# EECS 483: Compiler Construction

## Lecture 3:
## Complex Expressions, Evaluation Order, Basic Blocks, Continuations

**January 21, 2025**

## Announcements

- Assignment 1 is due next Friday, the 30th. Should get through all relevant material today.

- Office Hours reminder:

  Max: Monday and Thursday 3-4:30pm, Beyster 4628

  Yuchen: Wed 3-4pm, Friday 1:30-2:30pm, Beyster Atrium

## Learning Objectives

- Finish Simple Allocation scheme for local variables
- Address semantic questions in multi-argument operations
- How to compile nested recursive expressions to sequential assembly code
- Introduce the first version of our intermediate representation

## Compiling Let

In the interpreter, the value of each variable was stored in a HashMap.

In the compiled code, we correspondingly need to ensure that we have access to the value of each variable somewhere in **memory**

## x86 Memory Model

16 general-purpose 64-bit registers

- rax, rcx, rdx, rbx, rdi, rsi, rsp, rbp, r8-r15

Each holds a 64-bit value, so 128 bytes of extremely fast memory.

The abstract machine also gives us access to a large amount of memory, which is addressable by byte.

- Addresses are 64-bit values, though in current hardware only the lower 48-bits are used. This gives us access to 2^48 bytes of address space, or 128 terabytes.

## x86 Instructions: mov

mov dest, src

In a mov, the dest and src can be registers or memory addresses.

Use square brackets [ ] to "dereference" an address.

- mov rax, rdi copies the value stored in rdi to rax
- mov rax, [rdi] loads the memory at address rdi into rax
- mov [rax], rdi stores the value of rdi in the memory at address rax
- mov [rax], [rdi] - not allowed in x86 syntax

# x86 Memory Conventions

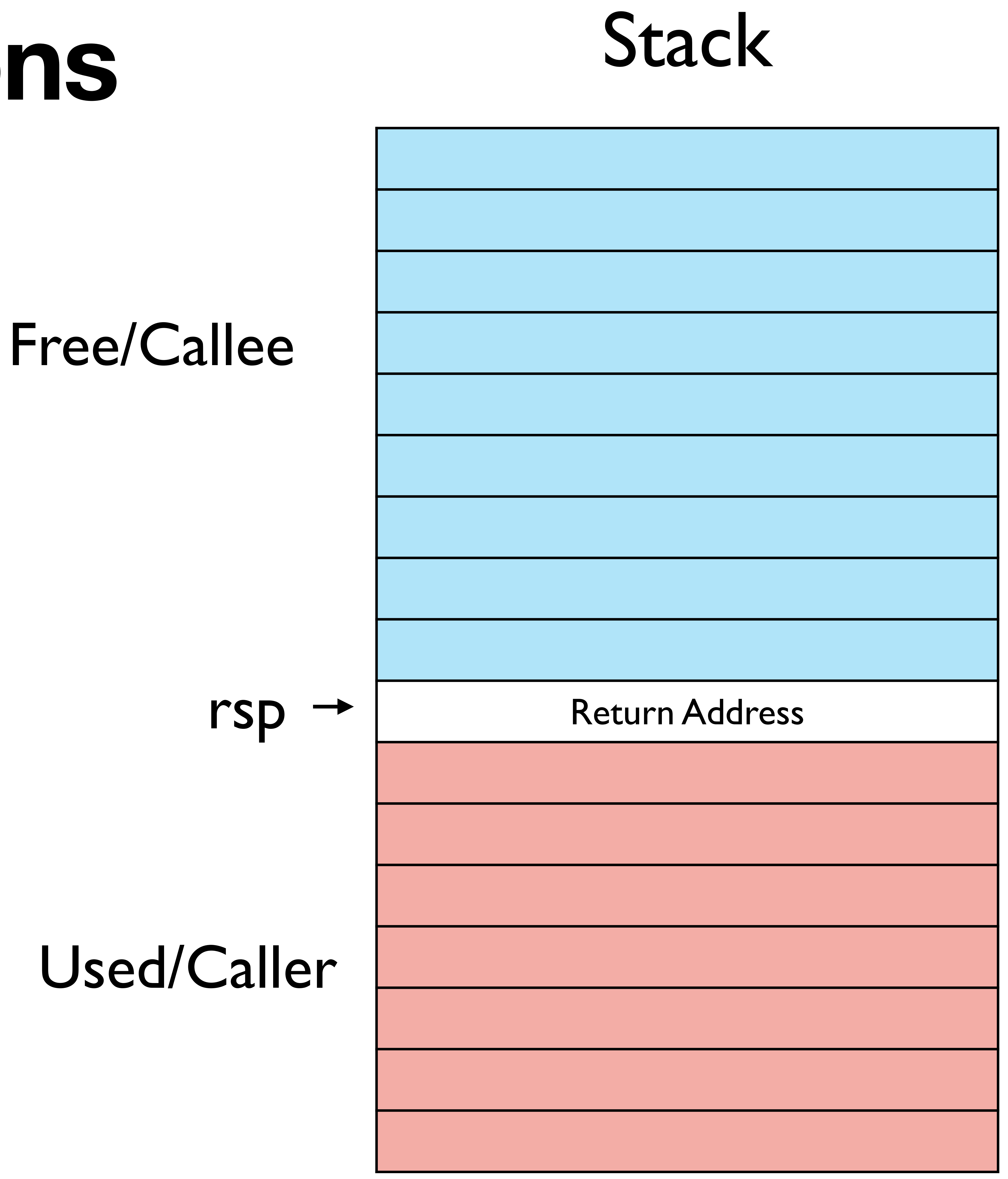
We access the stack using the "stack pointer" rsp.

The calling convention dictates that when a function is called, the stack pointer

1. Points to the return address of the caller

2. Lower memory addresses are free for the callee to use

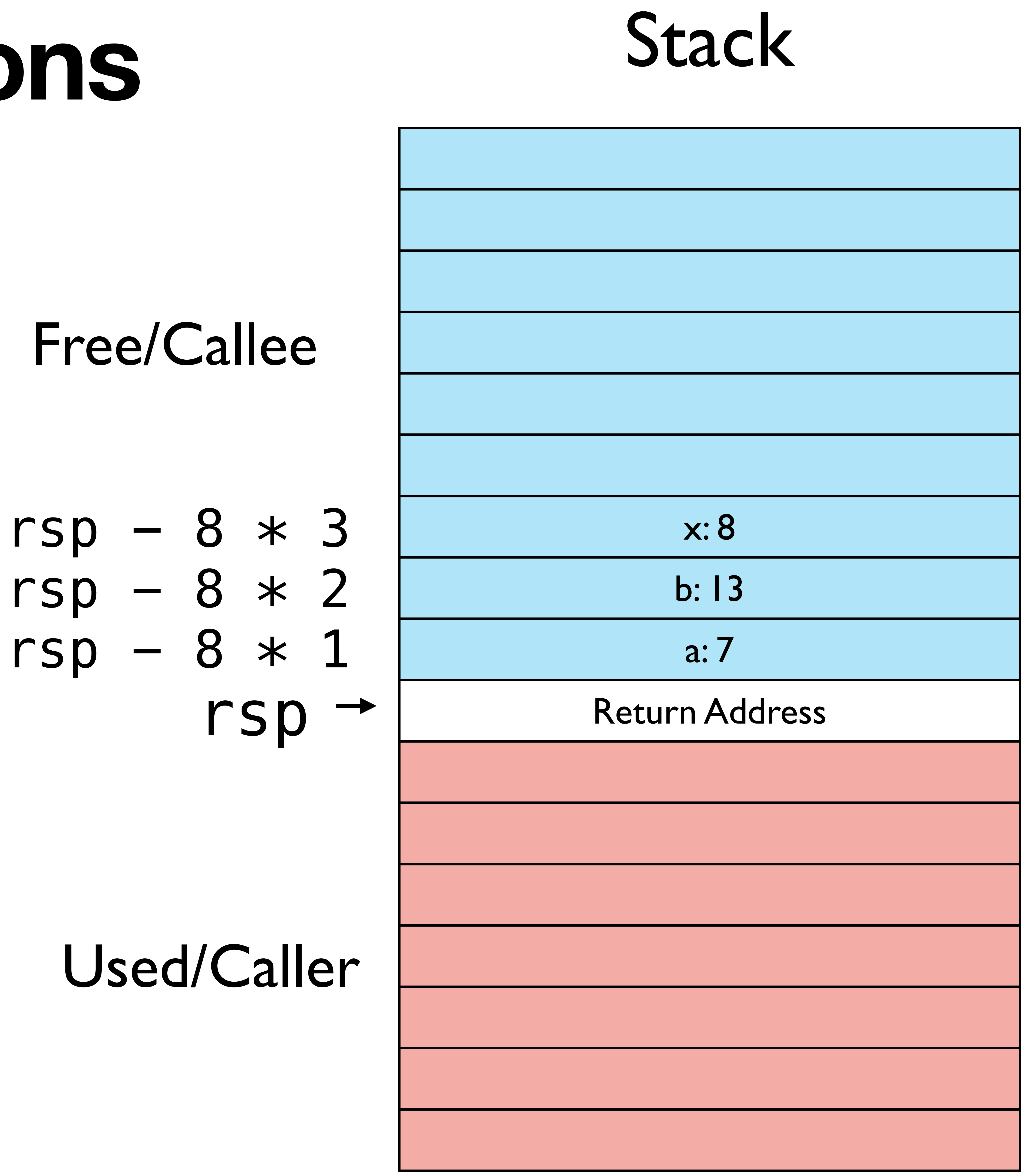
3. Higher memory addresses are owned by the caller



# x86 Memory Conventions

We use the free space on the stack to store our local variables

```
let a = 7 in
let b = 13 in
let x = add1(a) in
add1(x)
```



## Compiling Let

To compile our code, we need to establish a mapping of variable names to memory locations

## Compiling Let

To compile our code, we need to establish a mapping of variable names to memory locations

```
   let x = 10          /* [] */
in add1(x)             /* [ x --> 1 ] */
```

## Compiling Let

To compile our code, we need to establish a mapping of variable names to memory locations

```
   let x = 10          /* [] */
in let y = add1(x)     /* [x --> 1] */
in let z = add1(y)     /* [y --> 2, x --> 1] */
in add1(z)             /* [z --> 3, y --> 2, x --> 1] */
```

## Compiling Let

To compile our code, we need to establish a mapping of variable names to memory locations

```
   let a = 10                                 /* [] */
in let c =      let b = add1(a)               /* [a --> 1] */
           in let d = add1(b)                 /* [b --> 2, a --> 1] */
           in add1(b)                         /* [d --> 3, b --> 2, a --> 1] */
in  add1(c)                                   /* [c --> 4, d --> 3, b --> 2, a --> 1] */
```

Wasteful?

When a variable goes out of scope, its value is no longer needed

# Compiling Let

Only need to ensure that the memory locations are unique relative to the other variables that are currently in scope

```
   let a = 10                                /* [] */
in let c =      let b = add1(a)    /* [a --> 1] */
             in let d = add1(b)    /* [b --> 2, a --> 1] */
             in add1(b)                /* [d --> 3, b --> 2, a --> 1] */
in  add1(c)                                /* [c --> 2, a --> 1] */
```

How can you implement this in code? Again: designing the right kind of environment is the key

## Code Generation for Let

```
   let x = 10
in add1(x)
```

```
mov rax, 10
mov [rsp - 8*1], rax
mov rax, [rsp - 8*1]
add rax, 1
```

expressions store their result in rax

let bindings store rax on stack

variable lookups load from stack to rax

```
   let a = 10
in let c =     let b = add1(a)
            in let d = add1(b)
            in add1(b)
in  add1(c)
```

```
mov rax, 10
mov [rsp - 8*1], rax
mov rax, [rsp - 8*1]
add rax, 1
mov [rsp - 8*2], rax
mov rax, [rsp - 8*2]
add rax, 1
mov [rsp - 8*3], rax
mov rax, [rsp - 8*2]
add rax, 1
mov [rsp - 8*2], rax
mov rax, [rsp - 8*2]
add rax, 1
```

# Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. How can we generate that assembly code programmatically?

## Snake v0.1: "Adder"

Today: Finish Adder by adding binary arithmetic operations

## Snake v0.1: "Adder"

```
‹prog›: def main ( IDENTIFIER ) : ‹expr›
‹expr›:
    | NUMBER
    | ADD1 ( ‹expr› )
    | SUB1 ( ‹expr› )
    | IDENTIFIER
    | LET IDENTIFIER EQ ‹expr› IN ‹expr›
    | ‹expr› + ‹expr›
    | ‹expr› - ‹expr›
    | ‹expr› * ‹expr›
    | ( ‹expr› )
```

## Abstract Syntax

```
enum Prim {
    Add1,
    Sub1,
    Add,
    Sub,
    Mul,
}

enum Expression {
  ...
  Prim(Prim, Vec<Expression>),
}
```

no constructor for parentheses

## Precedence

Parser uses precedence rules (PEMDAS) to produce an AST

```
(2 - 3) + 4 * 5

(2 - 3) + (4 * 5)
```

both parse into the same AST:

```
Prim(Add,
  [Prim(Sub, [Number(2), Number(3)]),
   Prim(Mul, [Number(4), Number(5)])])
```

## Semantics

In an expression e1 op e2, do we evaluate e1 and then e2 or vice-versa?

Does it make a difference in Adder?

Does it make a difference in realistic extensions of Adder?

```
print(6) * print(7)
```

## Compiling Binary Operations

Why is compiling binary operations more complex than unary?

## Compiling Binary Operations

Why is compiling binary operations more complex than unary?

Recall: current strategy is to store intermediate results in rax

```
((4 - 3) - 2) * 5
```

```
mov rax, 4
sub rax, 3
sub rax, 2
mul rax, 5
```

## Compiling Binary Operations

```
(2 - 3) + (4 * 5)
```

```
mov rax, 2
sub rax, 3
?????
```

compound expressions have **implicit** intermediate results

solution: translate to a form where these intermediate results are explicit, and operations are only ever applied to **immediate** expressions (constants/variables)

```
let first = 2 - 3 in
let second = 4 * 5 in
first + second
```

# Intermediate Representation

We add a new pass lowering our AST into an **intermediate representation**.

An **intermediate representation** is a language used internally in the compiler.

Typically, humans don't write programs in the intermediate representation directly, only generated by compiler passes.

Intermediate representation should be "closer" to our target language but abstract over the complexities of assembly code.

The same IR can be used to translate to different backends, with common optimizations and transformations.

## Static Single Assignment v1: Basic Blocks

The intermediate representation we use in this course is called **Static Single Assignment** (**SSA**).

For Adder, we only need a fragment of SSA: we will compile the source to a single **basic block**.

## Static Single Assignment v1: Basic Blocks

SSA programs aren't written by humans so they don't need a "concrete syntax"

but to make debugging easier, we will print SSA programs in the style shown below:

```
entry(x):
  y = add 2 x
  z = sub 18 3
  w = mul y z
  ret w
```

## Static Single Assignment v1: Basic Blocks

Differences from Snake:

1. No left-nesting of let bindings
2. Arguments are immediate values, not complex expressions
3. Ends in explicit return

```
entry(x):
  y = add 2 x
  z = sub 18 3
  w = mul y z
  ret w
```

Differences from Assembly:

1. Immutable local variables, no registers/memory distinction
2. Calling convention is abstract

## Static Single Assignment v1: Basic Blocks

Live code: AST for SSA

# Static Single Assignment v1: Basic Blocks

Summary:

1. An SSA program consists of an entry point, a parameter and a block
2. A block is a sequence of primitive operations performed on immediately available values (variables or numbers) ending in a return statement.
3. Variables in SSA are **immutable**, just like our source language.
4. All bound variables in SSA should be globally unique.

# Static Single Assignment v1: Basic Blocks

Now we've reduced the compilation to two tasks:

1. "Lowering" our AST into an SSA program

2. Producing x86 assembly from an SSA program

## SSA to x86

Since SSA is essentially a simplified version of Adder, we can apply the same techniques for generating assembly code from SSA. The only extension is that we need to handle binary primitives.

## SSA to x86

```
entry(x):
  y = add 2 x
  z = sub 18 3
  w = mul y z
  ret w
```

```
;; entry(x):
mov [rsp - 8], rdi
;; y = add 2 x
mov rax, 2
mov r10, [rsp - 8]
add rax, r10
mov [rsp - 16], rax
;; z = sub 18 3
mov rax, 18
mov r10, 3
sub rax, r10
mov [rsp - 24], rax
;; w = mul y z
mov rax, [rsp - 16]
mov r10, [rsp - 24]
imul rax, r10
mov [rsp - 32], rax
;; ret w
mov rax, [rsp - 32]
ret
```

# Adder to SSA

```
(2 - 3) + (4 * 5)
```

```
x0 = 2
x1 = 3
x2 = sub x0 x1
x3 = 4
x4 = 5
x5 = mul x3 x4
x6 = add x2 x5
ret x6
```

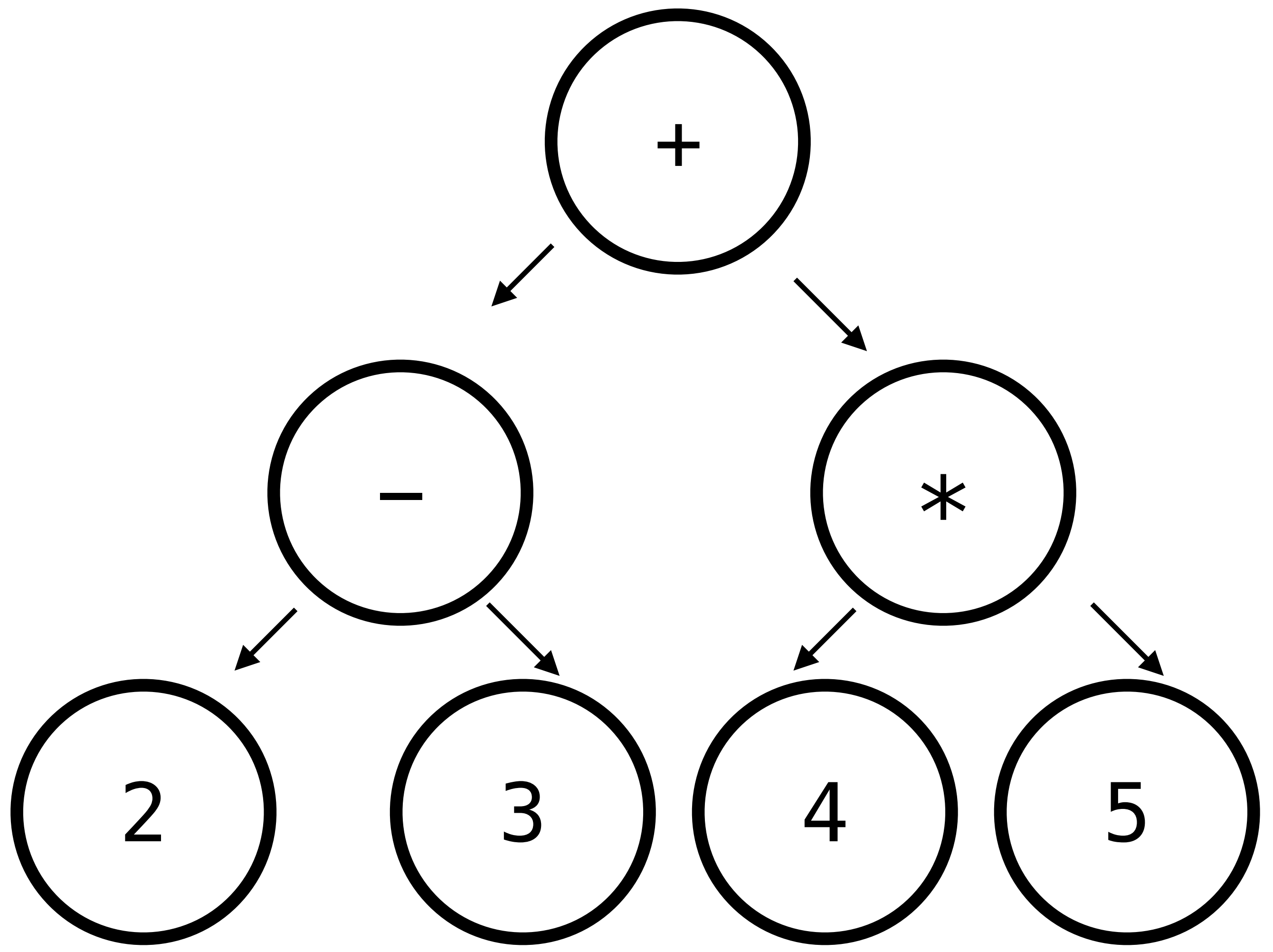
How to produce this code compositionally?

**Observation:** Each line corresponds to a different subexpression

**Observation:** A deeply nested subexpression 2 is the **top** of the AST in the output. We are converting a tree into a linked list using a "postorder" traversal

## Adder to SSA

```
(2 - 3) + (4 * 5)
```



```
x0 = 2
x1 = 3
x2 = sub x0 x1
x3 = 4
x4 = 5
x5 = mul x3 x4
x6 = add x2 x5
ret x6
```

$$\begin{tikzcd} 2 \arrow[r] & 3 \arrow[r] & - \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & * \arrow[r] & + \end{tikzcd}$$

solution: when compiling a sub-expression, we take "what code to run after" as an argument

"what to run after" is called the **continuation** of the expression

## Adder to SSA

Live Code

# Adder to SSA

Summary:

Translate Adder to SSA using **continuation-passing style:** expression lowering function is parameterized by a **continuation** consisting of

1. the name of the destination variable for the result.

2. a block of code to run **after** the compiled code places the result in the destination.

Need to generate **unique** names in this process to make sure that the generated variable names are all distinct and distinct from the original program variables