EECS 483: Compiler Construction

Lecture 9:
Non-tail Function Calls and Definitions, Lambda Lifting

February 11
Winter Semester 2026

Learning Objectives

Implement of SysV AMD64 Function Calls in x86

Implement Function **definitions** in x86

Discuss how to implement **variable capture** for **local** function definitions

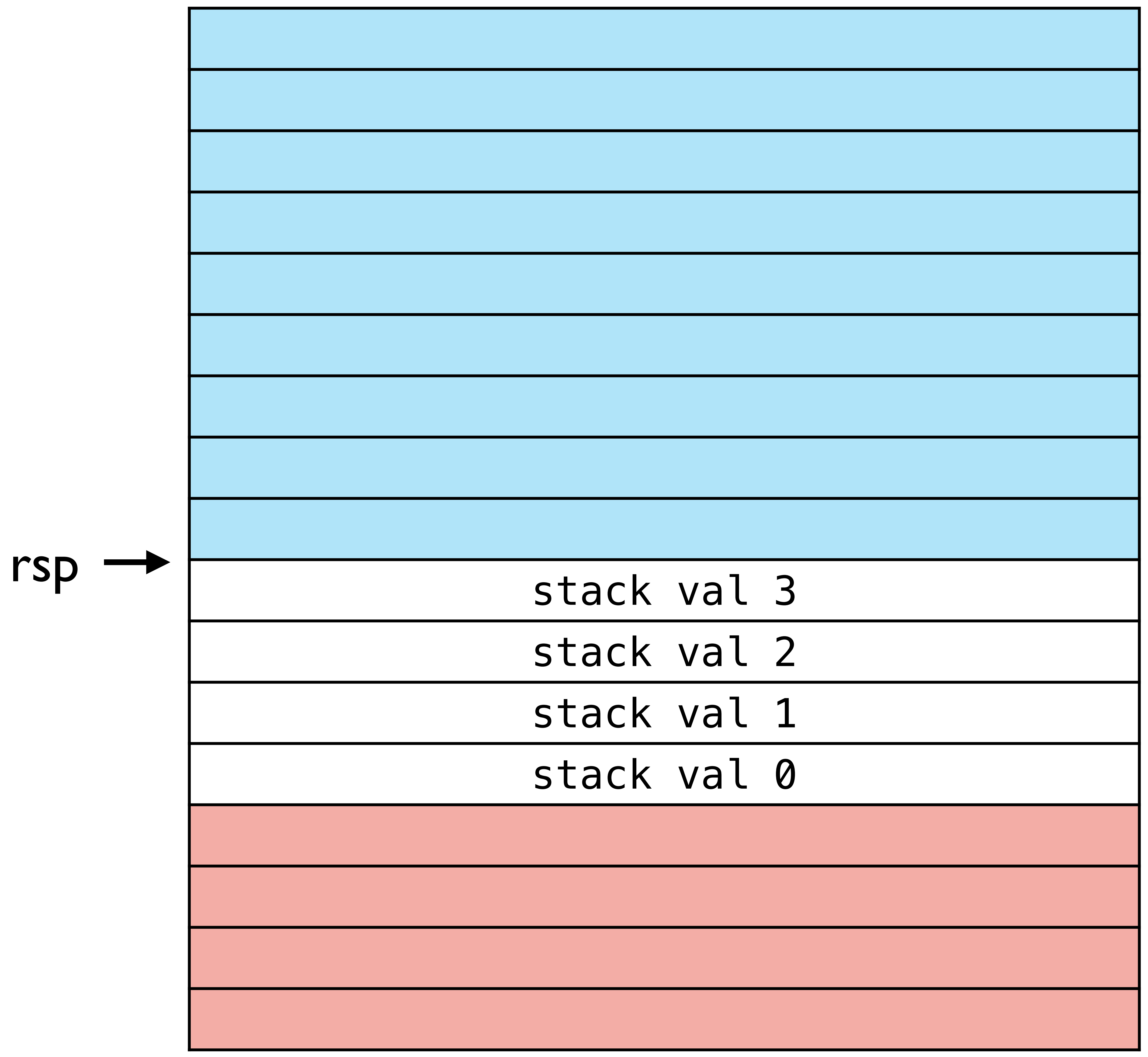
x86 Abstract Machine: The Stack

So far we have used rsp as a base pointer into our stack frame

But several instructions treat it as a "stack pointer" pointing to the **top** of a stack of values (growing downwards in memory address space

push/pop: dec/inc rsp by 8, store/load a value

call/ret: push/pop a return address, update instruction pointer



Calling Convention

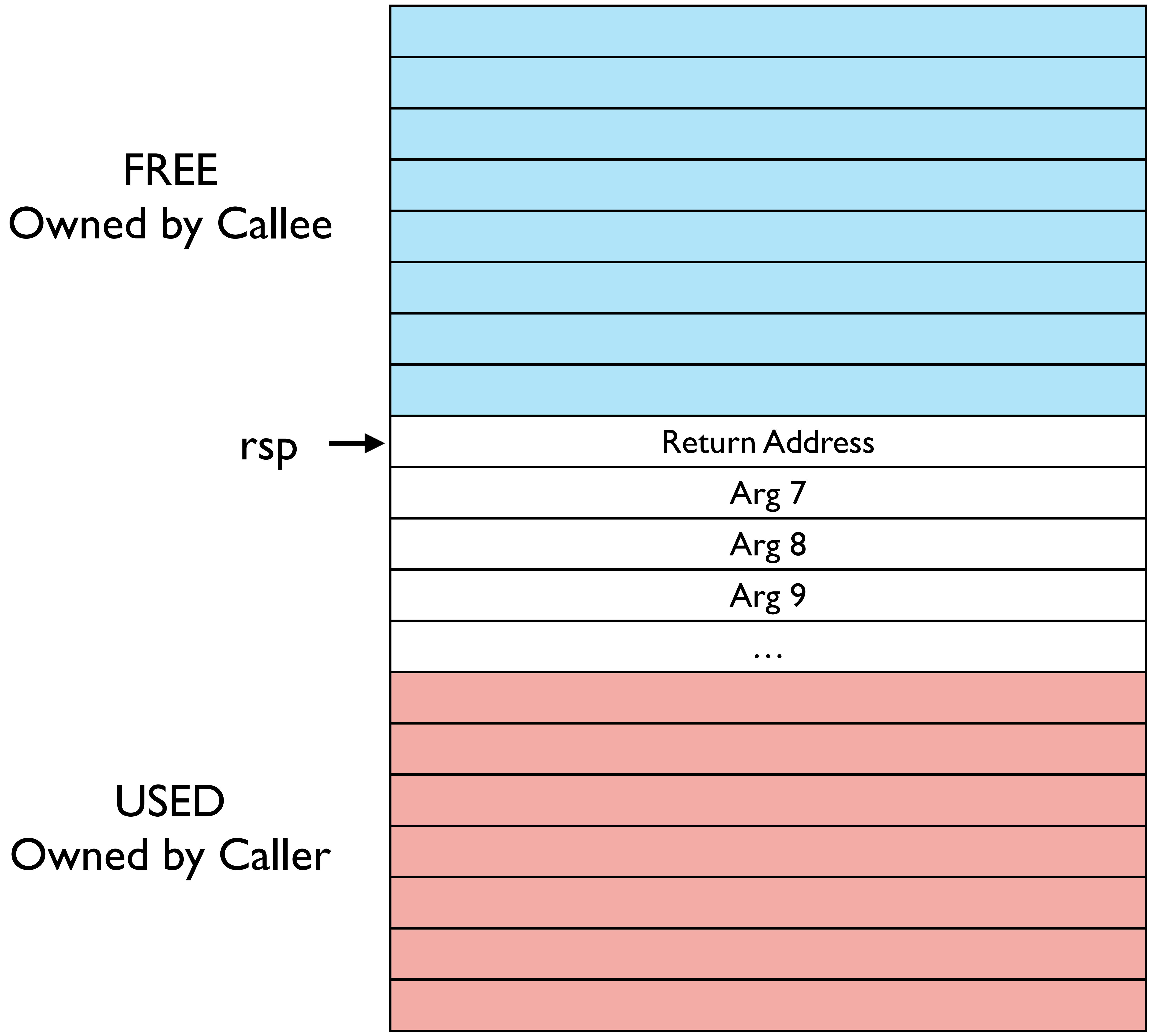
System V AMD 64

Calling protocol: When a called function starts executing the machine state is as follows:

1. Arguments 1-6 are stored in rdi, rsi, rdx, rcx, r8, r9
2. Arguments 7-N are stored in [rsp + 1 * 8], [rsp + 2 * 8],...[rsp + (N - 6) * 8]
3. rsp points to the return address.
4. Stack Alignment: rsp % 16 == 8

System V AMD 64

rdi	Arg 1
rsi	Arg 2
rdx	Arg 3
rcx	Arg 4
r8	Arg 5
r9	Arg 6
rsp	0xXX…X8



System V AMD 64

Returning protocol: When a called function returns to its caller

1. Return value is stored in rax
2. Registers rbx, rbp, r12-r15 are in their original state when the function was called **(non-volatile aka callee-save)**
3. Stack memory at higher addresses than rsp is in the original state when the function was called
4. Original value of rsp holds the return address, pop this address and jump to it

Stack Alignment

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

Needed for certain SSE instructions which require data alignment

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

WARNING: this is a common cause of implementation bugs that can be hard to track down.

If your compiler produces misaligned calls, it might be the case that the code errors on the autograder but not on your local machine.

Caller cleanup

In the SysV AMD 64 calling convention, the **caller** is responsible for "cleanup" of the arguments. That is, when a function returns, the arguments that are passed on the stack are still there, even though their values are not guaranteed to be preserved.

Why?

> Used to implement C-style variadic function. In C, a variadic function doesn't know how many arguments have been passed, so impossible for it to perform caller cleanup.

Downside:

> Impossible to perform tail call to a function that takes more stack-allocated arguments than the caller using SysV AMD 64 calling convention.

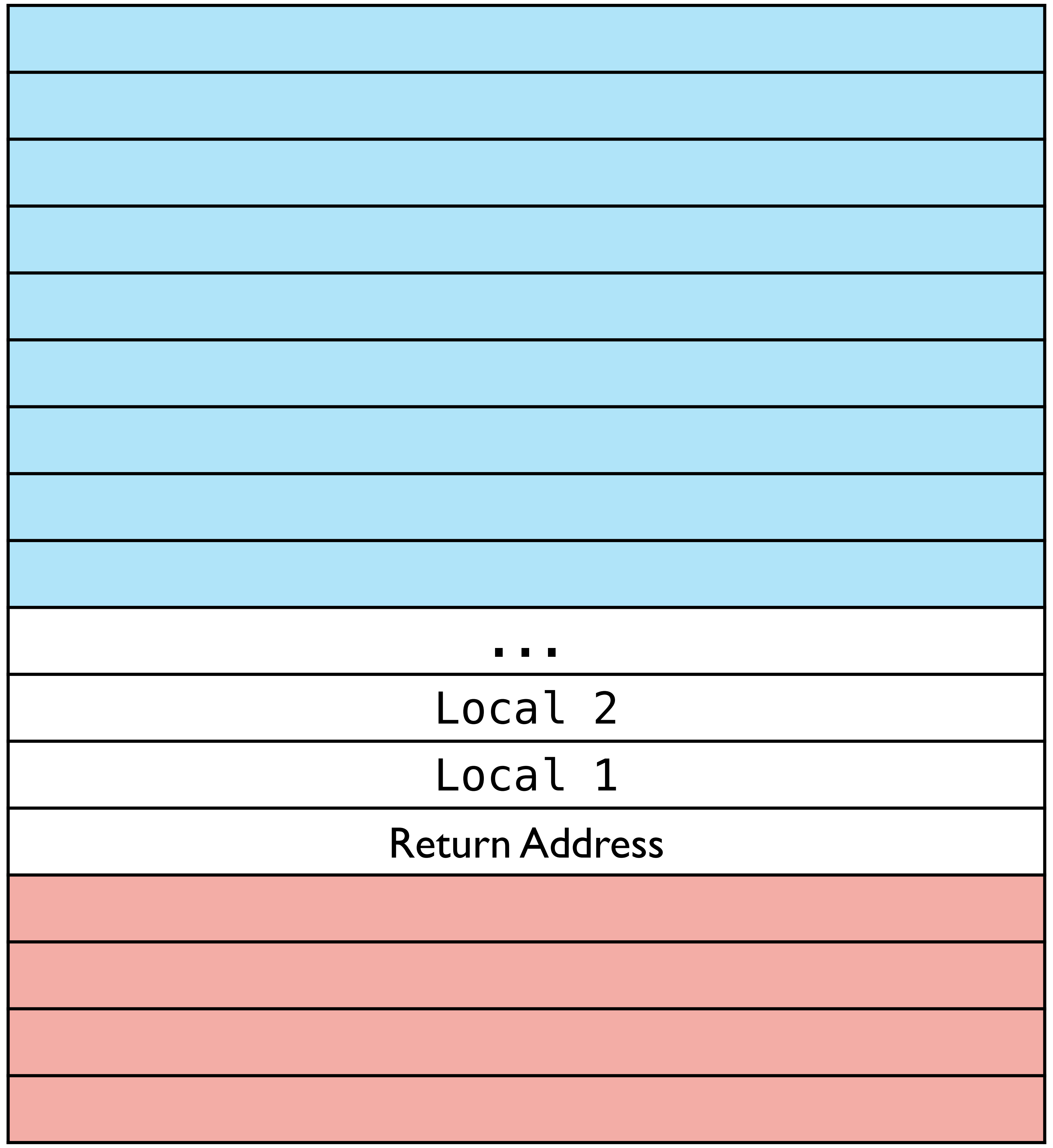
Stack Frames

The region of the stack used by a function is called the stack "frame".

When making a function call we need to ensure that the newly allocated stack frame for the callee is "above" all of our local variables, so that the callee doesn't overwrite their values.

So we need to know where the "top" of our stack frame is in order to determine **where** to create the callee's stack frame.

Two common strategies to do so.



Stack Frame Management: One Pointer

Strategy: use only a base pointer

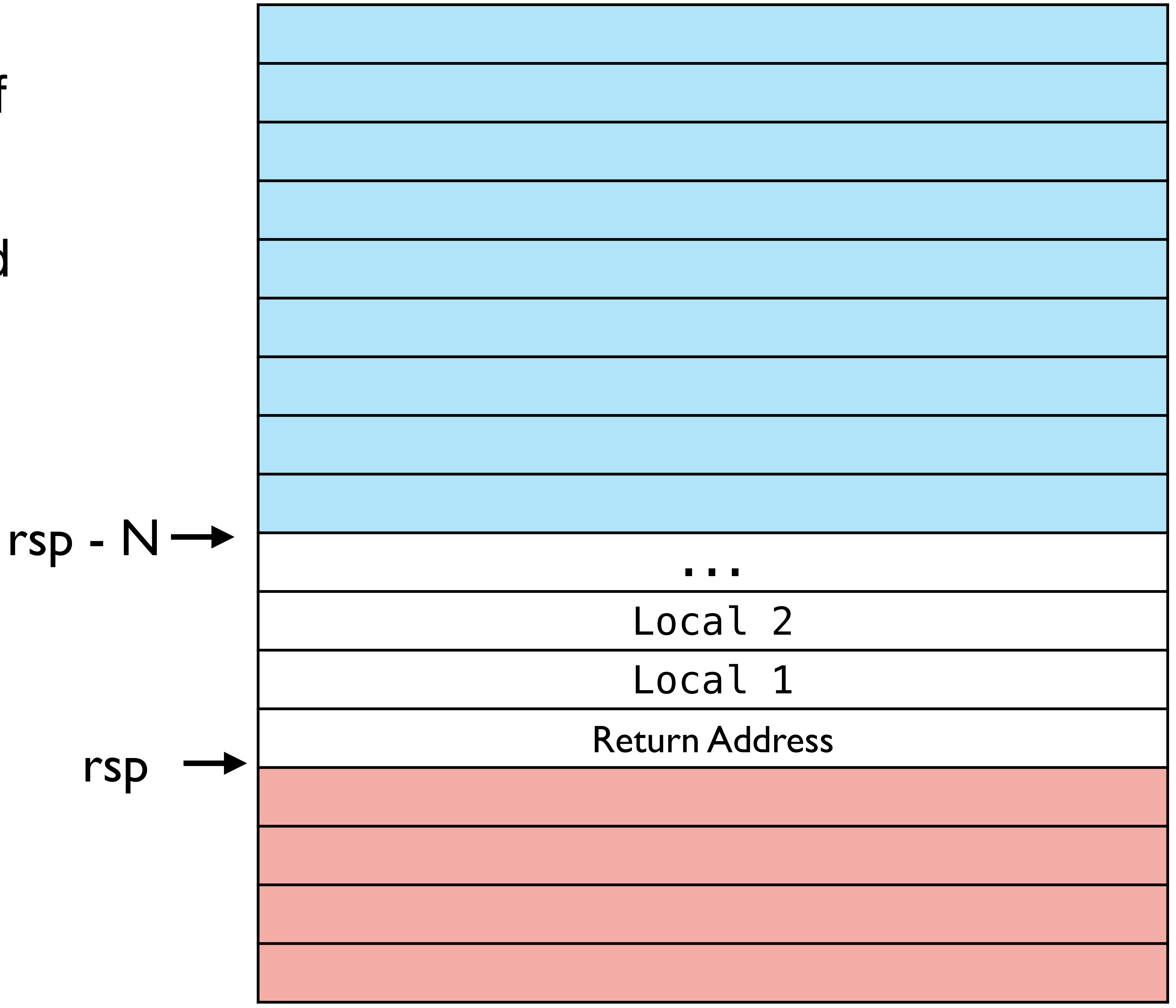
- rsp is the "base pointer" points to the base of the stack frame
- top of the stack frame is statically determined by how much space our locals take up

Benefit:

- frees up rbp to be used for other purposes

Downside:

- size of the stack frame must be statically computable (no alloca)
- complicates the implementation of stack walking mechanisms like debuggers/garbage collectors



Sys V AMD64 Calls

Live code: implement and compile function calls into Rust functions manually in assembly code

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)

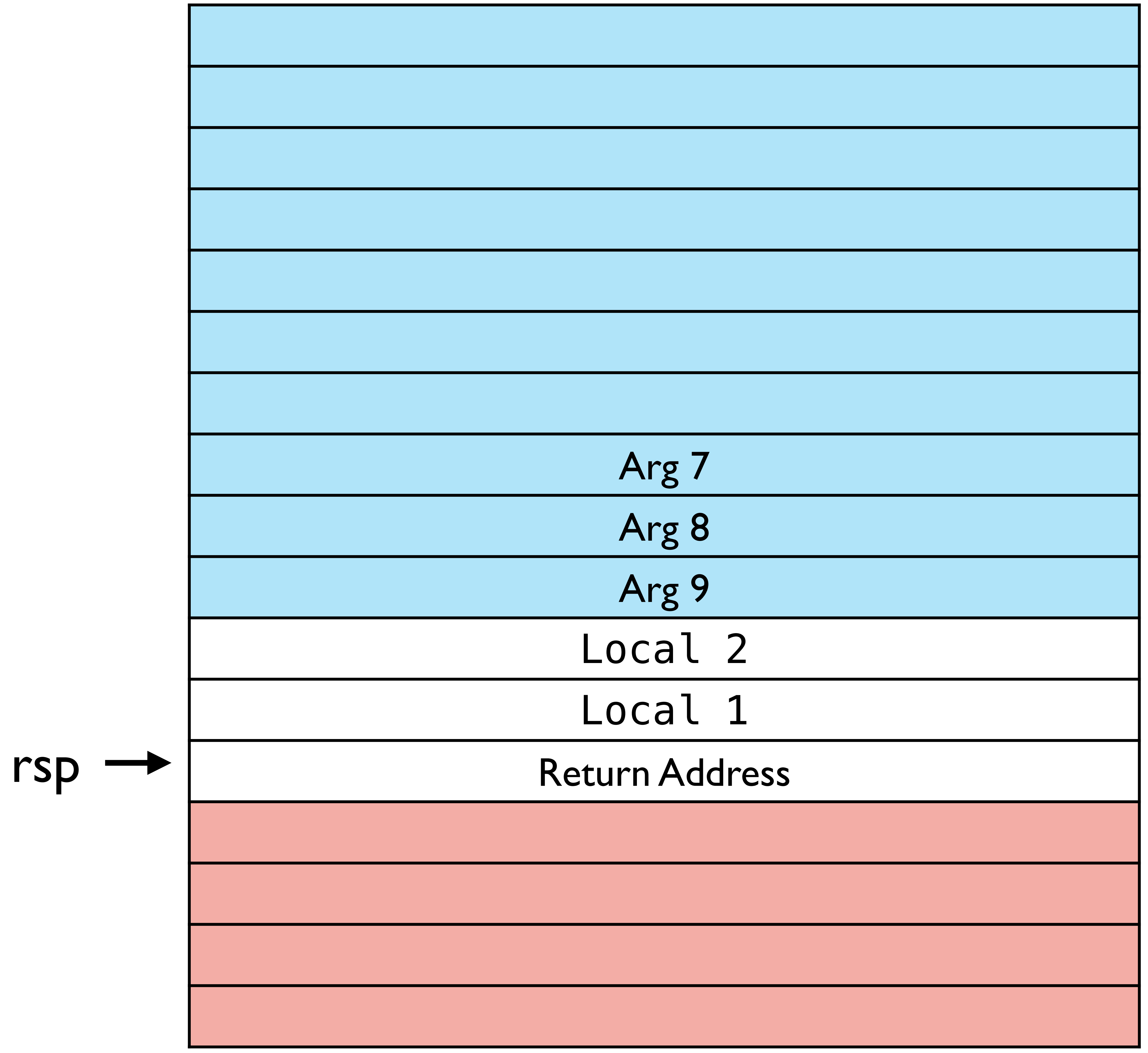
How do we know if we are aligned?

- **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
```

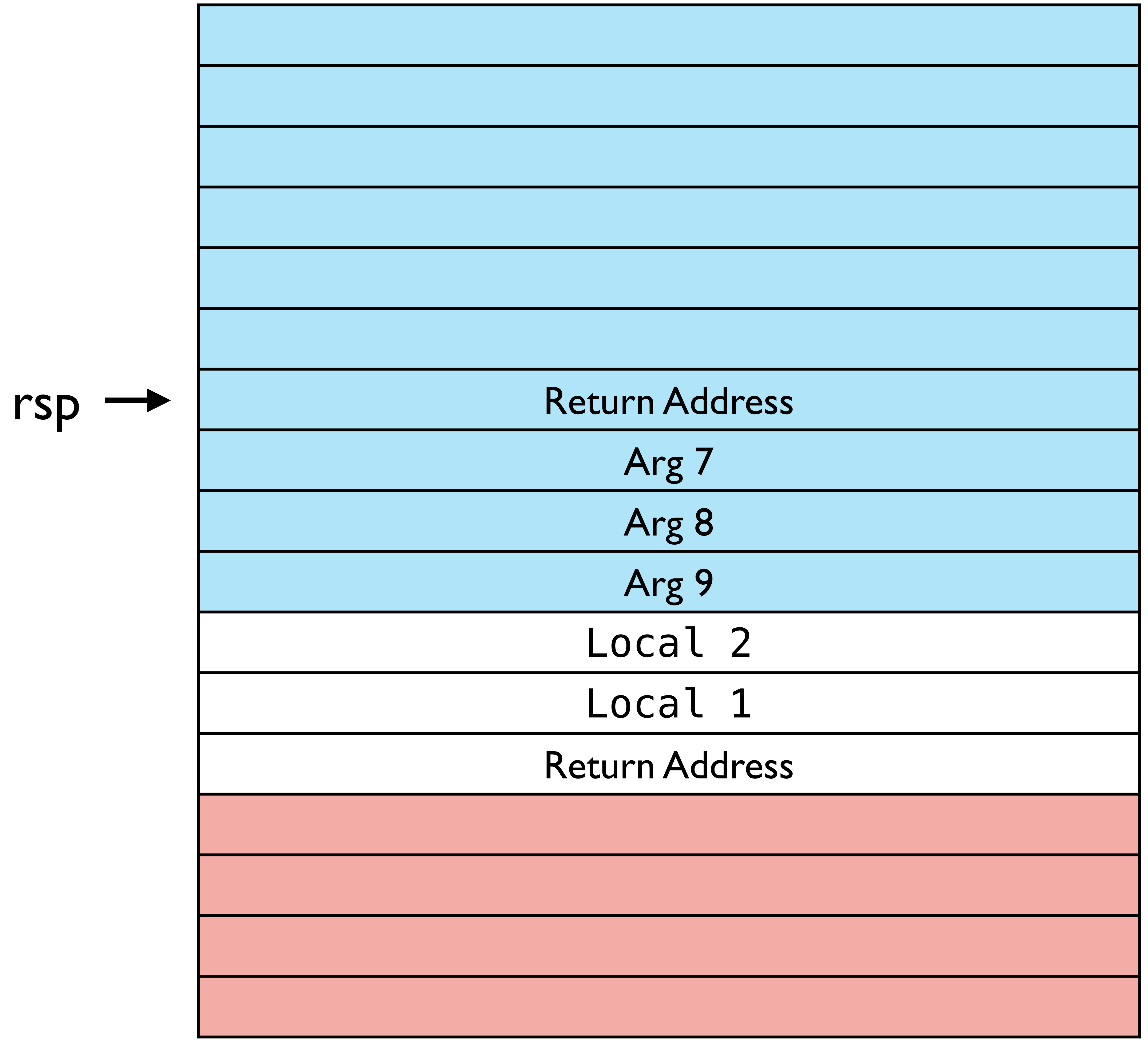
Aligned without padding



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
```

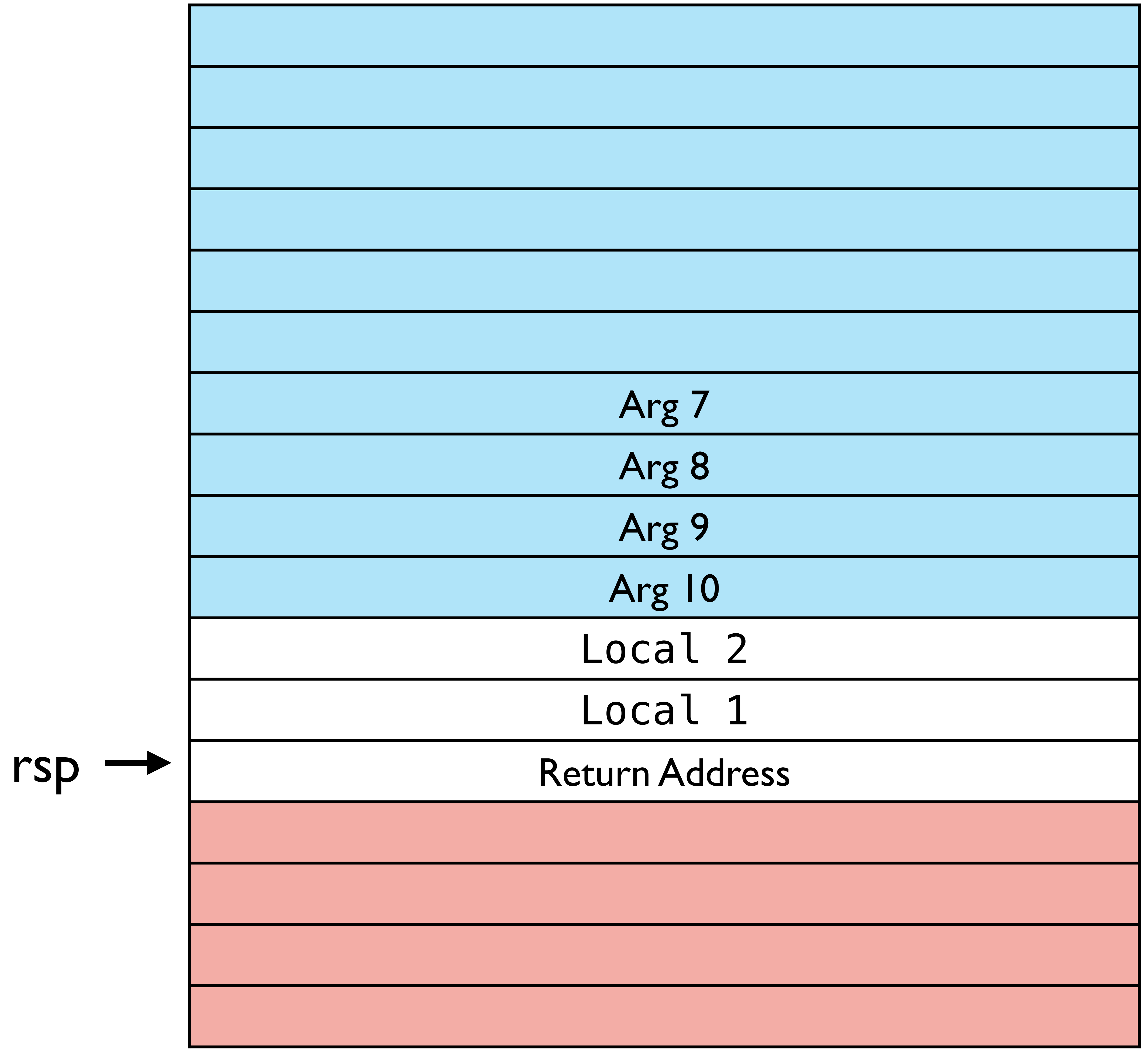
Aligned without padding



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
```

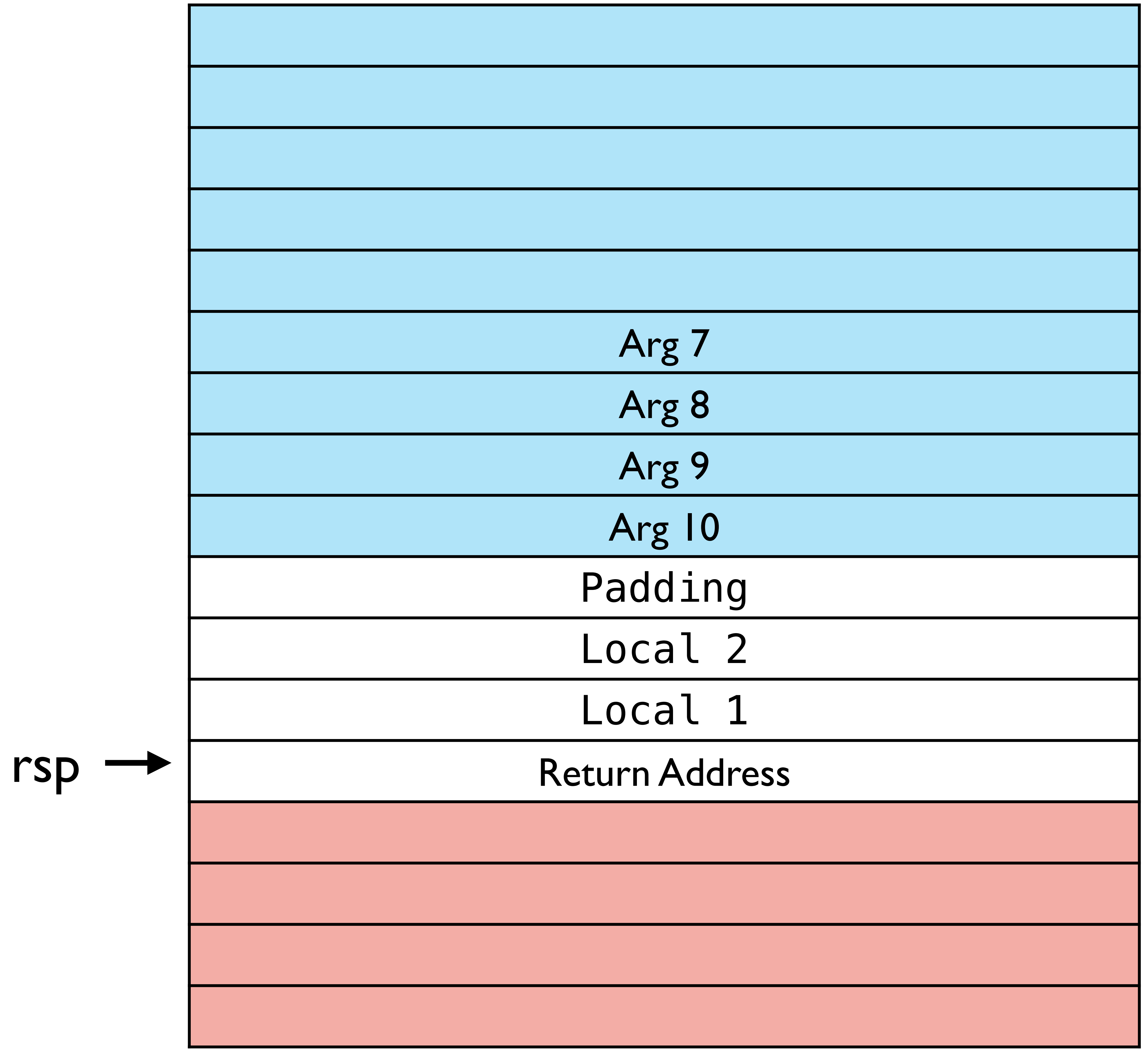
Misaligned without padding



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**

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

non-tail and tail call of the same function

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?

Example: max (live code)

Example: max (live code)

Make two **different** labeled code blocks in assembly:
- a block that is tail called, just as in Boa.
- a block that is called, which then moves the arguments to the stack and jumps to the tail call block.

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.

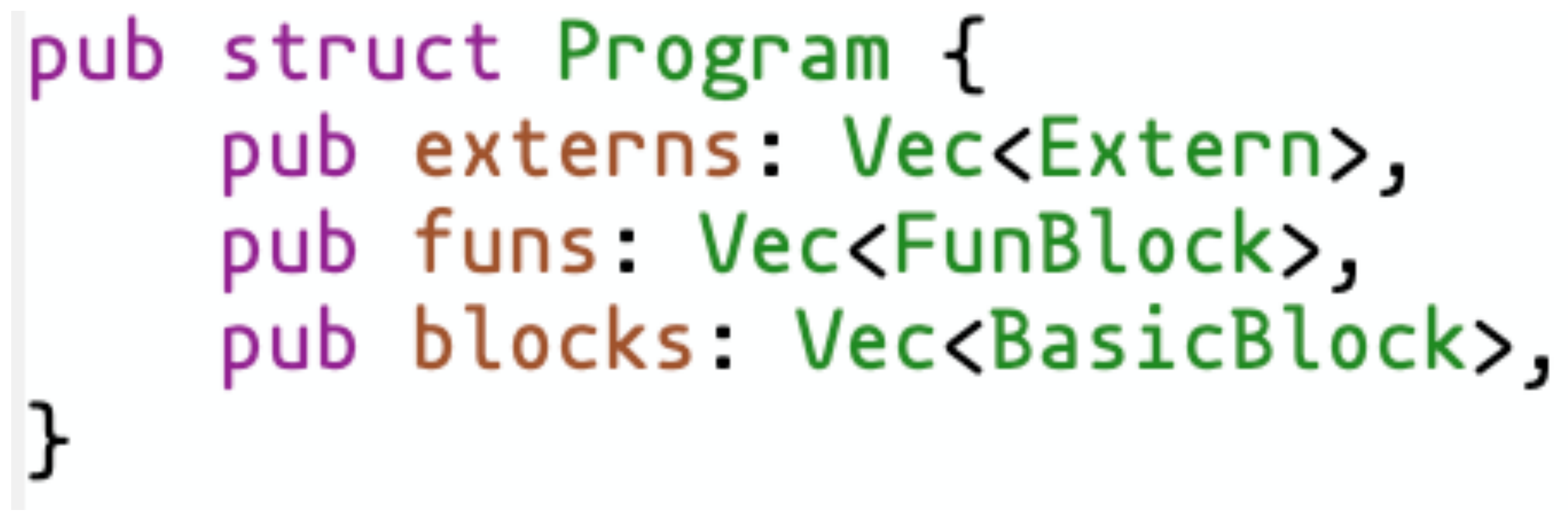
In code generation: compile these as moving the arguments from the SysV AMD64-designated locations to the stack.

Change to SSA: Abstract Syntax

An SSA program has three parts:
1. Extern declarations
2. Function blocks
3. Top-level Basic blocks

Side condition: one of the function blocks has the unmangled name "entry", corresponding to the main function in the source program.

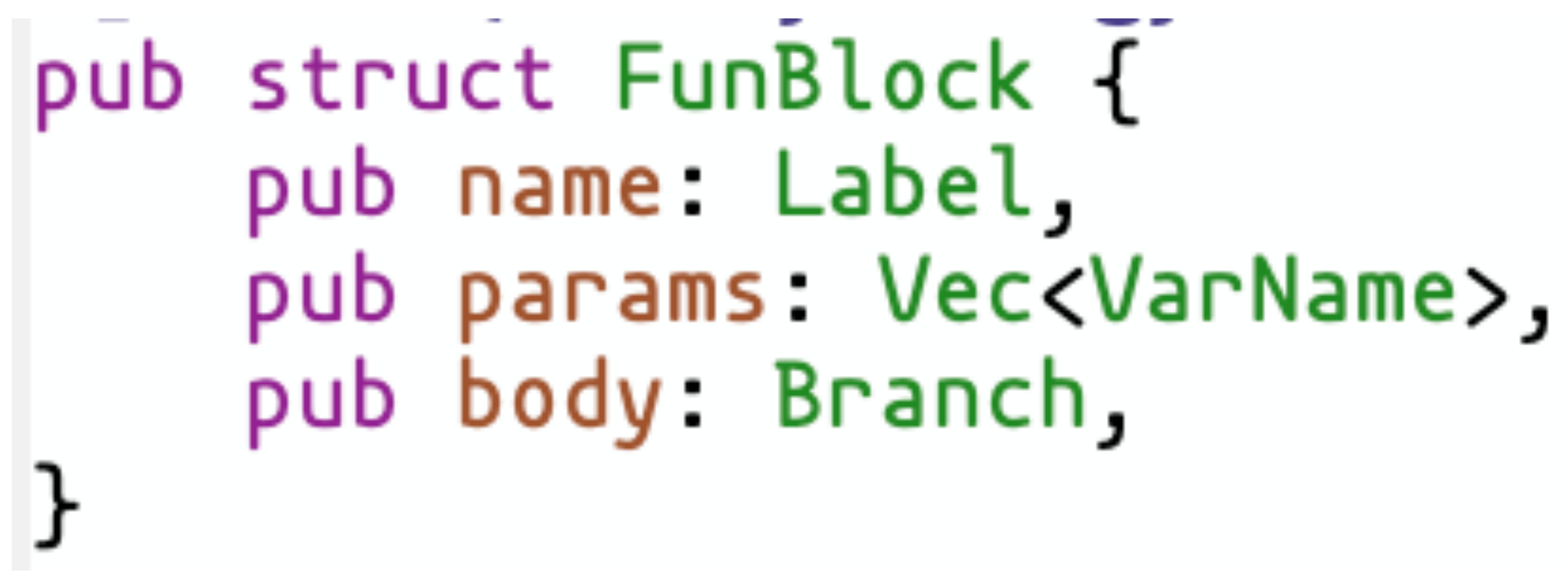
All of these are globally scoped: functions can branch to any of the top-level blocks and vice-versa the blocks can call any of the functions.



```
pub struct Program {
    pub externs: Vec<Extern>,
    pub funs: Vec<FunBlock>,
    pub blocks: Vec<BasicBlock>,
}
```

Change to SSA: Abstract Syntax

Function blocks always have the same structure: immediately branch to one of the top-level blocks



```
pub struct FunBlock {
    pub name: Label,
    pub params: Vec<VarName>,
    pub body: Branch,
}
```

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

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.

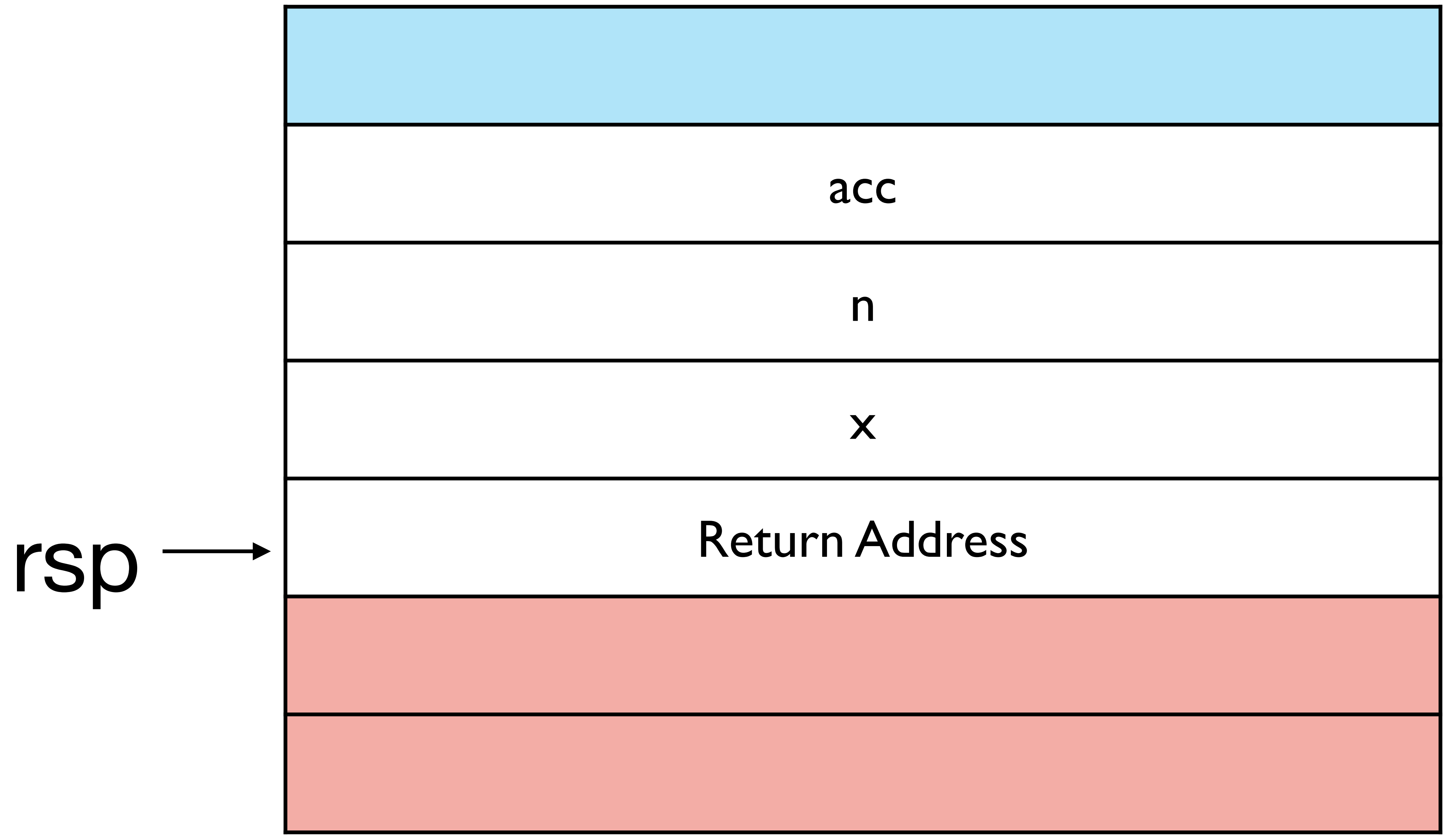
Why does this work?

Variable Capture

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

Variable Capture

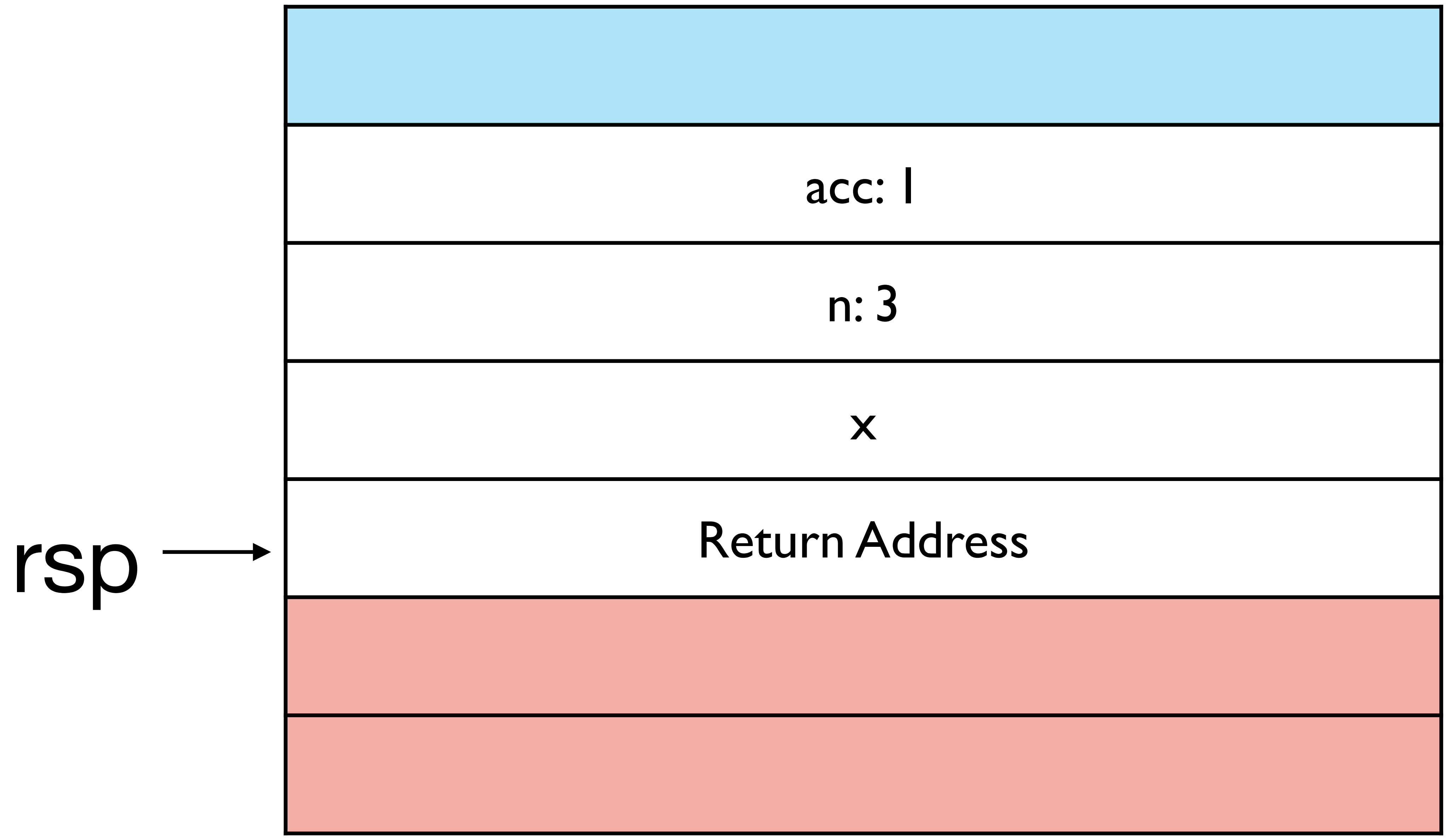
```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```



Variable Capture

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

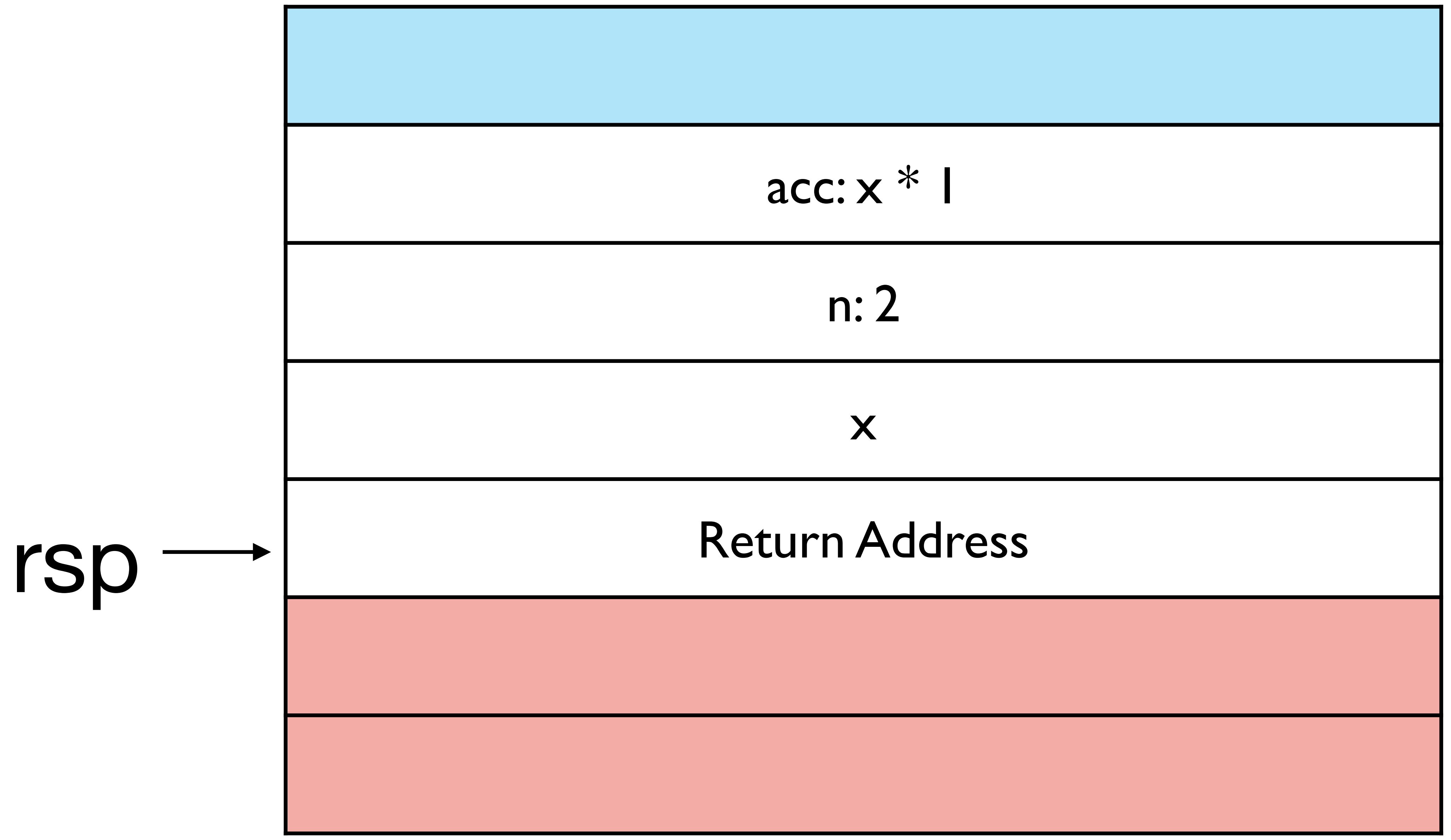
first iteration



Variable Capture

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

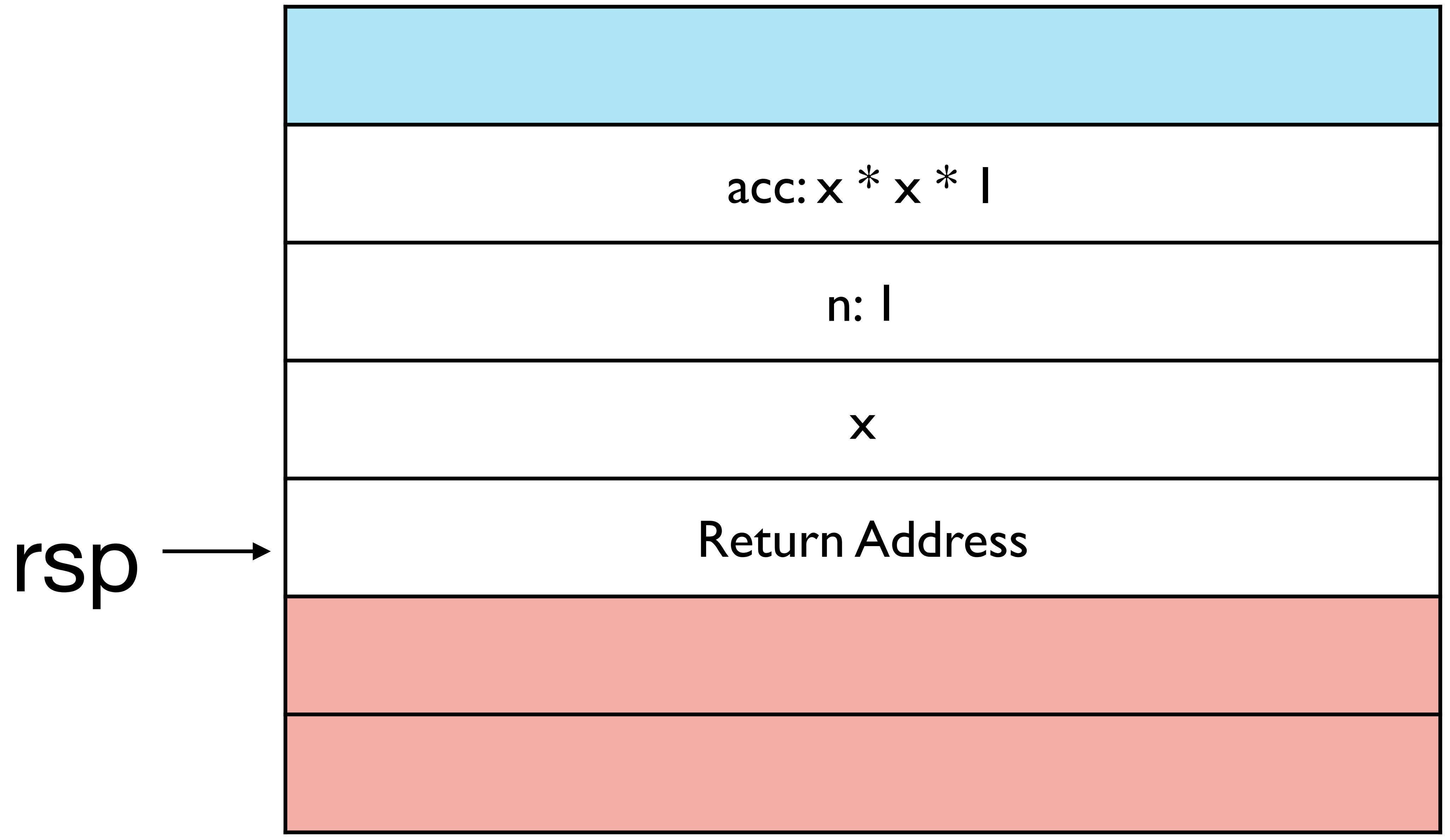
second iteration



Variable Capture

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

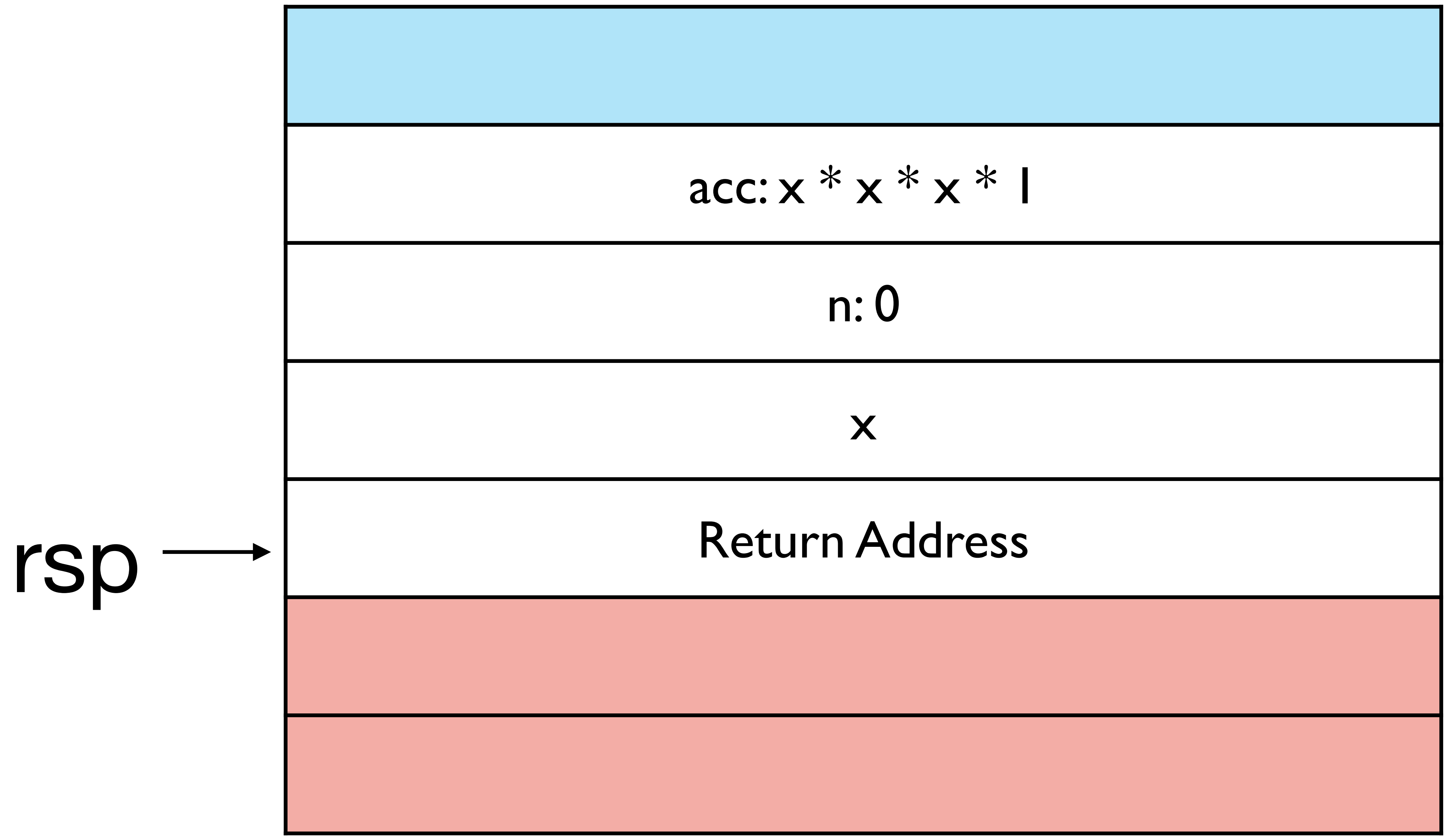
third iteration



Variable Capture

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

final iteration



Variable Capture

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

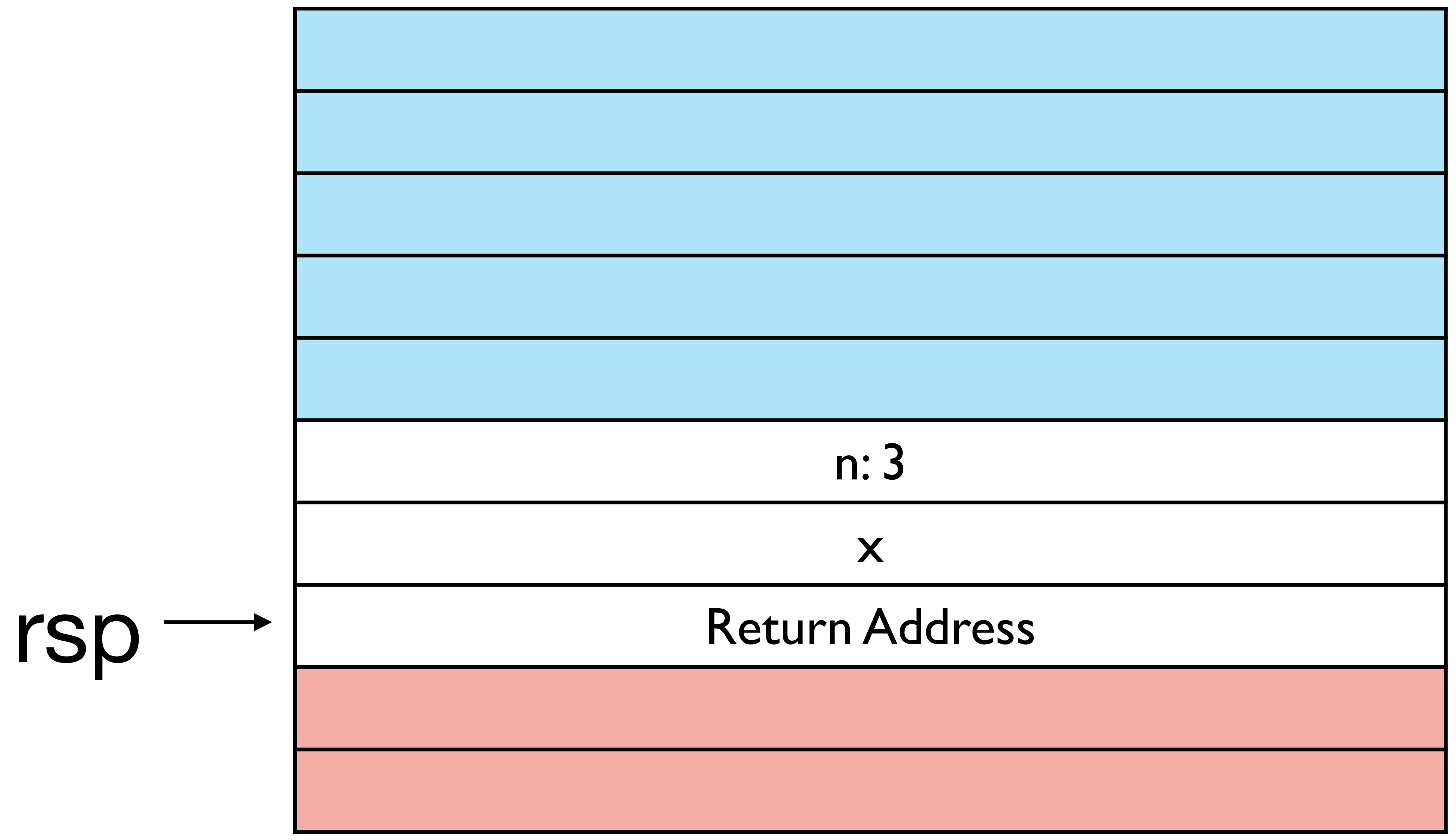
x is "captured" by the function **pow** in that it is used in the function because it is in scope when the function **pow** is defined.

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)
```

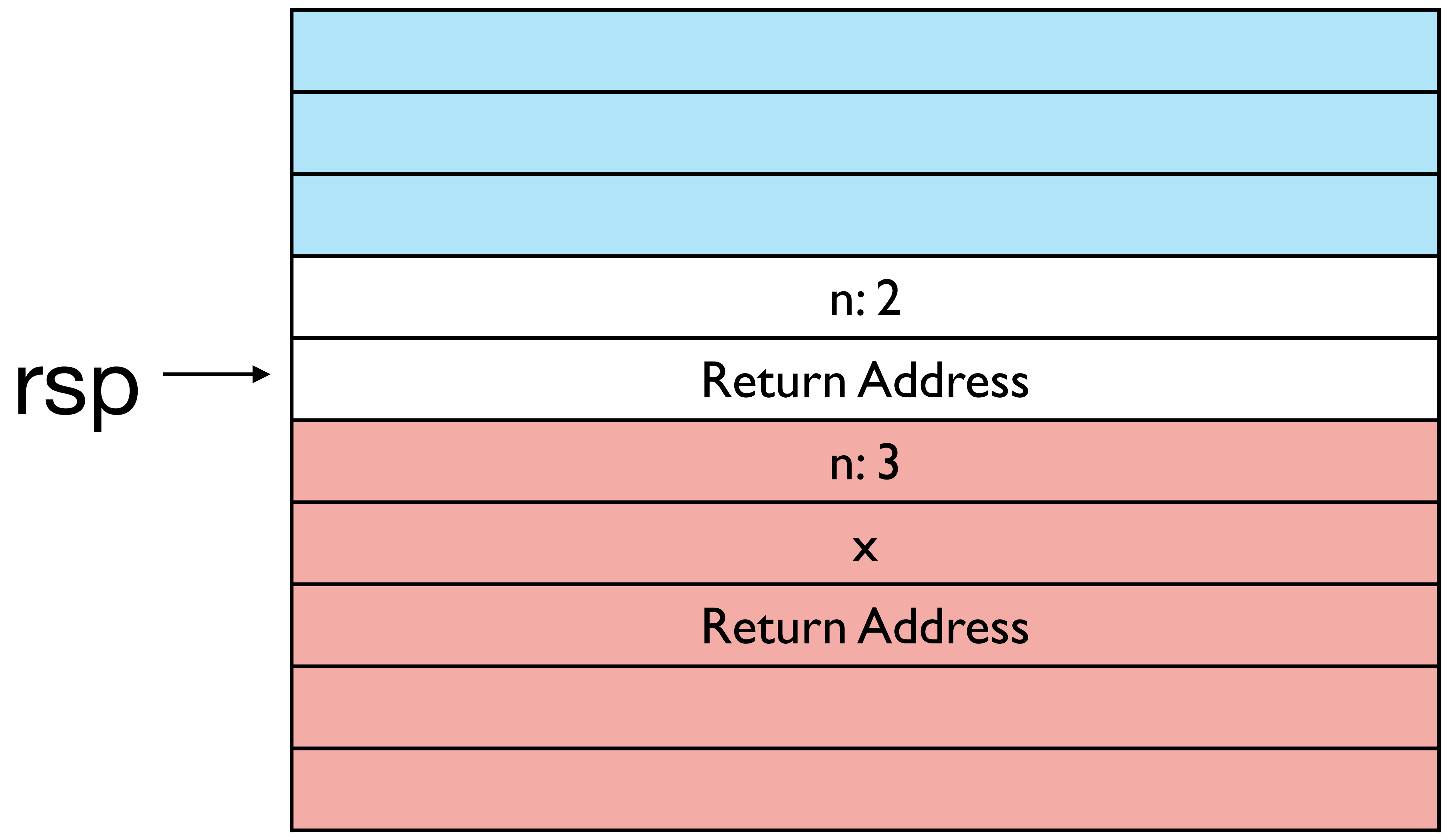
initial tail call



Variable Capture

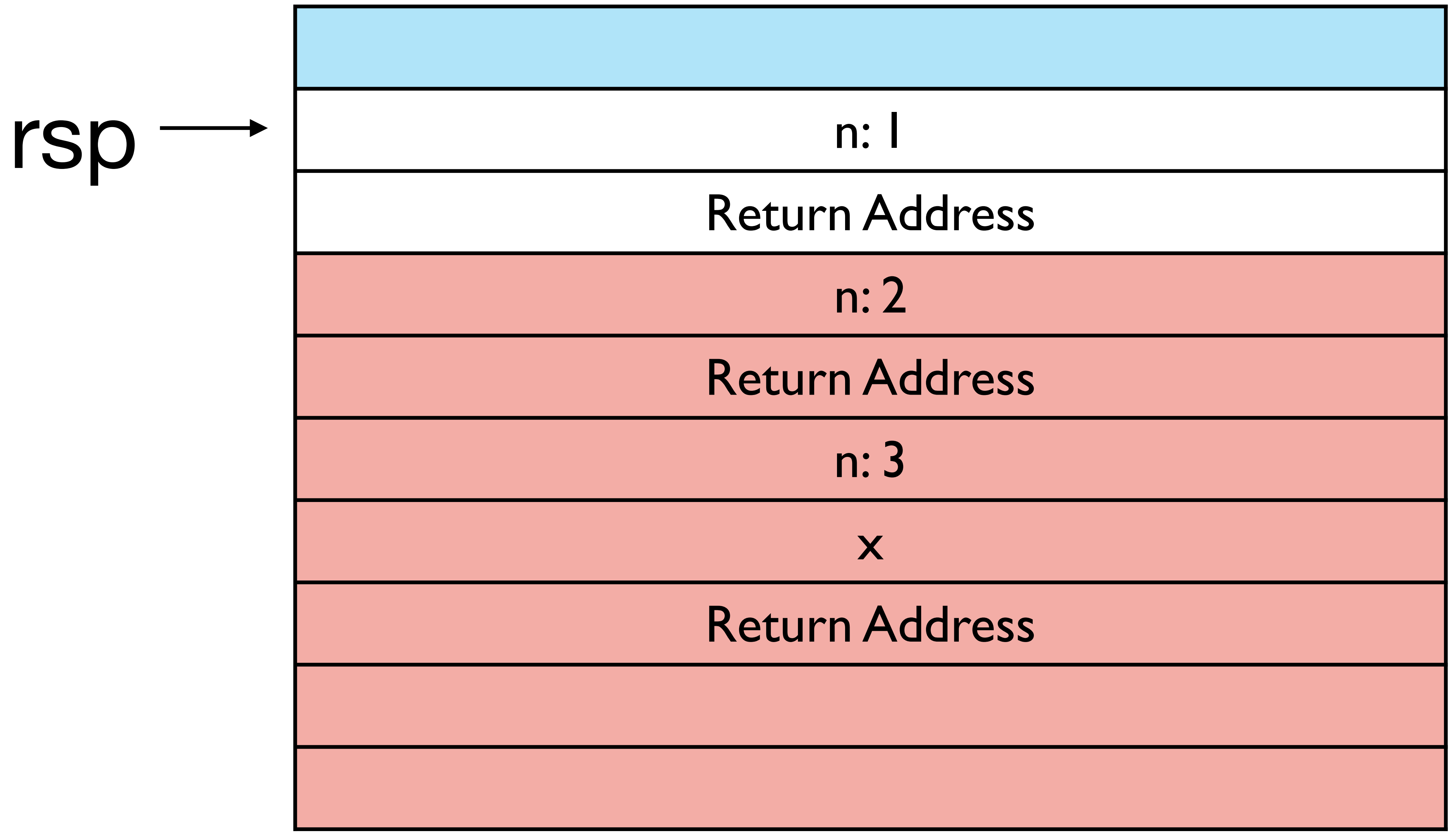
```
def main(x):
  def pow(n):
    if n == 0: 1 else: x * pow(n - 1)
  in
  pow(3)
```

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

Variable Capture

Variable capture in a tail-called function is not a problem, as the captured variables are still available in our local stack frame.

Variable capture in a called function is an issue: the distance from the current stack pointer to the location of the captured variable is not statically determined

How can we solve this issue?

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)
```

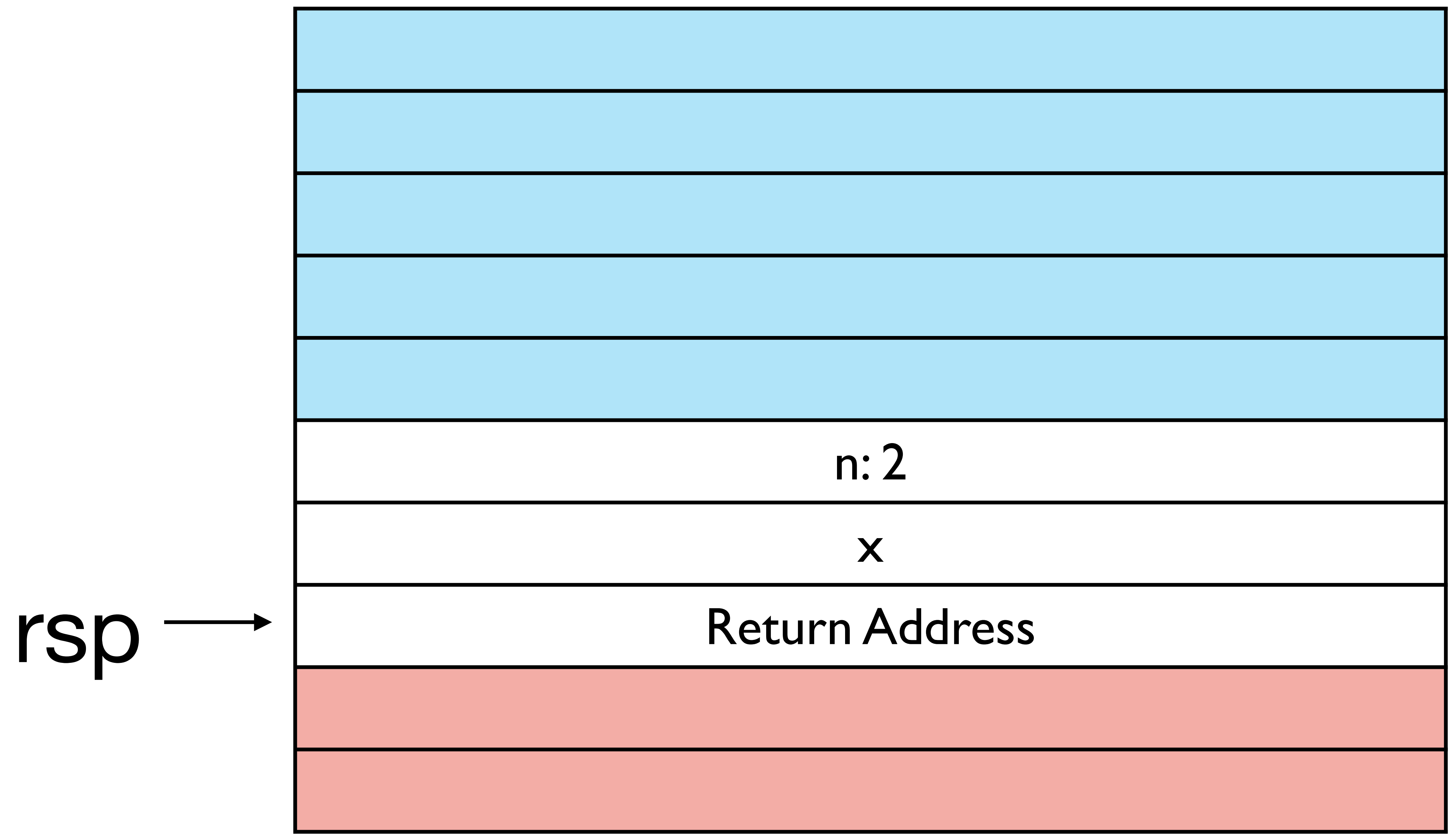
⬇

```
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)
```

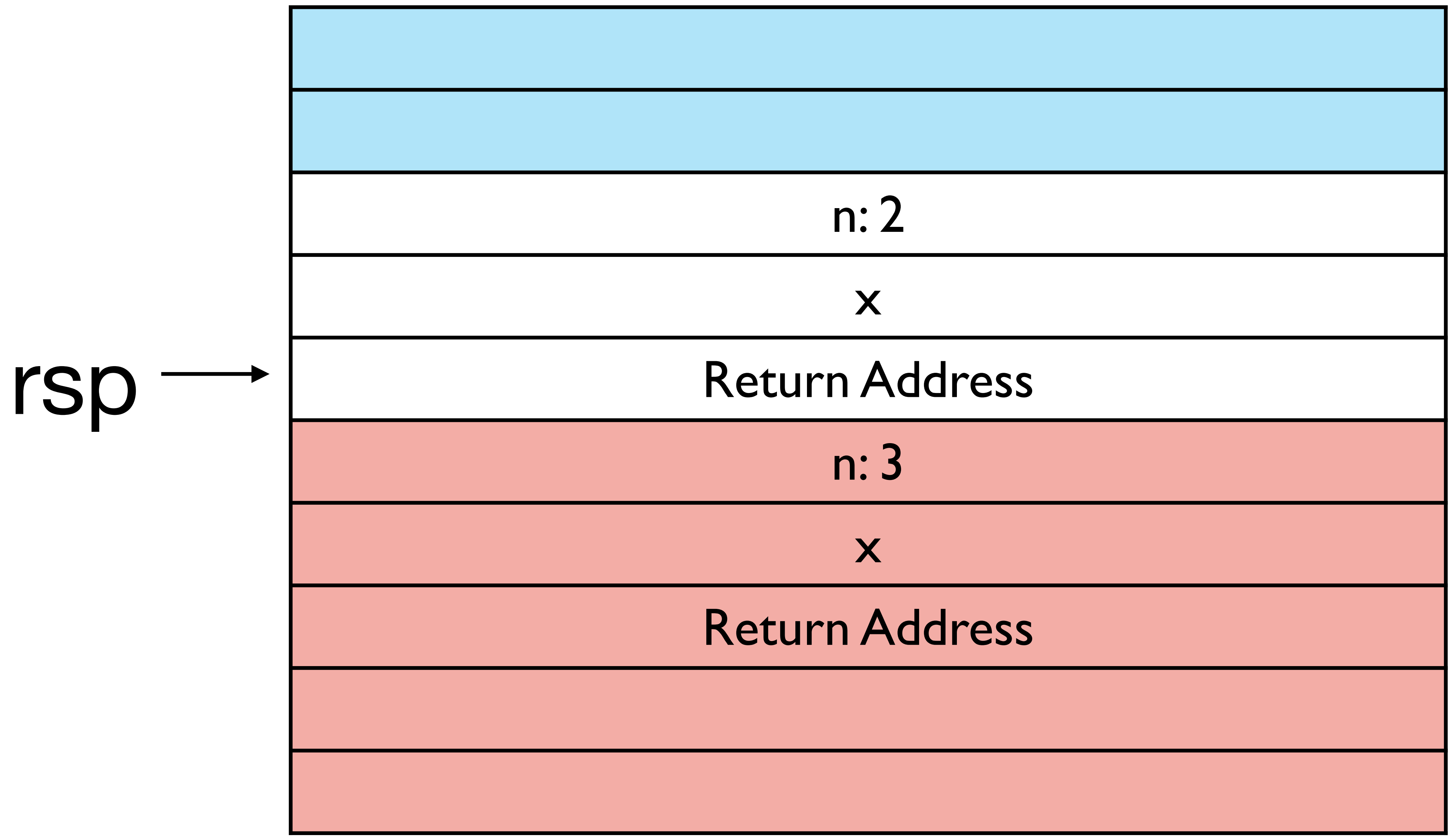
initial tail call



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.