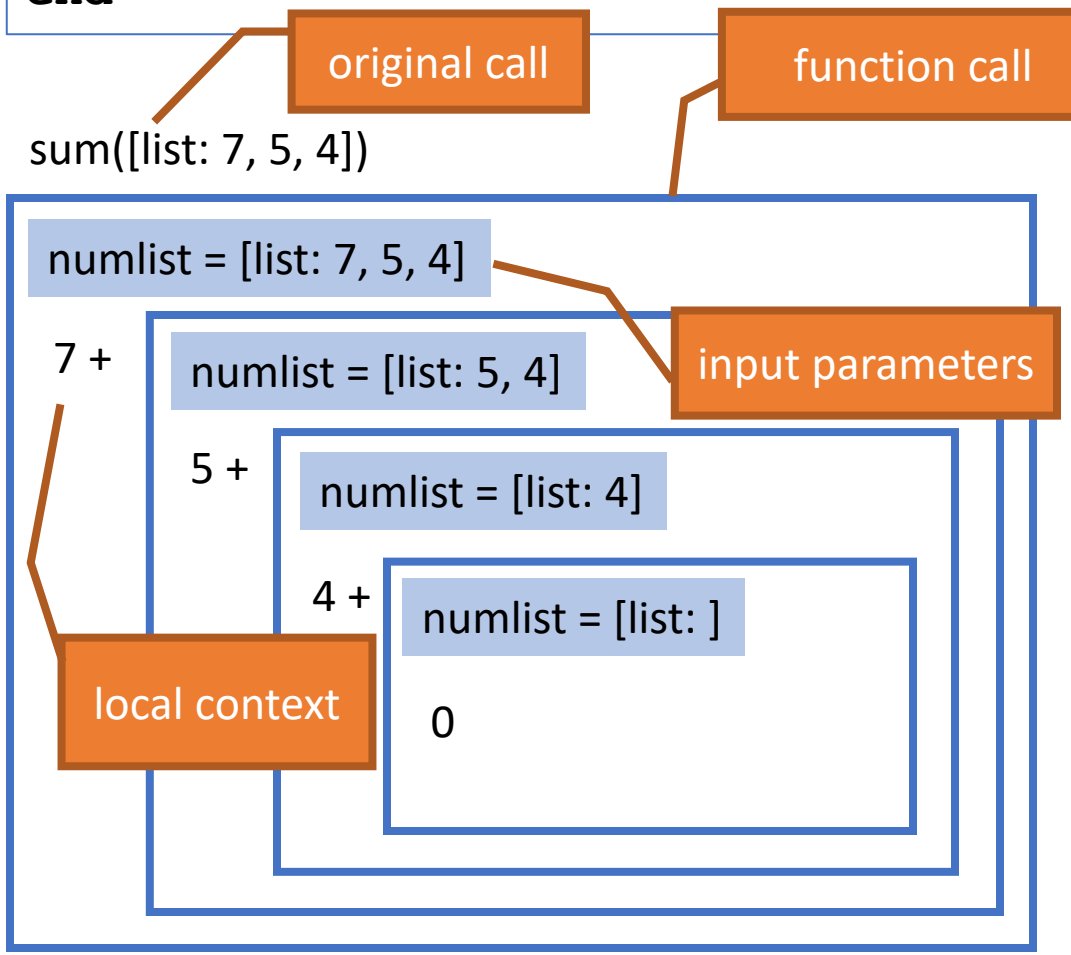
```
fun sum(numlist :: List<Number>) -> Number:
  doc: "any-negatives numbers in list"
  cases (List) numlist:
    | empty => 0
    | link(fst, rst) => fst + sum(rst)
  end
end
```



This slide shows a **FUNCTION CALL DIAGRAM**.

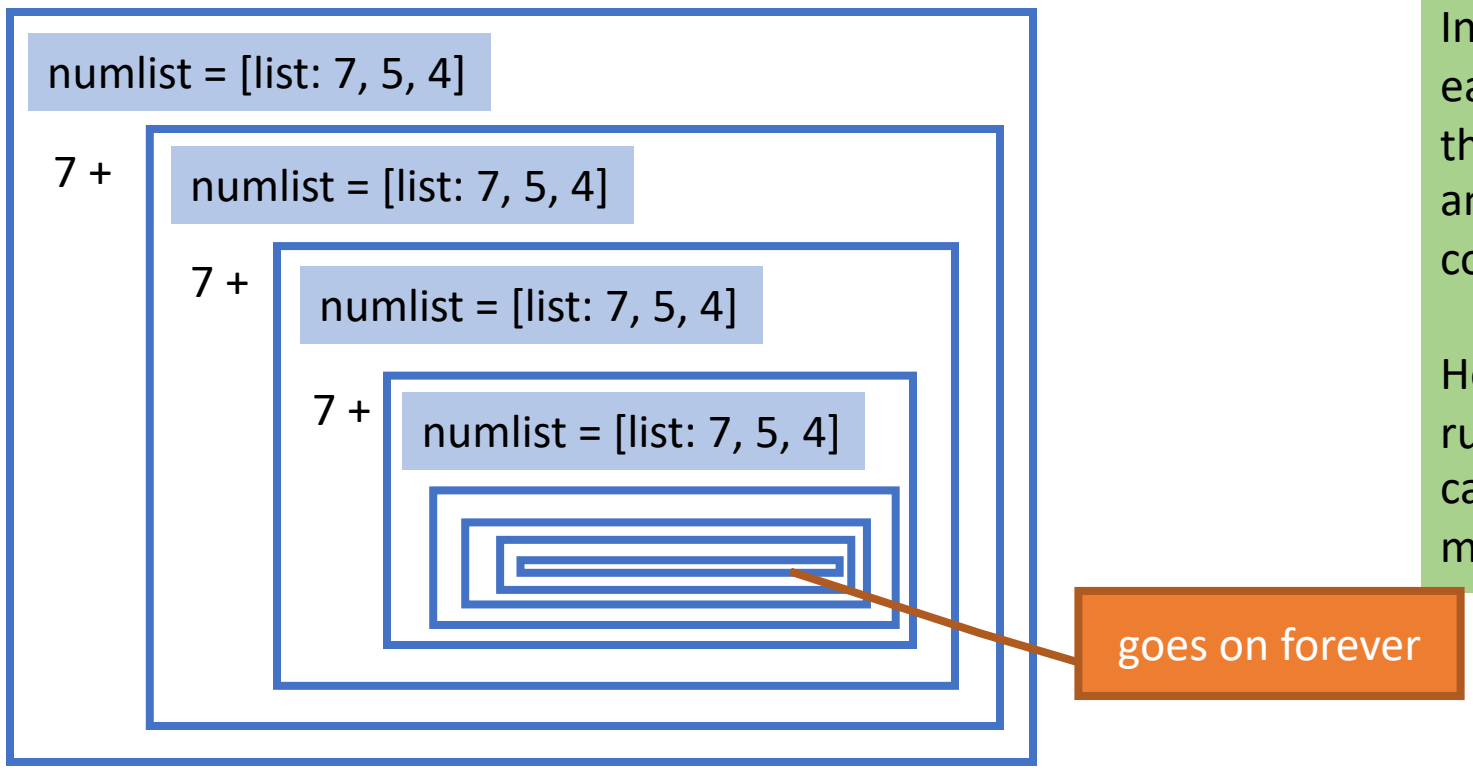
It starts with a concrete example.

Each time a function gets called, we draw a box. The parameters are listed in the shaded area inside the box. When the function call finishes, we erase the box (and its contents).

The **local context** is the part of the computation that is waiting on the result of the function call.

```
fun sum(numlist :: List<Number>) -> Number:
  doc: "any-negatives numbers in list"
  cases (List) numlist:
    | empty => 0
    | link(fst, rst) => fst + sum(numlist)
  end
end
```

sum([list: 7, 5, 4])



This slide shows what would happen on a broken version of sum that calls the function again on the ENTIRE list, rather than on the rest of the list, in the link case. (See the highlighting on the code for the change).

In the previous slide, numlist got shorter on each call to sum, eventually getting down to the empty list. Once the empty list is reached, an answer is returned that allows the entire computation to finish.

Here, since numlist never gets shorter, running the program keeps generating new calls to sum forever (or until Pyret runs out of memory in your browser).

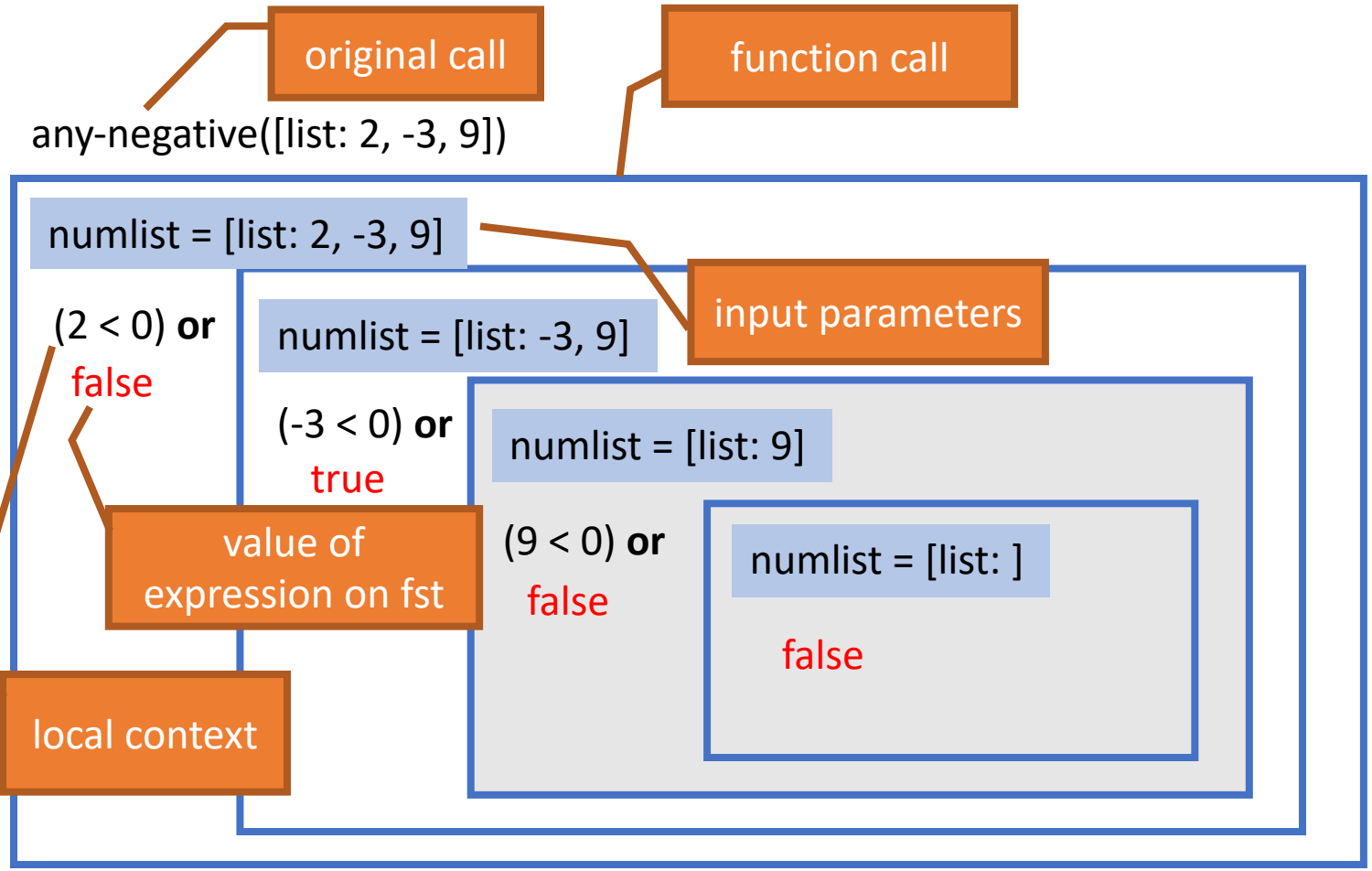
```
fun any-negative(numlist :: List<Number>) -> Boolean:
  cases (List) numlist:
    | empty => false
    | link(fst, rst) => (fst < 0) or any-negative(rst)
  end
end
```

You try it – work out the function-call diagram for the call to any-negative that's shown below the code.

(the solution is on the next page)

any-negative([list: 2, -3, 9])

```
fun any-negative(numlist :: List<Number>) -> Boolean:
  cases (List) numlist:
    | empty => false
    | link(fst, rst) => (fst < 0) or any-negative(rst)
  end
end
```



In reality, the gray boxes aren't run, because Pyret computes (-3 < 0), determines that it is true, then returns true as the value of the or without looking at the rest of the function. But that's a minor detail here.