EECS 483: Compiler Construction

Lecture 10:
Lambda Lifting, Dynamic Typing

February 16
Winter Semester 2026

Logistics

Assignment 2 Solution now available. Grades posted to Canvas.

Assignment 3 now available, due on the 27th

- extern functions, lambda lifting, SysVAMD64 calling convention
- finish the relevant material on lambda lifting today, but you can already get started implementing extern functions before fully understanding lambda lifting.
- Start early! More challenging to debug than previous assignments.

Learning Objectives

Correction about Alignment

Review idea of Lambda Lifting

Mechanics of lambda lifting: which functions to lift? which variables to capture?

Dynamic typing: semantics, implementation strategies

Corrections

Example call was misaligned last time. I've updated the previous weeks' slides. Let's review

Stack Alignment

When a function is called, rsp % 16 == 8

To ensure correct alignment: rsp % 16 == 0 **before** executing the **call** instruction (since call pushes an 8-byte address)

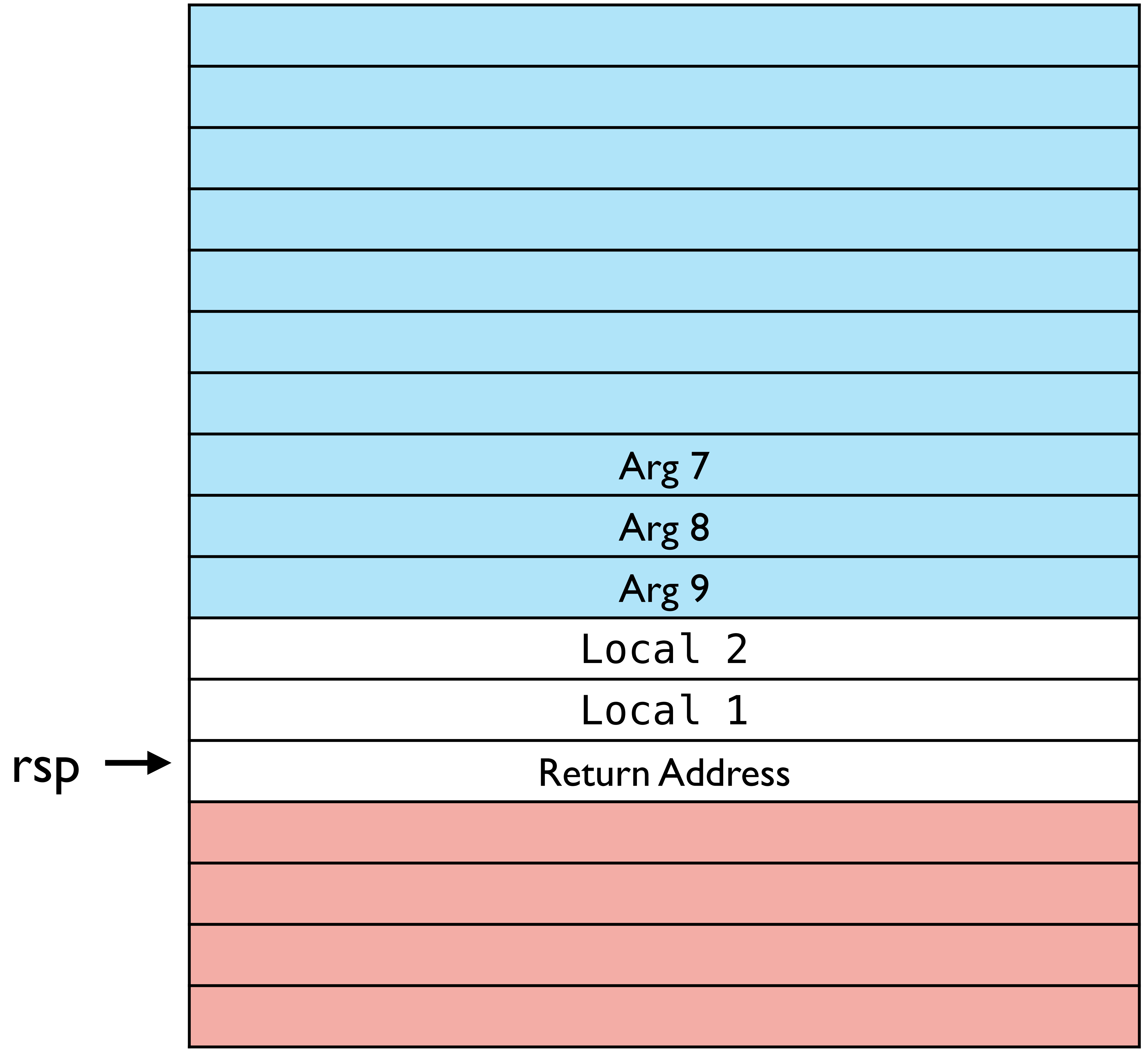
How do we know if we are aligned?

- **ASSUME** that you were called correctly, meaning rsp % 16 == 8. Need to make sure that when we call rsp % 16 == 0
- When creating the new stack frame account for everything we store on the stack:
  - number of locals L, number of stack-passed arguments A
  - if L + A is **odd** , the stack will be aligned if the arguments are pushed immediately after the locals
  - if L + A is **even**, we can add 8 bytes of "padding" to the locals

2 Locals, 3 stack args

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 8 * 5], arg7
mov QWORD [rsp - 8 * 4], arg8
mov QWORD [rsp - 8 * 3], arg9
sub rsp, 8 * 5
call big_fun
add rsp, 8 * 5
```

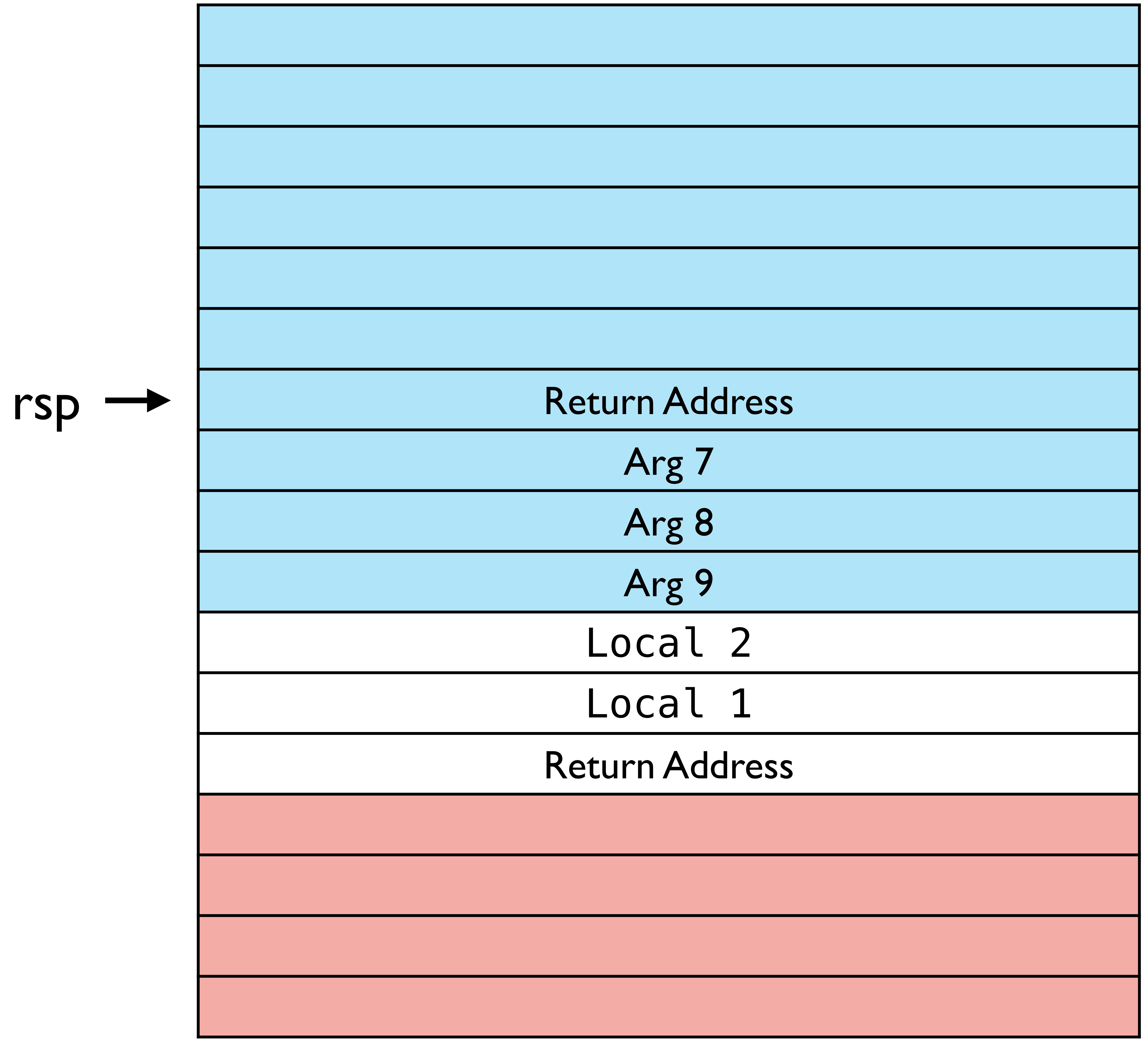
Aligned without padding



2 Locals, 3 stack args

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 8 * 5], arg7
mov QWORD [rsp - 8 * 4], arg8
mov QWORD [rsp - 8 * 3], arg9
sub rsp, 8 * 5
call big_fun
add rsp, 8 * 5
```

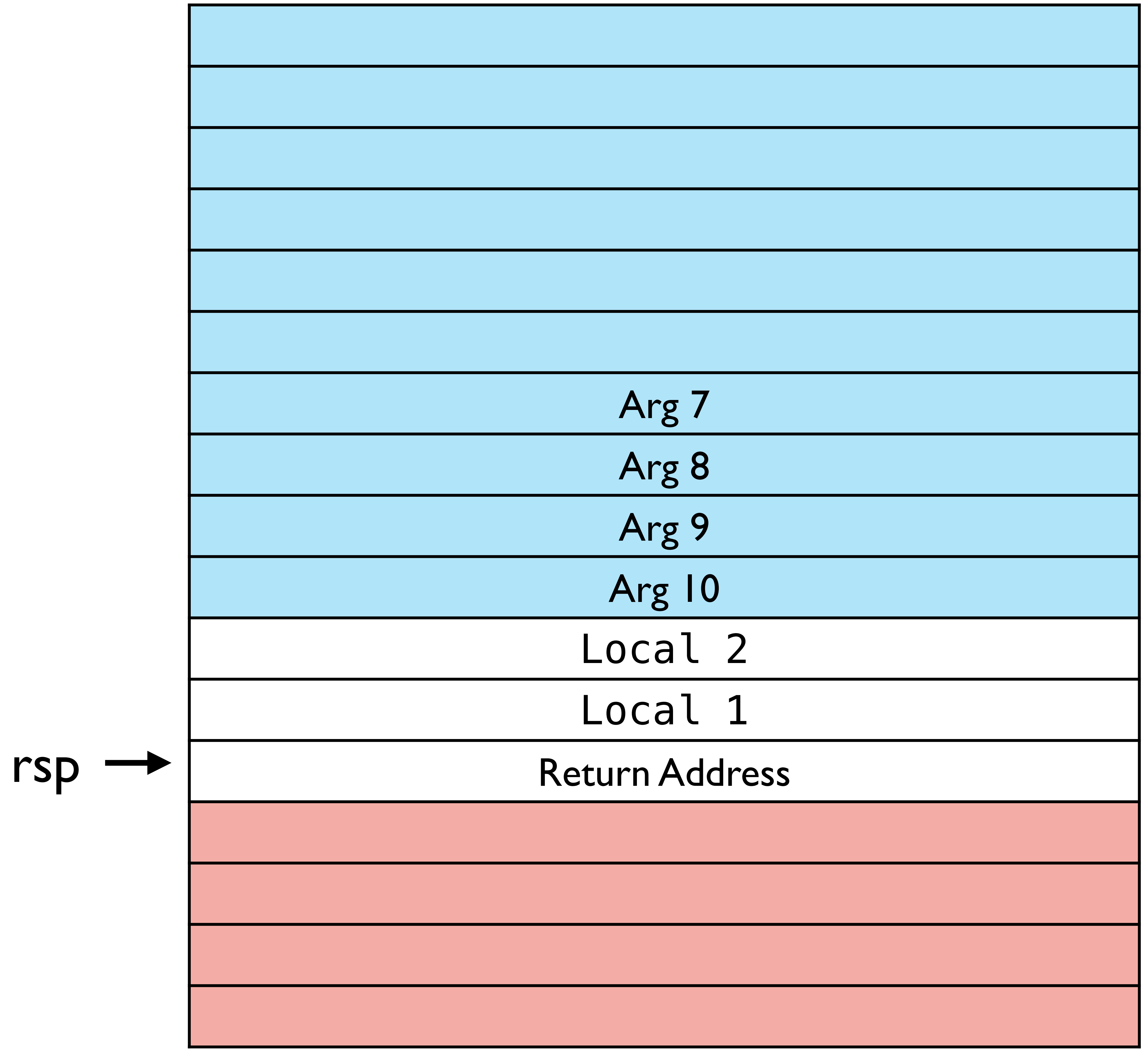
Aligned without padding



2 Locals, 4 stack args

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 8 * 6], arg7
mov QWORD [rsp - 8 * 5], arg8
mov QWORD [rsp - 8 * 4], arg9
mov QWORD [rsp - 8 * 3], arg10
sub rsp, 8 * 6
call big_fun
add rsp, 8 * 6
```

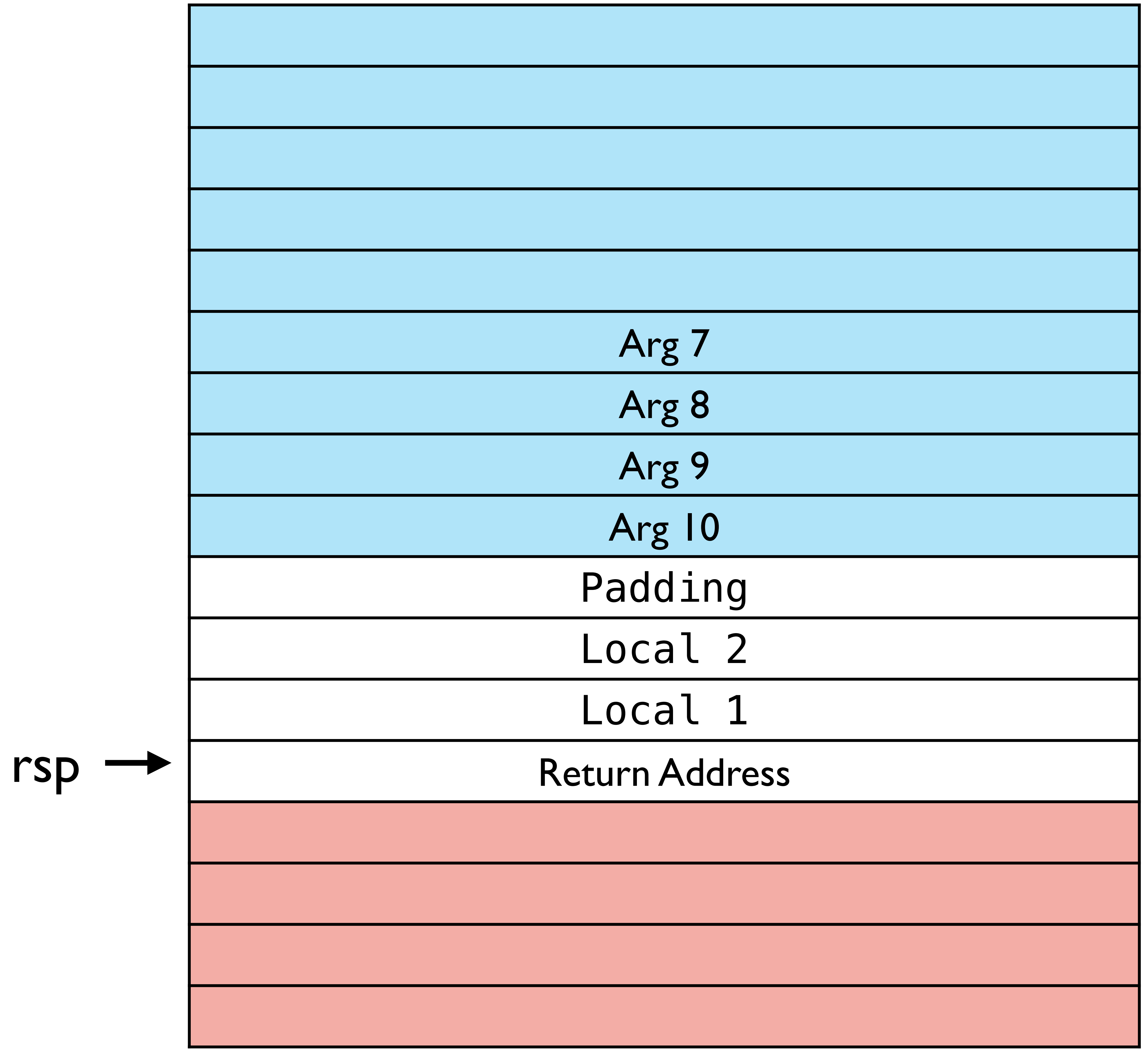
Misaligned without padding



2 Locals, 4 stack args

```
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
mov rcx, arg4
mov r8, arg5
mov r9, arg6
mov QWORD [rsp - 8 * 7], arg7
mov QWORD [rsp - 8 * 6], arg8
mov QWORD [rsp - 8 * 5], arg9
mov QWORD [rsp - 8 * 4], arg10
sub rsp, 8 * 7
call big_fun
add rsp, 8 * 7
```

Aligned with padding



Stack Alignment

When a function is called, rsp % 16 == 8

To ensure correct alignment: rsp % 16 == 0 **before** executing the **call** instruction (since call pushes an 8-byte address)

How do we know if we are aligned?

- **ASSUME** that you were called correctly, meaning rsp % 16 == 8. Need to make sure that when we call rsp % 16 == 0
- When creating the new stack frame account for everything we store on the stack:
    - number of locals L, number of stack-passed arguments A
    - if L + A is **odd** , the stack will be aligned if the arguments are pushed **immediately after** the locals
    - if L + A is **even**, we can add a "padding" gap of 8 bytes between our locals and the stack-passed arguments.

State of the Snake Language

Adder: Straightline Code

Boa: Conditionals + Tail-called functions = arbitrary "local" control flow.

Good for implementing the "bodies" of functions, but missing key features:

1. Interaction with the Operating System (stdin/stdout, file I/O, random number generation)

2. Reusable sub-procedures (functions with non-tail calls)

Add these in **Cobra**

Running Examples

Non-tail calls

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

captured variables in non-tail called function

Design Goals

Non-tail calls

We want to support the ability to **call** or **tail call** our internally defined functions.

We want **tail calls** to be implemented the same way as in Boa: this ensures tail-recursive functions are still compiled efficiently.

We want **calls** to be implemented using the System V AMD64 calling convention. This allows us to compile calls to Rust or Cobra functions the same way, simplifying code generation.

How do we get the best of both worlds?

Change to SSA

Previously we had one code block that would be called with the SysV calling convention: **main**

Generalize this to have many top level function blocks in SSA.
The body of a function block should immediately branch with arguments to an ordinary SSA block, which is compiled as before.

In code generation: compile these as moving the arguments from the SysV AMD64-designated locations to the stack.

SSA Generation Example

```
def main(x):
  def max(m,n):
    if m >= n: m else: n
  in
  max(10, max(x, x * -1))
```

```
block max_tail(m, n):
  ... as in Boa
block main_tail(x):
  tmp1 = x * -1
  tmp2 = call max_fun(x, tmp1)
  br max_tail(10, tmp2)
fun max_fun(m, n):
  br max_tail(m, n)
fun entry(x):
  br main_tail(x)
```

give the blocks and funs different names as we need to assign both of them labels in code generation

Functions vs Basic Blocks in SSA

In SSA, we make a distinction between **functions** and **parameterized blocks**

Both have a label and arguments, but the way they are used and compiled is different

Functions can only ever be the target of a **call**, using the System V AMD64 ABI

Blocks can only ever be the target of a **branch**, where arguments are placed at stack offsets

Blocks can be nested as sub-blocks within other blocks, can refer to outer scope

Functions are only ever **top-level**, only variables in scope inside are arguments

Code Generation for Function Blocks

```
fun max_fun(m, n):
  br max_tail(m, n)
```

➡

```
max_fun:
  mov [rsp - 8], rdi
  mov [rsp - 16], rsi
  jmp max_tail
```

Functions are just a thin wrapper around their blocks, mov arguments from where the calling convention dictates to where the block expects them to be (on the stack)

Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

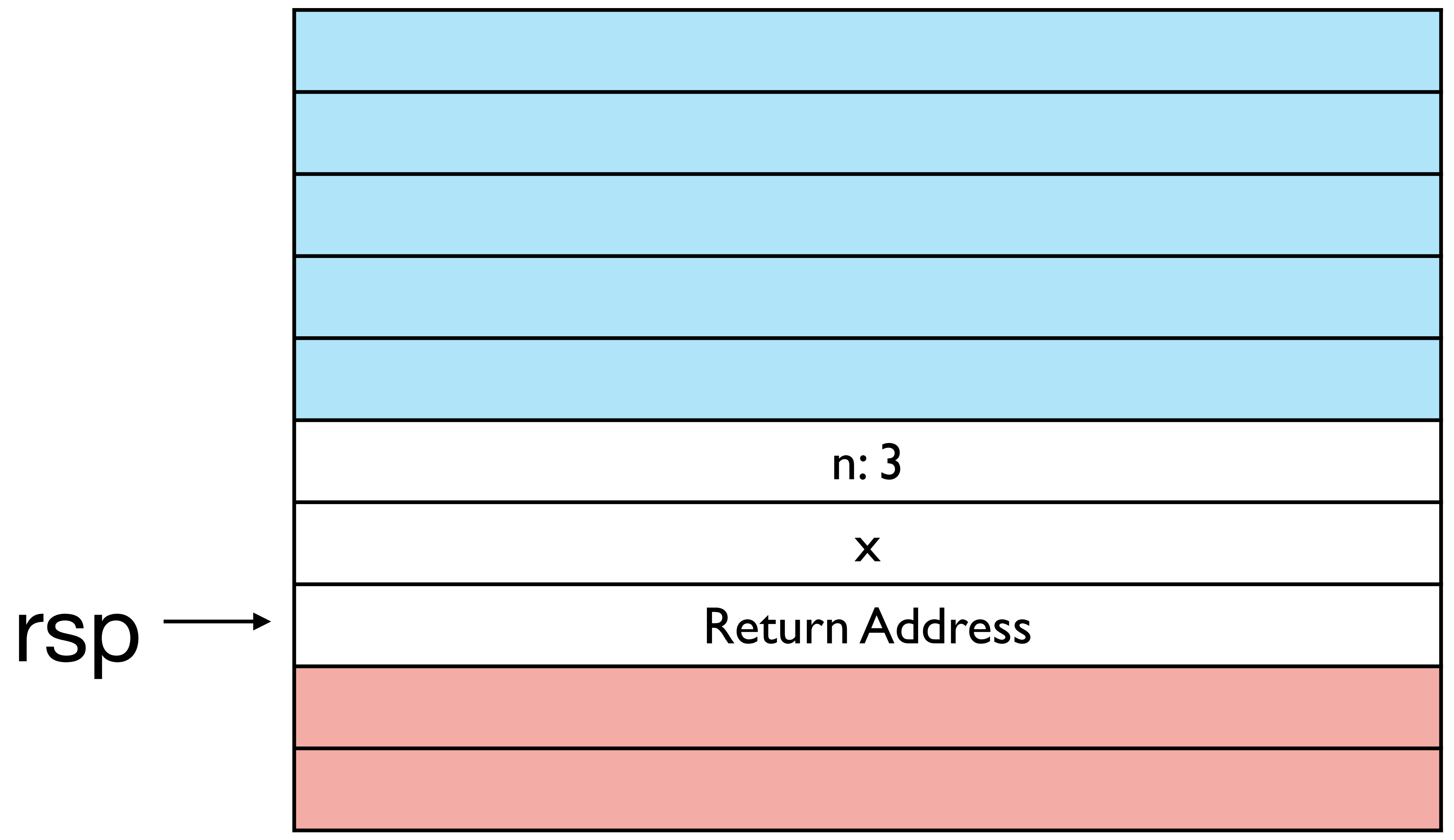
x is "captured" by the function **pow** in that it is used in the function because it is in scope when the function **pow** is defined.

But now **pow** is not tail-recursive. What happens?

Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

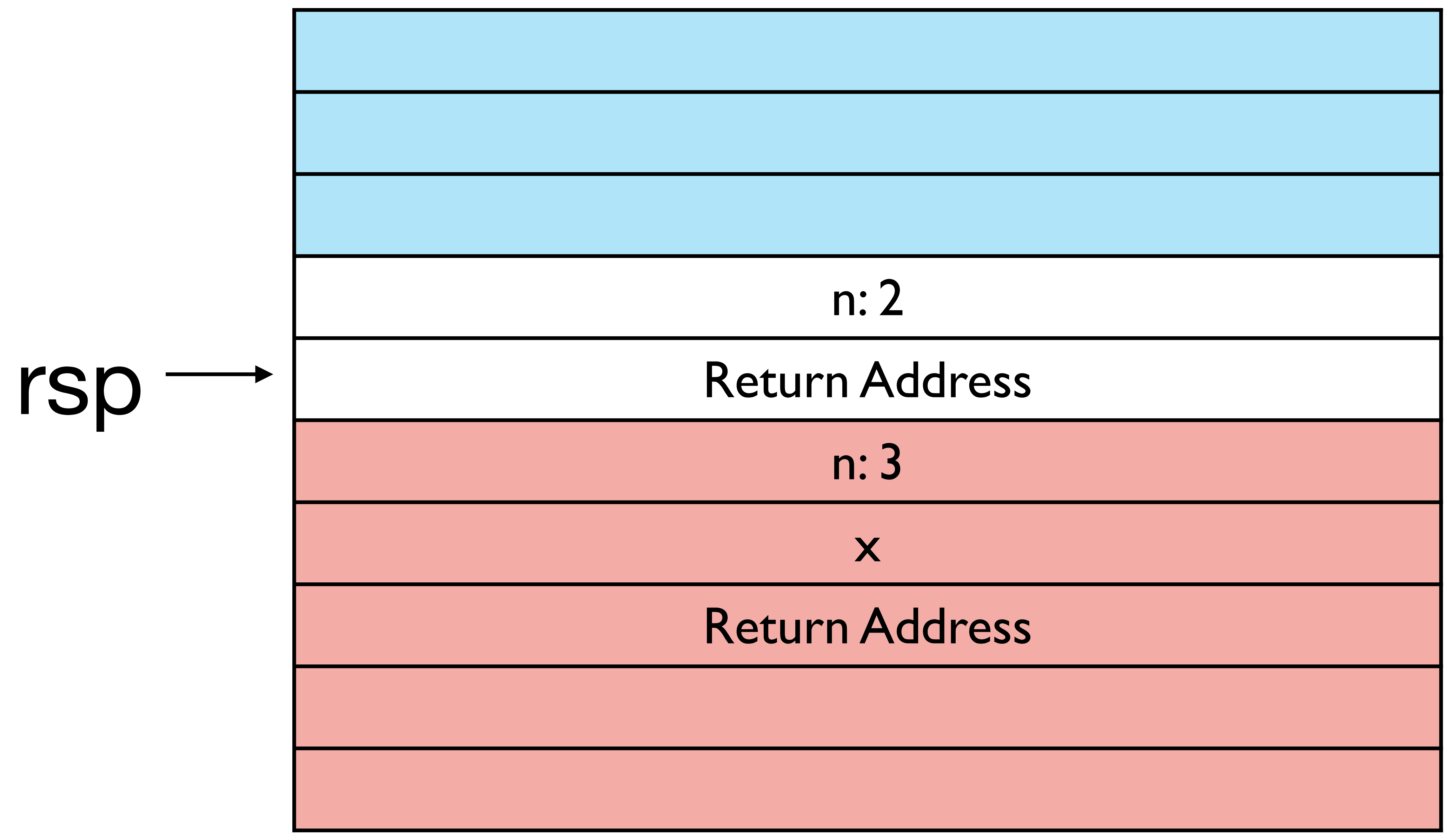
initial tail call



Variable Capture

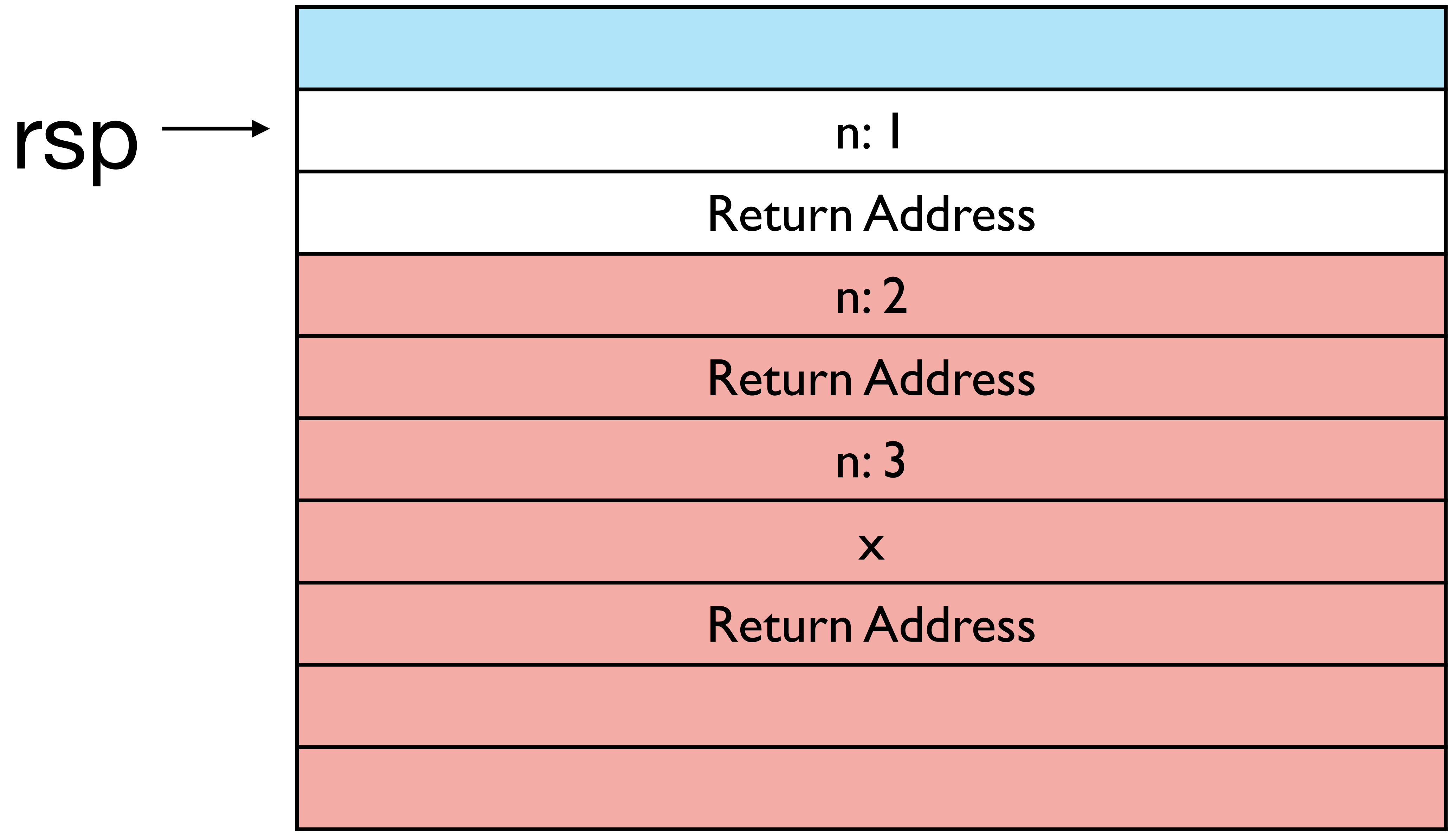
```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

first non-tail recursive call



Variable Capture

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```



second non-tail recursive call

Two Approaches to Variable Capture

Static Links (see Appel Chapter 6)
nested function definitions are passed an additional argument which is a pointer to the stack frame of the enclosing definition. To access captured variables, walk the stack to find the variable.

Lambda Lifting
nested function definitions are passed all captured variables as extra arguments.

Tradeoff:
in lambda lifting, accessing the value is fast because it is local, but with static links, it takes potentially many memory accesses.

with static links, passing the value is fast because it is just a single pointer, but with lambda lifting all captured values are copied

We implement lambda lifting as it is simpler to implement and generalizes easily to first-class functions.

Lambda Lifting (As AST to AST transform)

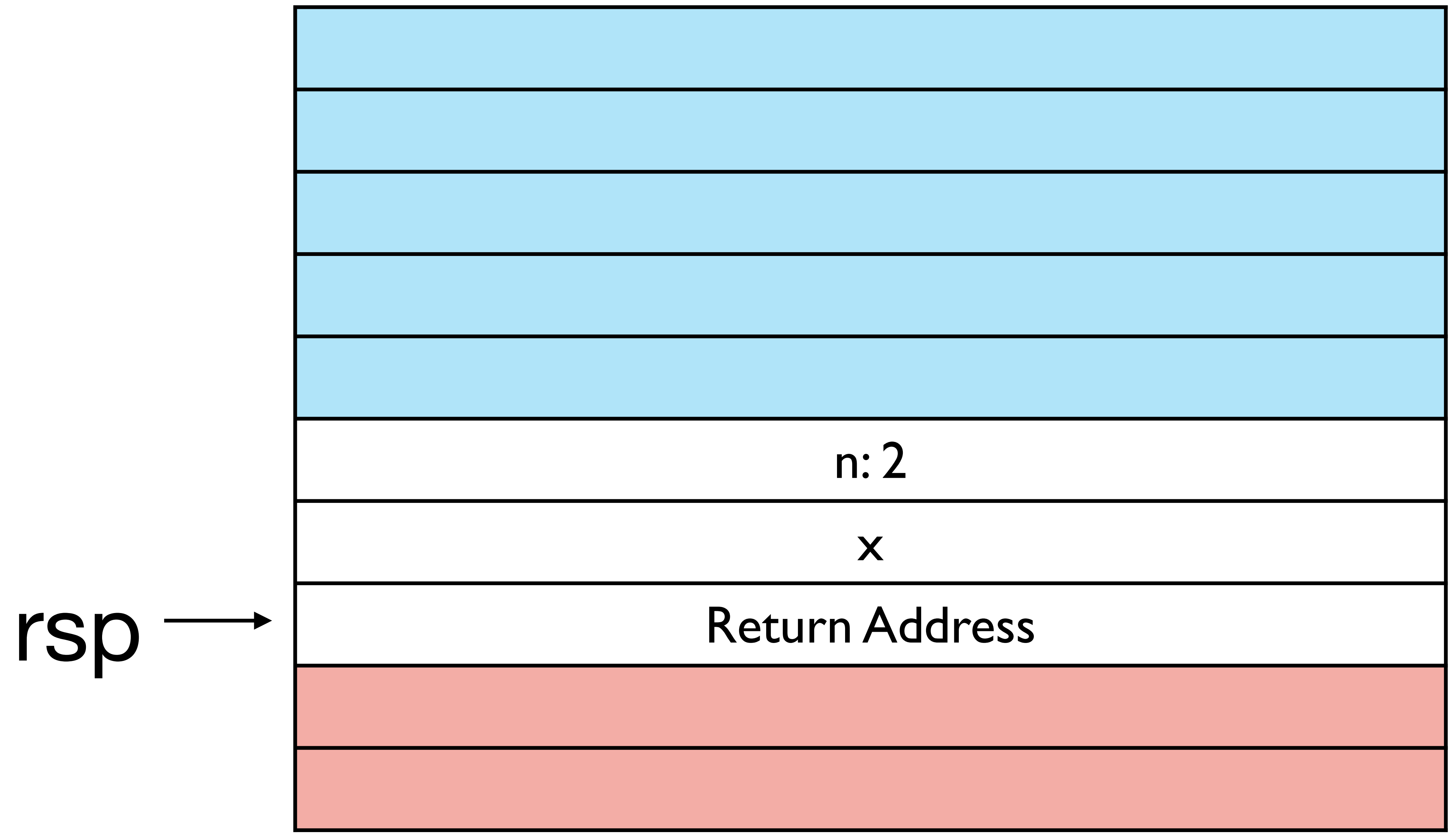
```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

```
def pow(x, n):
  if n == 0: 1 else x * pow(x, n - 1)
def main(x):
 pow(x, 3)
```

Variable Capture

```
def pow(x, n):
  if n == 0: 1 else x * pow(x, n - 1)
def main(x):
 pow(x, 3)
```

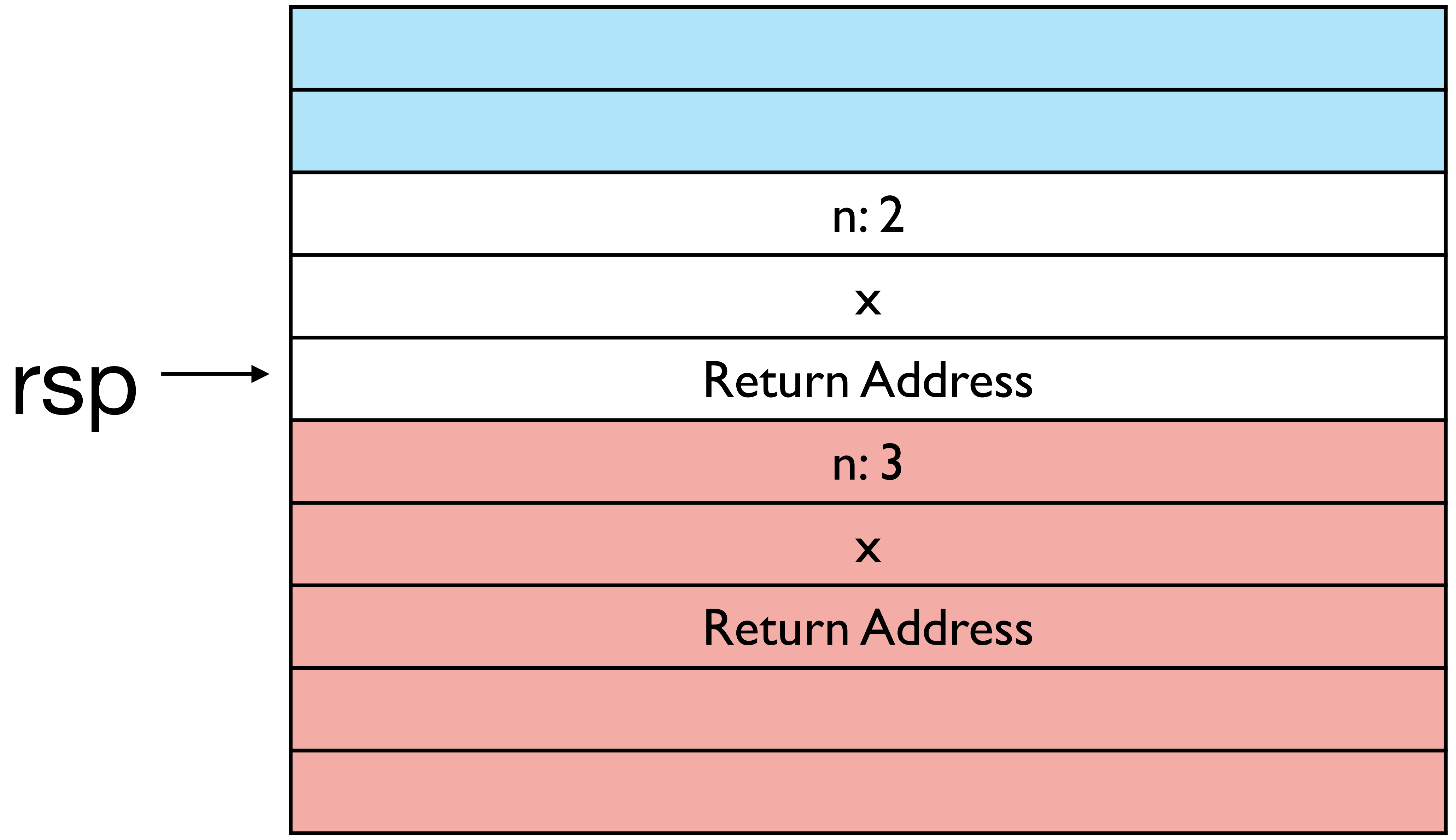
initial tail call



Variable Capture

```
def pow(x, n):
    if n == 0: 1 else x * pow(x, n - 1)
def main(x):
  pow(x, 3)
```

first non-recursive call



Lambda Lifting

Instead of an AST to AST transform, incorporate this in our AST to SSA transformation.

In the lowering to SSA, any function that is called must be "lifted" to the top-level, where all captured variables are added as extra arguments, and all **calls** or **branches** pass the additional arguments.

Lambda Lifting (As AST to SSA transform)

```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

```
block pow_tail(x, n):
  b = n == 0
  thn():
    ret 1
  els():
    n2 = n - 1
    r = call pow_call(x, n)
    tmp = x * r
    ret tmp
  cbr b thn() els()
block main_tail(x):
  br pow_tail(x, 3)
fun pow_call(x, n):
  br pow_tail(x, n)
fun main(x):
  br main_tail(x)
```

Lambda Lifting: Details

To implement lambda lifting, we need to address two questions.
1. Which functions need to be lifted?
2. Given a function to be lifted, which arguments need to be added?

For both of these we need to consider
1. Correctness
   Must ensure every function that must be lifted is lifted and that every argument that must be added is added, but we can **over-approximate** by lifting more than necessary and adding more arguments than necessary
2. Efficiency
   Lifting too many functions or adding too many arguments can impact runtime and space usage in our generated programs.

Correctness is **always** a must. Efficiency is best-effort.

Lambda Lifting: Who to Lift

What definitions need to be lifted?
- any function that is (non-tail) called needs to be lifted
- the tail-callable version of that function also needs to be lifted

Any other functions?

Yes: any function that is tail called by a lifted function must also be lifted, even if it is never non-tail called

Lambda Lifting: Who to Lift

Which of e,f,g,h,k need to be lifted in this example?

Answer:
h must be lifted because it is non-tail called
g, e must be lifted because they are tail called by a lifted function
f must be lifted because it is tail called by a lifted function
k does not need to be lifted

```
def main(a):
  def e(): a * 2 in
  def f(): a in
  def g(): f() in
  def h(b): if b: g() else: e() in
  def k(): h(a) + 1 in
  k()
```

Lambda Lifting: Who to Lift

```
def main(a):
  def e(): a * 2 in
  def f(): a in
  def g(): f() in
  def h(b): if b: g() else: e() in
  def k(): h(a) + 1 in
  k()
```

Call Graph

$$\begin{tikzcd}
 & \text{main} \arrow[d] & \\
 & k \arrow[d] & \\
 & h \arrow[dl] \arrow[dr] & \\
e & & g \arrow[d] \\
 & & f
\end{tikzcd}$$

1. Anything (besides main) that is called needs to be lifted

2. Anything reachable in the call graph from a lifted function needs to be lifted

Lambda Lifting: Who to Lift

1. Anything (besides main) that is called needs to be lifted

2. Anything reachable in the call graph from a lifted function needs to be lifted

Implement by worklist algorithm:

1. Build a call graph

2. Initialize worklist with the functions that are non-tail called

3. While the worklist is nonempty: pop a function off, add the function to the set of functions that need to be lifted, add sucessors that are not already identified as lifted to the worklist

Call Graph

$$\begin{tikzcd}
 & \text{main} \arrow[d] & \\
 & k \arrow[d] & \\
 & h \arrow[dl] \arrow[dr] & \\
e & & g \arrow[d] \\
 & & f
\end{tikzcd}$$

Lambda Lifting: What Args to Add

When we lift a function, we need to add extra arguments. But **which** arguments need to be added?

Answer: all the non-local variables that **must** be in the function's stack frame.

Which variables **must** be in the stack frame?
  Easy over-approximation: all the variables that are in scope at the function's definition site.

  Tempting but incorrect: just the variables that actually occur in the body of the lifted function?

Lambda Lifting: What Args to Add

```
def main(a):
  def e(): a * 2 in
  def f(): a in
  def g(): f() in
  def h(b): if b: g() else: e() in
  def k(): h(a) + 1 in
  k()
```

```
block e_tail(a):
  r = a * 2
  ret r
block f_tail(a):
  ret a
block g_tail(a):
  br f_tail(a)
block h_tail(a,b):
  cbr b g_tail(a) e_tail(a)
fun h_fun(a,b):
  br h_tail(a,b)
fun entry(a):
  k():
    tmp1 = call h_fun(a,a)
    tmp2 = tmp1 + 1
    ret tmp2
  br k()
```

Notice: need to add **a** to **g** even though it doesn't occur syntactically in the body of **g**

Lambda Lifting: What Args to Add

Adding all variables that are in scope is correct, but inefficient. Why?

Lambda Lifting: What Args to Add

The easiest way to correctly implement lambda lifting is to add **all** variables that are in scope as extra arguments.

But this can be inefficient:

```
def main(x):
  let a1 = ... in
  ...
  let a100 = ... in
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

$\Rightarrow$

```
fun pow_tail(a1,...a100,n):
  ...
fun entry(x):
  ...
```

every unnecessary variable is extra space in our stack frames!

Lambda Lifting: What Args to Add

When lambda lifting, we need to add captured variables to lifted functions.

Capturing all in-scope variables is correct, but can be wasteful.

Two options:

- Perform a **liveness analysis** before lambda lifting, to determine exactly which variables are used.

- Wastefully add all variables, and perform liveness analysis afterwards, at the SSA level, and perform **parameter dropping**.

We'll adopt the second approach. We'll discuss how to implement **liveness analysis** later when we cover optimizations.

Cobra Overview

Cobra is the language you implement in Assignment 3, **Procedures**

New source language features:

1. Extern functions
2. Non-tail calls to local functions or extern functions

Cobra: Frontend Changes

1. Remove errors for calling non-tail called functions
2. Ensure that extern functions are in scope
3. When performing name resolution, ensure that we treat extern function names specially as "non-mangled", as opposed to our local functions, which should be given unique names
4. Perform an analysis to determine which functions need to be lifted:
    1. easy but inefficient: just say all functions must be lifted
    2. harder but more efficient: analyze the call graph
5. Identify which variables need to be added to the lifted function
    1. easy but inefficient: add all variables that are in scope
    2. harder but more efficient: use **liveness analysis**

Cobra: Middle-end Changes

1. Tail calls to extern functions should be compiled to calls
2. Lift all functions that are identified in the frontend as needing to be lifted
3. For calls to internal functions
    1. If they are tail calls, compile to a branch with arguments
    2. If they are calls, compile to an SSA call
    3. If the function being called is lifted, make sure to add extra arguments
4. For calls to external functions
    1. Always compile to an SSA call

Cobra: Backend Changes

1. All calls are compiled using the System V AMD64 Calling Convention
2. Function definitions are compiled to a labeled block that moves their arguments from the location dictated by the System V Calling Convention to the stack and then **jmp** to the specified label.

Otherwise no change, the main work is handled by the lambda lifting pass

State of the Snake Language

Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures (pushdown automata if you don't use extern functions)

Remaining limitations:

1. Only data are ints (booleans are really just special ints)
2. Only ways to use memory are 64-bit local variables and the call stack

State of the Snake Language

Adder: Straightline Code (arithmetic circuits)

Boa: local control flow (finite automata)

Cobra: procedures (pushdown automata if you don't use extern functions)

Snake v4: **Diamondback**

1. Add new datatypes, use dynamic typing to distinguish them at runtime
2. Include heap-allocated variable-sized arrays, allowing for unrestricted memory usage

Computational power: Turing complete

Booleans in Boa/Cobra

In Boa/Cobra, booleans and integers weren't truly distinct datatypes.

- All integers could be used in logical operations
- All booleans could be used in arithmetic operations

Booleans in Boa/Cobra

The following are all valid programs with well-defined semantics in Boa/Cobra

```
-1 && 3

true + 5

7 >= false
```

Let's change the language semantics so these are **errors** instead.

Booleans in Boa/Cobra

Can we implement operations **isInt** and **isBool** that distinguish between integers and booleans?

```
isInt(true) == false

isInt(1)    == true
```

No, true and 1 have the exact same representation at runtime

Static vs Dynamic Typing

How would we implement a language where integers and booleans were considered disjoint?

1. Static Typing (C/C++, Java, Rust, OCaml)

   Identify the runtime types of all variables in the program

   Reject type-based misuse of values in the frontend of the compiler.

2. Dynamic Typing (JavaScript, Python, Ruby, Scheme)

   Use **type tags** to identify the type of data at runtime

   Reject type-based misuse of values at runtime, right before the operation is performed

Static Typing vs Dynamic Typing

Example 1:

```
    true + 5
```

Static typing: **compile time error**: true used where integer expected

Dynamic typing: **runtime error**: addition operation expects inputs to be integers

Static Typing vs Dynamic Typing

Example 2:

```
def main(x):
  x + 5
```

Static typing: need to declare a type for **x**, in this case **int**

Dynamic typing: succeed at runtime if **x** is an int, otherwise fail

Static Typing vs Dynamic Typing

Example 2:

```
def main(a):
  def complex_function(): ... in
  let x = if complex_function(): 1 else: true
  x + 5
```

Static typing: reject this program, even if **complex_function** always returns true

Dynamic typing: succeed at runtime if **complex_function** returns true, otherwise fail

Static Typing vs Dynamic Typing

Static Typing

- Easier on the compiler: if type information is reliable, we can use that to inform the runtime representation of our compiled values
- Easier on the programmer? Types document the code, aid in tooling, design

Dynamic Typing

- Easier on the programmer? Complex patterns that are difficult to assign static types are possible

Static Typing vs Dynamic Typing

Poll: Is static typing or dynamic typing better?

My opinion:

> I prefer static typing, but both are popular enough to be worth studying and implementing well.

In Assignment 4, we'll implement dynamic typing

In Assignment 5, perform optimizations to reduce the runtime overhead of dynamic typing

Revisit syntactic aspects of static typing and the relation with static analysis later in the course.

Semantics of Dynamic Typing

Live code interpreter

Semantics of Dynamic Typing

- A Snake value is not just an int anymore. It is **either** an int or a boolean, and we need to be able to tell the difference at runtime in order to determine when we should error and how to implement isInt, isBool.
- Many operations can now produce runtime errors if type tags are incorrect, need to specify
  - what the appropriate error messages are
  - evaluation order between expressions executing and type tags
    - true + (let _ = print(3) in 3)
      - does this print 3 before it errrors?

Representing Dynamically Typed Values

In Adder/Boa/Cobra, all runtime values were integers.

In Diamondback, a runtime value must have both a type tag and a value that matches the type tag

How should we represent tags and values in our compiled program?

Representing Dynamically Typed Values

Approach 1: Values as 8 bytes, Tags as extra data

A snake value is 9 bytes

- the first byte is a tag: 0x00 for integer, 0x01 for boolean. Use a full byte to keep our values byte-aligned
- the remaining 64 bits are the underlying integer, bool or pointer

Upside: Faithful representation of our Rust interpreter

Downside: 1 byte memory overhead for all values plus padding, calling convention and architecture are 8-byte oriented, tedious to implement pervasively

Representing Dynamically Typed Values

Approach 2: Values as pointers

- A snake value is a 64-bit pointer to an object on the **heap**
  - value stored on the heap can then be whatever size we want, the pointer is always 64 bits.
  - store a tag and value on the heap similarly to previous approach.

A value stored in this way is called **boxed**.

Representing Dynamically Typed Values

Approach 2: Values as pointers

- A snake value is a 64-bit pointer to an object on the **heap**
  - value stored on the heap can then be whatever size we want, the pointer is always 64 bits.
  - store a tag and value on the heap similarly to previous approach.

Upside: uniform implementation, 64-bit values can be compiled as before

Downside: Adds memory overhead pervasively.

Representing Dynamically Typed Values

Approach 3: compromise

A snake value is a 64-bit value.

- Use the least significant bits of the value as a **tag**.
- Represent simple data like integers, booleans within the 64-bits
- Represent large datatypes like arrays, closures, structs as pointers to the heap

Upside: use stack allocation more often

Downside: can't fit 64 bits and a tag...

Roughly the approach used in high-performance Javascript engines (v8) as well as some garbage-collected typed languages (OCaml)

Representing Dynamically Typed Values

To implement our compiler, we need to specify

1. How each of our Snake values are represented at runtime
2. How to implement the primitive operations on these representations

Integers

Implement a snake integer as a 63-bit signed integer followed by a 0 bit to indicate that the value is an integer

Number	Representation
1	`0b00000000_0000.....0000_00000010`
6	`0b00000000_0000.....0000_00001100`
-1	`0b11111111_1111.....1111_11111110`

I.e., represent a 63-bit integer **n** as the 64-bit integer 2 * n

Booleans

The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b01` to distinuish from integers and other datatypes

Use the remaining 62 bits to encode true and false as before as 1 and 0

Number	Representation
`true`	`0b00000000_0000.....0000_00000101`
`false`	`0b00000000_0000.....0000_00000001`

2^62 - 2 bit patterns are therefore "junk" in this format

Boxed Data

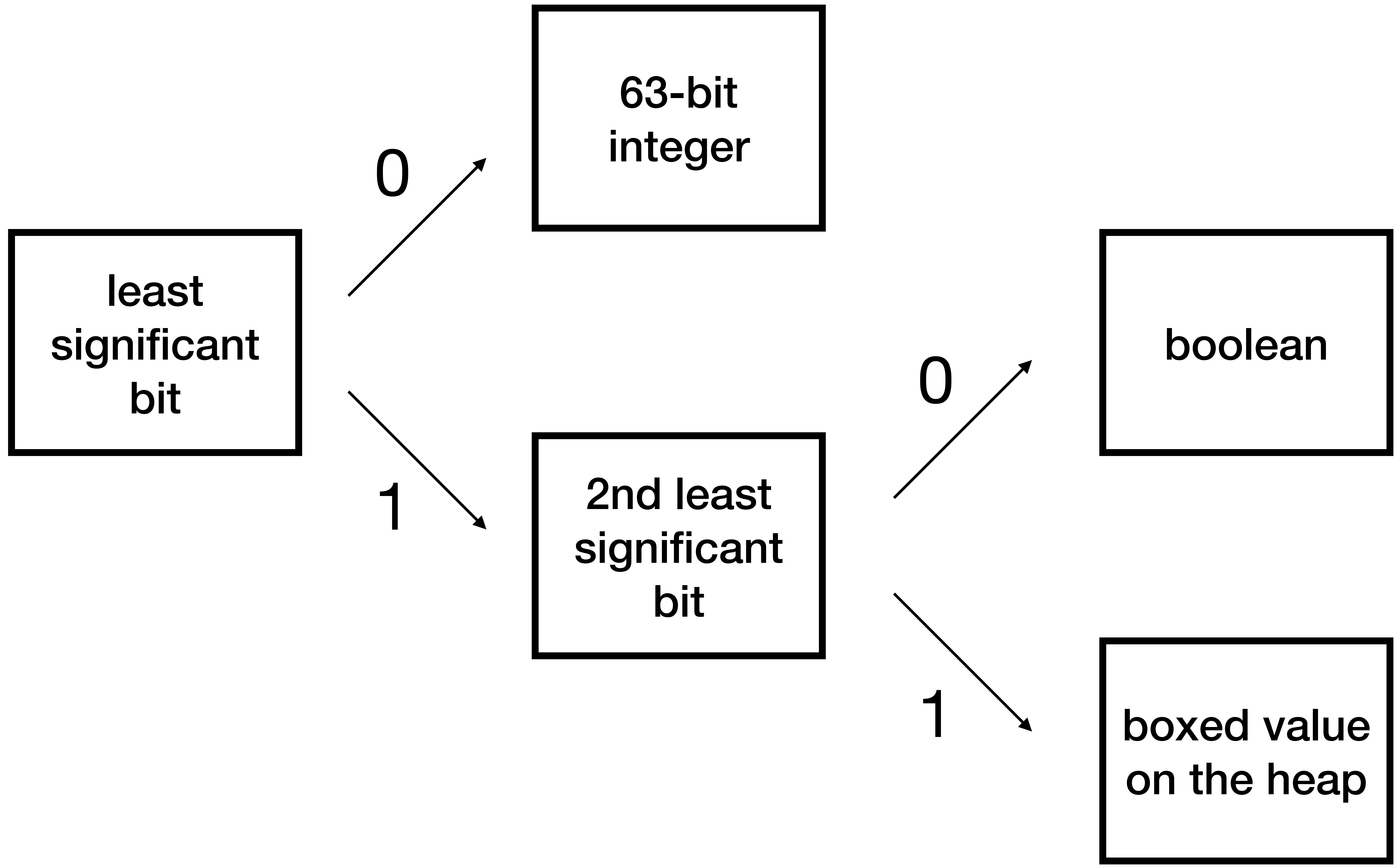
The least significant bit must be 1 to distinguish from integers

Use least significant bits `0b11` to distinguish from booleans.

Use remaining 62-bits to encode a pointer to the data on the heap

- Why is this ok? Discuss more thoroughly on Wednesday

Representing Dynamically Typed Values



Implementing Dynamically Typed Operations

We need to revisit our implementation of all primitives in assembly code to see how they should work with our new datatype representations.

1. Arithmetic operations (add, sub, mul)
2. Inequality operations (<=, <, >=, >)
3. Equality
4. Logical operations (&&, ||, !)

As well as supporting our new operations isInt and isBool

Implementing Dynamically Typed Operations

In dynamic typing, implementing a primitive operation has two parts:

1. How to check that the inputs have the correct type tag
2. How to actually perform the operation on the encoded data

Implementing Dynamically Typed Operations

Live code