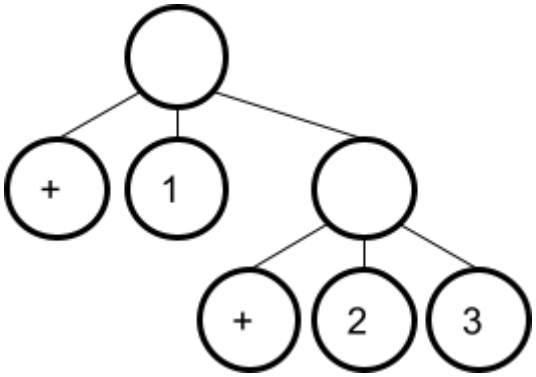
CS164 - Section 7

Task 1: Visualizing S-Expressions in the Interpreter and Compiler

We can visualize the s-expressions that our compiler works on as trees.
For example, *(+ 1 (+ 2 3))* can be visualized as follows:



1.1.1 Draw the order in which your interpreter visits the nodes of the tree.
1.1.2 Draw the flow of results in the interpreter as it evaluates the expressions. For example, where does the calculation 2 + 3 flow from and to?

1.2 Draw the s-expression tree for the following expression:
(let ((x 1)) (let ((x 2) (y x)) y))

1.2.1 Draw the order in which your interpreter visits the nodes of the tree.
*1.2.2 Annotate the value of the **env** environment variable in the interpreter at the various nodes of the tree.*
1.2.3 Draw the flow of results in the interpreter as it evaluates the expressions.

Task 2: Visualizing Memory

In this task, we will practise how to visualize the stack and the heap.

2.1.1 Use the class compiler to compile the example program from the beginning of task 2:
`compile_and_run "(+ 1 (+ 2 3))";;`
2.1.2 Open up the generated assembly: `cat program.s`
2.1.3 Go through the assembly from top to bottom and fill out the following stack diagram (on the left):

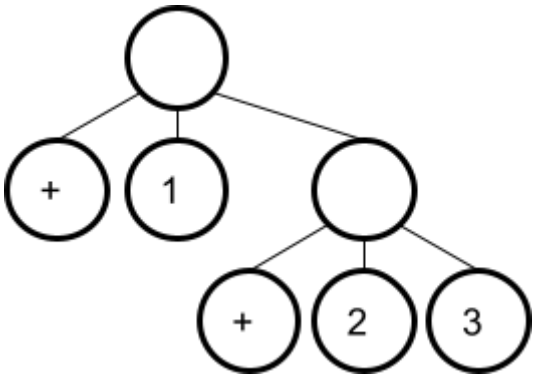
Stack

*(rsp - 16):	
*(rsp - 8):	
*(rsp):	< return address >

Registers

rax:

r8:



2.1.4 In the s-expression tree on the right, mark the value of the stack_index variable in the compiler at the various nodes.

2.2.1 Use the class compiler to compile the following program:
`compile_and_run "(right (pair 1 (left (pair 2 true))))";;`
2.2.2 Draw the s-expression tree for the program

2.2.3 Go through the assembly from top to bottom and fill out the following stack and heap diagrams.

Stack

*(rsp - 16):	
*(rsp - 8):	
*(rsp):	< return address >

Heap

Registers

rax:

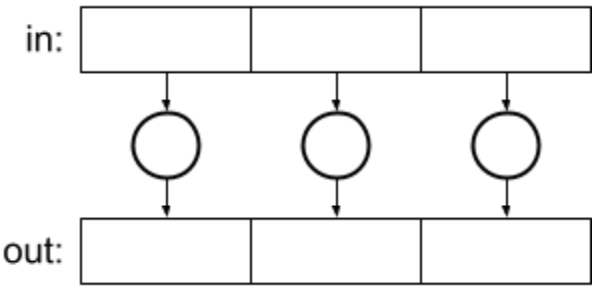
r8:

rsp:

2.2.4 Write down your thoughts on why there isn't an equivalent to the stack_index variable in the compiler for the heap. How are heap addresses managed compared to stack addresses?

Task 3: Visualizing Functional Programming

We visualize lists as a sequence of boxes, functions as circles and dataflow as arrows.
Thus we can visualize the functionality of **List.map** like this:



Please draw your own diagrams for the following functions. You might have to look up descriptions of these functions or ask your group members if you are not familiar with them. Feel free to develop your own style and add any details that you think are important without making the visualization hard to read.

3.1 List.hd : 'a list -> 'a

3.2 List.tl: 'a list -> 'a list

3.3 List.rev: 'a list -> 'a list

3.4 List.concat: 'a list list -> 'a list

3.5 List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

3.6 List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

*3.7 List.partition : ('a -> bool) -> 'a list -> 'a list * 'a list*

*3.8 List.split : ('a * 'b) list -> 'a list * 'b list*

*3.9 combine : 'a list -> 'b list -> ('a * 'b) list*