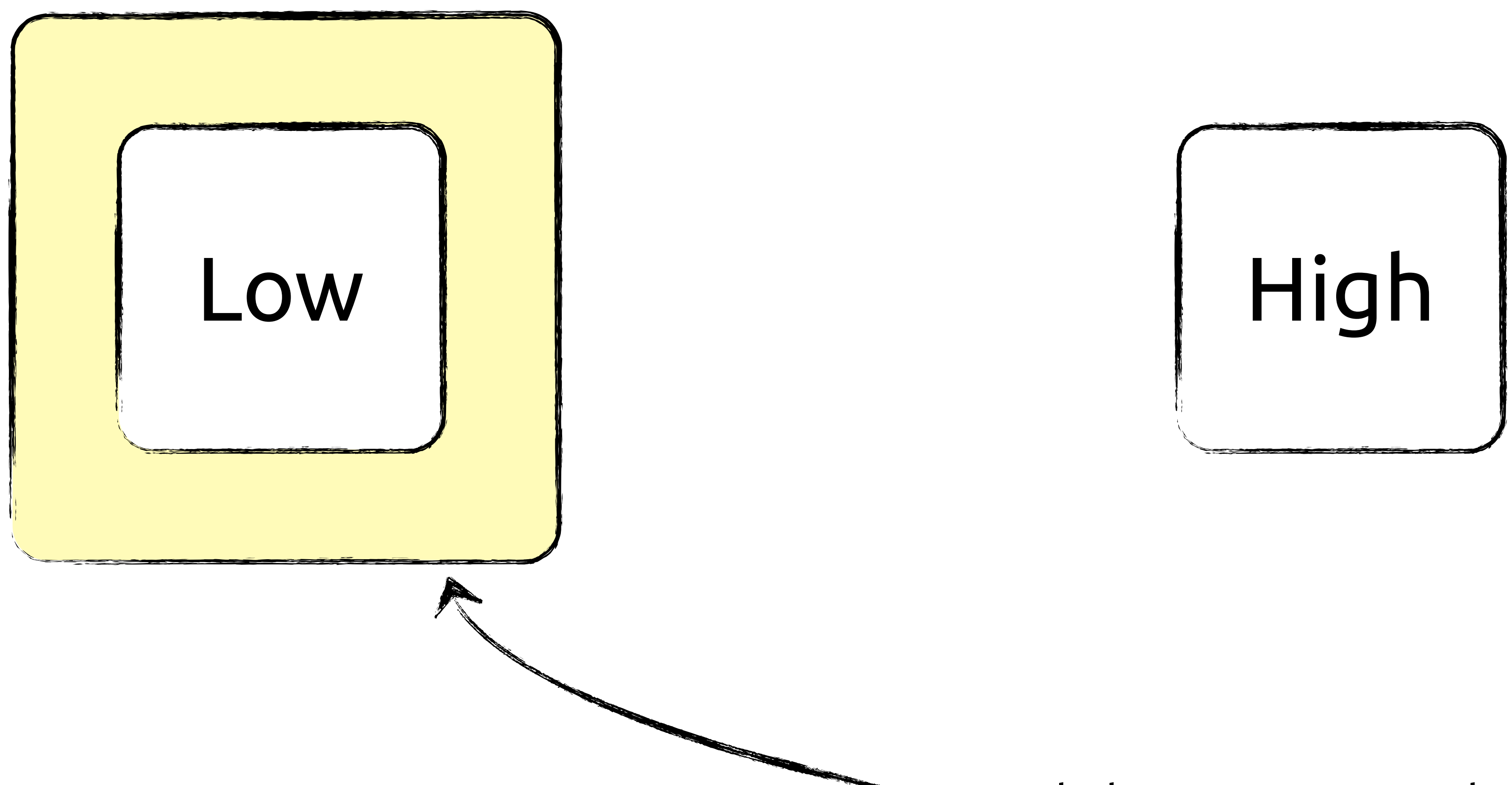
Low

High

Integrating existing languages is problematic

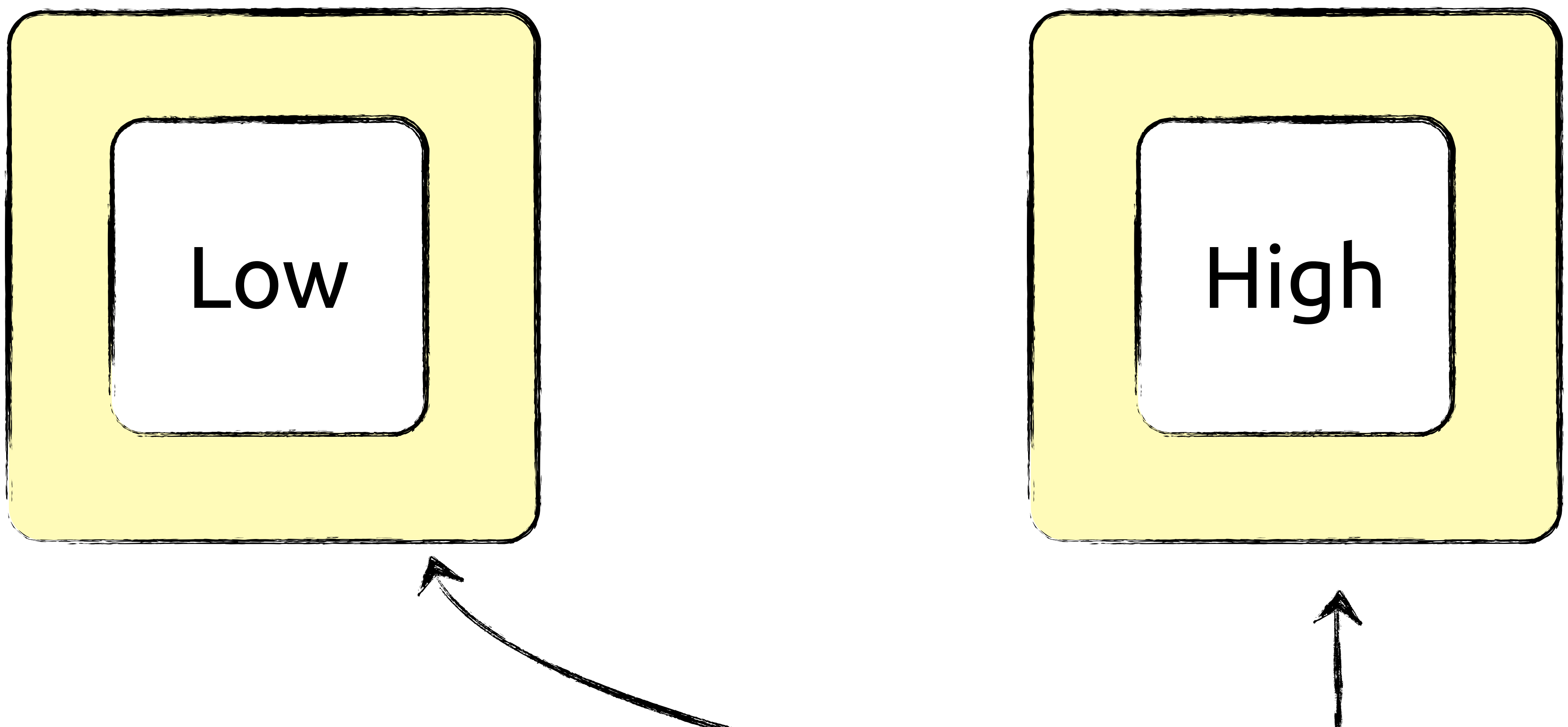
Each language makes tradeoffs,
adding extra complexity

Integrating existing languages is problematic



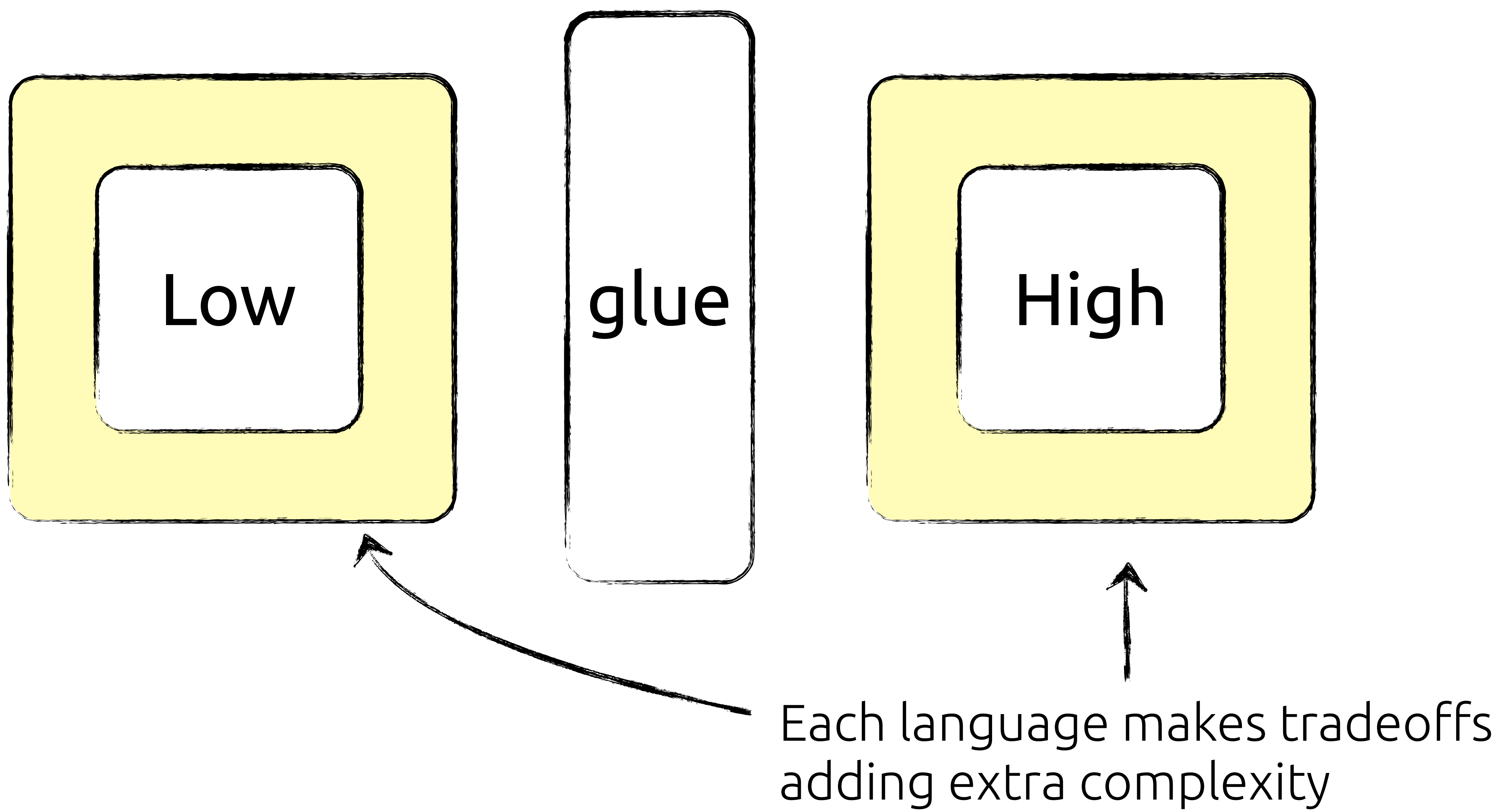
Each language makes tradeoffs, adding extra complexity

Integrating existing languages is problematic

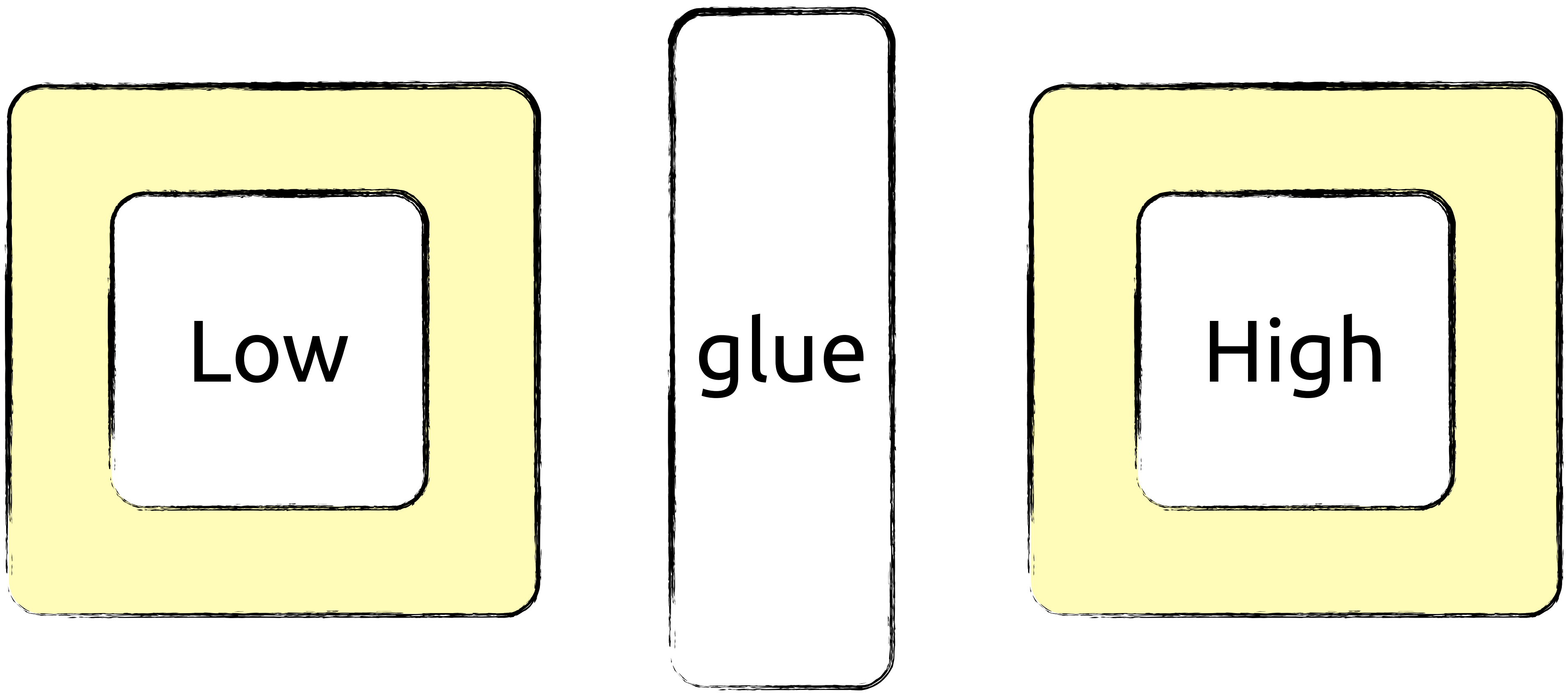


Each language makes tradeoffs,
adding extra complexity

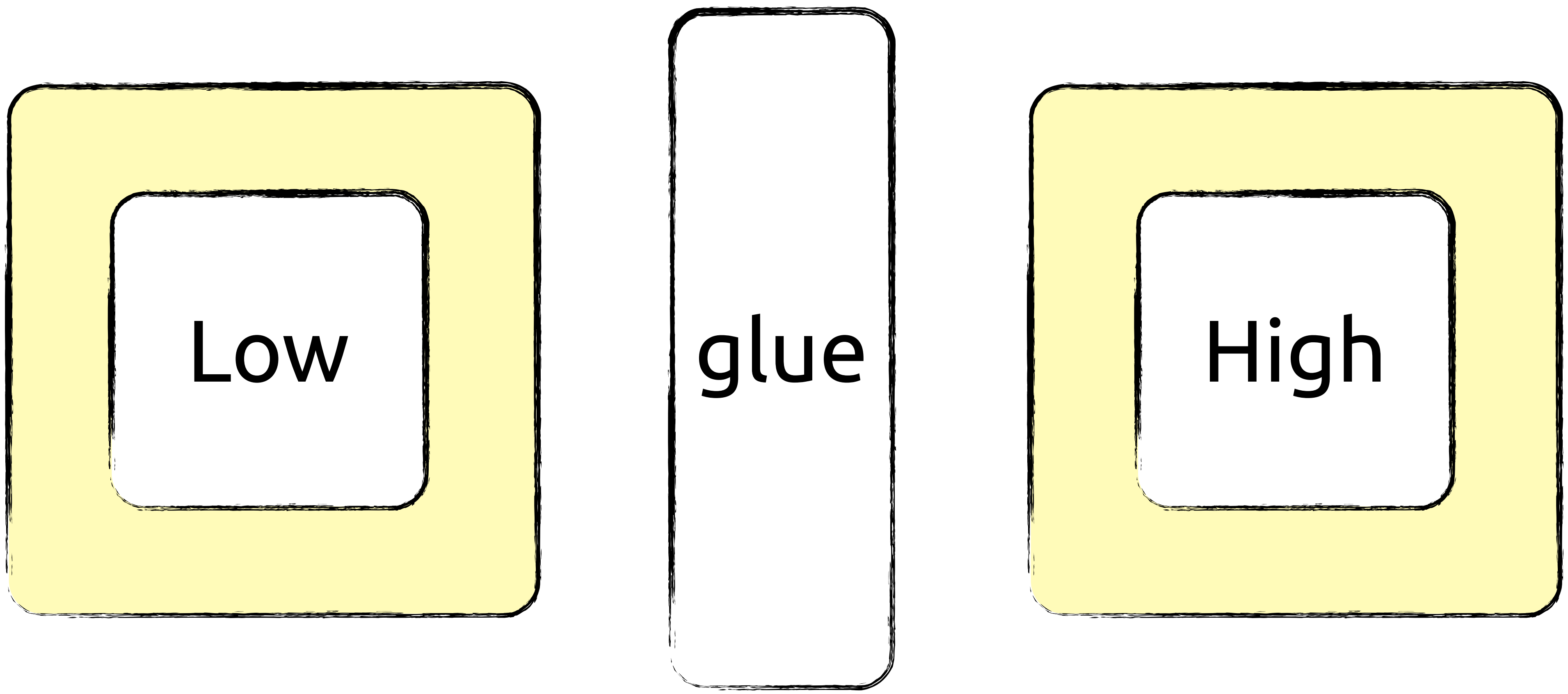
Integrating existing languages is problematic



We should design with the expectation of two languages!

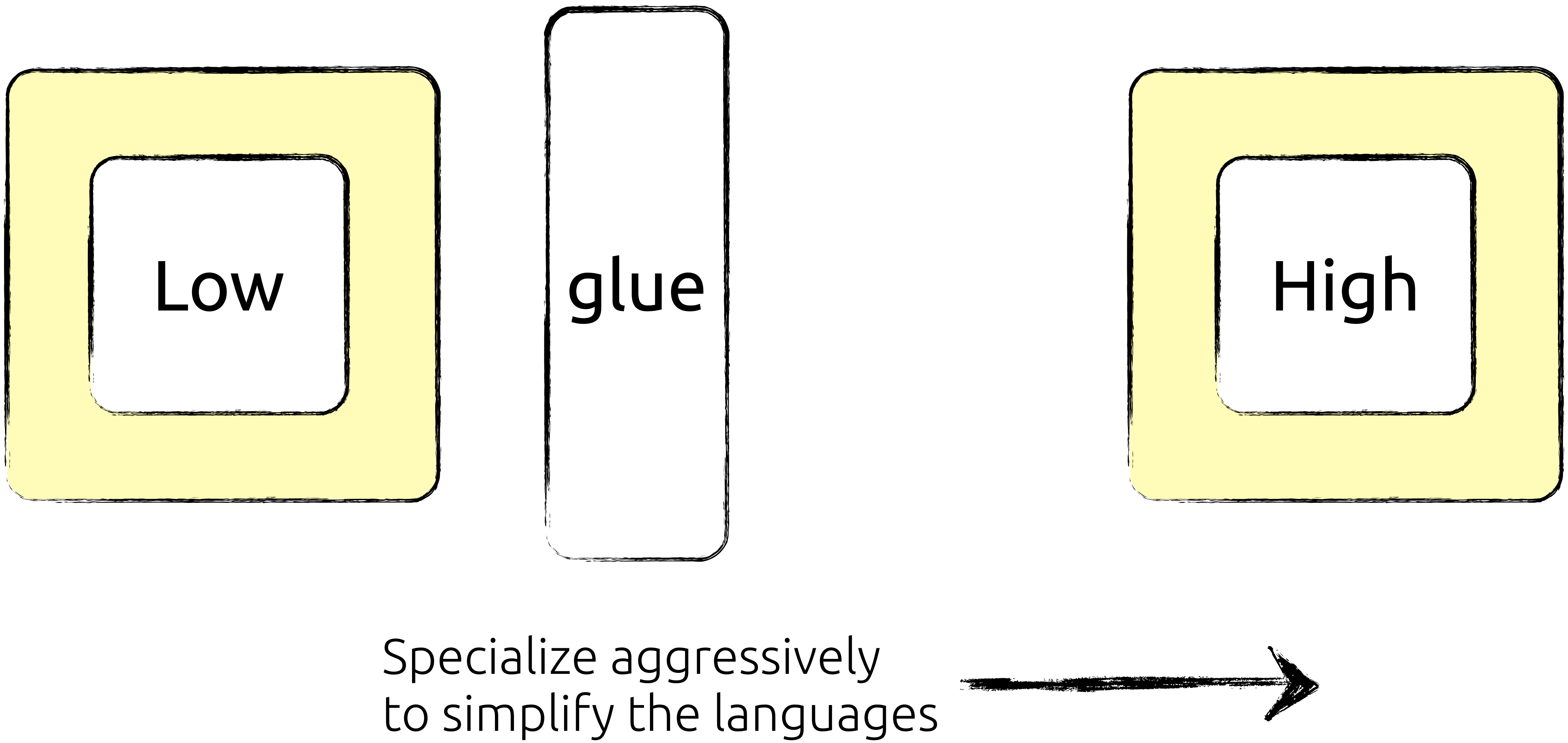


We should design with the expectation of two languages!

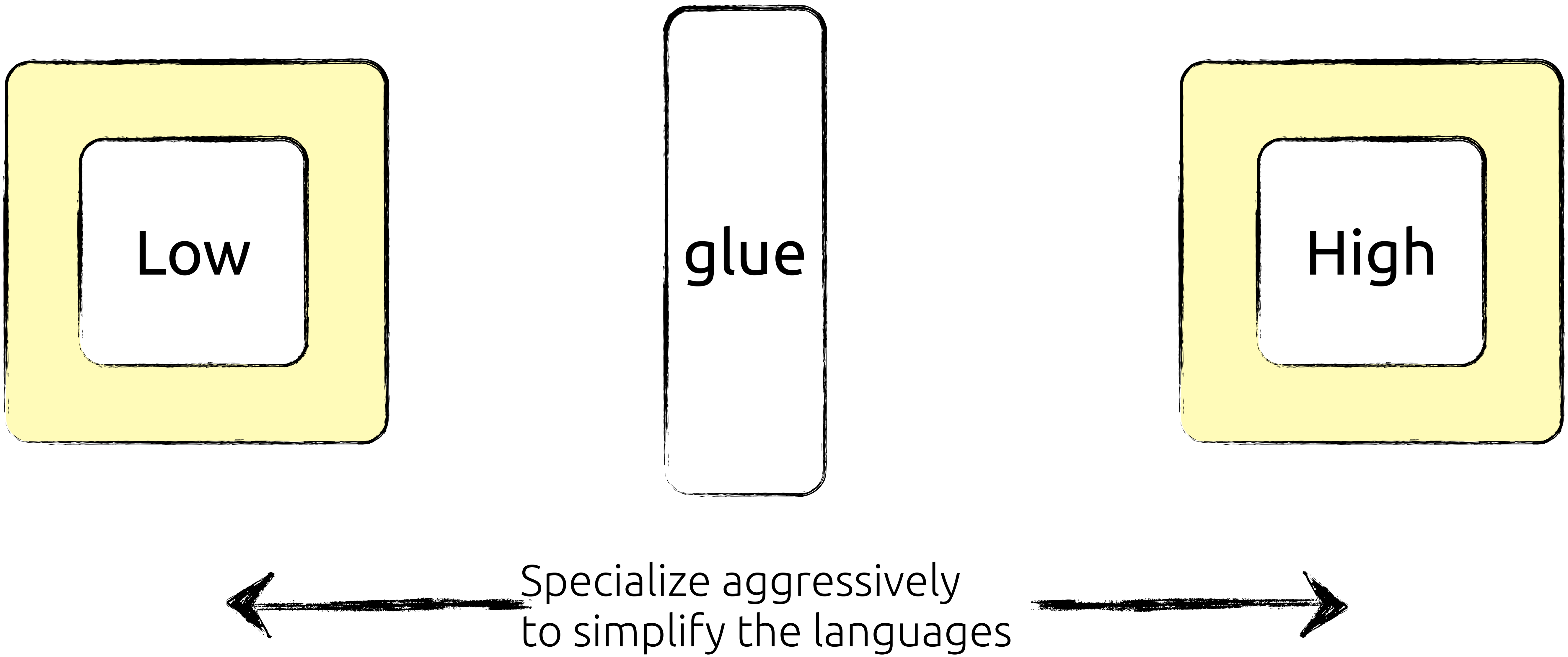


Specialize aggressively
to simplify the languages

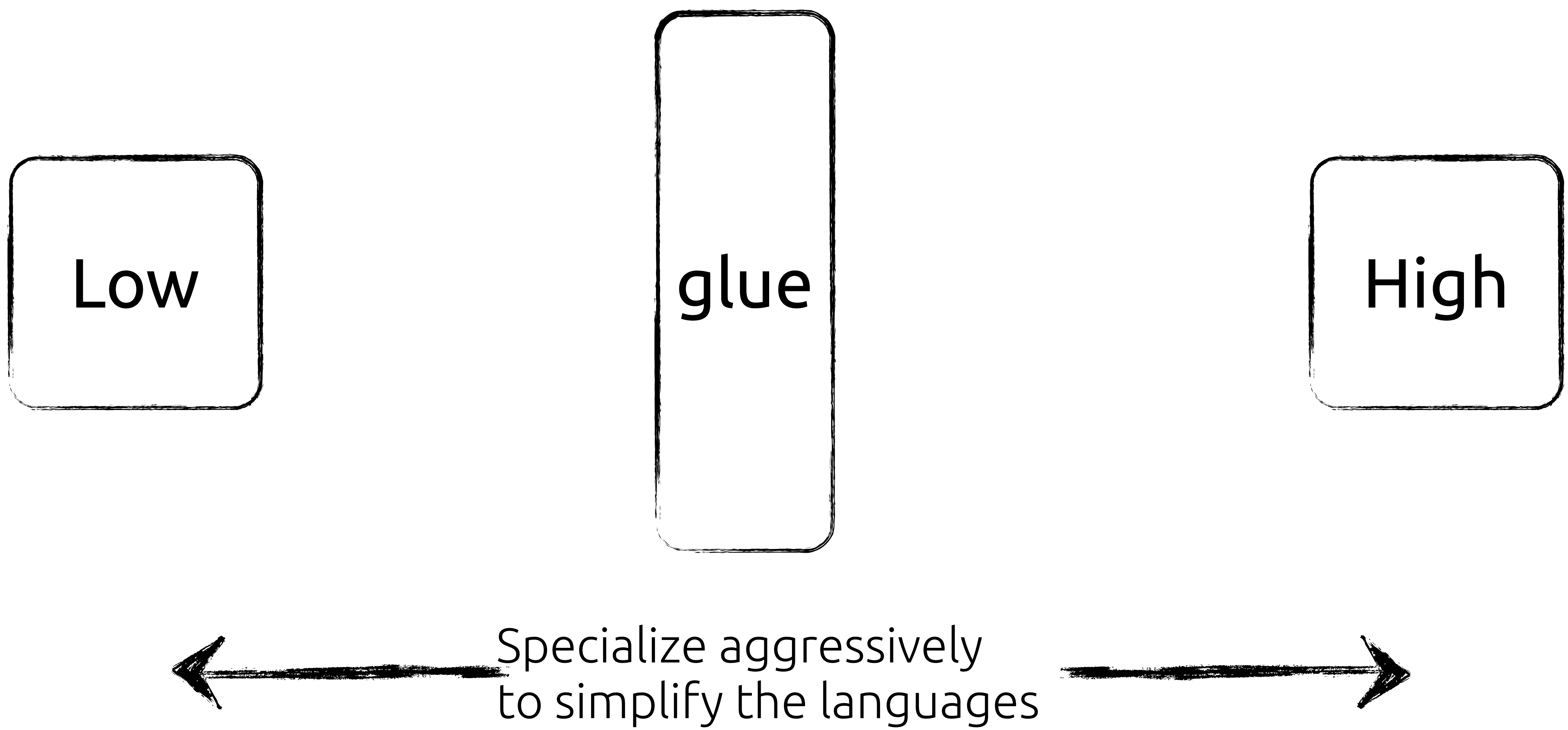
We should design with the expectation of two languages!



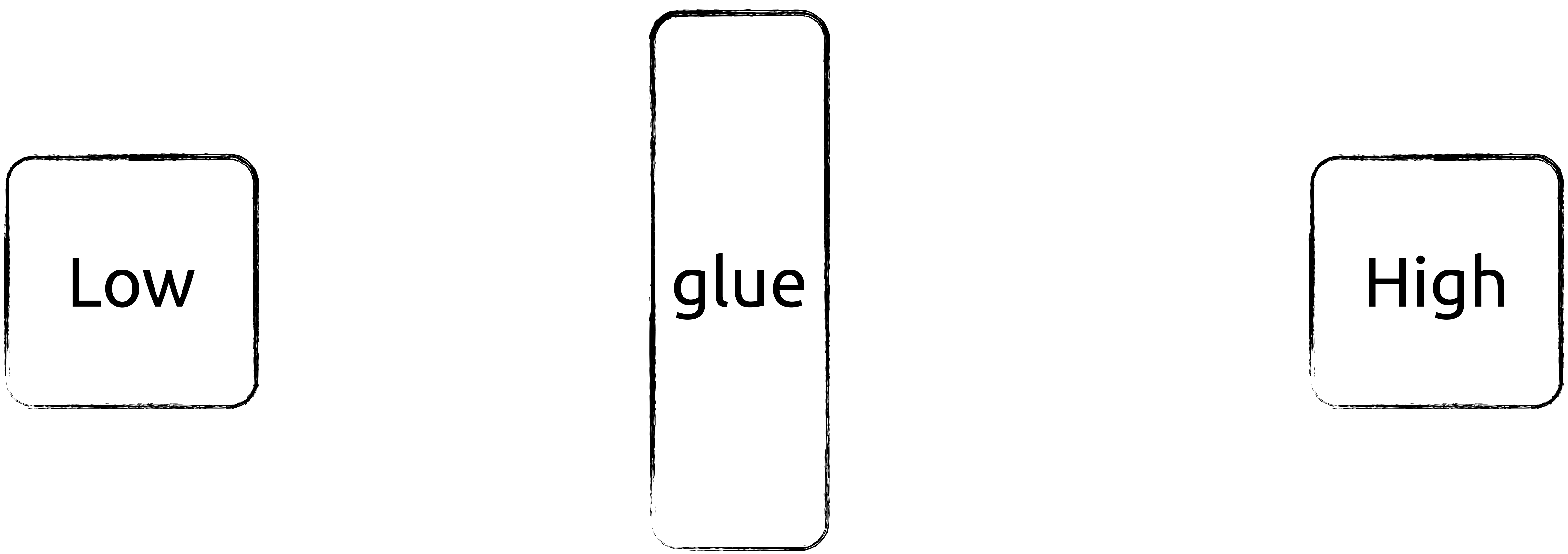
We should design with the expectation of two languages!



We should design with the expectation of two languages!

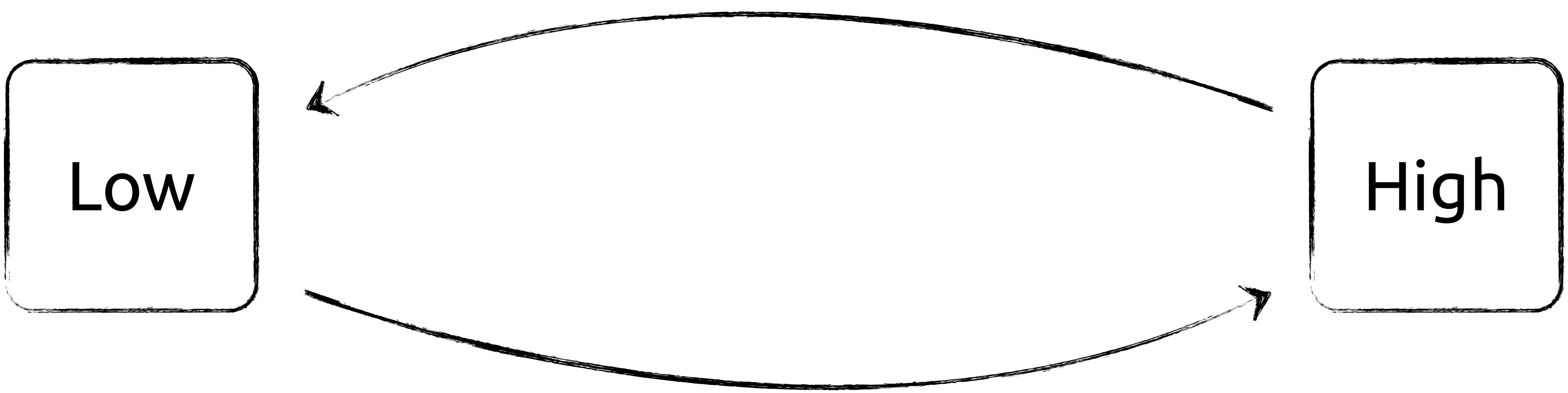


Designing with the expectation of Two Languages



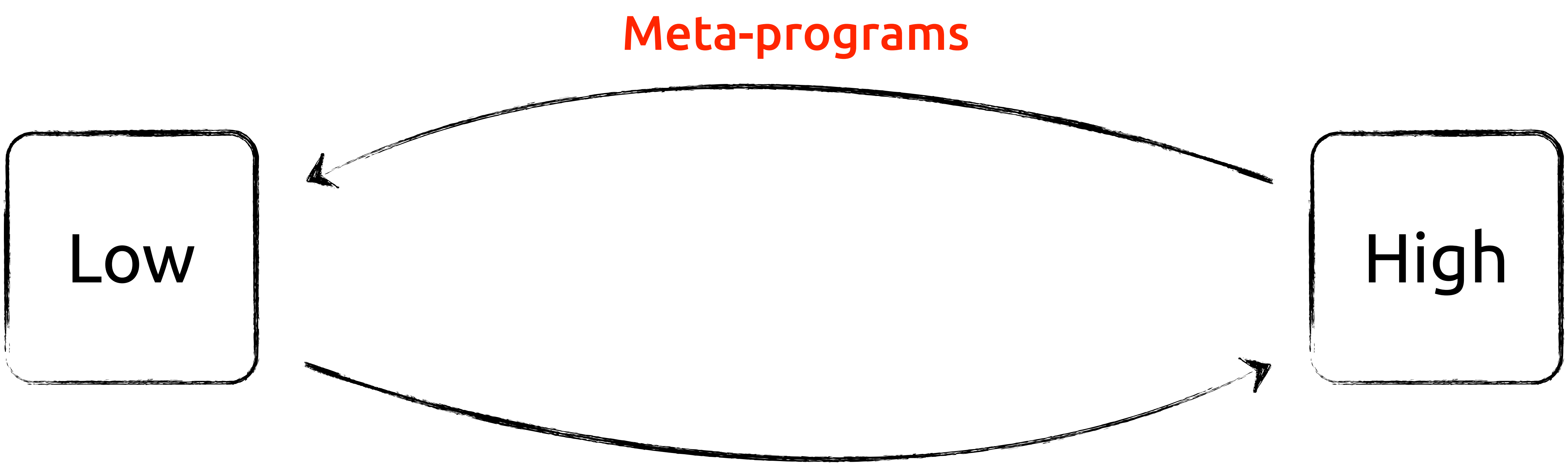
Make each language aware of the other to remove glue.

Designing with the expectation of Two Languages



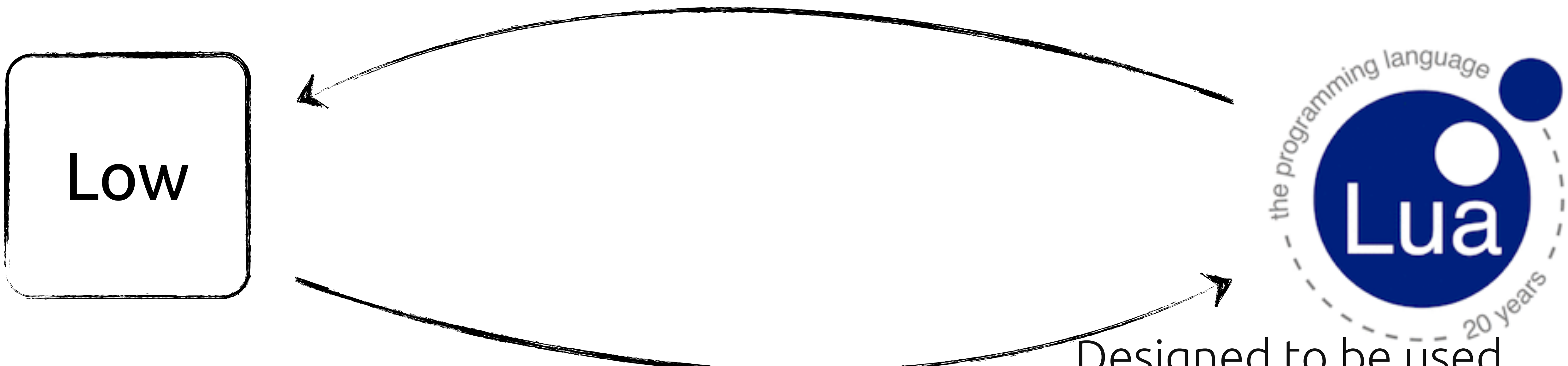
Make each language aware of the other to remove glue.

Designing with the expectation of Two Languages



Meta-program the low-level language
to produce high-performance
code from concise descriptions.

Designing with the expectation of Two Languages



Designed to be used with
low-level languages such as C
[Ierusalimschy et al. 11]

Designing with the expectation of Two Languages

New low-level language
designed
to work with high-level
languages

Designed to be used
with
low-level languages such
as C
[Ierusalimschy et al. 11]

Example: Lua

```
--this is a comment.
--top level is Lua code:
function add(a,b)
    return a + b
end
print(add(3,4)) --7
```

Example: Lua + Terra

```
--this is a comment.
--top level is Lua code:
function add(a,b)
    return a + b
end
print(add(3,4)) --7

--terra introduces a low-level terra function
terra addt(a : int, b : int) : int
    return a + b
end

print(addt(3,4)) --7
```

Terra function called from Lua

Types and semantics are similar to C

```
struct FloatArray {
    data: &float;
    N : int;
}

--get an element from the array
terra FloatArray:get(i : int) : float
   return self.data[i]
end
```

Types and semantics are similar to C

Aggregate type

```
struct FloatArray {
    data: &float;
    N : int;
}


--get an element from the array
terra FloatArray:get(i : int) : float
   return self.data[i]
end
```

Types and semantics are similar to C

Aggregate type

```
struct FloatArray {
    data: &float;
    N : int;
}
```

Pointer

```
--get an element from the array
terra FloatArray:get(i : int) : float
   return self.data[i]
end
```

Types and semantics are similar to C

Aggregate type

```
struct FloatArray {
    data: &float;
    N : int;
}
```

Pointer

```
--get an element from the array
terra FloatArray:get(i : int) : float
   return self.data[i]
end
```

Method declaration (sugar)

Terra’s Design

Compartmentalized Runtimes

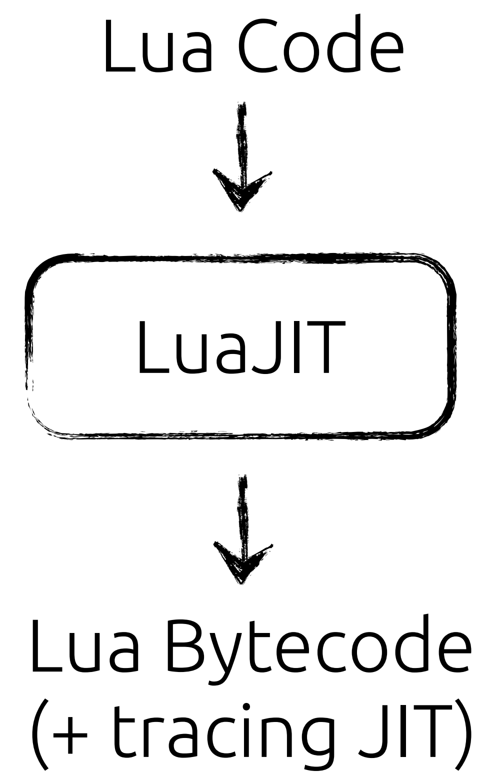
Terra Runtime

Lua Runtime

Compartmentalized Runtimes

Terra Runtime

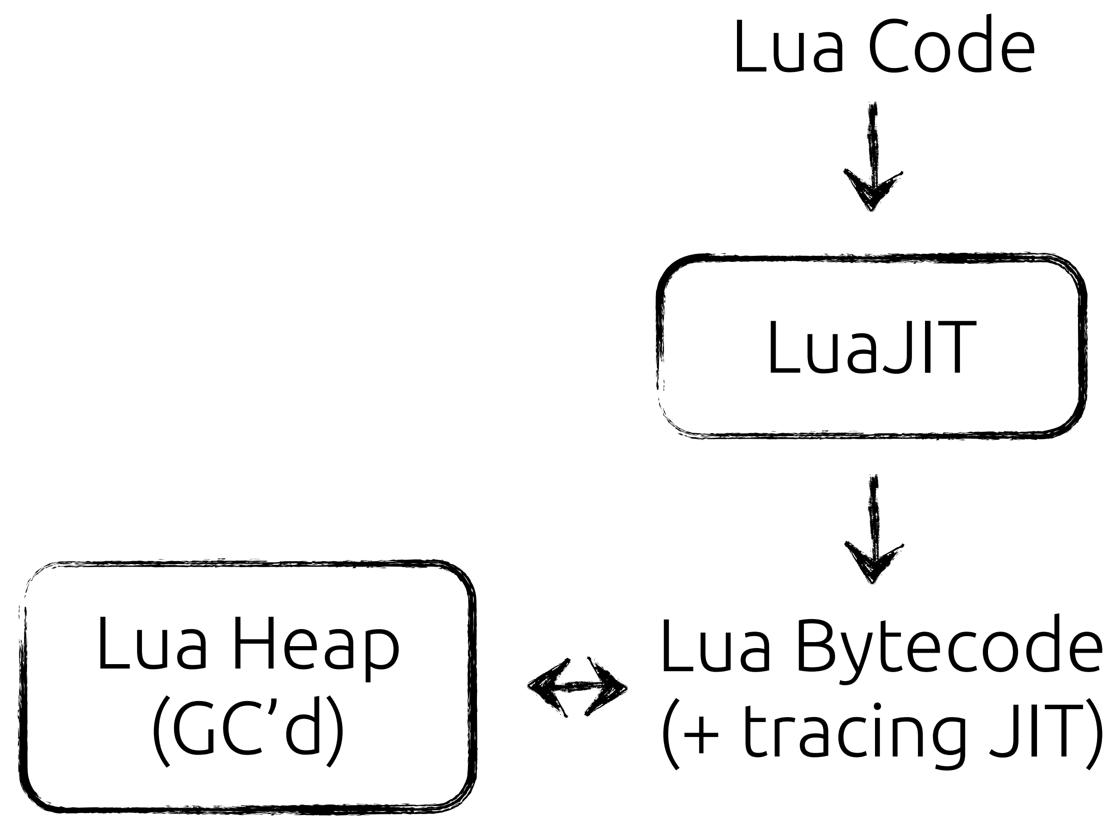
Lua Runtime



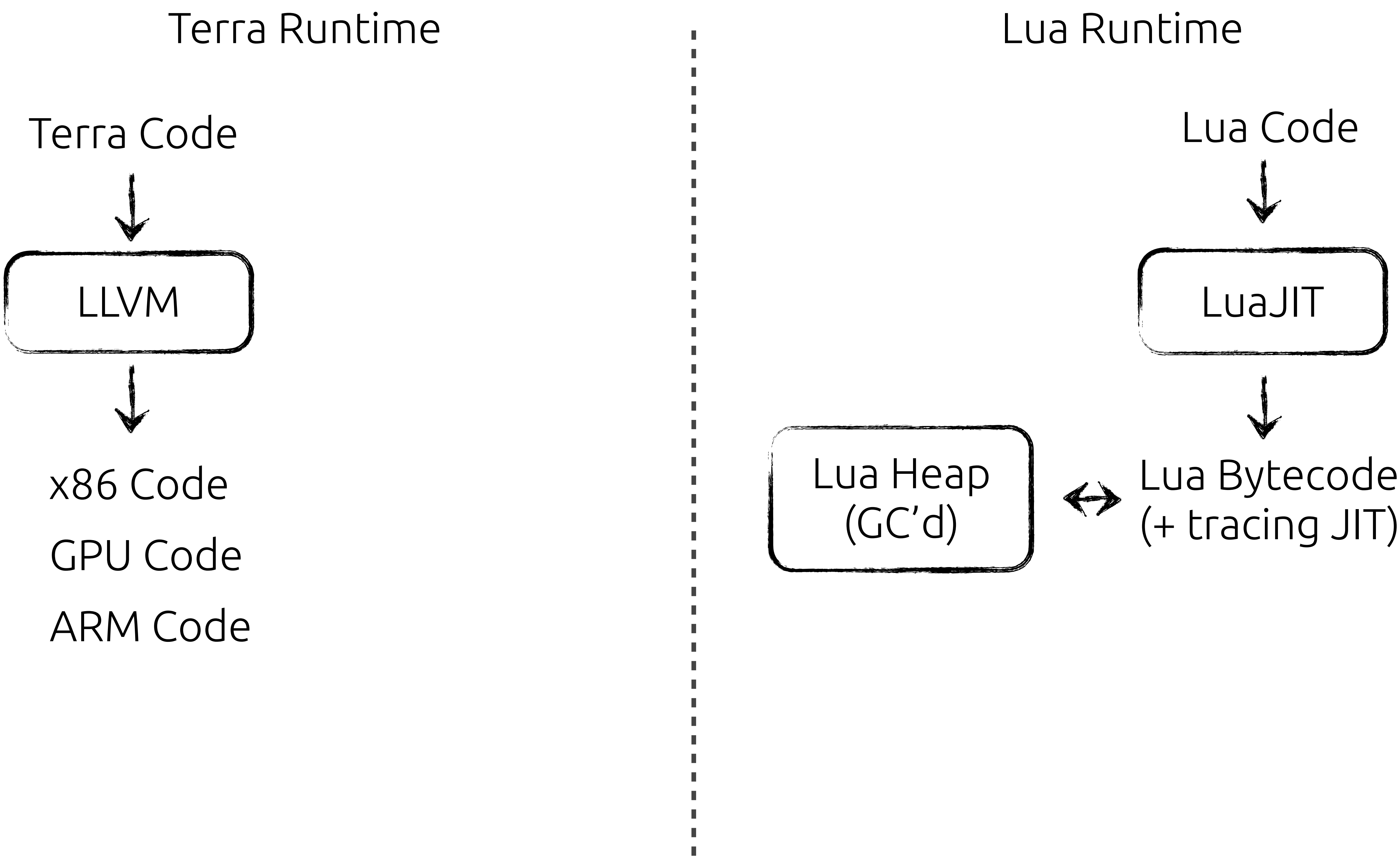
Compartmentalized Runtimes

Terra Runtime

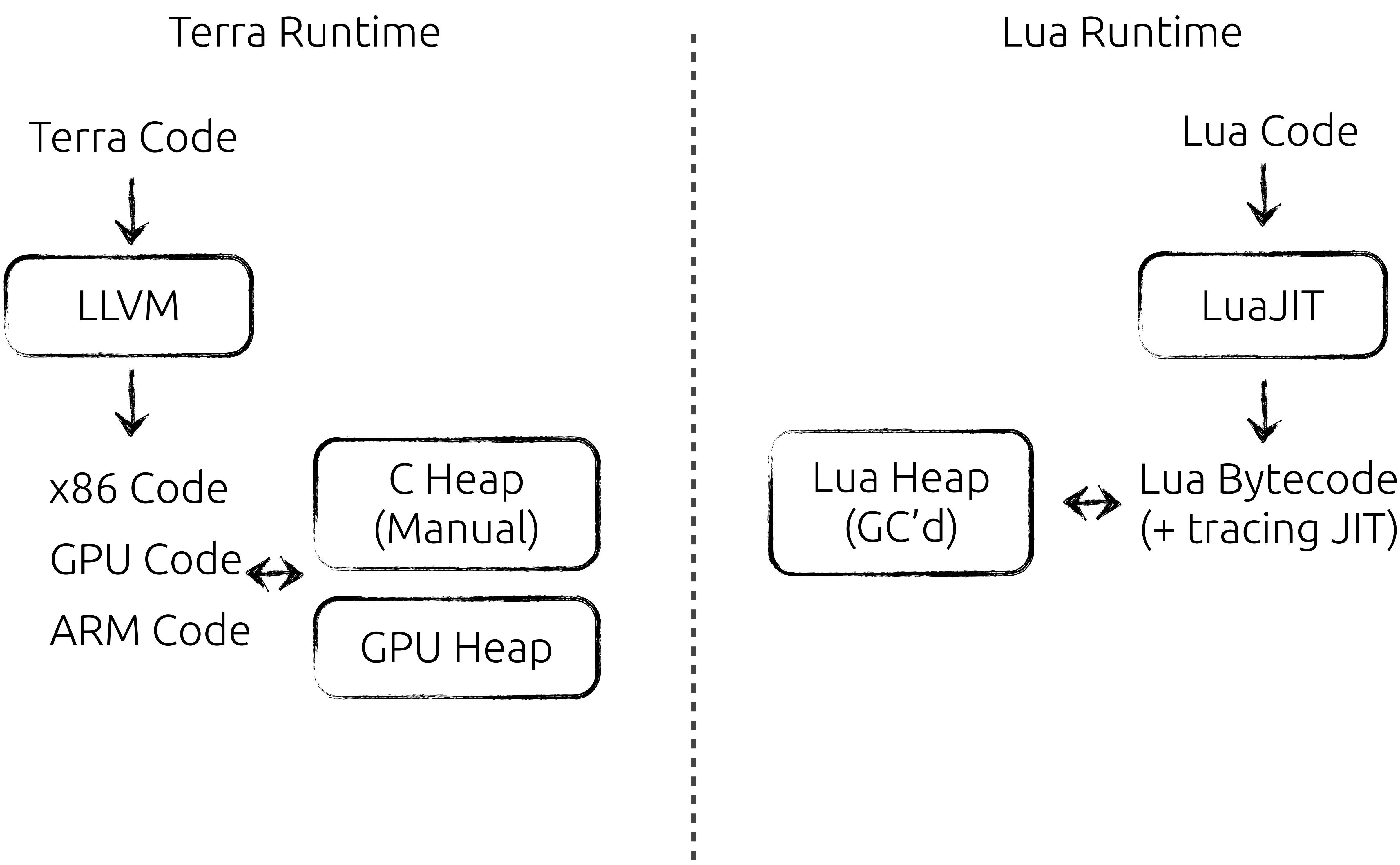
Lua Runtime



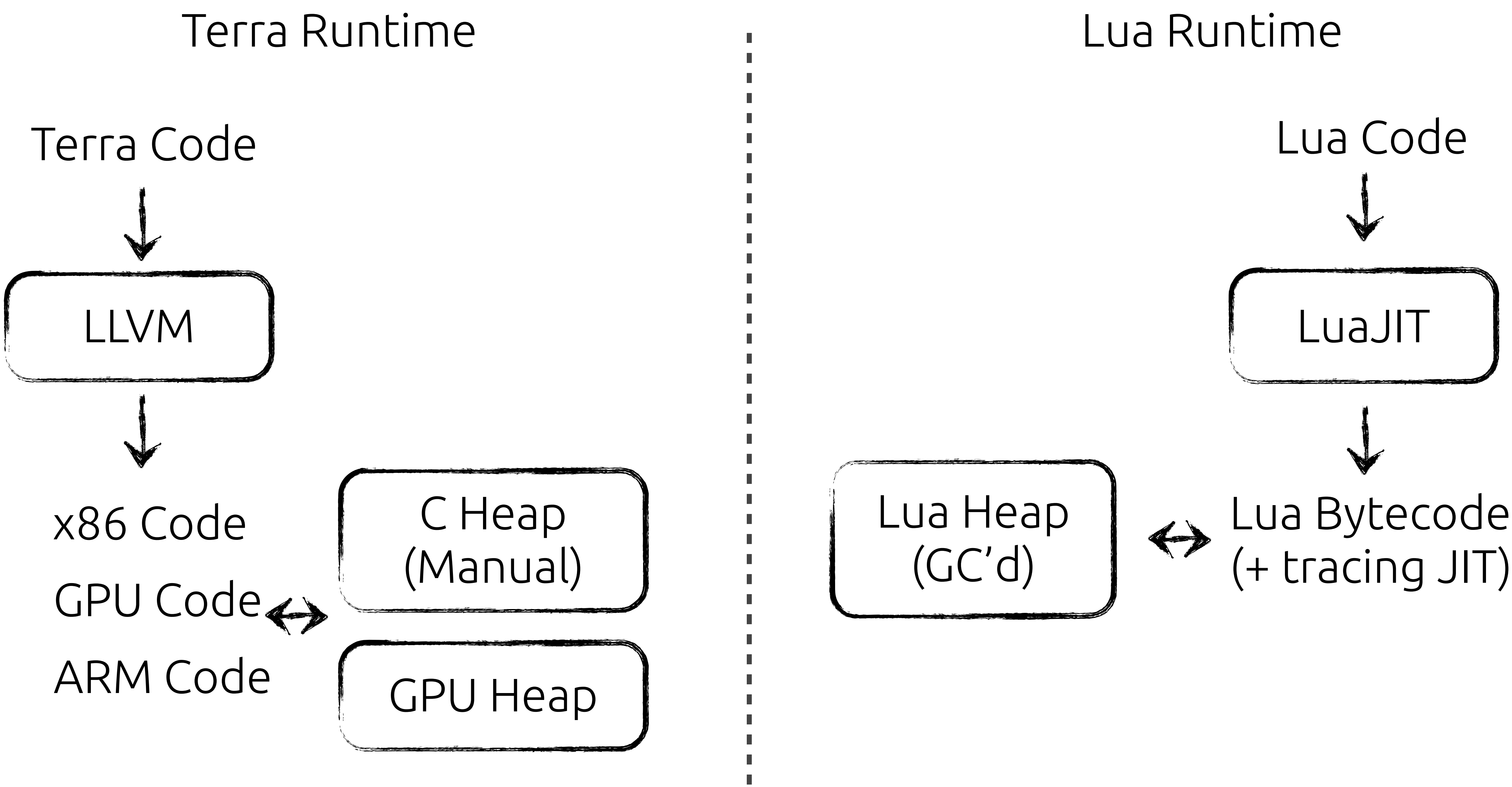
Compartmentalized Runtimes



Compartmentalized Runtimes



Compartmentalized Runtimes



Separation ensures Terra can always produce fast code.

Clean interface between languages

Clean interface between languages

```
terra addt(a : int, b : int) : int
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
   return addt(a,1)
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
   return addt(a,1)
end
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
   return addt(a,1)
end
```

When called from one another, values are translated from one language

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
   return addt(a,1)
end
```

When called from one another, values are translated from one language to another using rules adapted from LuaJIT's FFI

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
   return addt(a,1)
end
```

When called from one another, values are translated from one language to another using rules adapted from LuaJIT's FFI

```
print(addt(1,2)) -- on call: lua number -> int
```

Clean interface between languages

```
terra addt(a : int, b : int) : int
    return a + b
end
```

Terra functions are first class Lua values:

```
print(addt)
> <terra function>
```

Terra uses Lua's lexical environment to resolve symbols:

```
terra add1(a : int) : int
   return addt(a,1)
end
```

When called from one another, values are translated from one language to another using rules adapted from LuaJIT's FFI

```
print(addt(1,2)) -- on call: lua number -> int
                 -- on return: number -> int
```

Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Ex. Templating:

```
local struct ArrayType {
  data : &float;
  N : int;
}
terra ArrayType:get(i: int) : float
  return self.data[i]
end
```

Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Ex. Templating:

```
local struct ArrayType {
  data : &ElemType;
  N : int;
}
terra ArrayType:get(i: int) : ElemType
  return self.data[i]
end
```

Meta-programming

All Terra entities (types, functions, expressions, symbols) are first-class Lua values.

Ex. Templating:

```
function Array(ElemType)
  local struct ArrayType {
    data : &ElemType;
    N : int;
  }
  terra ArrayType:get(i: int) : ElemType
    return self.data[i]
  end
  return ArrayType
end
FloatArray = Array(float)
```

Terra is meta-programmed from Lua using
multi-stage programming (e.g., from MetaOCaml)

```
function gen_square(x)
  return `x * x
end
```

```
terra mse(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end
```

Terra is meta-programmed from Lua using **multi-stage programming** (e.g., from MetaOCaml)

```
function gen_square(x)
  return `x * x
end
```

In Lua, a **quotation** creates a Terra expression.

Like a `"string literal"` for code.

```
terra mse(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end
```

Terra is meta-programmed from Lua using **multi-stage programming** (e.g., from MetaOCaml)

```
function gen_square(x)
  return `x * x
end
```

In Lua, a **quotation** creates a Terra expression.

Like a `“string literal”` for code.

```
terra mse(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end
```

In Terra, an **escape** splices the value of a Lua expression into Terra code.

Like a string interpolation operator `“hello, %s”`

Evaluation Semantics

```
print(“lua execution”)

function gen_square(x)
  return `x * x
end



terra sqd(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end


print(mse(3,2))
```

Evaluation Semantics

```
print("lua execution")
> lua execution
function gen_square(x)
  return `x * x
end
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

```
terra sqd(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end
```

```
print(mse(3,2))
```

Evaluation Semantics

```
print("lua execution")
> lua execution
function gen_square(x)
  return `x * x
end
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float)
  return [gen_square(a)] - [gen_square(b)]
end


print(mse(3,2))
```

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float): float
  return [        `a * a ] - [gen_square(b)]
end


print(mse(3,2))
```

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float)
  return [      `a * a ] - [ `b * b      ]
end


print(mse(3,2))
```

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.

```
terra sqd(a: float, b: float)
  return      a * a   -    b * b
end


print(mse(3,2))
```

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end

terra sqd(a: float, b: float)
  return          a * a    -    b * b
end

print(mse(3,2))
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.

3. The Terra function is **evaluated as Terra**

Evaluation Semantics

```
print("lua execution")

function gen_square(x)
  return `x * x
end

terra sqd(a: float, b: float)
  return          a * a    -     b * b
end

print(mse(3,2))
> 5
```

1. Lua code **evaluates** normally until it reaches a Terra function or quote expression

2. The Terra expression is **specialized**, by evaluating all *escaped* Lua expressions.

3. The Terra function is **evaluated as Terra**

Backwards compatibility with C

```
local C = terralib.includec("stdio.h")
-- or for more than one header:
local C = terralib.includecstring [[
#include<stdio.h>
#include<stdlib.h>
]]

-- C is now a Lua table of Terra wrapper functions
-- for each C function:
C.printf("hello, world\n") -- Terra called from Lua
terra hello()
  C.printf("hello, world\n") -- Terra called from Terra
end
```

Try to put all of your C includes into one call because each call is very expensive since it spins up a C compiler.

Pointers and using C's heap

```
var a : int = 1
var pa : &int = &a
@pa = 4
var b = @pa
var b2 = pa[0] -- same
```

```
-- To allocate data in Terra, use C's malloc:
C = terralib.includec("stdlib.h")
terra doit()
    var a = [&int](C.malloc(sizeof(int) * 2))
    @a,@(a+1) = 1,2
    a[0] = 1 -- syntax sugar
end
```

`void*` in C is equivalent to `&opaque` in Terra

Pointers and using C's heap

```
var a : int = 1
var pa : &int = &a
@pa = 4
var b = @pa
var b2 = pa[0] -- same

-- To allocate data in Terra, use C's malloc:
C = terralib.includec("stdlib.h")
terra doit()
    var a = [&int](C.malloc(sizeof(int) * 2))
    @a,@(a+1) = 1,2
    a[0] = 1 -- syntax sugar
end
```

A cast

`void*` in C is equivalent to **&opaque** in Terra

Back to our image processing example

```
local r =   (a + a:shift(-1,0)
               + a:shift(0,1)
               + a:shift(0,-1)
               + a:shift(1,0)) / 5.0
```

Allocate data using Terra types rather than Lua

Even when using Lua, you can create handles to Terra types. These are usually called "cdata" in plain LuaJIT. (http://luajit.org/ext_ffi.html)

```
local function alloc_image_data(w,h)
    local data = C.malloc(3*w*h)
    return terralib.cast(&uint8,data)
end
local function loadppm(filename)
    ...
    local data = alloc_image_data(image.width,image.height)
    for i = 0,image.width*image.height - 1 do
        data[3*i],data[3*i+1],data[3*i+2] =
           parseNumber(),parseNumber(),parseNumber()
    end
    ...
end
```

Using an intermediate representation

Lua data structure that represents the computation we want to do:

```
(a + a:shift(1,0))/2

-- use operator overload to build the table
-- 'result'
local load_a = { kind = "load", data = <cdata> }
local shift = { kind = "shift", value = load,
                sx = 1, sy = 0 }
local add = { kind = "+", lhs = load_a, rhs = shift }
local const = { kind = "const", value = 2 }
local div = { kind = "/", lhs = add, const }
```

Building Our IR

Each image object internally tracks its IR

```
function image:shift(sx,sy)
    local width,height = self.width,self.height
    local result = image.new(width,height)
    result.tree = { kind = "shift", sx = sx, sy = sy,
                    value = self.tree }
    return result
end
```

Making an Image

A function turns an image represented by IR into a concrete ‘reified’ image:

```
function image:reify()
    local result = image.new(self.width,self.height)
    result.tree = { kind = "load",
                    data = alloc_image_data(self.width,self.height)
                  }

    local compiled_function =
       compile_image_ir(self.width,self.height,self.tree)

    compiled_function(result.tree.data)
    return result
end
```

Compiling our Image IR

```
local function compile_image_ir(W,H,tree)
    local function gen_tree(tree,x,y,c)
        ...
    end
    local terra body(data : &uint8)
        for y = 0,H do
          for x = 0,W do
            for c = 0,3 do
              data[3*(y*W + x) + c] = [ gen_tree(tree,x,y,c) ]
            end
          end
        end
    end
    return body
end
```

```
-- a helper function
local terra load_data(data : &uint8, x: int, y: int, c: int): float
    if x < 0 or x >= W and y < 0 or y >= H then
        return 0.f
    end
    return data[3*(y*W + x) + c]
end
```

```
-- a helper function
local terra load_data(data : &uint8, x: int, y: int, c: int): float
    if x < 0 or x >= W and y < 0 or y >= H then
        return 0.f
    end
    return data[3*(y*W + x) + c]
end

local function gen_tree(tree,x,y,c)
    if tree.kind == "const" then
        return `float(tree.value)
    elseif tree.kind == "load" then
        return `load_data(tree.data,x,y,c)
    elseif tree.kind == "+" then
        local lhs = gen_tree(tree.lhs,x,y,c)
        local rhs = gen_tree(tree.rhs,x,y,c)
        return `lhs + rhs
    ...
    elseif tree.kind == "shift" then
        local xn,yn = `x + tree.sx,`y + tree.sy
        return gen_tree(tree.value,xn,yn,c)
    end
end
```

Results

Our Lua implementation: 0.27 MP/s

Naive C loop doing the same thing: **48.2 MP/s**

Our Terra loop: **39.1 MP/s**

(Still slower by a bit because the C loop was smarter about bounds checking.)

Results

Our Lua implementation: 0.27 MP/s

Naive C loop doing the same thing: **48.2 MP/s**

Our Terra loop: **39.1 MP/s**

(Still slower by a bit because the C loop was smarter about bounds checking.)

Question: how can we do better?

Terra in Details

Resources for learning Terra

terralang.org

- Getting Started Guide
- API reference (includes more detailed descriptions)
- Research papers

Meta-programming Details

Quotations:

```
local short_quote = `3 + 4 --only an expression
local long_quote = quote -- can include expressions
                     C.printf("hi\n")
                     var a = 4
                   in 3 + a end
```

Escapes:

```
terra my_function()
  -- short escape
  var a = [ short_quote ]
  escape -- long escape
    for i = 1,10 do
      emit quote
         C.printf("hi %d",[i])
      end
    end
  end
end
```

Meta-program Anything

Multiple Expressions:

```
local short_quotes = {`3 + 4,`5+6 }
terra returntwo()
  a_function_with_two_arguments(short_quotes) -- pastes both
  return short_quotes -- returns both as a tuple
end
```

Multiple Statements:

```
local hi = quote C.printf("hi\n") end
local stuff = {hi,hi}
terra chatty()
  [hi]
end
```

Meta-program Anything

Use a variable before the quote that defines it:

```
local a = symbol(int,"a") -- type and name are optional
local addone = quote
  a = a + 1
end
terra useit()
  var [a] = 0 -- don't make new 'a', define the symbol a
  [addone]
  return a -- 1
end
```

Multiple arguments to a function:

```
local args = { symbol(), symbol(), symbol()}

terra useit(another_arg : int, [args])
  return another_arg + [args[1]] + [args[2]] + [args[3]]
end
```

Meta-program Anything

Field and method names:

```
struct Complex {
  real : float
  imag : float
}
-- or, via meta-programming:
local entries = { {“real”, float}, {“imag”, float} }
Complex = terralib.types.newstruct(“Complex”)
Complex.entries = entries

terra Complex:add(rhs : Complex) : Complex
    return {self.real + rhs.real, self.imag + rhs.imag}
end

local string_add,string_imag = “add”,”imag”
terra use_complex(c : Complex)
  var c2 = c:[string_add](c)
  return c2.[string_imag]
end
```

Variables

This is actually a Lua expression

```
local myluavalue = 6
terra foo()
  var b : float = 1.f -- type explicitly specified
  var a = 1.0 -- double type inferred from RHS
  var c : int, d = 3,4
  var d = myluavalue -- myluavalue is constant
end

-- Globally accessible Terra variable
local myglobal = global(int,3)
terra setglobal()
   myglobal = 4
end
terra getglobal()
  return myglobal
end
```

Most of the time, you will not need to use global variables.
Instead Terra objects can be store in Lua values and passed as arguments when necessary.

Control Flow: If

Must be booleans

```
if a or b and not c then
    C.printf("then\n")
elseif c then
    C.printf("elseif\n")
else
    C.printf("else\n")
end
```

Loops

```
var a = 0
while a < 10 do
    C.printf("loop\n")
    a = a + 1
end

repeat
    a = a - 1
    C.printf("loop2\n")
until a == 0

while a < 10 do
    if a == 8 then
        break
    end
    a = a + 1
end
```

For

0-indexed language so for-loop is not inclusive of upper bound

```
for i = 0,10 do
    C.printf("%d\n",i)
end

for i = 0,10,2 do
    c.printf("%d\n",i) --0, 2, 4, ...
end
```

Gotos

```
::loop::
C.printf("y\n")
goto loop
```

Almost exclusively used for when *generating code* for things that do not have structured control flow.

Functions: Multiple Returns in Terra

```
terra sort2(a : int, b : int) : {int,int}
    if a < b then
        return a, b
    else
        return b, a
    end
end

terra doit()
    -- the multiple returns are returned
    -- in a 'tuple' of type {int,int}:
    var ab : {int,int} = sort2(4,3)
    -- tuples can be pattern matched,
    -- splitting them into separate variables
    var a : int, b : int = sort2(4,3)
    --now a == 3, b == 4
end
doit()
```

Functions: Mutual Recursion

When a Terra function is created it needs to know about all the identifiers it references:

```
terra isodd -- declare isodd as a Terra function
terra iseven(n : uint32)
    if n == 0 then
        return true
    else
      -- OK! isodd is declared
        return isodd(n - 1)
    end
end
and terra isodd(n : uint32)
    if n == 0 then
        return false
    else
        return iseven(n - 1)
    end
end
```

Primitive Types

- Integers: `int int8 int16 int32 int64`
- Unsigned integers: `uint uint8 uint16 uint32 uint64`
- Boolean: `bool`
- Floating Point: `float double`

Primitive Operators

- Arithmetic: `- + * / %`
- Comparison: `< <= > >= == ~=`
- Logical: `and or not`
- Bitwise: `and or not ^ << >>`

```
true and false --Lazily evaluated logical and
1 and 3        --Eagerly evaluated bitwise and
```

Function Pointers

```
terra add(a : int, b : int) return a + b end
terra sub(a : int, b : int) return a - b end
terra doit(usesub : bool, v : int)
    var a : {int,int} -> int
    if usesub then
        a = sub
    else
        a = add
    end
    return a(v,v)
end
```

Fixed Length Arrays

```
var a : int[4]
a[0],a[1],a[2],a[3] = 0,1,2,3
var a = array(1,2,3,4) -- a has type int[4]
var a = arrayof(int,3,4.5,4) -- a has type int[3]
                                 -- 4.5 will be cast to an int
```

Vectors

```
terra saxpy(a :float,  X : vector(float,3), Y : vector(float,3),)
   return a*X + Y
end

var a = vector(1,2,3,4) -- a has type vector(int,4)
var a = vectorof(int,3,4.5,4) -- a has type vector(int,3)
                                  -- 4.5 will be cast to an int
```

Structs

Only user-defined data type.

Analog in Terra to Lua's tables.

```
struct Complex {
  real : float
  imag : float
}
terra doit()
    var c : Complex
    c.real = 4
    c.imag = 5
end
```

Structs

Only user-defined data type.

Analog in Terra to Lua's tables.

```
struct Complex {
  real : float
  imag : float
}
terra doit()
    var c : Complex
    c.real = 4
    c.imag = 5
end
```

```
struct B -- declaration
struct A {
  b : &B
}
struct B {
  a : &A
}
```

There is no -> Operator

```
terra doit(c : Complex)
    var pc = &c
    return pc.real --sugar for (@pc).real
end
```

Syntax Sugar for Struct Creation

```
-- a pair of floats
var a : tuple(float,float) = {3.f, 4.f}

-- an anonymous struct
var b = { real = 3.0, imag = 2.0 }
var c = Complex(b) -- cast
var d = Complex { real = 3.0, imag = 2.0 } -- also a cast
```

Syntax Sugar for Methods

```
struct Complex { real : double, imag : double }
Complex.methods.add = terra(self : &Complex, rhs : Complex) : Complex
    return {self.real + rhs.real, self.imag + rhs.imag}
end

terra doit()
    var a : Complex, b : Complex = {1,1}, {2,1}
    var c = a:add(b) -- sugar for Complex.methods.a(a,b)
    var ptra = &a
    var d = ptra:add(b) --also works
end

--same as before:
terra Complex:add(rhs : Complex) : Complex
    return {self.real + rhs.real, self.imag + rhs.imag}
end
```

Terra Entities as Lua objects

Since all Terra entities are Lua objects, we can introspect them from Terra:

```
> terra foo() return 4 end
> foo:printpretty() -- use foo:printpretty(false)
                    -- to see debug _before_ typechecking
[string "stdin"]:1:     foo = terra() : int32
                            return 4
                        end
> myquote = `3 + 4
> myquote:printpretty()
[string "stdin"]:1:     3 + 4
> aterratype = &int
> print(aterratype)
&int32
> foo:disas()
assembly for function at address 0x9b50010
0x9b50010(+0):   mov  eax, 4
0x9b50015(+5):   ret
```

Hygiene

Variables are still lexically scoped:

```
function use_quote(q)
  return quote
    var a = false
  in q end
end
terra my_function()
  var a = true
  return [ quote(`a) ] -- returns true
end
```

Casts

Rules for type casts are mostly the same as C, but the syntax is different.

Apply the Terra type object *as a function*:

```
terra todouble(a : int)
  return double(a)
end
```

If you need to use Lua code to get the Type object, you will need to escape the expression

```
terra todoublepointer(a : &opaque)
  return [&double](a)
end
local doublepointer = &double
terra todoublepointer(a : &opaque)
  return doublepointer(a) -- same as above
end
```

Programmatically decide memory layout of types

```
terra example()

  var s : Student
  s:setname("bob")
  s:setyear(4)

end
```

Programmatically decide memory layout of types

```
terra example()

  var s : Student
  s:setname("bob")
  s:setyear(4)

end
```

Like a high-level language: generate types using dynamic information.

Programmatically decide memory layout of types

```
terra example()

  var s : Student
  s:setname("bob")
  s:setyear(4)

end
```

```
Student.metamethods.__getentries =
function()
  file = io.open("students.dat","r")

  create ORM by populating type with fields
  described in database file

  return entries
end
```

Like a high-level language: generate types using dynamic information.

Programmatically decide memory layout of types

```
terra example()

  var s : Student
  s:setname("bob")
  s:setyear(4)

end
```

```
Student.metamethods.__getentries = 
function()
  file = io.open("students.dat","r")

  create ORM by populating type with fields
  described in database file

  return entries
end
```

name: rawstring	year: int

use generated layout in compiler optimizations

Like a high-level language: generate types using dynamic information.
Like a low-level language: optimize code using memory layout.

Programmatically decide memory layout of types

```
terra example()

  var s : Student
  s:setname("bob")
  s:setyear(4)

end
```

```
Student.metamethods.__getentries =
function()
  file = io.open("students.dat","r")

  create ORM by populating type with fields
  described in database file

  return entries
end
```

name: rawstring	year: int

use generated layout in compiler optimizations

Object behavior can also be meta-programmed.

Like a high-level language: generate types using dynamic information.
Like a low-level language: optimize code using memory layout.

Using compilers like LLVM to dynamically generate code tedious and verbose

```
float solve(float a, float b, float c) {
  return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```

Using compilers like LLVM to dynamically generate code tedious and verbose

```
float solve(float a, float b, float c) {
  return (-b + sqrt(b*b - 4*a*c)) / (2 * a);
}
```

```
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFSub(float_call, float_b)
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```

Using compilers like LLVM to dynamically generate code tedious and verbose

```
//types

std::vector<Type*> SolveTy_args;
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
SolveTy_args.push_back(Type::getFloatTy(C));
FunctionType* SolveTy = FunctionType::get(Type::getFloatTy(C),SolveTy_args)

std::vector<Type*>SqrtTy_args;
SqrtTy_args.push_back(Type::getFloatTy(C));
FunctionType* SqrtTy = FunctionType::get(Type::getFloatTy(C),SqrtTy_args);

PointerType* PtrSqrtTy = PointerType::get(SqrtTy, 0);

//function declarations

Function* func_solve = Function::Create(SolveTy,
                    GlobalValue::ExternalLinkage,
                    "solve", M);
Function* func_sqrtf = Function::Create(SqrtTy,
                    GlobalValue::ExternalLinkage,
                    "sqrtf", M);

// constants
ConstantFP* const_float_3 = ConstantFP::get(C, 4.f);
ConstantFP* const_float_4 = ConstantFP::get(C, 5.f);

// function definition
Function::arg_iterator args = func_solve->arg_begin();
Value* float_a = args++;
Value* float_b = args++;
Value* float_c = args++;

BasicBlock* label_entry = BasicBlock::Create(C, "entry",func_solve,0);
IRBuilder<> * B(label_entry);
Value* float_mul = B.CreateFMul(float_b, float_b);
Value* float_mul1 = B.CreateFMul(float_a, const_float_3);
Value* float_mul2 = B.CreateFMul(float_mul1, float_c);
Value* float_sub3 = B.CreateFSub(float_mul, float_mul2);
Value* float_call = B.CreateCall(func_sqrtf, float_sub3);
Value* float_add = B.CreateFSub(float_call, float_b)
Value* float_div = B.CreateFMul(float_add, const_float_4);
Value* float_mul4 = B.CreateFMul(float_div, float_a);
B.CreateReturn(float_mul4);
```

```
) {
2 * a);
```