Welcome to Lecture 6: Functions as Data + Lambdas

Class will start at 10:10.

In the meantime, we will go around. Tell me your name, where you're from, & favorite mythical creature.

Today’s Topics

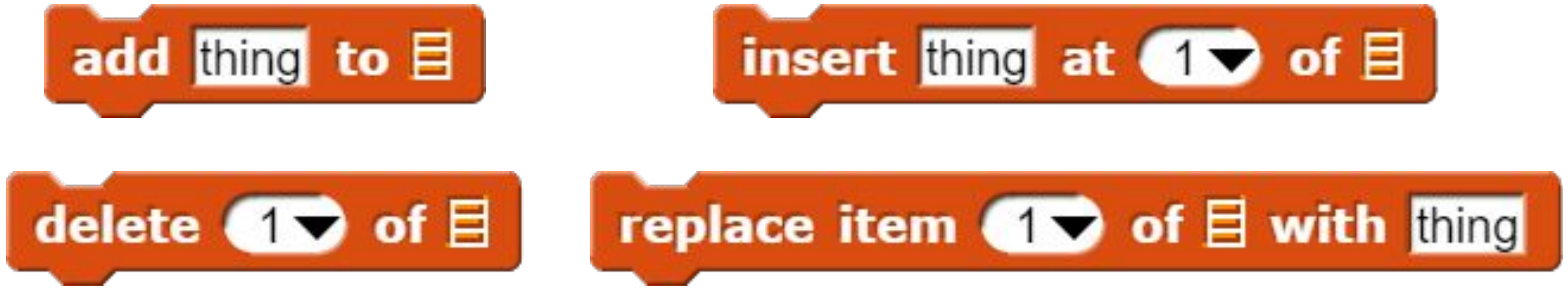
- Announcements
- Review
- List Scope
- Mutability vs Immutability
- Functions as input
- Lambdas
- Call, run, and rings in Snap*!*

Announcements

- Victoria’s OH is 6 to 7PM on Mon + Wed (Hybrid: online and in soda-777)
- Victoria’s SUPPORT OH 7 to 8PM on Mon (Hybrid: online and in soda-777)
- Computers: You can always use the computers in SDH-200, you will sign into your account here: https://acropolis.cs.berkeley.edu/~account/webacct/
- We are removing “duplicates” for Lab 4: Lists + Loops
  - So don’t worry if you didn’t get credit!

Review from Last Lecture

- Mutability vs Immutability
  - Mutability: Object can be changed after created
  - Immutability: Object CANNOT be changed after created
  - Lists are one of few data types that are mutable in Snap*!*
  - Only these functions can mutate a list:



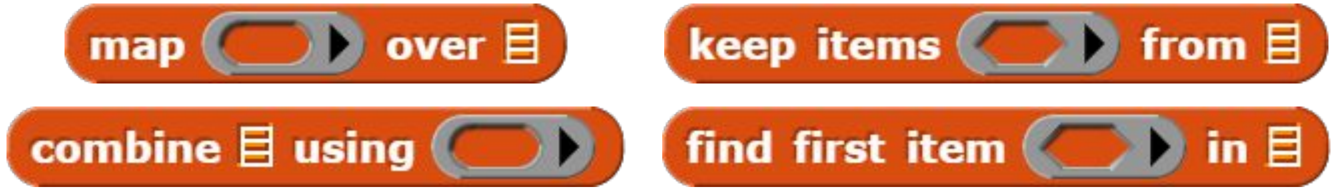
```
add thing to ☰
insert thing at 1 ▾ of ☰
delete 1 ▾ of ☰
replace item 1 ▾ of ☰ with thing
```

Review from Last Lecture

- Higher Order Functions (HOFs)
  - Definition:

Review from Last Lecture

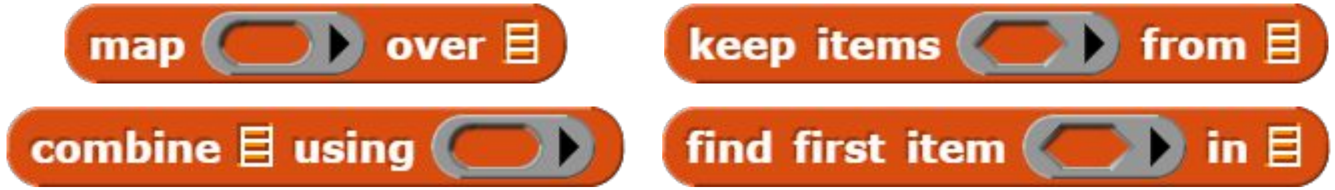
- Higher Order Functions (HOFs)
  - Definition: A function whose input is a function
  - Built in HOFs in Snap*!*:



  - But, we can make our own!
  - Do the Built-in HOFs, return new values/lists or modify the input list?

Review from Last Lecture

- Higher Order Functions (HOFs)
    - Definition: A function whose input is a function
    - Built in HOFs in Snap*!*:

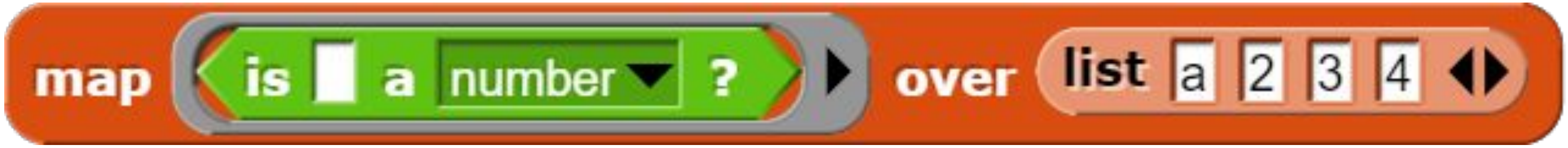


    - But, we can make our own!
    - Do the Built-in HOFs, return new values/lists or modify the input list?
        - Return new values / list!

Review from Last Lecture

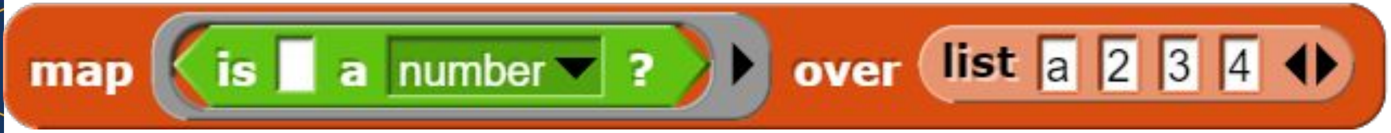
map ( ) over ☰	keep items ( ) from ☰
Performs a function on EACH item in the input list. Will return same size list.	Filters out items if the conditions evaluates to true. Will return less than or equal to input list
combine ☰ using ( )	find first item ( ) in ☰
Reduces input list based on function by applying function on all items. Didactic function! Unintended behavior if less or more than 2 inputs	Finds the first item from the list where the condition evaluates to true.

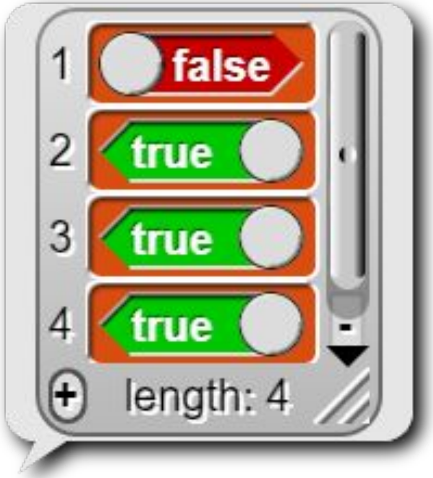
Review from Last Lecture



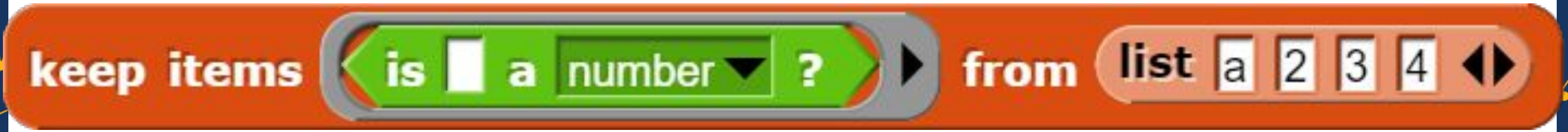
```
map (is [ ] a [number ▾] ?) over (list [a] [2] [3] [4])
```

Review from Last Lecture



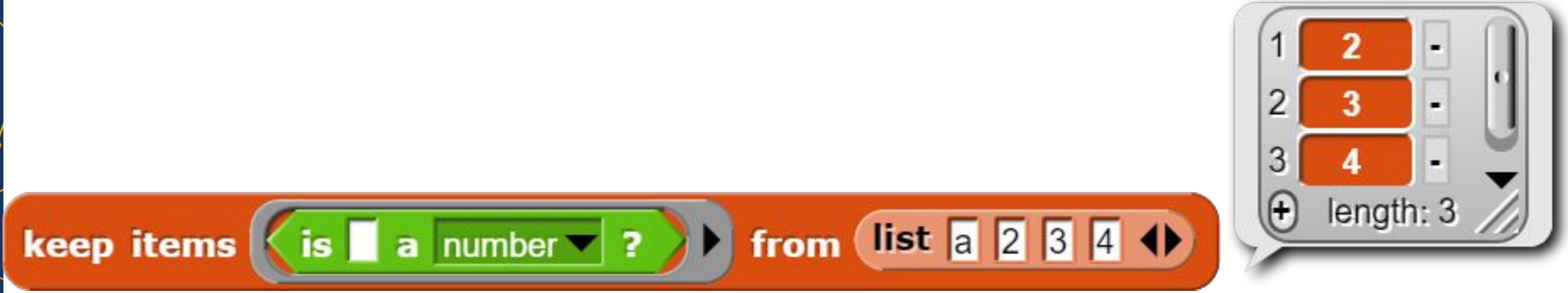


Review from Last Lecture

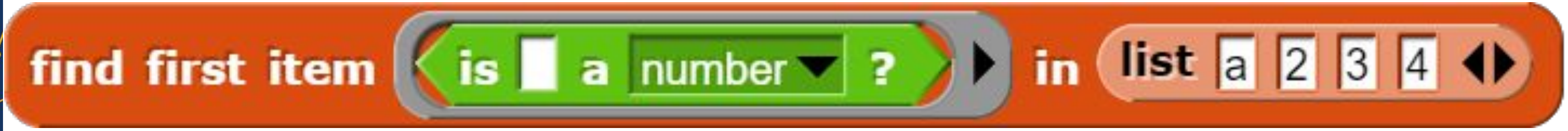


```
keep items ( is [ ] a number ? ) from list a 2 3 4
```

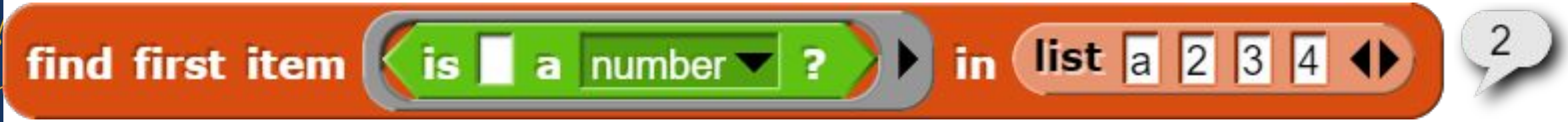
Review from Last Lecture



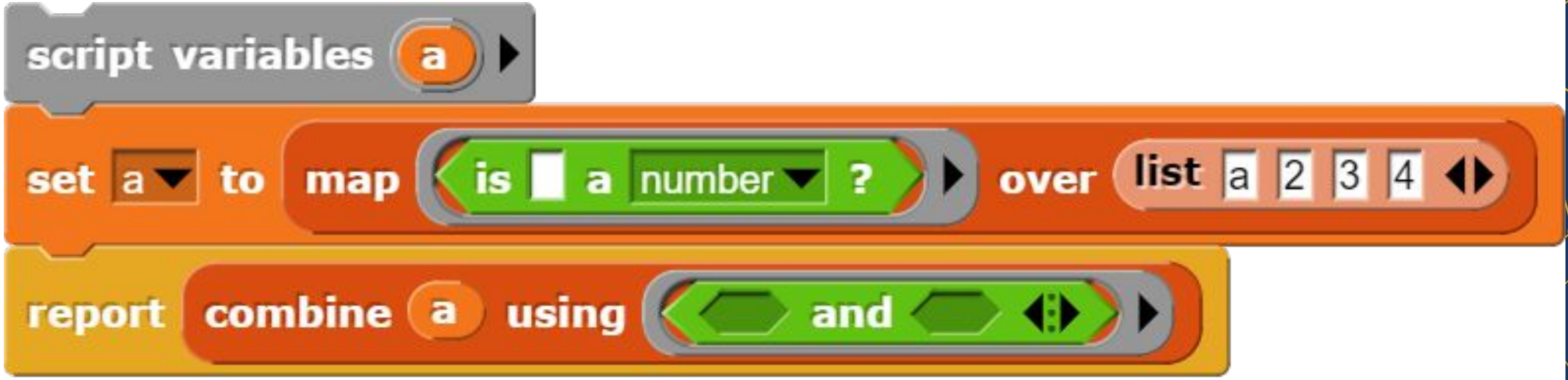
Review from Last Lecture



Review from Last Lecture

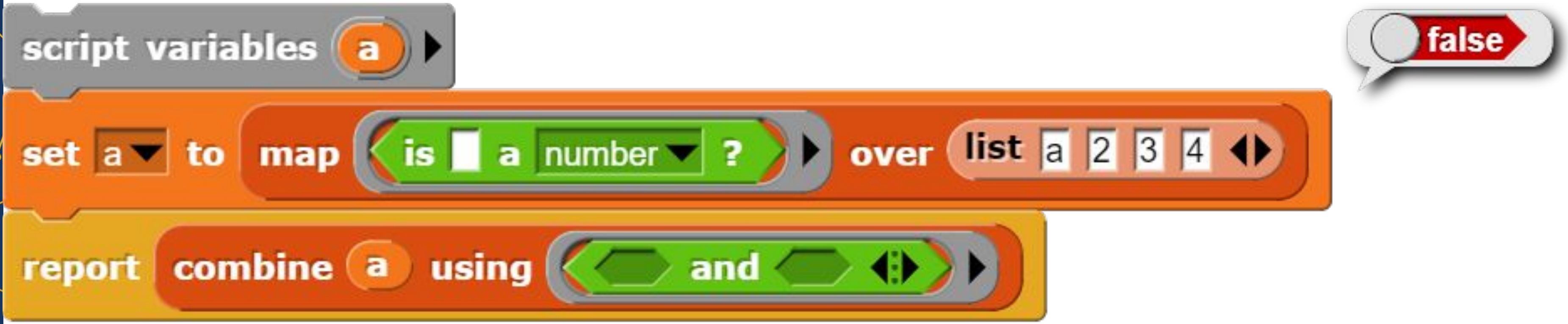


Review from Last Lecture



```
script variables (a)
set a to map (is [ ] a number ?) over list a 2 3 4
report combine (a) using (< > and < >)
```

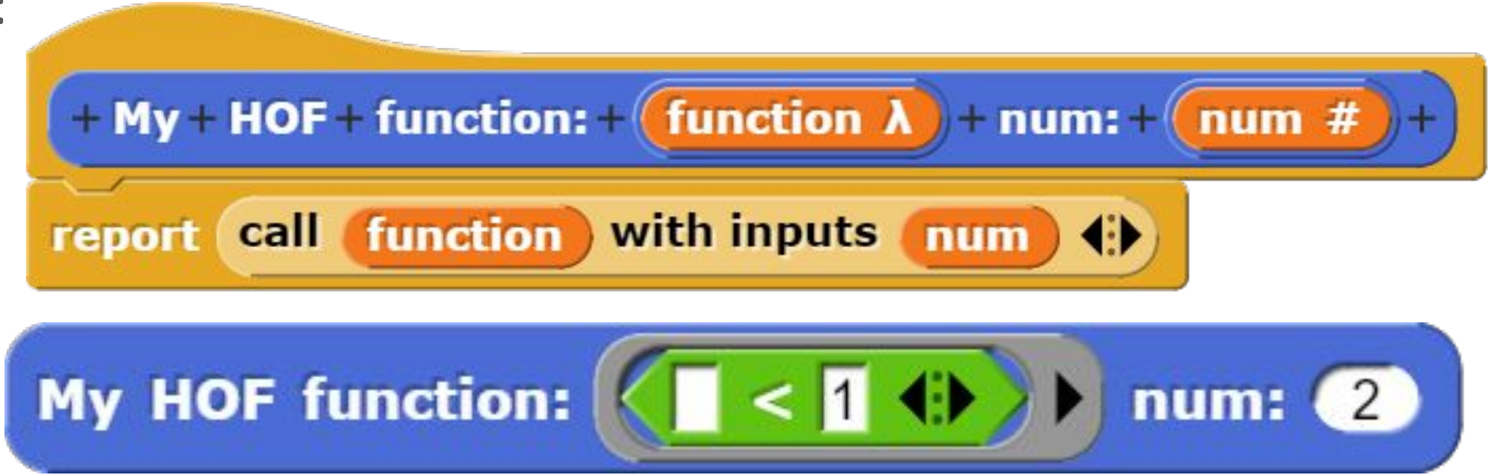
Review from Last Lecture



```
script variables (a)
set a to map (is [] a number ?) over (list a 2 3 4)
report combine (a) using (<> and <>)
```

Functions as Data

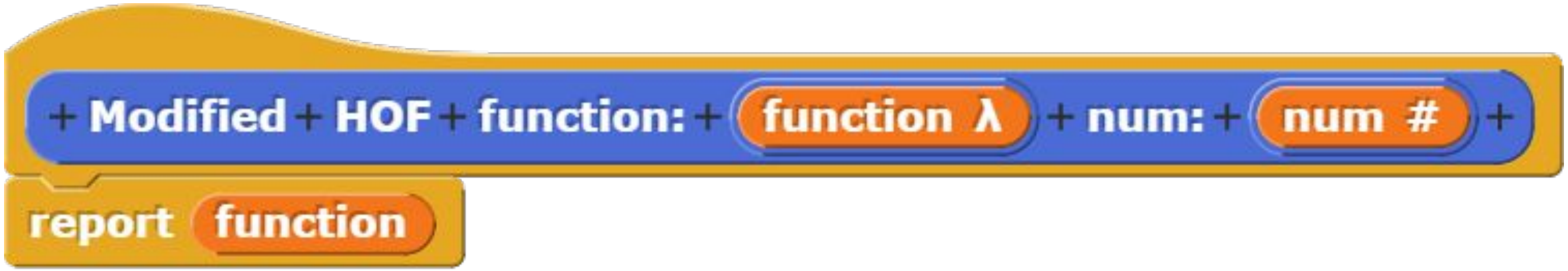
- We can make our own HOFs
- The input to our function will be a function!
- Example:

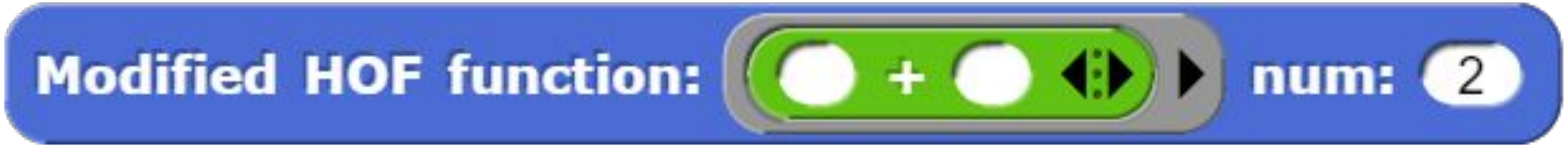

+ My + HOF + function: + function λ + num: + num # +
report call function with inputs num
My HOF function: < 1 num: 2

- Call function: Invokes ANY function with inputs dynamically (i.e. we specify function and inputs at runtime)
- Must call function manually with “call” or “run” function

Functions as Data

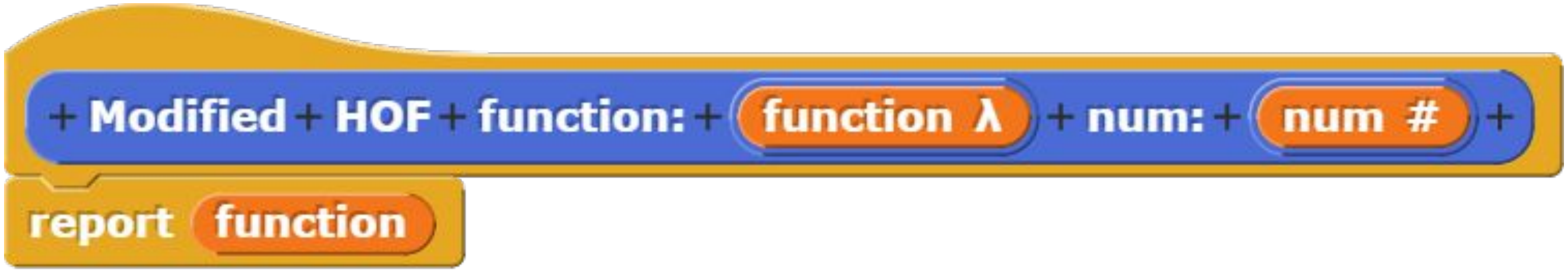
- Why is “call” necessary?

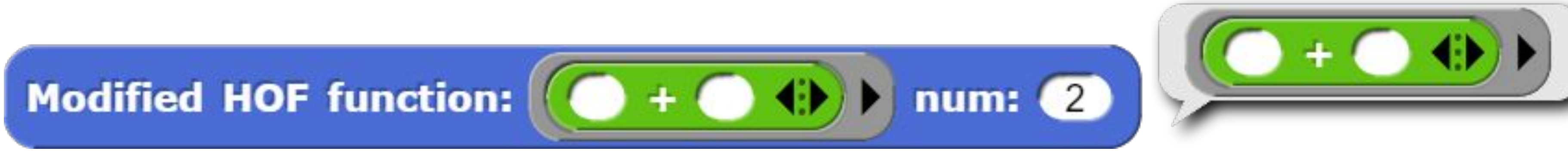




Functions as Data

- Why is "call" necessary?

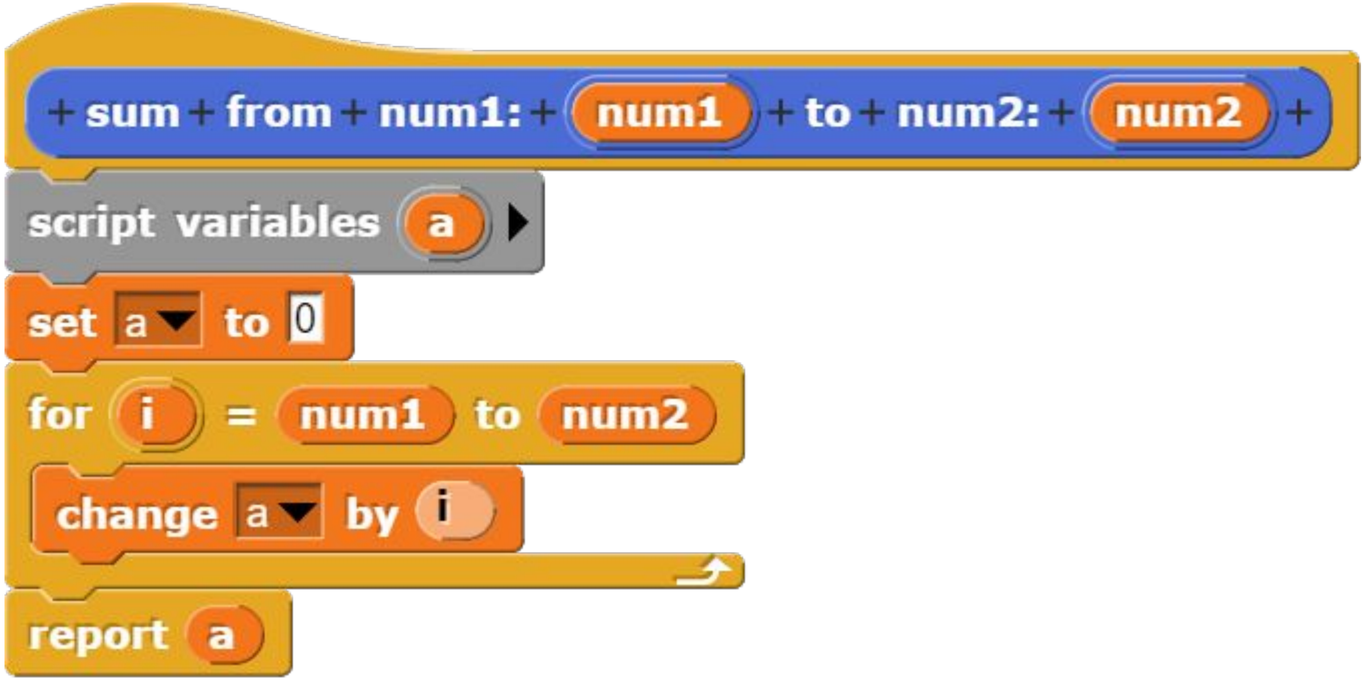




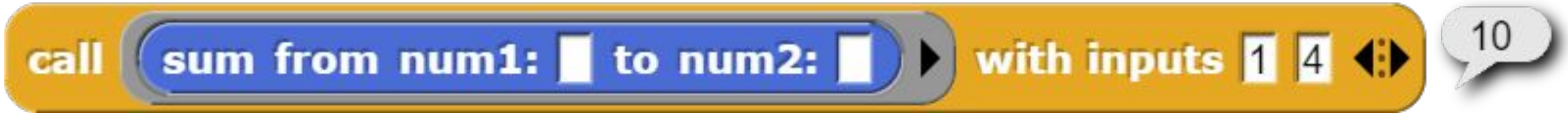
- Allows us to invoke the function
- Allows us to pass in inputs to function

Functions as Data

- Call Example

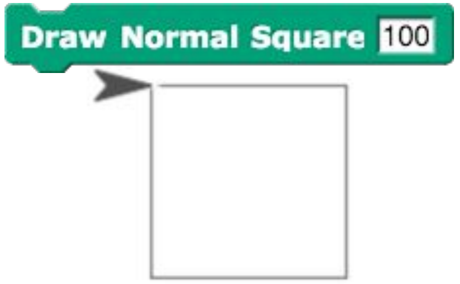


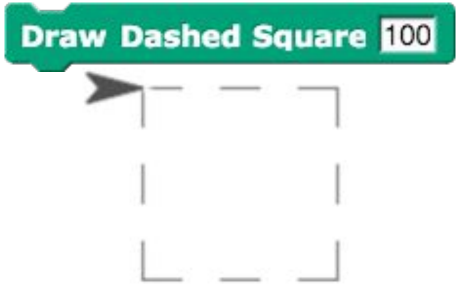
```
+ sum + from + num1: + (num1) + to + num2: + (num2) +
script variables (a)
set a to 0
for i = (num1) to (num2)
    change a by (i)
report (a)
```

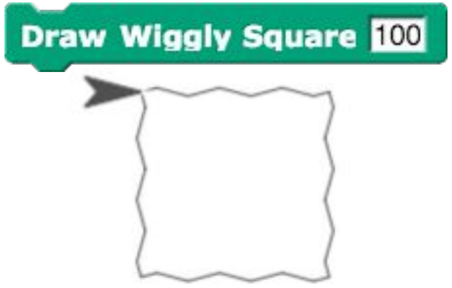


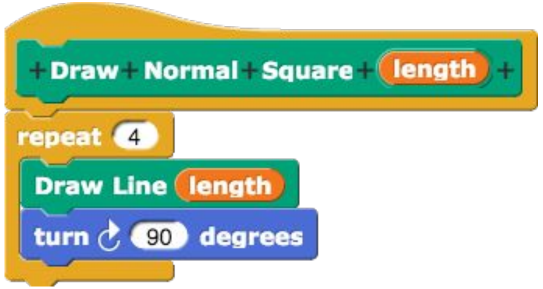
```
call (sum from num1: [ ] to num2: [ ]) with inputs 1 4
```

Build a Drawing HOF!

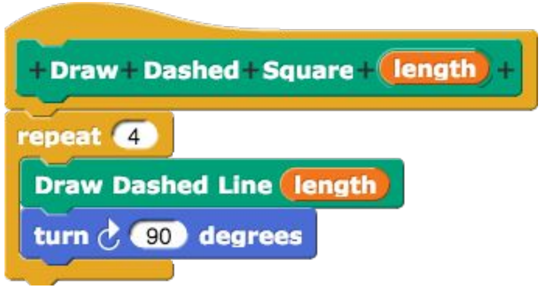




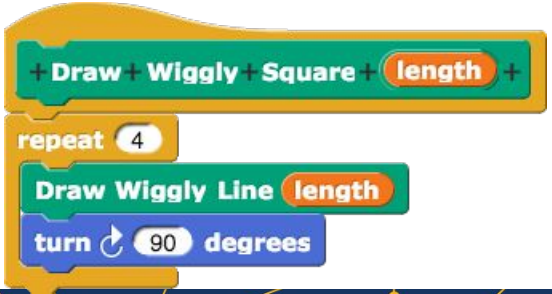




```
+Draw+Normal+Square+(length)+
repeat 4
    Draw Line (length)
    turn ↻ 90 degrees
```

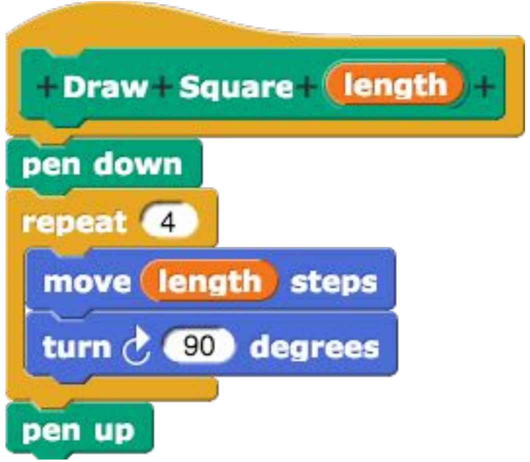


```
+Draw+Dashed+Square+(length)+
repeat 4
    Draw Dashed Line (length)
    turn ↻ 90 degrees
```



```
+Draw+Wiggly+Square+(length)+
repeat 4
    Draw Wiggly Line (length)
    turn ↻ 90 degrees
```

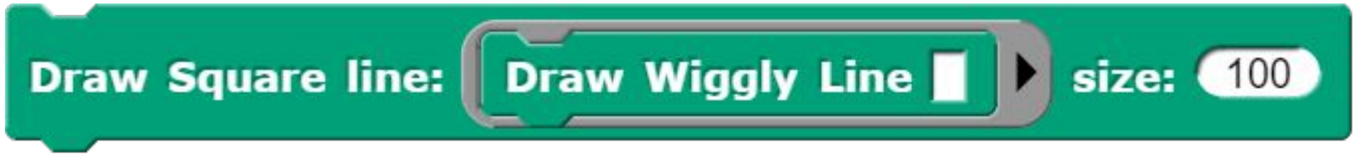
- Not very efficient
- Tedious if repeated
- Let’s generalize Draw Square into a HoF.

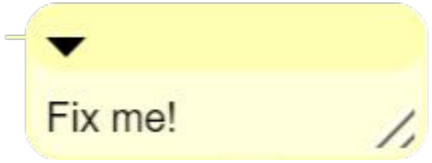


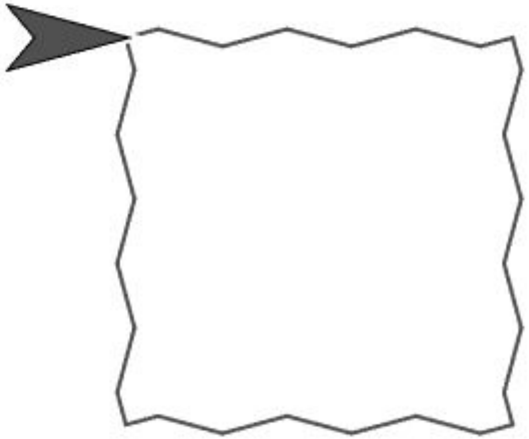
```
+Draw+Square+(length)+
pen down
repeat 4
    move (length) steps
    turn ↻ 90 degrees
pen up
```

Build a Drawing HOF!

Objective: Generalize square to draw any line type!

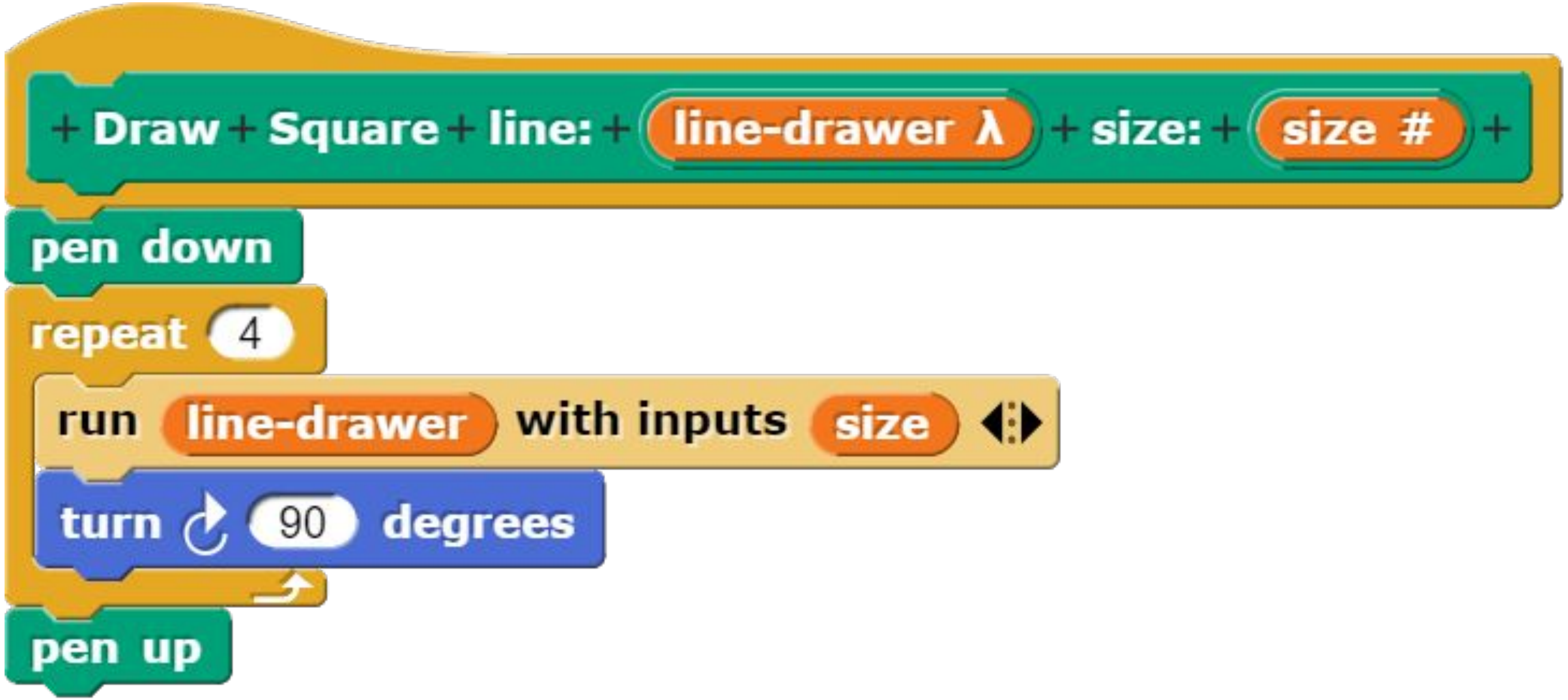

Draw Square line: Draw Wiggly Line size: 100


Fix me!



Build a Drawing HOF!

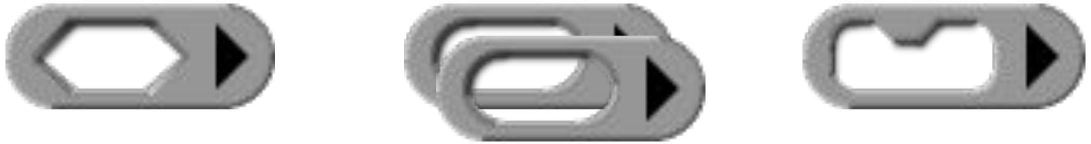
Objective: Generalize square to draw any line type!



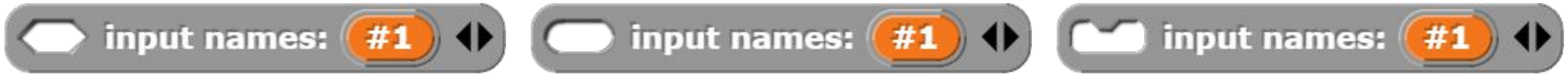
```
+ Draw + Square + line: + (line-drawer λ) + size: + (size #) +
pen down
repeat (4)
    run (line-drawer) with inputs (size) ◂▸
    turn ↻ (90) degrees
pen up
```

Lambdas

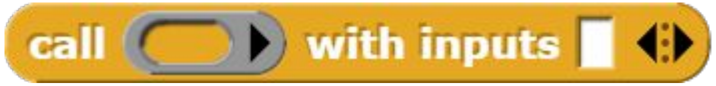
- Defn: A temporary, anonymous function that disappears after use
- In Snap*!*, we denote lambdas by:



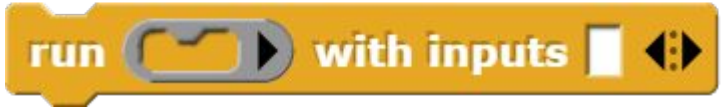
- We can also create temporary local variables for the lambdas:



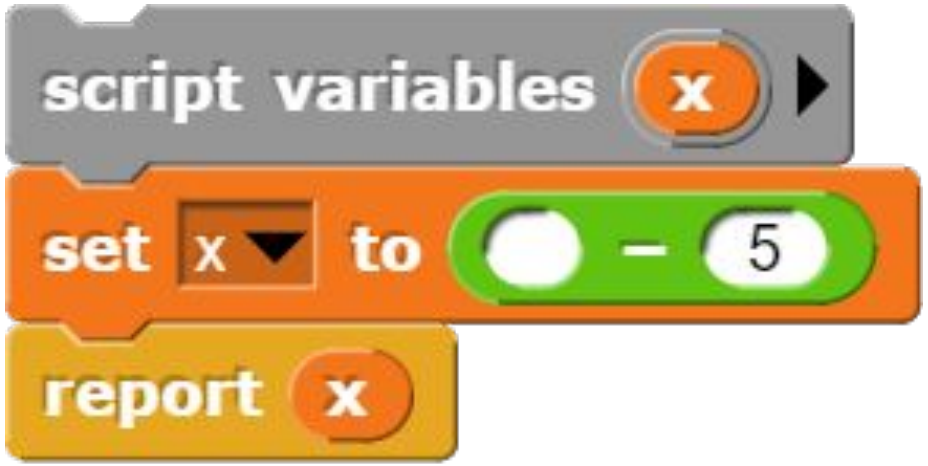
- Lambdas create functions - but they're not invoked / called!
- To invoke / call, we need:



OR

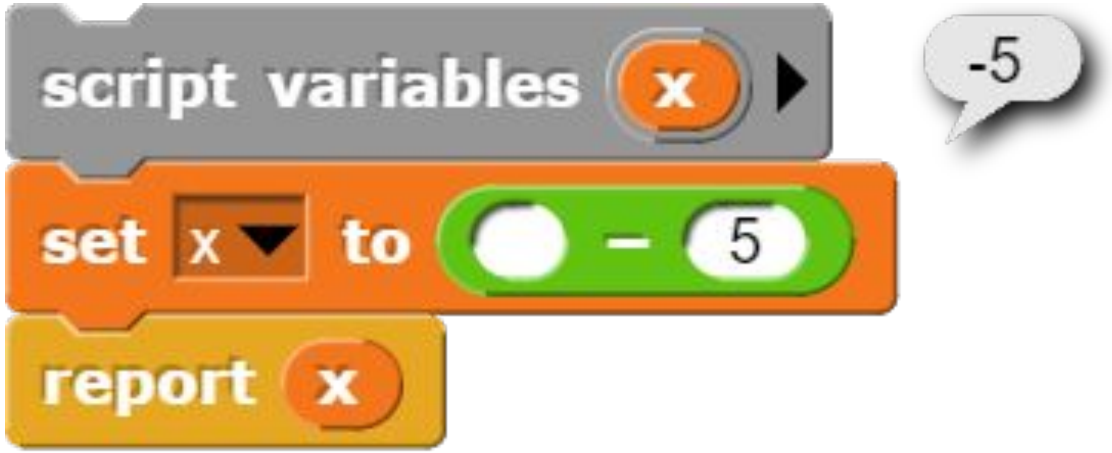


Guess that Output!

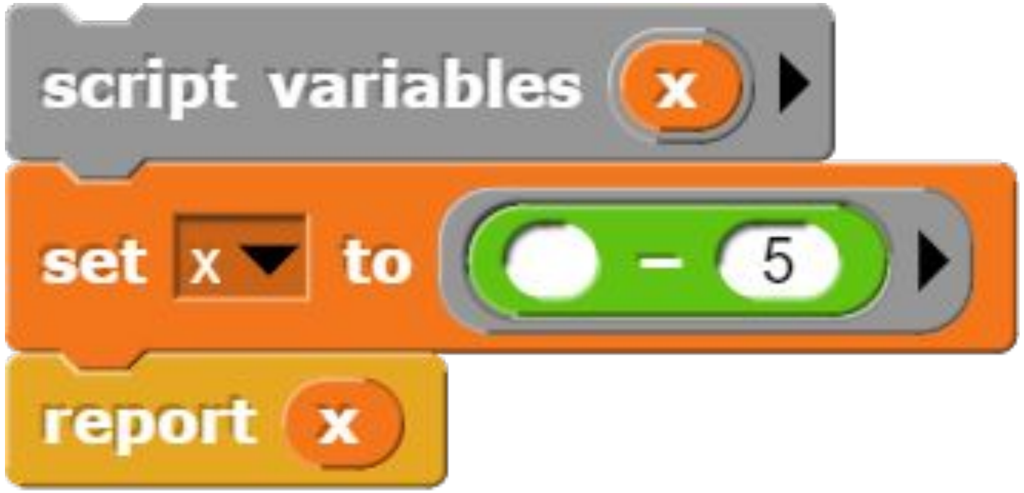


```
script variables (x) ▶
set x ▼ to ( ( ) − (5) )
report (x)
```

Guess that Output!

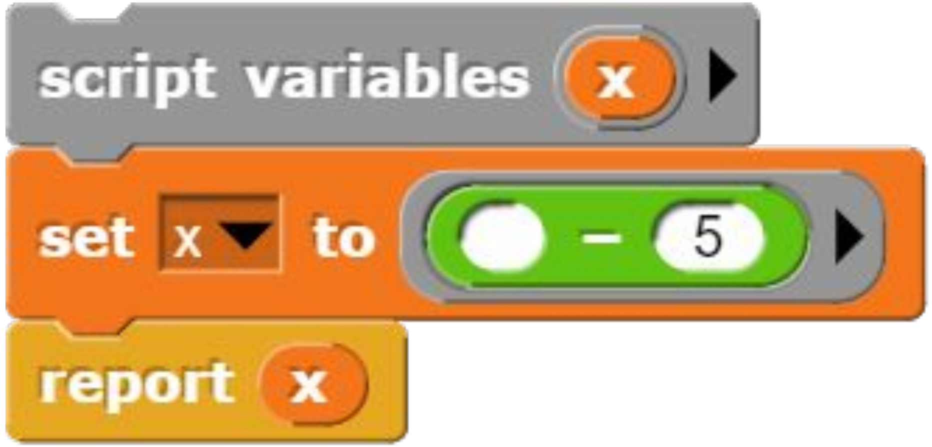

```
script variables (x)
set x to ( ( ) - (5) )
report (x)
```

Guess that Output!

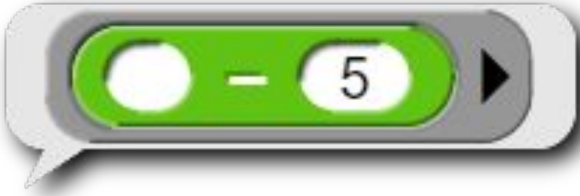


```
script variables (x)
set x to ( ( ) − (5) )
report (x)
```

Guess that Output!

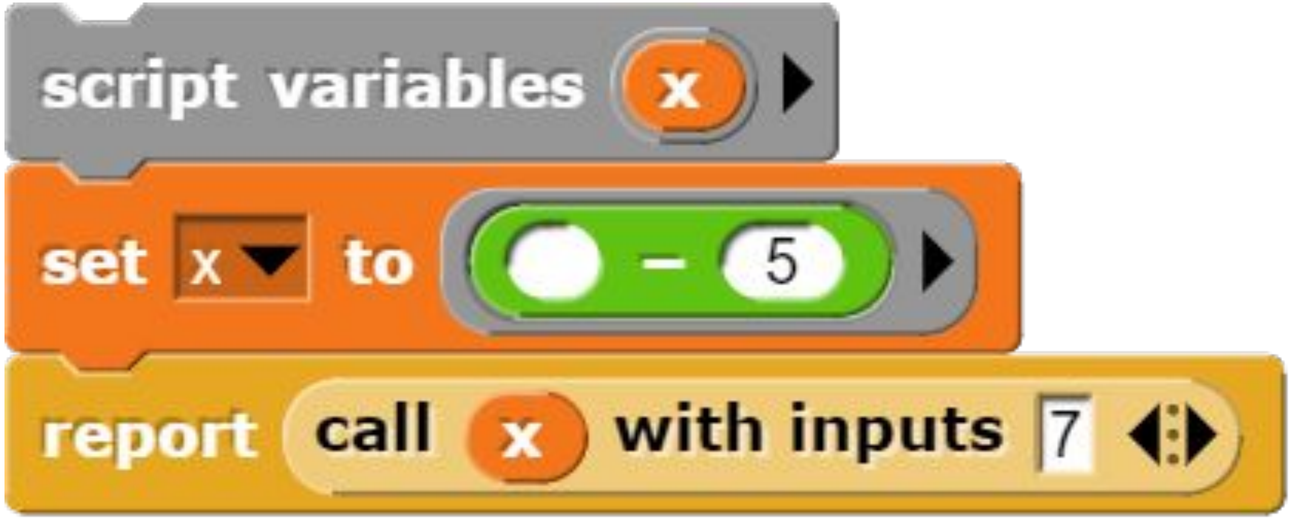


```
script variables (x)
set [x] to (( ) − (5))
report (x)
```



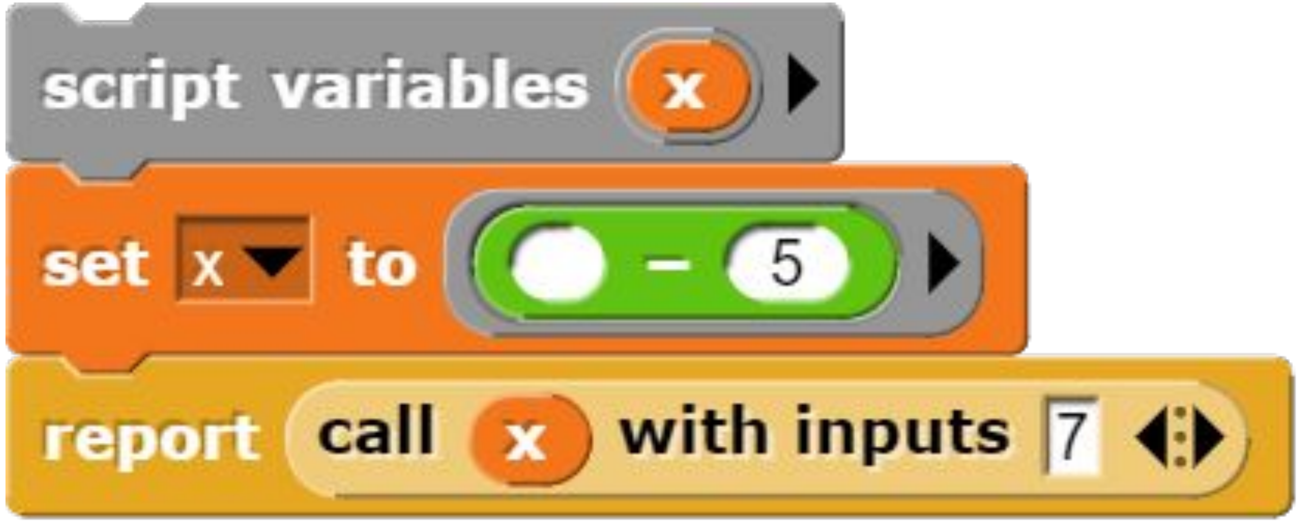
```
(( ) − (5))
```

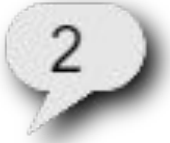
Guess that Output!



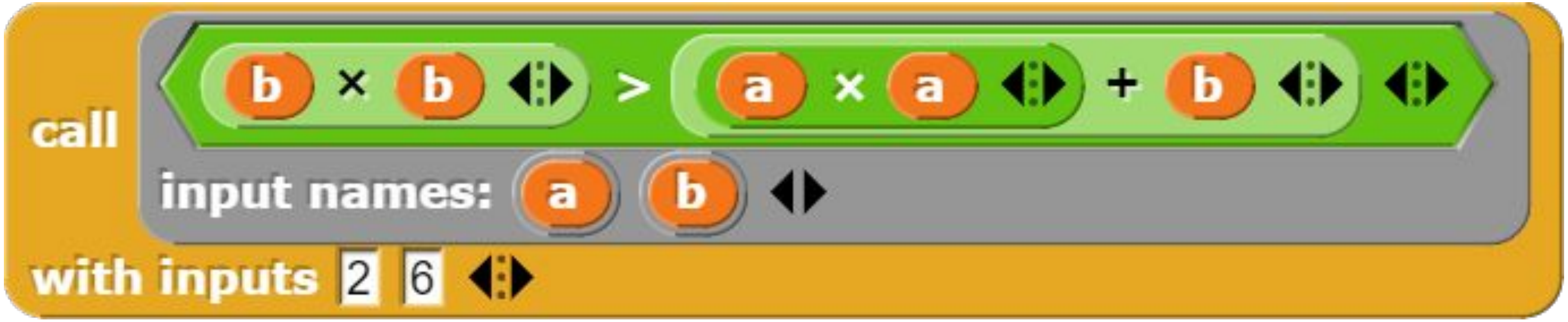
```
script variables (x)
set x to (( ) − (5))
report (call (x) with inputs 7)
```

Guess that Output!

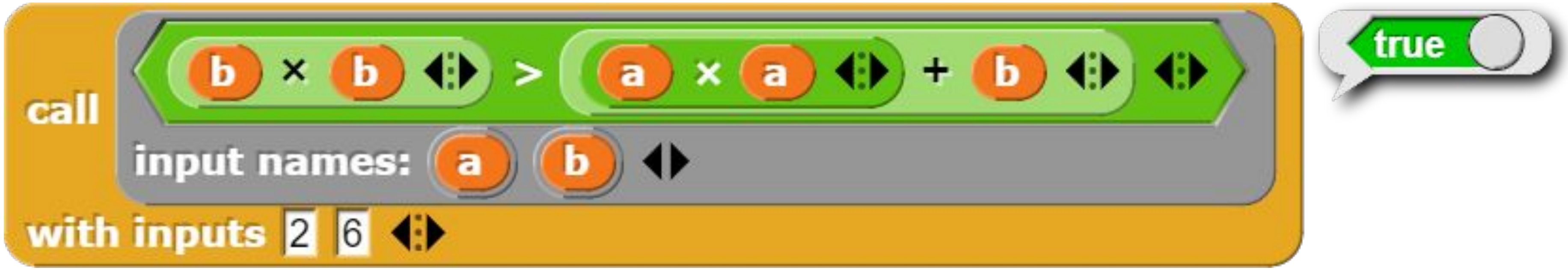

```
script variables (x) ▸
set x ▾ to ( ( ) − (5) ) ▸
report call (x) with inputs [7] ◂▸
```



Guess that Output!



Guess that Output!



Evaluates to:

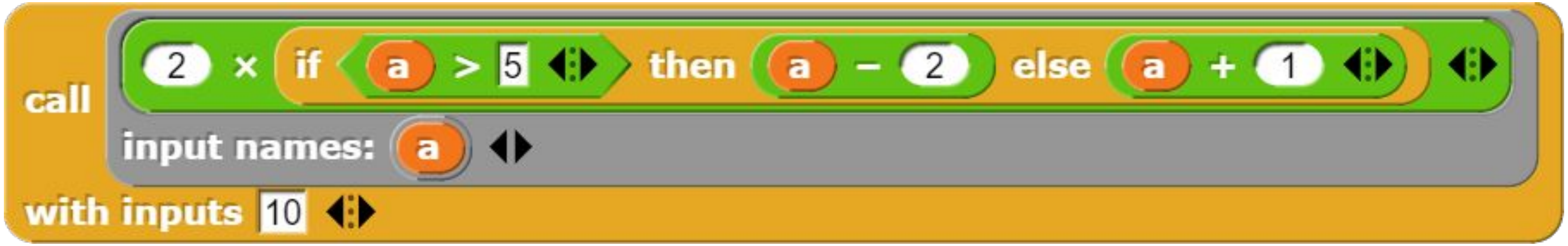
(6 * 6) > ((2 * 2) + 6)

(36) > ((4) + 6)

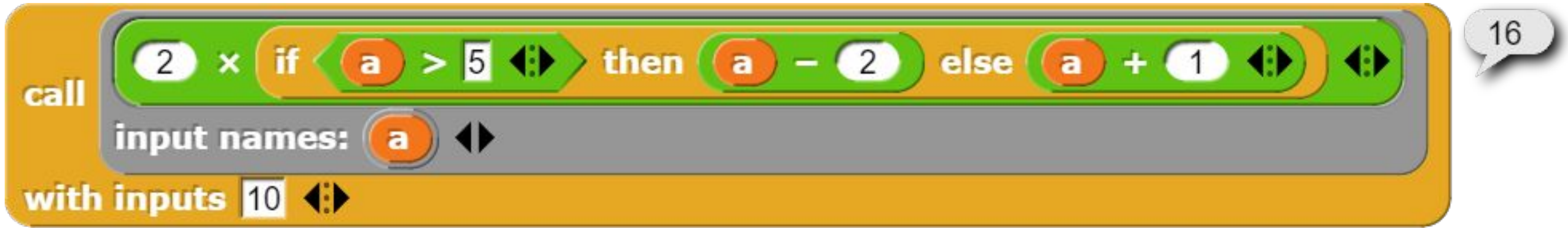
(36) > (10)

true

Guess that Output!


```
call (2 × (if ⟨(a) > 5⟩ then ((a) − (2)) else ((a) + (1))))
  input names: (a)
with inputs 10
```

Guess that Output!



Evaluates to:

2 * (if (10 > 5) then (10 - 2) else (10 + 1))

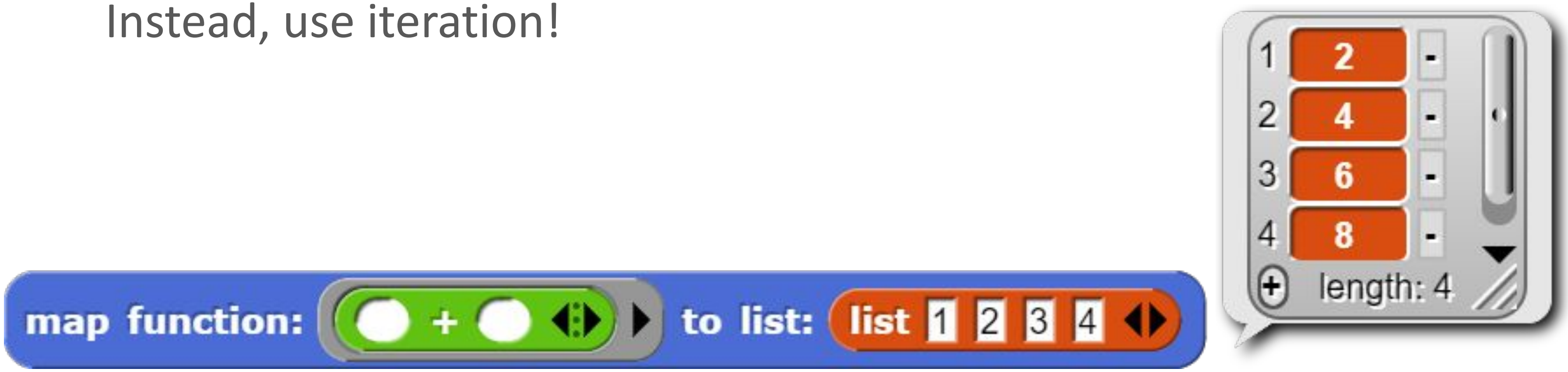
2 * (if (true) then (8) else (11))

2 * (8)

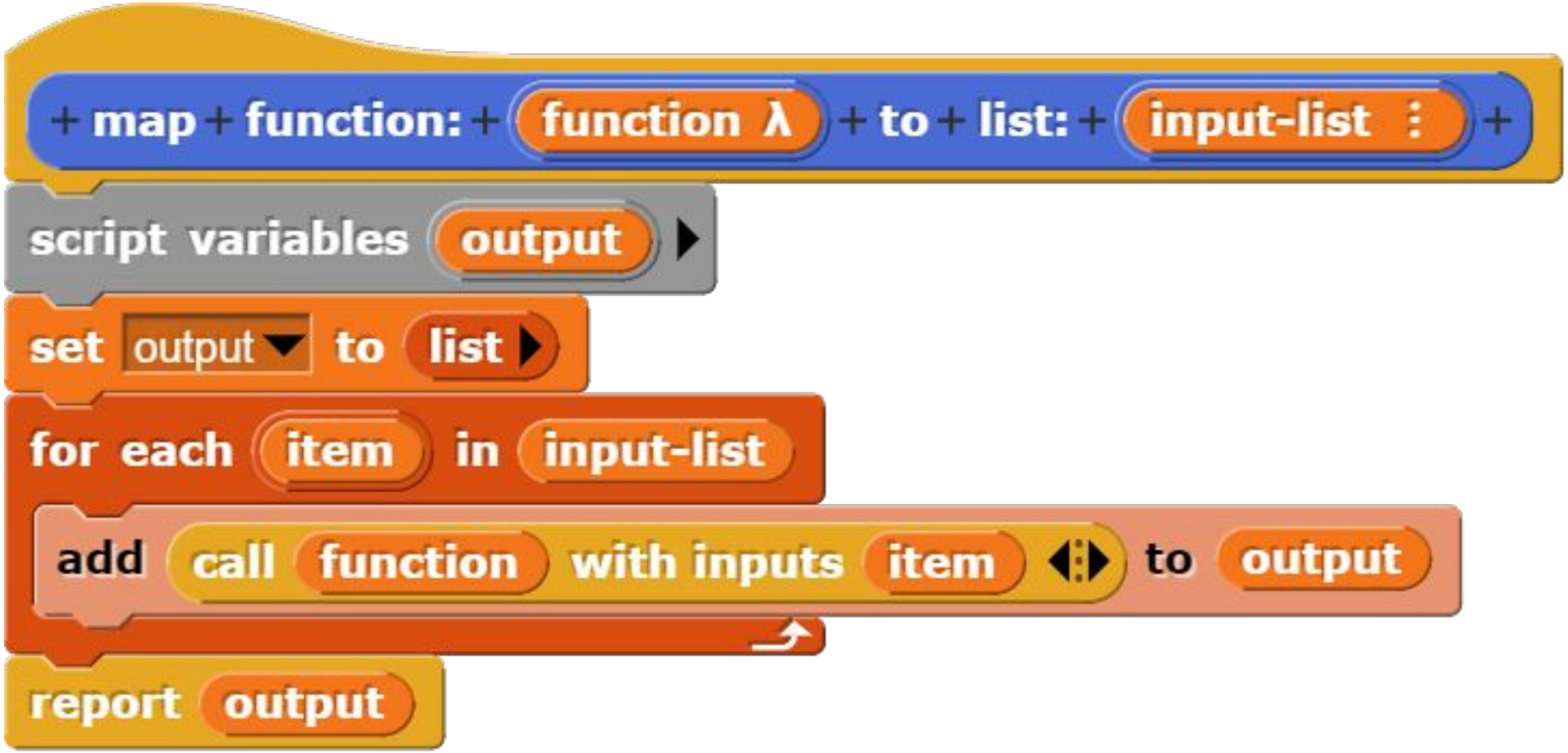
16

Build that Function!

- Objective: Create the map block from scratch. You should be invoking a function on every item from a list. Do this without using map. Instead, use iteration!



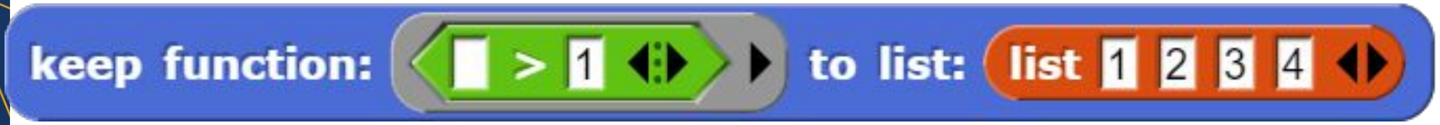
Build that Function!

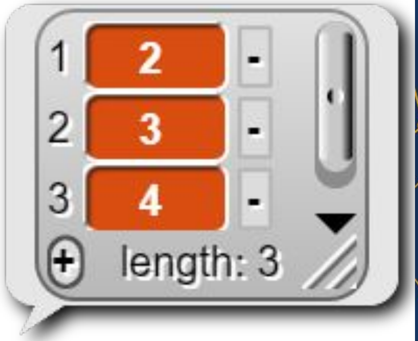


```
+ map + function: + (function λ) + to + list: + (input-list ⋮) +
script variables (output) ▶
set [output ▼] to (list ▶)
for each (item) in (input-list)
    add (call (function) with inputs (item) ◀⁝▶) to (output)
report (output)
```

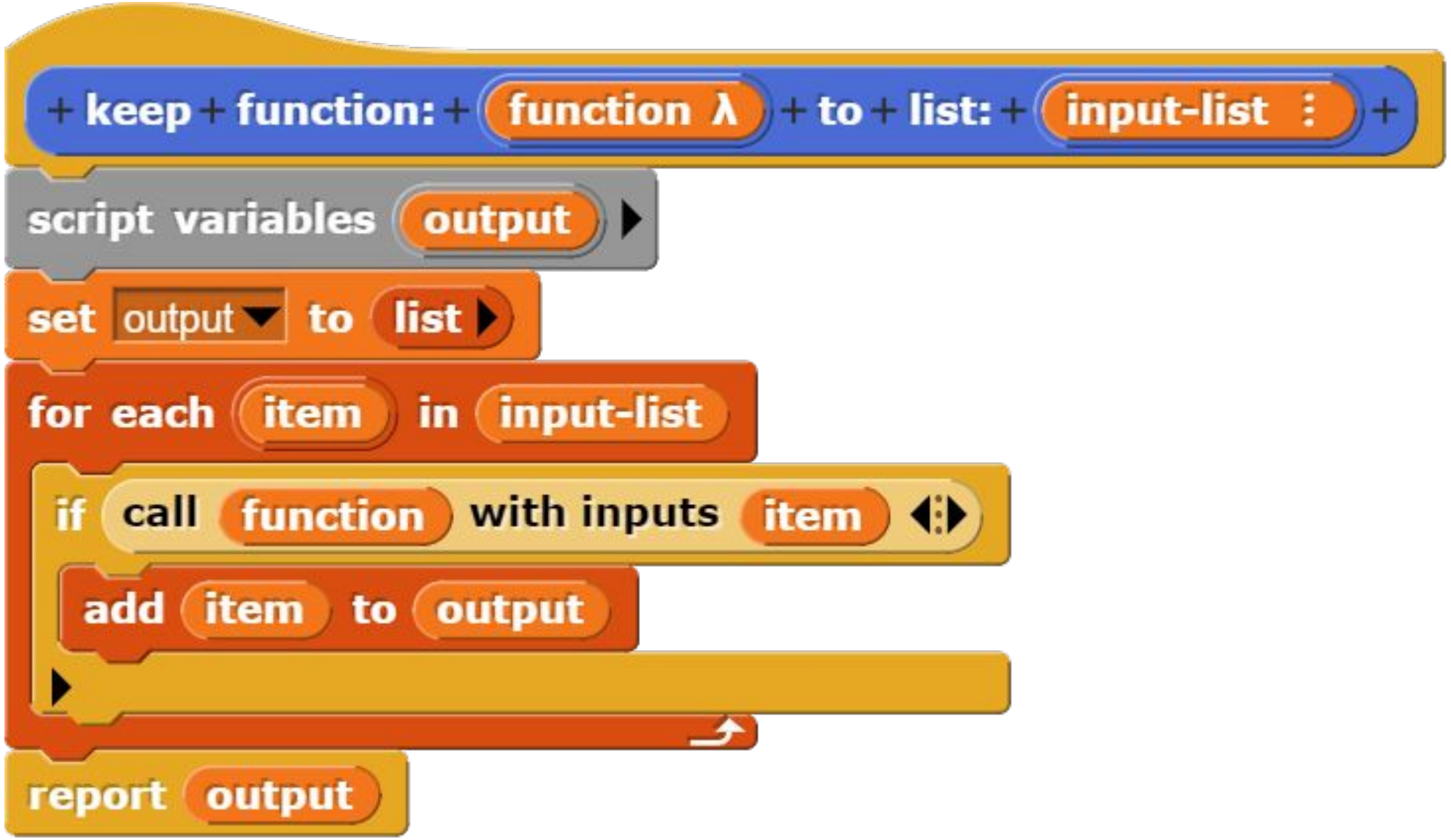
Build that Function!

- Objective: Create the keep block from scratch. You should be invoking a function on every item from a list. Do this without using map. Instead, use iteration!





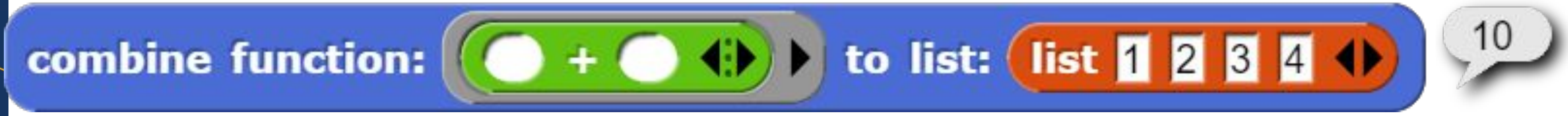
Build that Function!



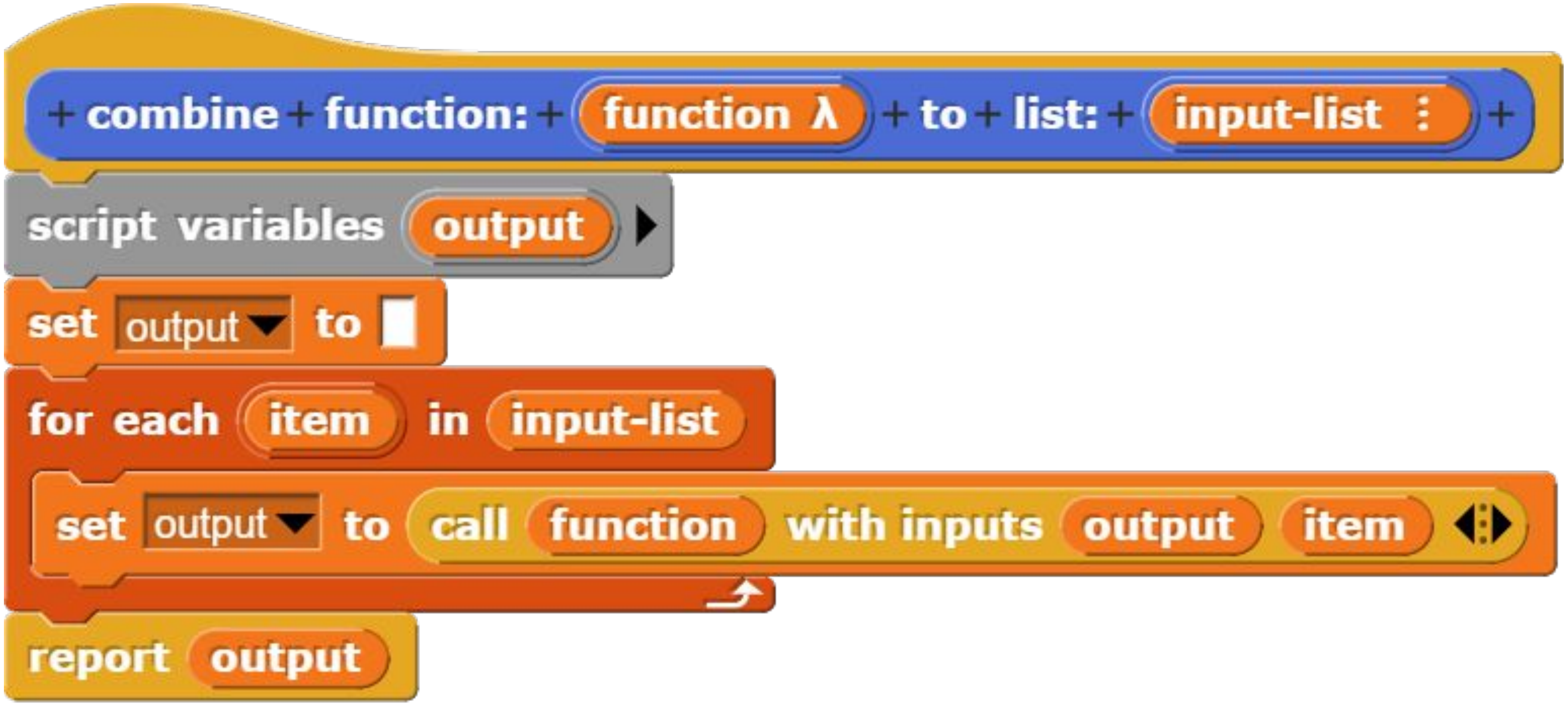
```
+ keep + function: + (function λ) + to + list: + (input-list ⋮) +
script variables (output) ▸
set output ▾ to (list ▸)
for each (item) in (input-list)
    if (call (function) with inputs (item) ◂▸)
        add (item) to (output)
    ▸
report (output)
```

Build that Function!

- Objective: Create the combine block from scratch. You should be invoking a function on every item from a list. Do this without using map. Instead, use iteration!



Build that Function!



```
+ combine + function: + (function λ) + to + list: + (input-list ⋮) +
script variables (output) ▸
set [output ▾] to [ ]
for each (item) in (input-list)
    set [output ▾] to (call (function) with inputs (output) (item) ◂▸)
report (output)
```