EECS 483: Compiler Construction

Lecture 2:
Variables, Scope and Memory

January 15, 2025

Announcements

- First homework assignment will be released tonight.

  Some material will be covered in next week's class, but can get started on parts of it after today's lecture

  This week's discussion will go over the infrastructure in the starter code.

- Max is out of town next week for a conference (POPL), lecture on the 22nd will be posted on Canvas.

  - Yuchen will be holding office hours 3-4:30pm on Thursday the 23rd in Beyster Atrium in place of Max.

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: add immutable variables to Adder, to allow saving results of intermediate computations

Snake v0.1: "Adder"

‹prog›: `def` `main` `(` *IDENTIFIER* `)` `colon` ‹expr›

‹expr›:

- | *NUMBER*
- | `add1` `(` ‹expr› `)`
- | `sub1` `(` ‹expr› `)`
- | *IDENTIFIER*
- | `let` *IDENTIFIER* `=` ‹expr› `in` ‹expr›

Examples

```
def main(x):
  let y = sub1(x) in
  let z = add1(add1(y)) in
  add1(z)
```

Examples

```
def main(x):
  let z =
      let y = sub1(x) in
      add1(add1(y)) in
  add1(z)
```

Let is an **expression** form, just like add1 and sub1

Examples

```
def main(x):
  let z = add1(add1(let y = sub1(x) in y)) in
  add1(z)
```

Let is an **expression** form, just like add1 and sub1

Expressions vs Statements

In most languages in the C style, variable bindings belong to a separate syntactic class of **statements**.

In languages with a functional programming style, it is more common to allow most syntactic constructs.

Rust is somewhere in the middle

```
def main(x):
  let z = add1(add1(let y = sub1(x) in y)) in
  add1(z)
```

```
fn funny(x: i64) -> i64 {
    let z = {
        add1(add1({
            let y = sub1(x);
            y
        }))
    };
    add1(z)
}
```

Example?

```
def main(x):
  y
```

Does this example match our grammar?

Should it be considered a valid program?

Compiler Frontends

Even after parsing, there are some conditions on the syntax that still remain to be checked. This is inherent: to be implemented efficiently, parsers use computationally restrictive languages that are not capable of performing all of the **semantic analysis** necessary to check if the input program is valid

Semantic Analysis

Examples:

- Scope checking (today)
- Type checking
- Borrow checking

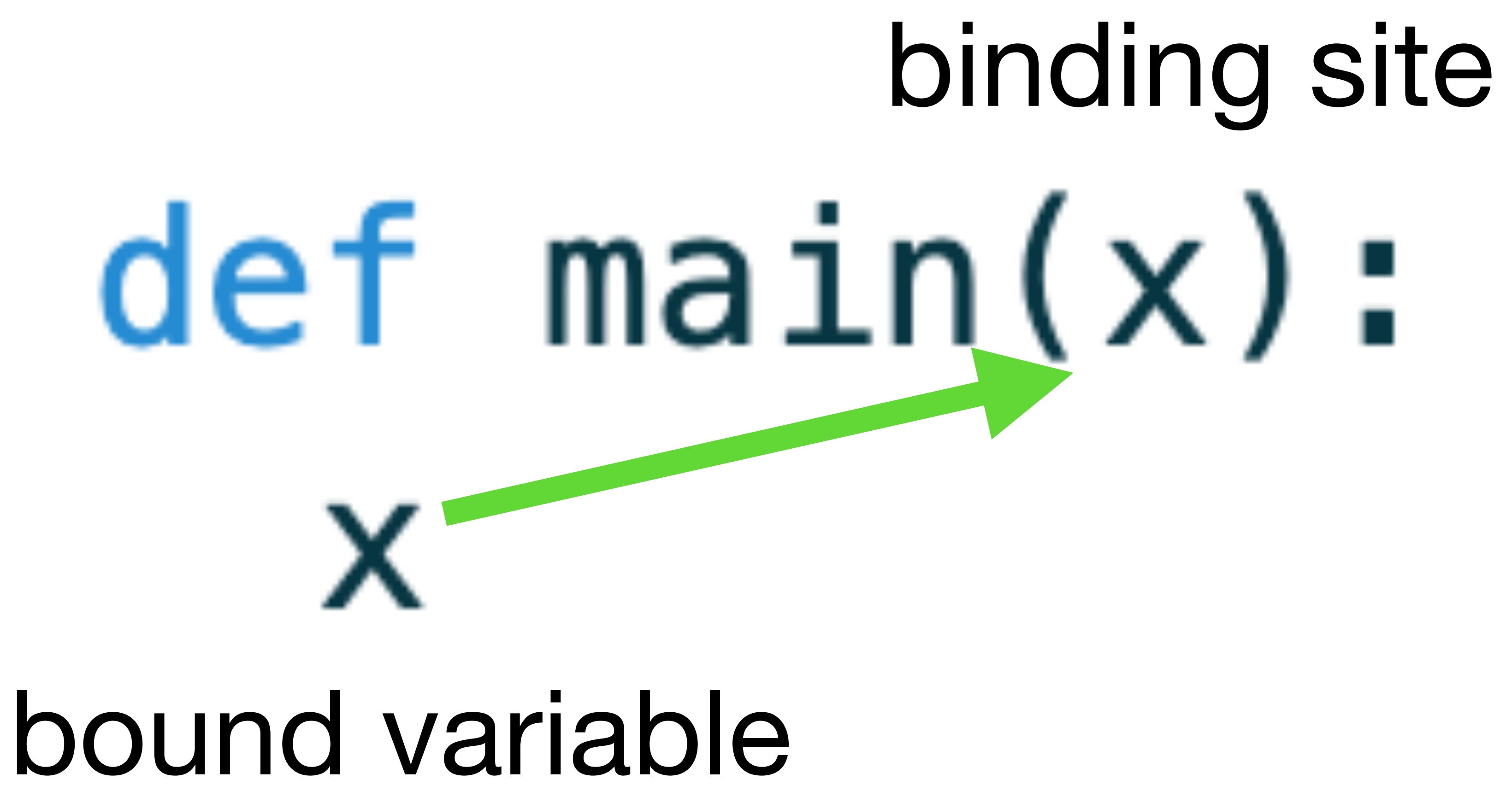
EECS 490 covers type checking in more detail.

Free and Bound Variables

```
def main(x):
    y
```

We say this program is invalid because the y is a **free variable**, meaning it has not been defined

Free and Bound Variables



The usage of x here is valid because it occurs within the scope of a binding site that binds the variable name x. We call such a usage a **bound** variable

Free and Bound variables

```
def main(x):
let y =
  let z = x in add1(y)
  in
let w = sub1(x) in
sub1(z)
```

There are 8 variables in this program. Which ones are binding sites, which ones are free variables and which ones are bound variables?

Live Code: Scope Checking

To define scope rigorously, let's define a scope checker in Rust.

Variable Names are Tricky

We use variable names as a way to refer back to binding sites. But because names are implemented as strings, sometimes the same name is used to refer to multiple binding sites.

```
def main(y):
let x =
  let x = sub1(y)
  in add1(x)
in add1(add1(x))
```

Variable Names are Tricky

We use variable names as a way to refer back to binding sites. But because names are implemented as strings, sometimes the same name is used to refer to multiple binding sites.

```
def main(y):
let x =
  let x = sub1(y)
  in add1(x)
in add1(add1(x))
```

Shadowing

Should this be allowed?

```
let x = 1 in
let x = 2 in
x
```

Shadowing

Should this be allowed?

We say the second binding **shadows** the first

```
let x = 1 in
let x = 2 in
x
```

If a binding is shadowed, it's impossible to refer to it in the source program!

Live Code: Interpreter

Now let's define the semantics of our language rigorously by defining an interpreter in Rust.

Beta Reduction

A common rewrite we can apply to our ASTs is called beta reduction

```
let x = e1 in e2
```

rewrites to

```
e2
```

with all occurrences of x replaced by e1

Beta Reduction

```
let x = y in
let z = add1(x) in
add1(add1(z))
```

rewrites to

```
let z = add1(y) in
add1(add1(z))
```

Beta Reduction

Is there any situation where this rewrite is **not** correct? I.e., where the two different expressions have different behaviors?

```
let x = e1 in e2
```

rewrites to

```
e2
```

with all occurrences of x replaced by e1

Beta Reduction

Is there any situation where this rewrite is **not** correct? I.e., where the two different expressions have different behaviors?

```
def main(y):
  let x = y in
  let y = 17 in
  add1(x)
```

```
def main(y):
  let y = 17 in
  add1(y)
```

we say that the inner binding of y has **captured** the occurrence of y on the inside

Unique Variable names

Shadowing is convenient for programmers, but ultimately harmful to compilers. For this reason compilers typically implement a variable renaming phase that makes sure that all binding occurrences are globally unique

```
def main(y#0):
  let x#0 = y#0 in
  let y#1 = 17 in
  add1(x#0)
```

```
def main(y#0):
  let y#1 = 17 in
  add1(y#0)
```

Ensuring that all variables are unique ensures we can move code around without worrying about capture.

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 Instructions: mov

```
mov dest, src
```

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

Addresses can be not just registers, but offsets from registers

```
mov rax, [rsp - 8 * 3]
```

x86 Memory Conventions

Registers give us access to 128 bytes, and byte-addressable memory gives us access to 128 terabytes.

But that memory needs to be **shared** by different components of the process (functions, objects, allocator, garbage collector, etc).

We can't just start writing to a random portion of memory

1. That memory might be used by another component, like our caller, and we would break the invariants of that component

2. Hardware supports mechanisms for process isolation, so most of the memory space will be invalid for us to access, causing the dreaded **segmentation fault**

x86 Memory Conventions

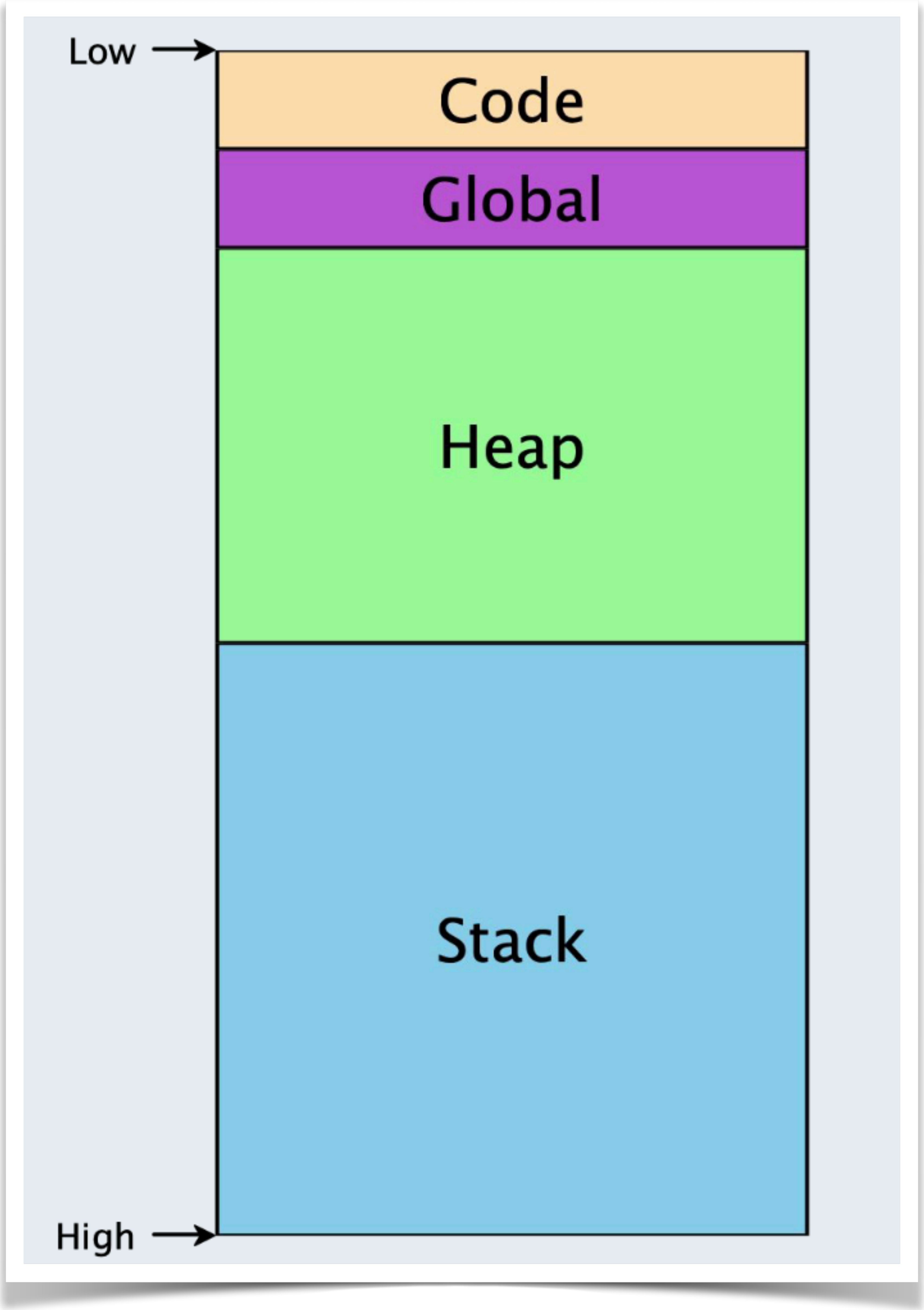
Memory in x86 processes is divided into 4 portions

1. Read-only memory containing the source code. (.text section)

2. Globals

3. Heap

4. The call Stack



x86 Memory Conventions

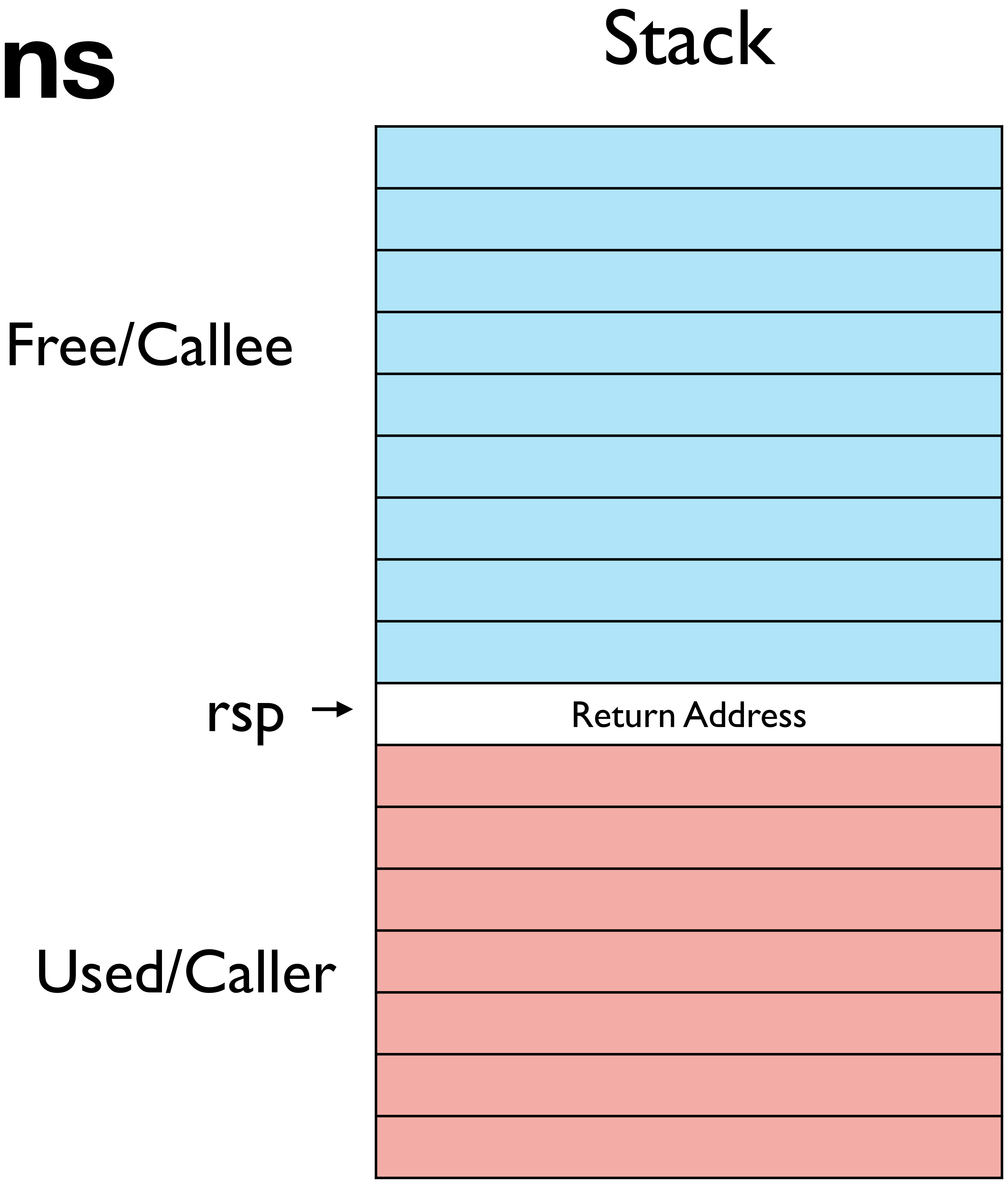
We access the stack using the "stack pointer" rsp.

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

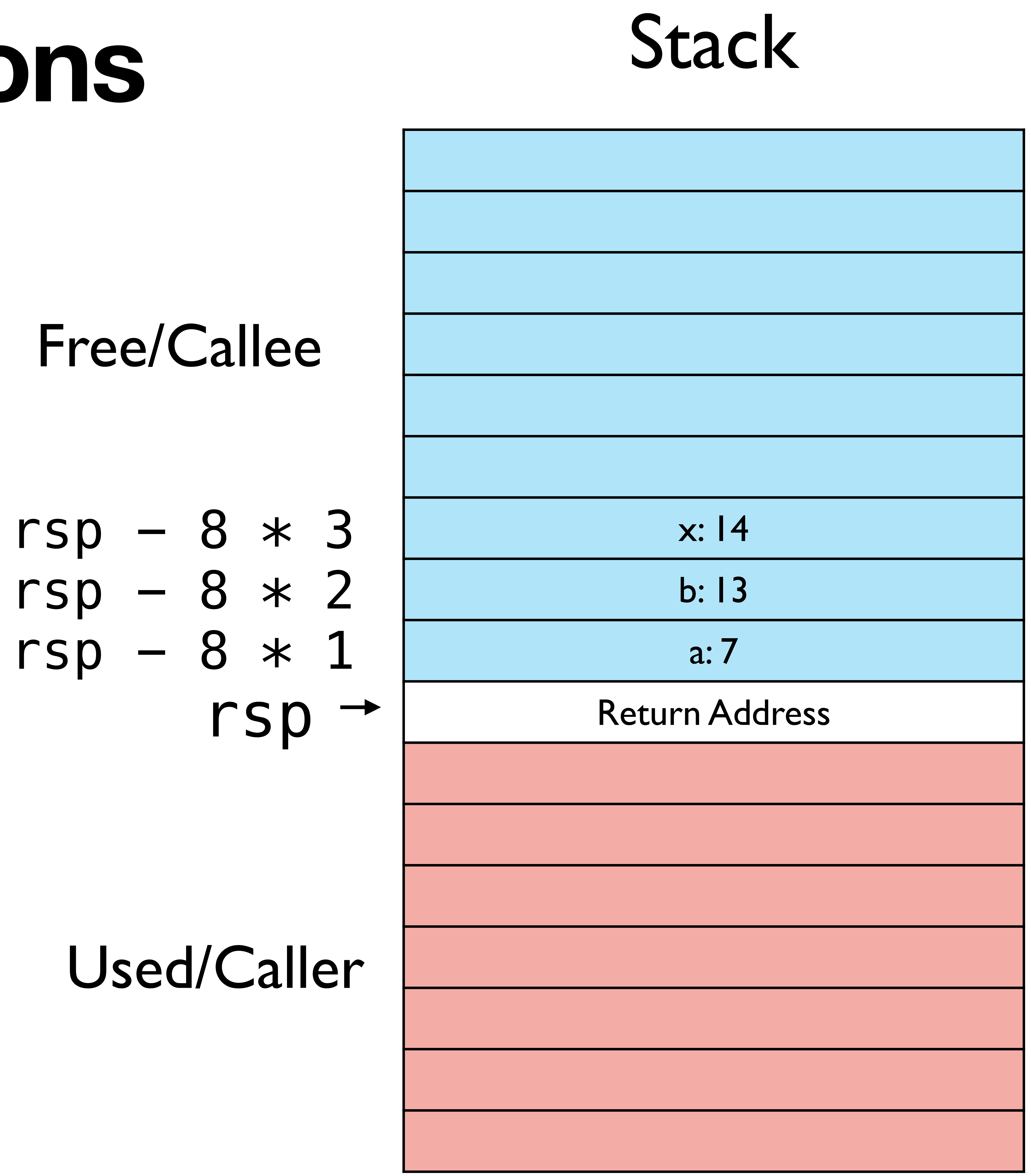
3. Higher memory addresses are owned by the caller



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

```
   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
```

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

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