EECS 483: Compiler Construction

Lecture 9:
Non-tail Function Definitions, Lambda Lifting

February 12
Winter Semester 2025

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.

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>,
}
```

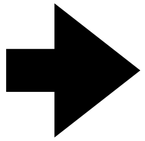
Change to SSA: Abstract Syntax

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

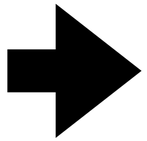
In SSA, we make a distinction between **functions** and **parameterized blocks**
Both have a label and arguments, but the way they are used and compiled is different

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.

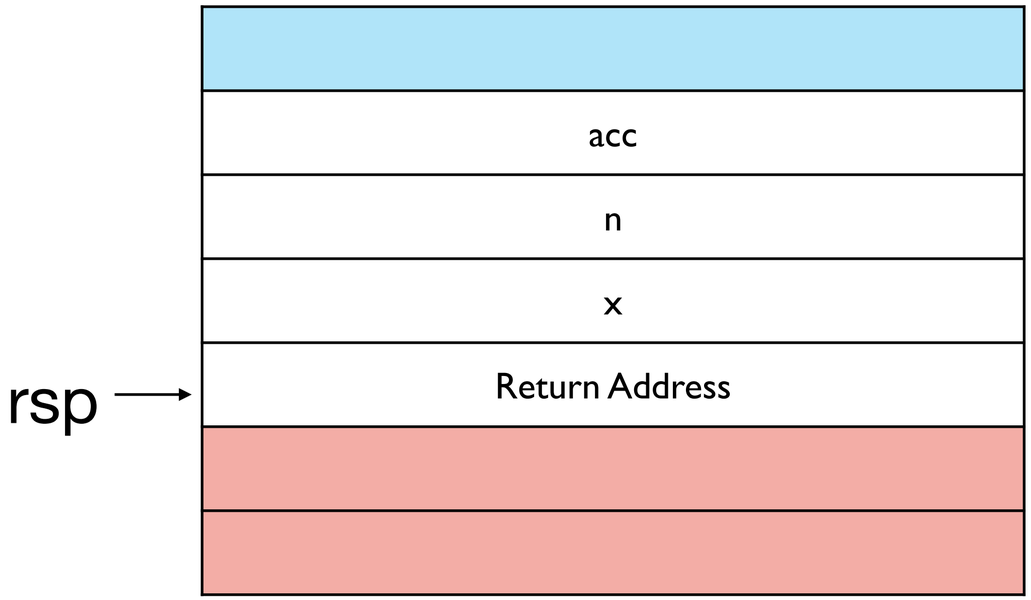
Why does this work?

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

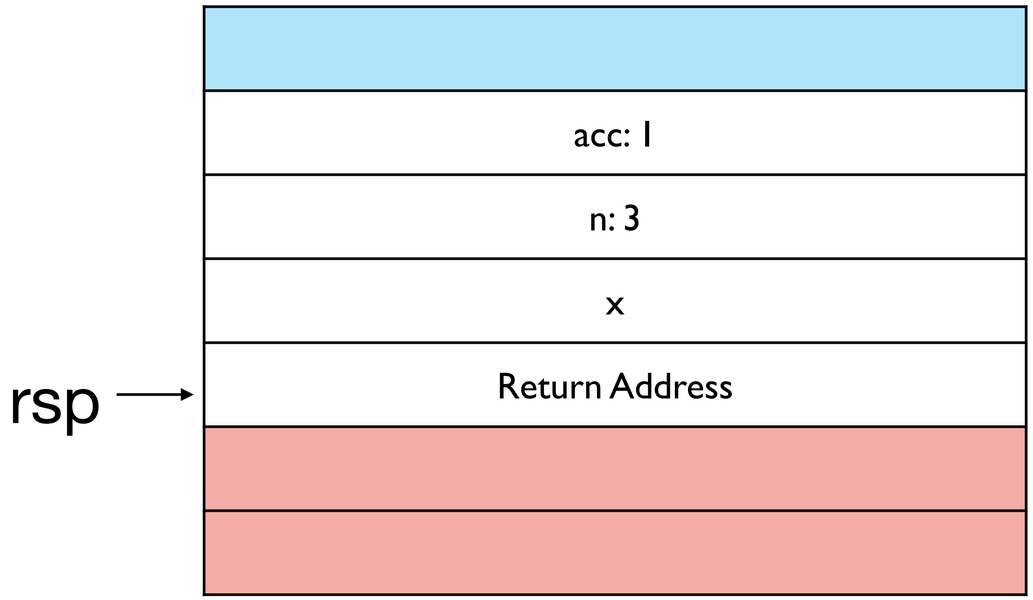
```
def main(x):
  def pow(n, acc):
    if n == 0: acc else: pow(n - 1, x * acc)
  in
  pow(3, 1)
```



Variable Capture

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

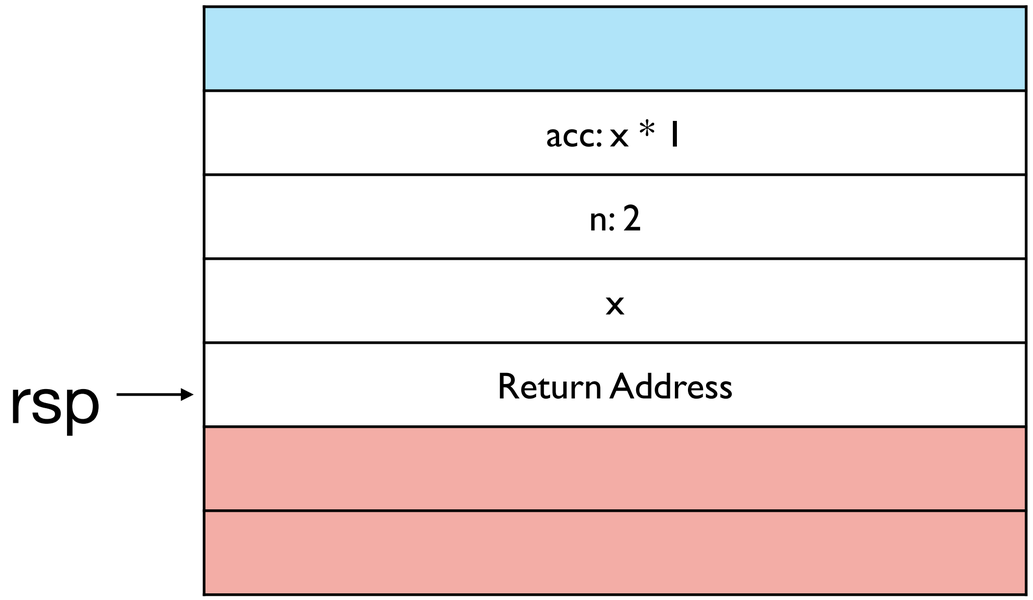
first iteration



Variable Capture

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

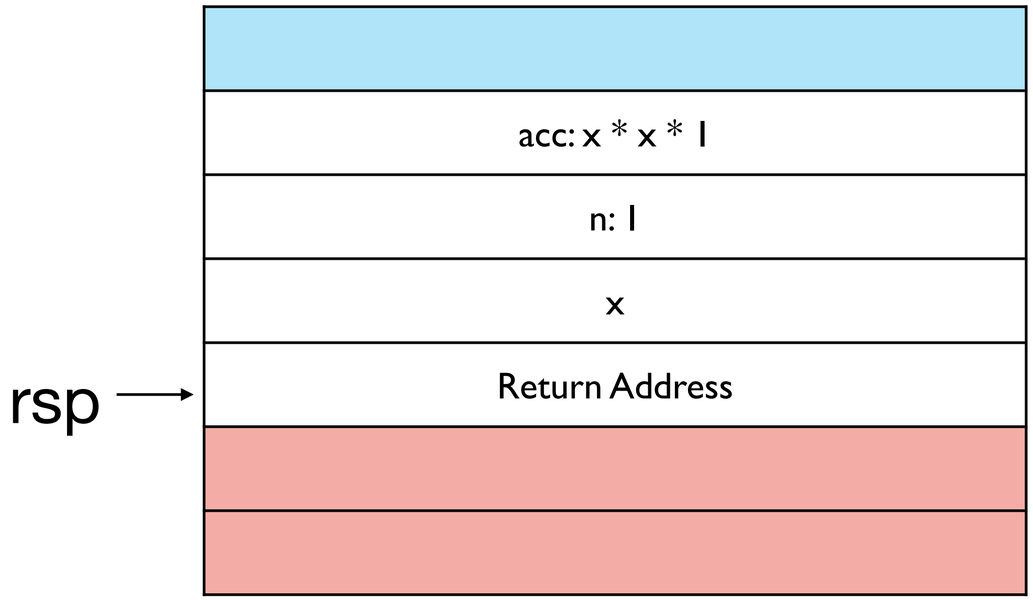
second iteration



Variable Capture

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

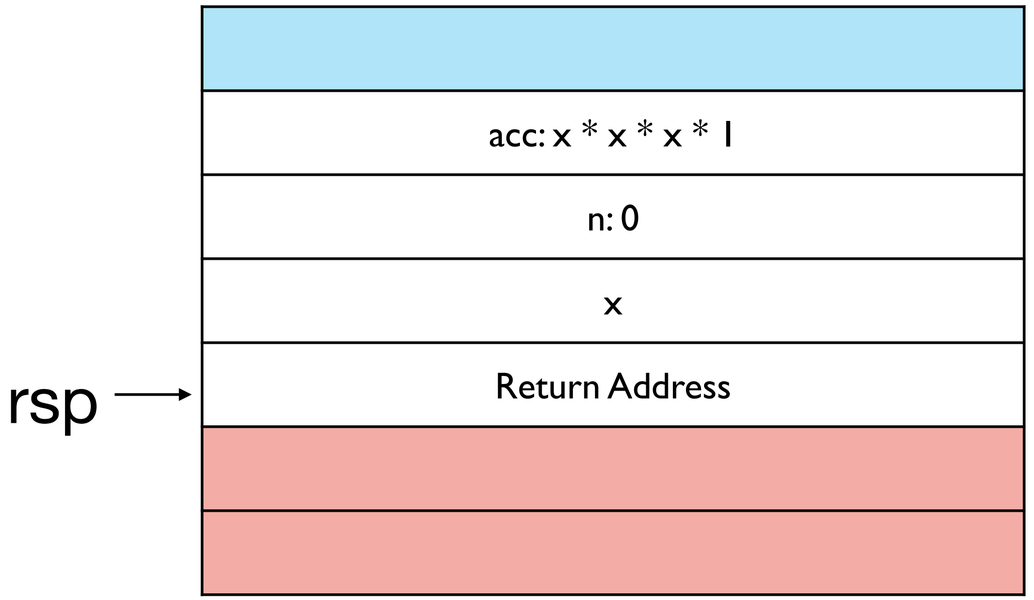
third iteration



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)
```

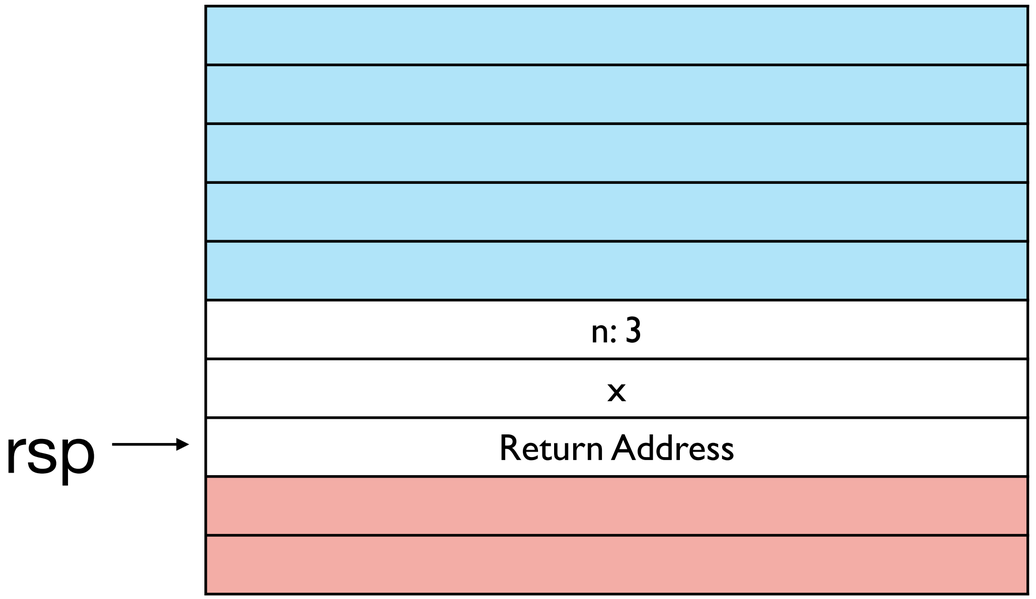
x is "captured" by the function **pow** in that it is used in the function because it is in scope when the function **pow** is defined.

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)
```

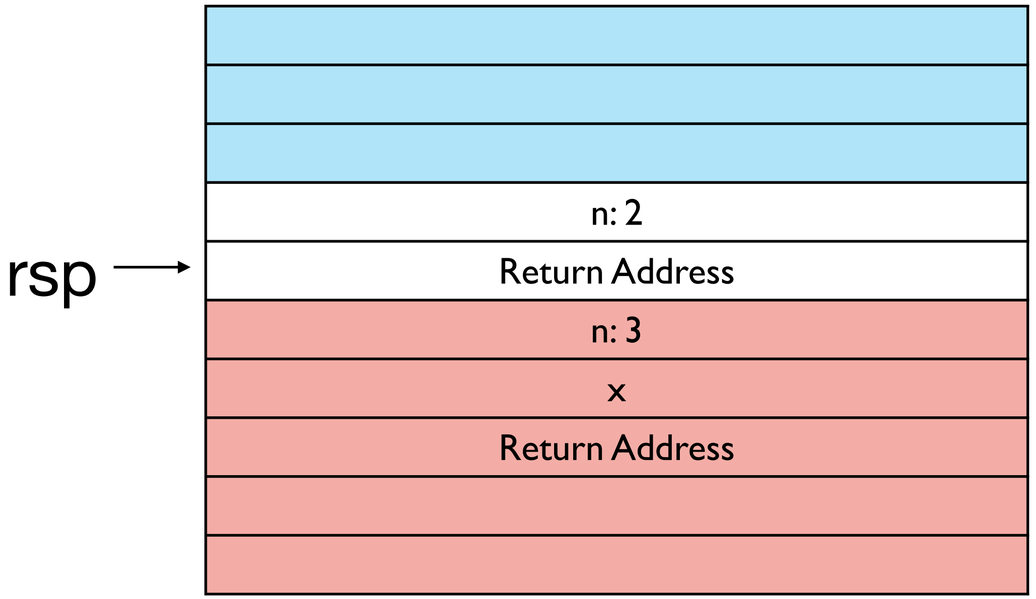
initial tail call



Variable Capture

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

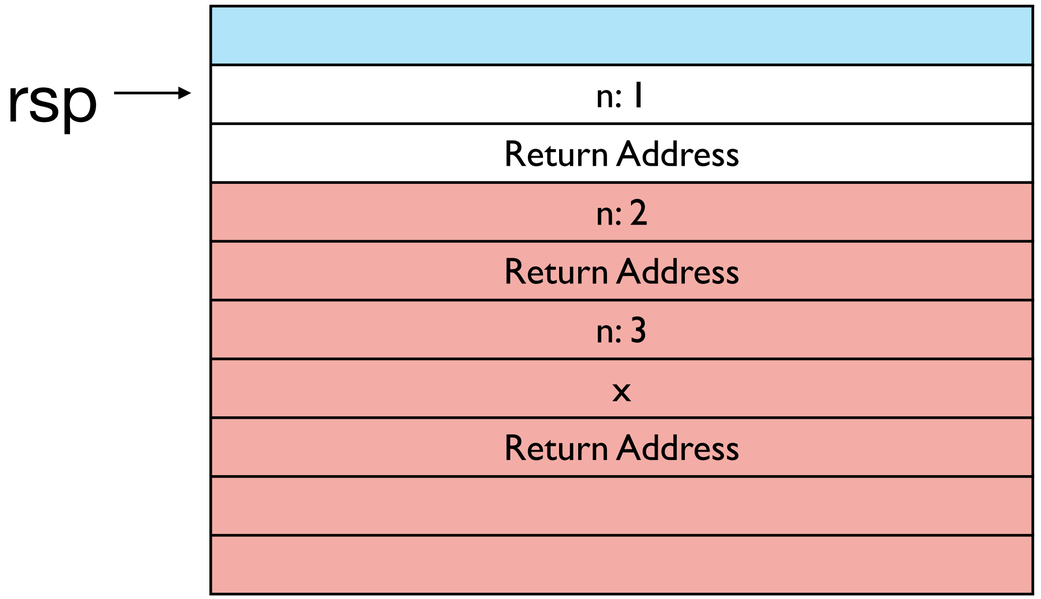
first non-tail recursive call



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

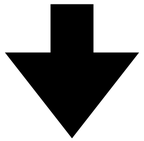
Variable capture in a tail-called function is not a problem, as the captured variables are still available in our local stack frame.

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

Can solve this by **copying** the value into each stack frame.
Implement as a code transformation: add all the captured variables as **extra arguments**. This process is called **lambda lifting**.

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)
```

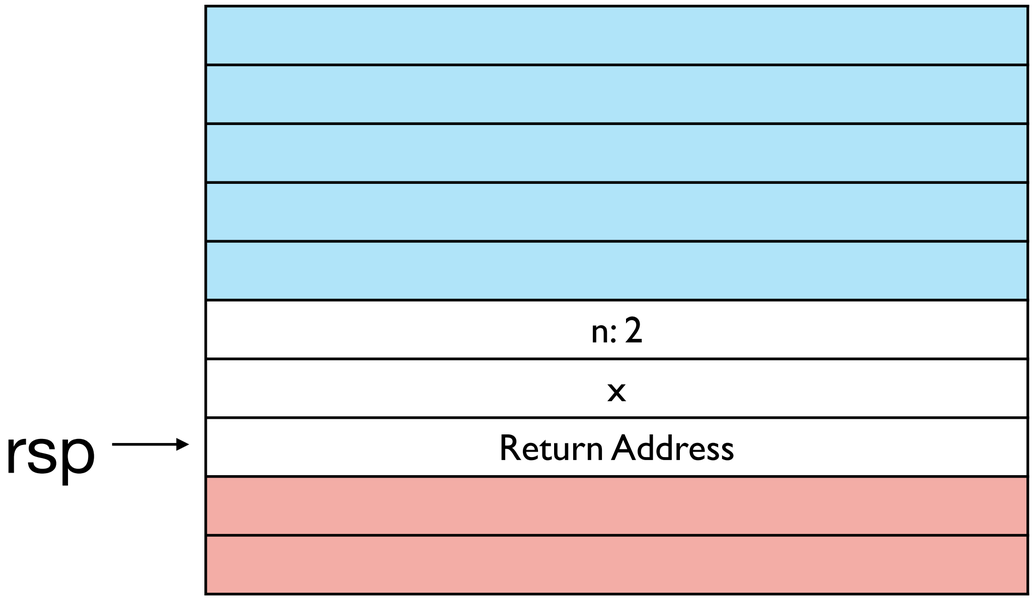


```
def pow(x, n):
  if n == 0: 1 else x * pow(x, n - 1)
def main(x):
 pow(x, 3)
```

Variable Capture

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

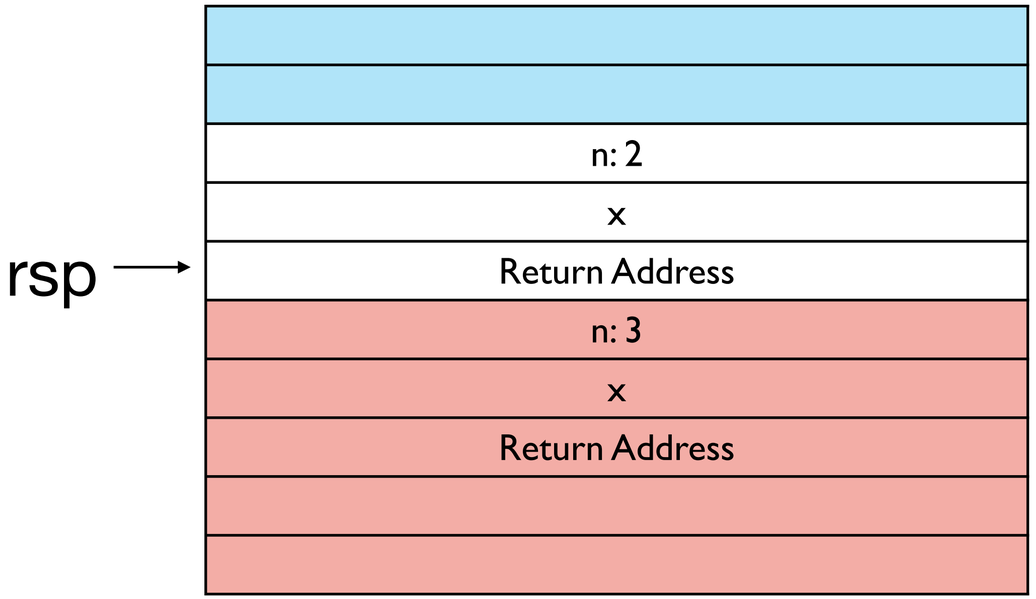
initial tail call



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



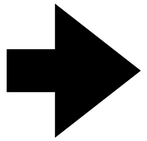
Lambda Lifting

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?

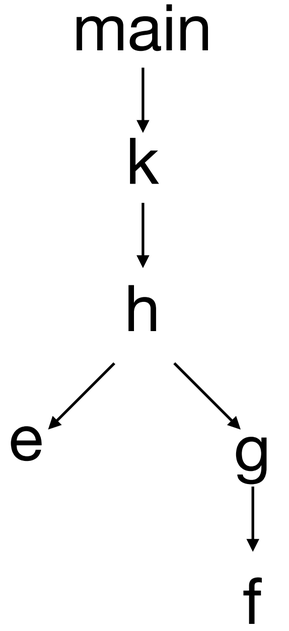
Answer:
h must be lifted because it is non-tail called
g, e must be lifted because they are tail called by a lifted function
f must be lifted because it is tail called by a lifted function
k does not need to be lifted

```
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



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

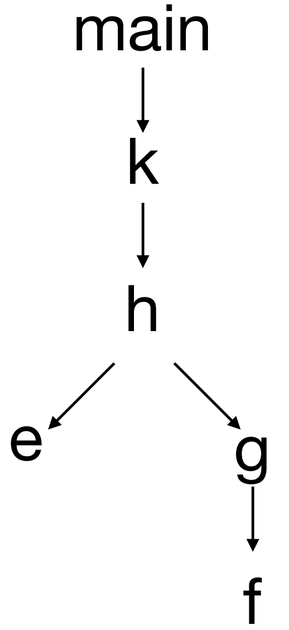
Implement by worklist algorithm:

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



Lambda Lifting: What Args to Add

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

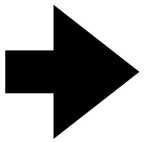
Answer: all the non-local variables that **must** be in the function's stack frame.

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

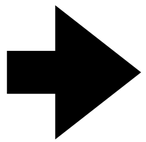
Adding all variables that are in scope is correct, but inefficient. Why?

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)
```



```
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

New source language features:

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

Cobra: Frontend Changes

1. Remove errors for calling non-tail called functions
2. Ensure that externed 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