Section 11: First-Class Functions

1. What happens when we call a function?

Complete the following diagram, which represents the stack immediately after the function call `(f 3 6 9)`. *Assume that `rsp` is aligned to a multiple of 16 when we compile the call. By "`rsp`" in the left column, we mean the value of `rsp` before the function call.*

Address	Value
`rsp - 56`	
`rsp - 48`	
`rsp - 40`	
`rsp - 32`	
`rsp - 24`	
`rsp - 16`	
`rsp - 8`	
`rsp`	<return address for caller's frame>

Where does the stack frame of the callee start?

How would the stack frame change if we dynamically communicate the number of arguments like we did in Homework 6?

2. What's the use of function pointers without lambdas?

Write a program in our language that only works once we support function pointers (do *not* include any lambda expressions at this point).

What can function pointers be useful for?

3. Handling lambdas without free variables

The C11 standard supports function pointers, but no lambdas. How could you change the following functions to be a valid C11 code while still maintaining its functionality?

```c
typedef struct node {
   int value;
   struct node* next;
} node;

node* map(node* list, int (*f)(int)) {
   if(list == NULL) return NULL;
   node* next = map(list->next, f);
   node* new = malloc(sizeof(node));
   new->next = next;
   new->value = f(list->value);
   return new;
}

node* square(node* list) {
   return map(list, [](int n) { return n * n; });
   //              ---------------------------- This is the problematic
   //                                           section b/c lambdas are
   //                                           not supported in C11.
}
```

Interactive Version (runs in your browser!): https://godbolt.org/z/jjhcafffT

4. Handling lambdas with free variables

We will now explore what happens when we compile the following lambda in an environment where the variables **x** and **y** are defined:

```
(lambda (z) (+ z (+ y x)))
```

4.1 Compiling the creation of a lambda

4.1.1 Setting up the heap

On the left is the stack layout at the site where the lambda is created.

On the right, fill in the values that are transferred to the heap when we compile the lambda. *Assume that the label of the lambda implementation is called* ***`_lambda_1`***.

Stack

Address	Value
`rsp - 16`	y
`rsp - 8`	x
`rsp`	<return address>

Heap

Address	Value
`rdi`	
`rdi + 8`	
`rdi + 16`	

Why can't we just inline the values of **x** and **y** in the body of the lambda?

4.1.2 Stack layout going into the body

We now want to explore the stack and heap layout that we would expect to see right after we jump to the label of the lambda function. Please show where **x**, **y** and **z** are located on the stack.

Stack

Address	Value
`rsp - 24`	
`rsp - 16`	
`rsp - 8`	
`rsp`	<return address>

Who is responsible for placing **x** and **y** there? ▢ caller ▢ callee

Who is responsible for placing **z** there? ▢ caller ▢ callee

How do we obtain the values of **x** and **y**?

4.2 Compiling a call to a lambda

Finally, let's talk about how a lambda function is called. Feel free to look at the **`compile.ml`** file of the class compiler, starting at the line **`| Call (f, args) when not is_tail ->`**.

Please fill in the blanks in the following pseudocode that describes how to call a lambda function.

1. Compile the arguments and place them on the ____________________.
2. Compile the ____________________ expression.
3. Ensure that **`rax`** is tagged with ____________________.
4. Copy the content of **`rax`** to the ____________________.
    a. This is important for the function to be able to retrieve the ____________________ and copy them onto the stack.
5. Load the address of the function into **`rax`** from the ____________________ by subtracting the ____________________ type tag from ____________________.
6. Perform a call to the address now stored in ____________________, modifying the stack pointer before and after for alignment purposes.

Here's some blank diagrams that you can use to help yourself visualize the algorithm!

Stack

Address	Value
`rsp - 24`	
`rsp - 16`	
`rsp - 8`	
`rsp`	

Heap

Address	Value
`rdi`	
`rdi + 8`	
`rdi + 16`	
`rdi + 24`	

5. The spectrum of first-class functions

Assume that variadic arguments and the ***`apply`*** *function from Homework 6 are not supported.*

	No first-class functions	**First-class functions:** Functions pointers w/o lambdas	Lambdas (w/o free variables)	Lambdas (w/ free variables)
Is the function label known at the call site?				
Is the number of expected arguments known at the call site?				
Where is the code for the function generated?				
How is the function label created?				
How do we know which address to jump to at the call site?				
Do we need to use the heap? Why?				