Putting Together and Taking Apart Lists

Real Quick Warm-up: `map`

Go to the starter file from yesterday.

The helper function from yesterday:

```
fun web-com-email(email):
    split-string-all(email, “@”).get(1) == “web.com”
end
```

Run the file then try:

```
map(web-com-email, emails)
```

What is map’s contract? `map:: (A -> B), List<A> -> List<B>`

What is its purpose statement? Takes in a list of type A and a function, and returns a list of the results of the function for each

5.2.1 Making Lists and Taking Them Apart

So far we've seen one way to make a list: by writing `[list: ...]`. While useful, writing lists this way actually hides their true nature. Every list actually has two parts: a *first* element and the *rest* of the list. The rest of the list is itself a list, so it too has two parts…and so on.

Consider the list `[list: 1, 2, 3]`. Its first element is `1`, and the rest of it is `[list: 2, 3]`. For this second list, the first element is `2` and the rest is `[list: 3]`.

> ***Do Now!***
>
> Take apart this third list.

For the third list, the first element is `3` and the rest is `[list: ]`, i.e., the empty list. In Pyret, we have another way of writing the empty list: `empty`.

Lists are an instance of *structured data*: data with component parts and a well-defined format for the shape of the parts. Lists are formatted by the first element and the rest of the elements. Tables are somewhat structured: they are formatted by rows and columns, but the column names aren't consistent across all tables. Structured data is valuable in programming because a predictable format (the structure) lets us write programs based on that structure. What do we mean by that?

Programming languages can (and do!) provide built-in operators for taking apart structured data. These operators are called *accessors*. Accessors are defined on the structure of the datatype alone, independent of the contents of the data. In the case of lists, there are two accessors: `first` and `rest`. We use an accessor by writing an expression, followed by a dot (`.`), followed by the accessor's name. As we saw with tables, the dot means "dig into". Thus:

```
l1 = [list: 1, 2, 3]
e1 = l1.first
l2 = l1.rest
e2 = l2.first
l3 = l2.rest
e3 = l3.first
l4 = l3.rest
```

On your handouts, try to work out the missing values, then check them in Pyret…

```
check:
  e1 is 1
  e2 is 2
  e3 is 3
  l2 is [list: 2, 3]
  l3 is [list: 3]
  l4 is empty
end
```

Accessors give a way to take data apart based on their structure (there is another way that we will see shortly). Is there a way to also *build* data based on its structure? So far, we have been building lists using the `[list: ...]` form, but that doesn't emphasize the structural constraint that the `rest` is itself a list. A structured operator for building lists would clearly show both a `first` element and a `rest` that is itself a list. Operators for building structured data are called *constructors*.

```
link(1, [list: 2, 3]) is link(1, link(2, [list: 3]))
```

The constructor for lists is called `link`. It takes two arguments: a `first` element, and the list to build on (the `rest` part). Here's an example of using `link` to create a three-element list.

```
link(1, link(2, link(3, empty)))
```

The `link` form creates the same underlying list datum as our previous `[list: ...]` operation, as confirmed by the following check:

```
check:
  [list: 1, 2, 3] is link(1, link(2, link(3, empty)))
end
```

Do Now!

Use the `link` form to write a four-element list of fruits containing `"lychee"`, `"dates"`, `"mango"`, and `"durian"`.

This means we actually have two structural features of lists, both of which are important when writing programs over lists:

1. Lists can be empty or non-empty

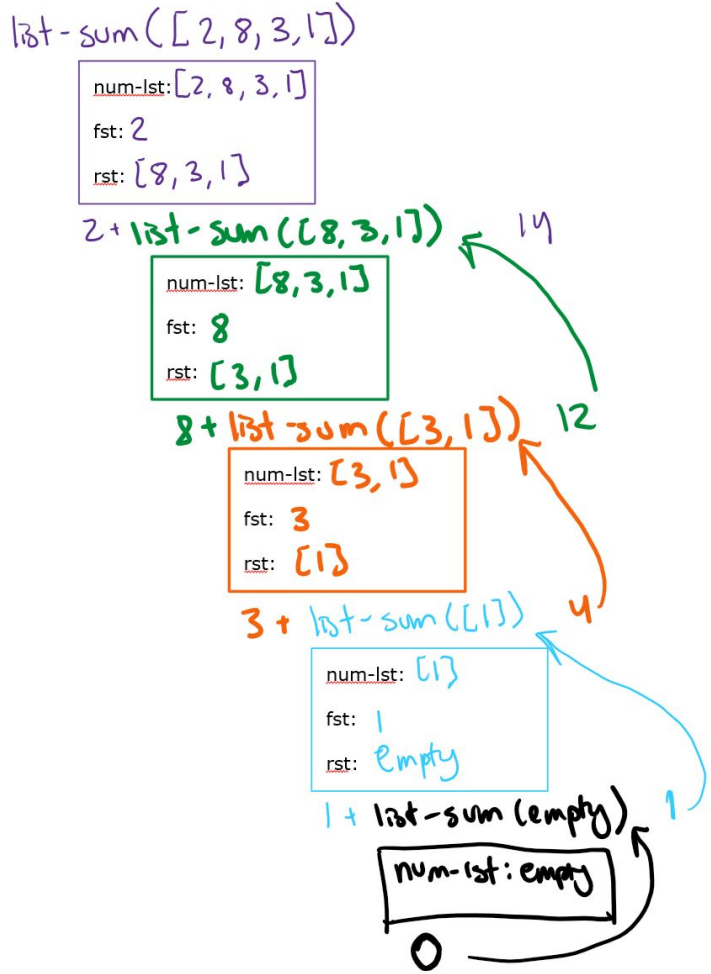
2. Non-empty lists have a first element and a rest of the list

Let's leverage these two structural features to write some programs to process lists!

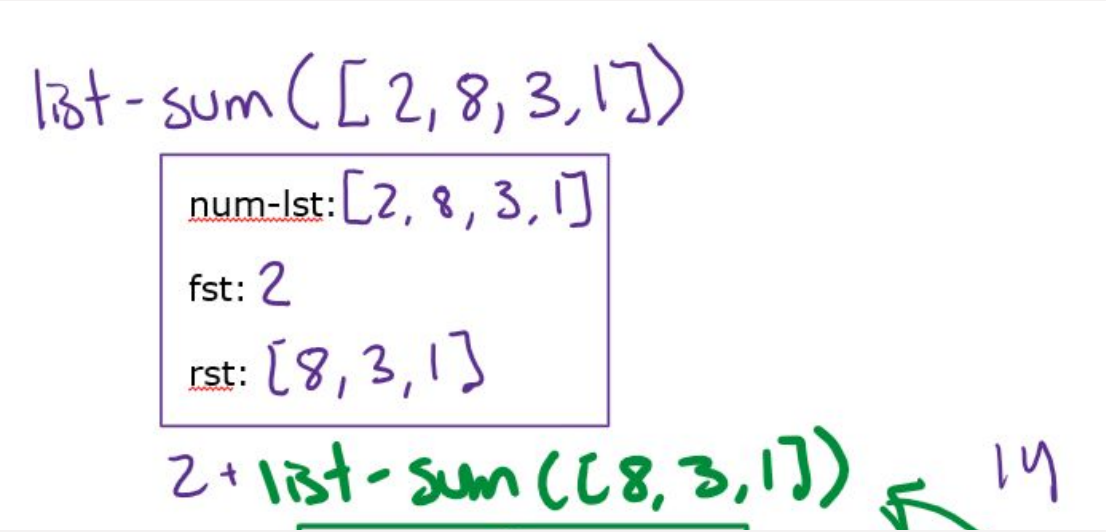
A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.

```
nums = [list: 2, 3, 8, 1]
example:
   list-sum(nums) is 14
   list-sum(nums) is 2 + 3 + 8 + 1
   list-sum(nums) is 2 + list-sum([list: 3, 8, 1])
   list-sum(nums) is 2 + 3 + 9
   list-sum(nums) is 2 + 3 + list-sum([list: 8, 1])
   list-sum(nums) is 2 + 3 + 8 + list-sum([list: 1])
   list-sum(nums) is 2 + 3 + 8 + 1 + list-sum([list: ])
end
```

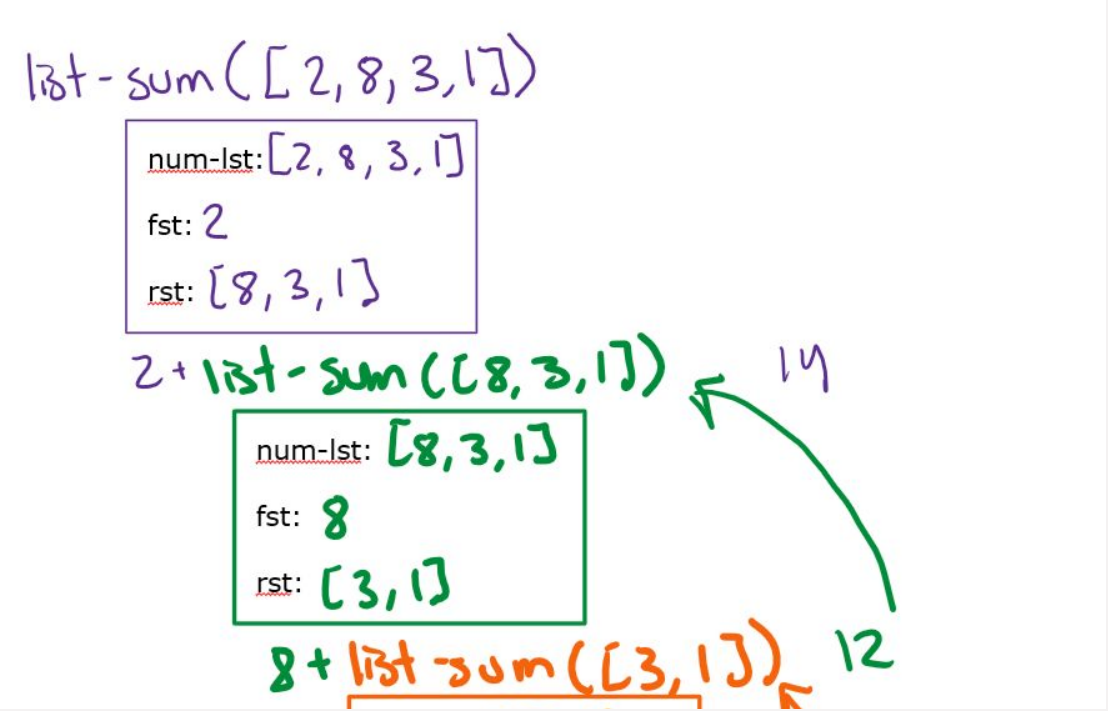
A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.



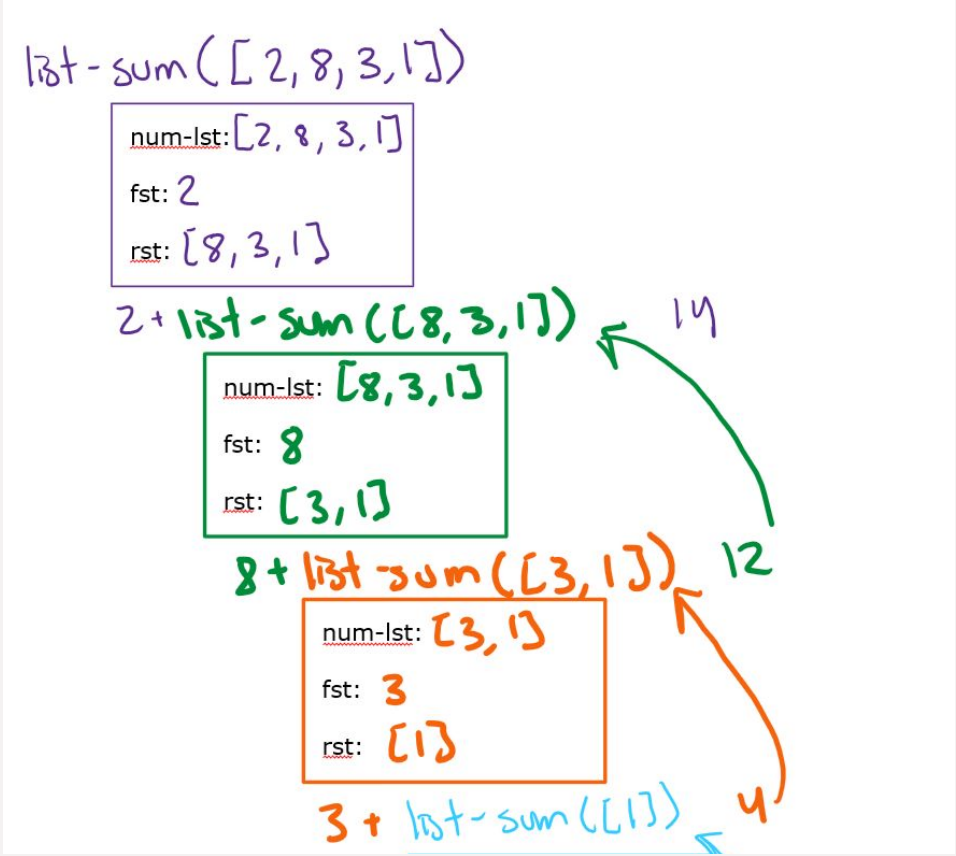
A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.



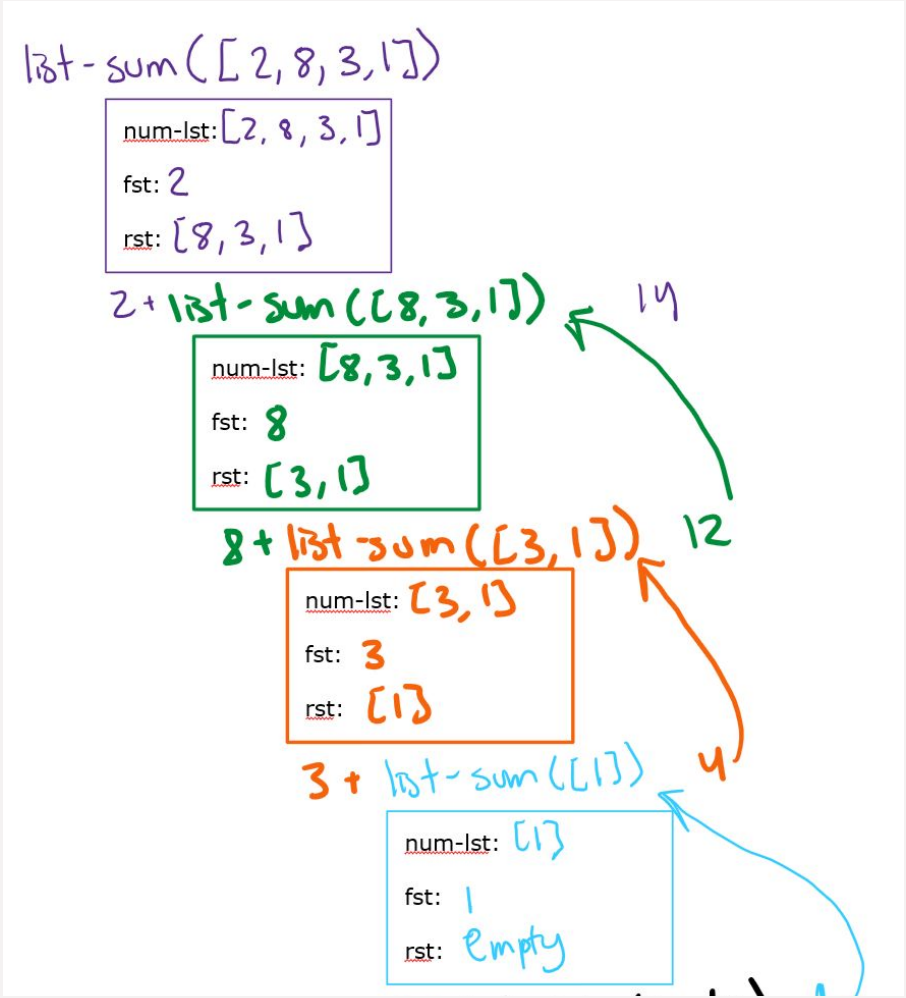
A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.



A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.



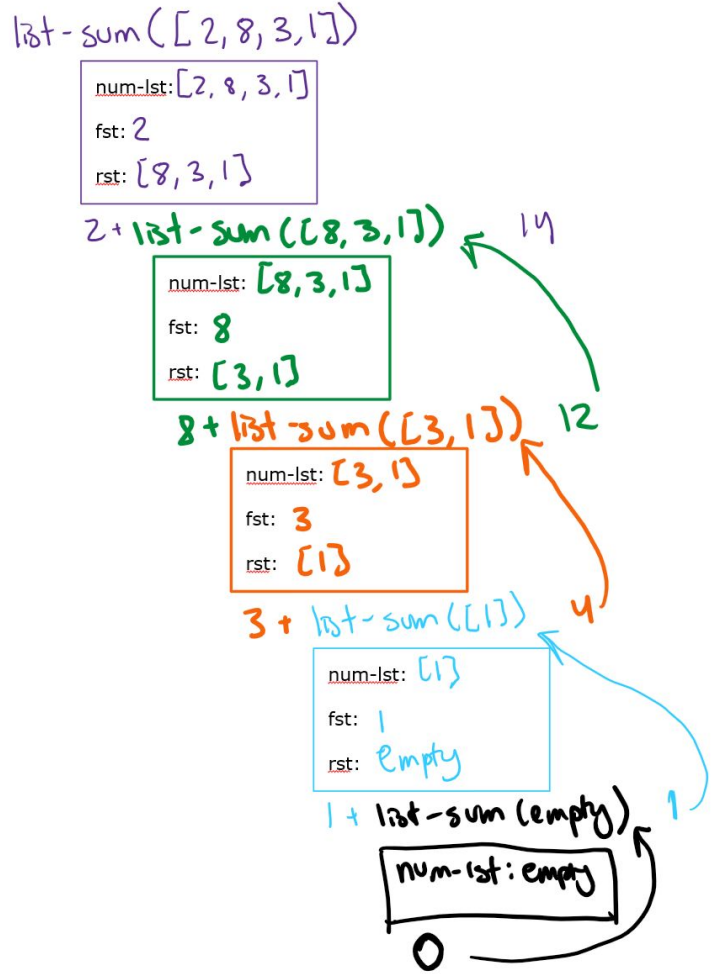
A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.



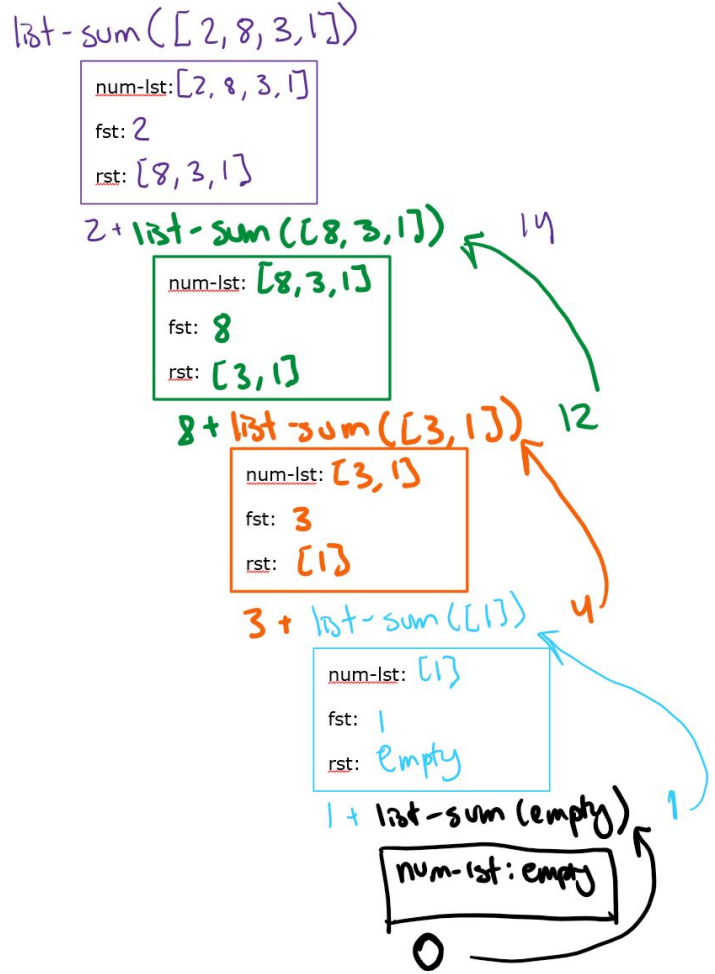
A function called `list-sum` that consumes a list of `Numbers`, and returns the sum of the elements of the `List`.

Pick off the first value, and add it to the list-sum computed by the rest of the list.

If the list is empty, the result is zero



```
fun list-sum(num-list):
    cases (List) num-list:
    | empty => 0
    | link(first, rest) =>
    first + list-sum(rest)
    end
end
```



draw-row:
Make a function that takes in a list of Strings containing color values. Draw a row of shapes of the same type and size, so that the colors appear in the same order that they do in the list.

Here is a helper that you can use to draw a shape given a color:

```
fun draw-square(c):
     squares(50, “outline, c”)
end

color-list = [list: “red”, “green”, “blue”]
```

Contract: draw-row :: List<String> -> Image

Purpose Statement: Takes in `List` of color strings, output `Image`. he image should be a row of shapes colored in the order of the `List`.

```
example:
     draw-row(color-list) is
          beside(draw-square(“red”),
               beside(draw-square(“green”),
                    beside(draw-square(“blue”),
                         empty-image)))

end
```

```
beside :: (
   img1 :: Image,
   img2 :: Image
)
-> Image
```

Constructs an image by placing `img1` to the left of `img2`.

Examples:

```
››› beside(rectangle(30, 60, "solid", "orange"),
    ellipse(60, 30, "solid", "purple"))
```

Nested Table Diagram

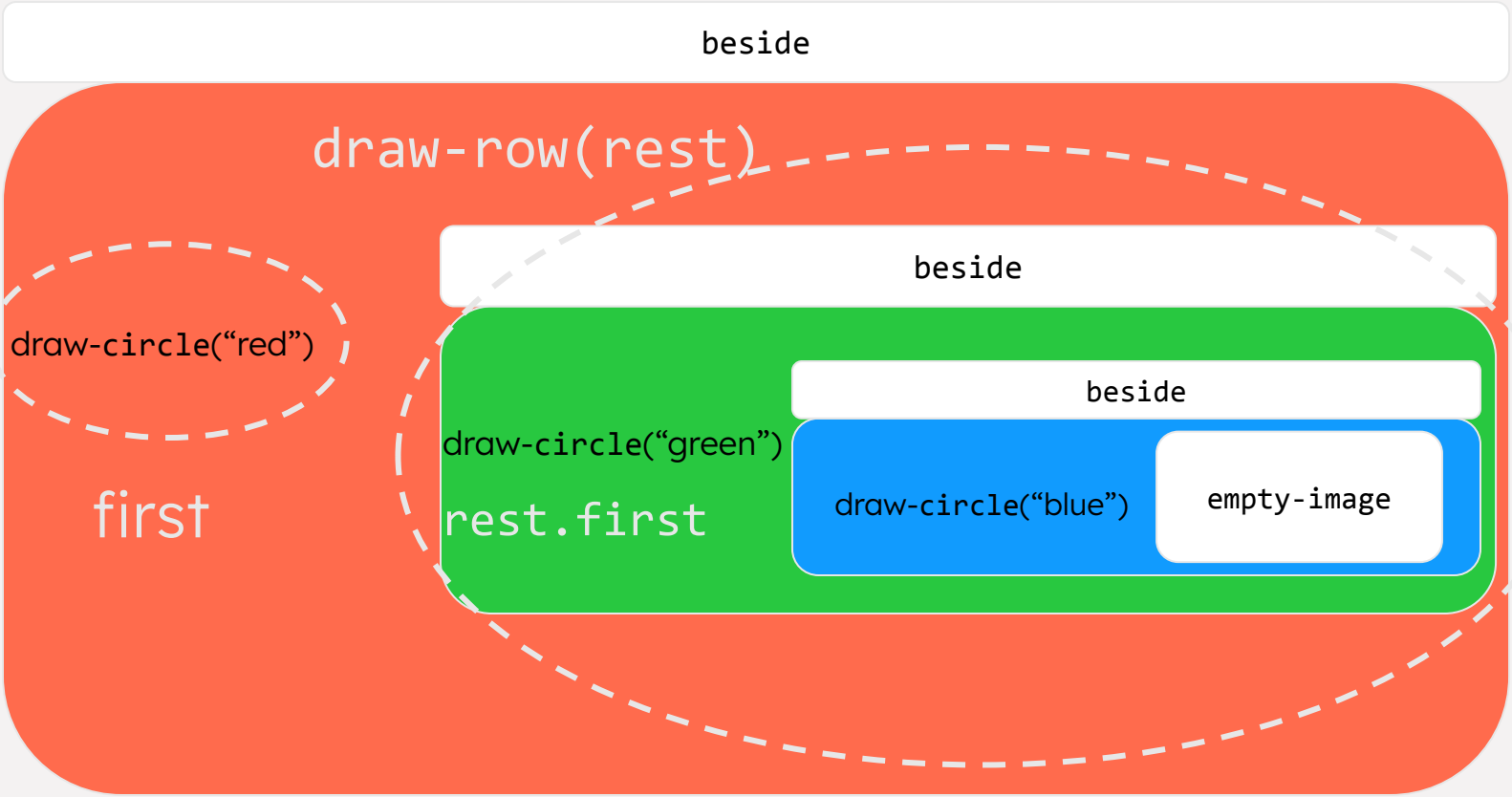
```
draw-row([list: “red”, “green”, “blue”]):
first: “red”
rest: [list: “green”, “blue”]
```

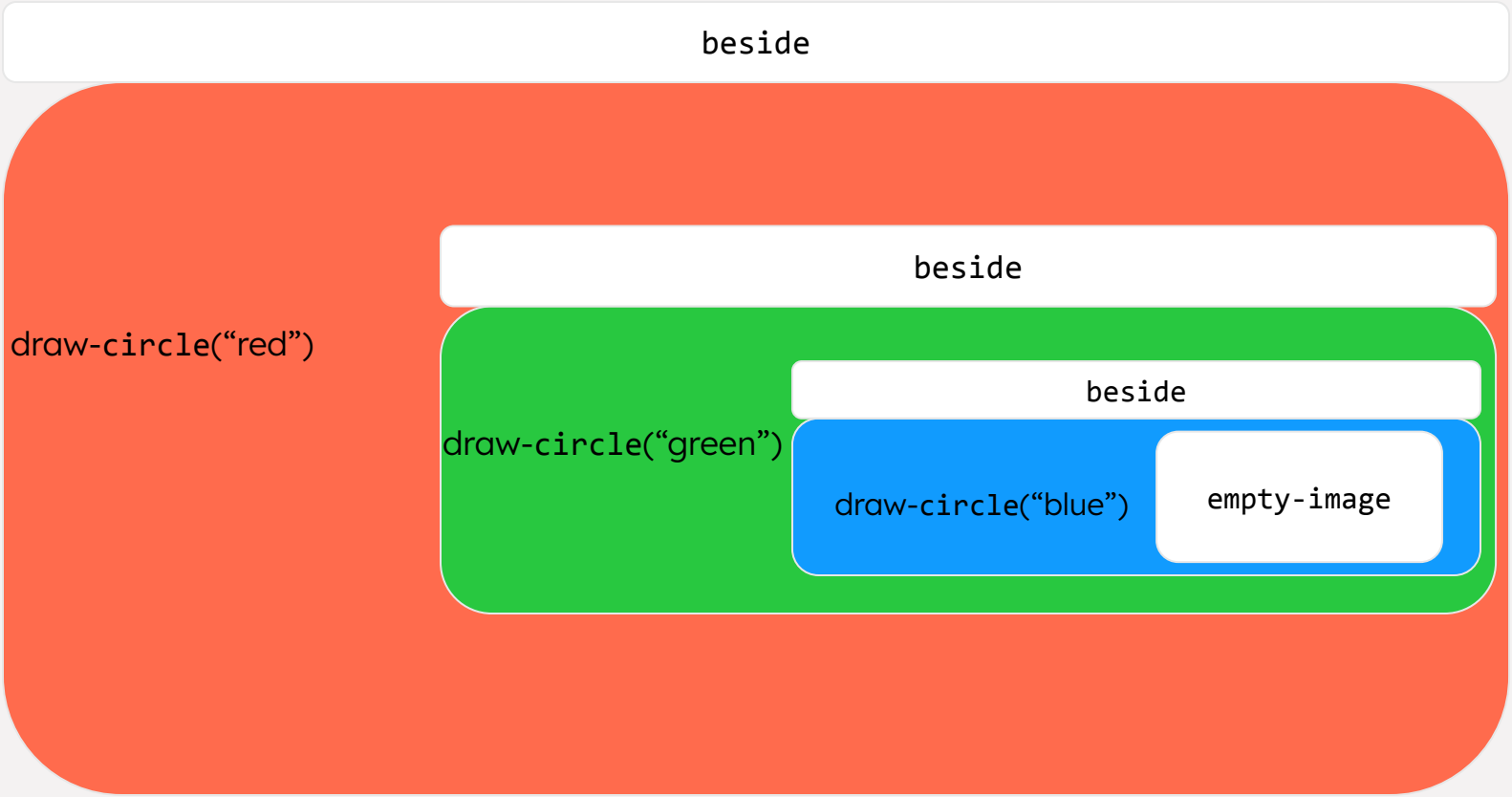
```
draw-row([list: “green”, “blue”]):
first: “green”
rest: [list: “blue”]
```

```
draw-row([list: “blue”])
first: “blue”
rest: [list: ]
```

```
draw-row([list: ])
empty
```

draw-row([list: “red”, “green”, “blue])




beside
draw-circle("red")
beside
draw-circle("green")
beside
draw-circle("blue")
empty-image

`draw-cell`

Directions: Take in a `String`. Draw `water-img` if the String is “o” and if not, draw `grass-img`

Contract: `draw-cell :: String -> Image`
Purpose: Input a `String`. Output `water-img` if the String is “o” and if not, draw `grass-img`

Examples:

```
examples:
    draw-cell(‘o’) is water-img
    draw-cell(‘x’) is grass-img
    draw-cell(‘1’) is grass-img
end
```

Function:

```
fun draw-cell(s):
    if s == ‘o’: water-img
    else: grass-img
    end
end
```

Just drawing one square (cell) at a time, so that we can make a row with it!

Strategy: Developing Functions Over Lists

Leverage the structure of lists and the power of concrete examples to develop list-processing functions.

- Pick a concrete list with (at least) three elements. Write a sequence of examples for each of the entire list and each suffix of the list (including the empty list).
- Rewrite each example to express its expected answer in terms of the `first` and `rest` data of its input list. You don't have to use the `first` and `rest` operators in the new answers, but you should see the `first` and `rest` *values* represented explicitly in the answer.
- Look for a pattern across the answers in the examples. Use these to develop the code: write a **`cases`** expression, filling in the right side of each `=>` based on your examples.

This strategy applies to structured data in general, leveraging components of each datum rather than specifically `first` and `rest` as presented so far.

Draw-row

Draw a row from a list of strings! The row should look like the list!

Task 5: Build the `BACKGROUND` image for a maze (the part of the image that does NOT change over time). Don't worry about including the widgets or portals. Your `BACKGROUND` image should be built using a function and `maze-grid` – you should **not** be manually constructing the maze.

Note: The maze design will be imported as a list of lists, such as the following ("x" is a wall/grass, "o" is open space/water):

```
small-maze =
  [list: [list: "x", "x", "o", "x"],
         [list: "x", "o", "o", "x"],
         [list: "x", "o", "x", "x"],
         [list: "x", "o", "x", "x"],
         ]
```

While later in the project you will read in such a list of lists from a Google Sheet, for this task you are only required to build a maze background image from a small example. Use `small-maze` for this.

How can you do this? Notice that the list of lists resembles a grid; the maze background will also be a grid, just made from image icons (with a "grass" icon in place of "x" and a "water" icon in place of "o").

Strategy: Developing Functions Over Lists

Leverage the structure of lists and the power of concrete examples to develop list-processing functions.

- Pick a concrete list with (at least) three elements. Write a sequence of examples for each of the entire list and each suffix of the list (including the empty list).

draw-row

Contract: `draw-row :: List<String> -> Image`

Purpose Statement: Takes in a `List` of `Strings` and outputs an `Image`. The `Image` should look like `water-img` and `grass-img` in the order of the `List`.

Examples:

```
l1 = [list: “x”, “x”, “o”, “x”]

draw-row([list: “x”, “x”, “o”, “x”]) is beside(grass-img,
                                               beside(grass-img,
                                                   beside(water-img, grass-img))
draw-row([list:      “x”, “o”, “x”]) is beside(grass-img,
                                               beside(water-img, grass-img))
draw-row([list:           “o”, “x”]) is beside(water-img, grass-img)
draw-row([list:                “x”]) is grass-img
draw-row([list:                   ]) is empty-image
```

Strategy: Developing Functions Over Lists

Leverage the structure of lists and the power of concrete examples to develop list-processing functions.

- Pick a concrete list with (at least) three elements. Write a sequence of examples for each of the entire list and each suffix of the list (including the empty list).
- Rewrite each example to express its expected answer in terms of the `first` and `rest` data of its input list. You don't have to use the `first` and `rest` operators in the new answers, but you should see the `first` and `rest` *values* represented explicitly in the answer.

Examples:

```
l1 = [list: “x”, “x”, “o”, “x”]

draw-row([list: “x”, “x”, “o”, “x”]) is beside(grass-img,
                                              draw-row([list: “x”, “o”, “x”]))
draw-row([list:      “x”, “o”, “x”]) is beside(grass-img, draw-row([list: “o”, “x”]))
draw-row([list:           “o”, “x”]) is beside(water-img, draw-row([list: “x”]))
draw-row([list:                “x”]) is beside(grass-img, draw-row([list:]))
draw-row([list:                   ]) is empty-image


beside(draw-cell(first), draw-row(rest))
```

Strategy: Developing Functions Over Lists

Leverage the structure of lists and the power of concrete examples to develop list-processing functions.

- Pick a concrete list with (at least) three elements. Write a sequence of examples for each of the entire list and each suffix of the list (including the empty list).
- Rewrite each example to express its expected answer in terms of the `first` and `rest` data of its input list. You don't have to use the `first` and `rest` operators in the new answers, but you should see the `first` and `rest` *values* represented explicitly in the answer.
- Look for a pattern across the answers in the examples. Use these to develop the code: write a **`cases`** expression, filling in the right side of each `=>` based on your examples.

This strategy applies to structured data in general, leveraging components of each datum rather than specifically `first` and `rest` as presented so far.

What did we do today? Draw rows of squares

- Drew a bunch of squares
- Draw-row function
- Draw-cell
- Took in a list
- Eventually a maze
- Used lists to make a draw-function
-

`draw-table`

Now we want to use the ability to create rows form the lists within small-maze, and draw the whole maze.

For this, we can use the functions we already made, `draw-cell` and `draw-row`. In the starter file, I gave variable names to the rows within `small-table`, so that we don't have to write out the contents of each list:

Contract: `draw-table :: List<List<String>> -> Image`

Purpose Statement: Takes in a `List` of `String` `Lists`,
and returns a stack of the `draw-rows` for each `String` `List`.

Examples:

```
48  small-maze =
49    [list: [list: "X", "X", "O", "X"],
50           [list: "X", "O", "O", "X"],
51           [list: "X", "O", "X", "X"],
52           [list: "X", "O", "X", "X"],
53           ]
54
55  maze-list0 = small-maze.get(0)
56  maze-list1 = small-maze.get(1)
57  maze-list2 = small-maze.get(2)
58  maze-list3 = small-maze.get(3)
59
```

1. Write examples for each suffix of the list (Hint: we can use `above` like we did with `beside`)

```
small-maze = [list: ml0, ml1, ml2, ml3]
examples:
draw-table([list: ml0, ml1, ml2, ml3]) is
draw-table([list: ml1, ml2, ml3]) is
above(draw-row(ml1), above(draw-row(ml2), draw-row(ml3)))
draw-table([list: ml2, ml3]) is above(draw-row(ml2), draw-row(ml3))
draw-table([list: ml3]) is draw-row(ml3)
draw-table([list: ]) is empty-image
end
```

2. Rewrite the examples so that the first, rest structure is easy to

3. Use examples to figure out the cases!

```
48  small-maze =
49    [list: [list: "x", "x", "o", "x"],
50           [list: "x", "o", "o", "x"],
51           [list: "x", "o", "x", "x"],
52           [list: "x", "o", "x", "x"],
53           ]
54
55  ml0 = small-maze.get(0)
56  ml1 = small-maze.get(1)
57  ml2 = small-maze.get(2)
58  ml3 = small-maze.get(3)
59
```

1. Write examples for each suffix of the list (Hint: we can use `above` like we did with `beside`)

```
small-maze = [list: ml0, ml1, ml2, ml3]
```

2. Rewrite the examples so that the first, rest structure is easy to see

```
examples:
draw-table([list: ml0, ml1, ml2, ml3]) is above(draw-row(ml0),
                                                draw-table([list: ml1, ml2, ml3]))
draw-table([list: ml1, ml2, ml3]) is above(draw-row(ml1), draw-table([list: ml2, ml3]))
draw-table([list: ml2, ml3]) is above(draw-row(ml2), draw-table([list: ml3]))
draw-table([list: ml3]) is above(draw-row(ml3), draw-table([list: ]))
draw-table([list: ]) is empty-image
end
```

3. Use examples to figure out the cases!
   empty => empty-image, above(draw-row(first), draw-table(rest))

Put this on one of your handouts and make sure I can find it!

What did we learn? Processing Lists on our own (11/20)

- Used recursive code and processed List<List<String>>
- Built grids for our games or other functions
- Learned how to create a maze using draw-row and draw-table
- Used draw-cell, draw-row, draw-table. Drew boxes, put them beside each other, then stacked them on top of each other
- Created a table of a different data type
- Can be used for our games for screens of our devices
- Used the ideas of circles of eval to stack and do our examples
- Recursive code
- Recursion and created an image from a table.
- Having empty data types
- Learned how to count to 14 and color squares

Mr. Wolf:

- A recursive function calls itself on data repeatedly until it gets to an endpoint, after which it begins to compute the results of those function calls
- On a list, the recursive function calls itself until we get to the empty list.
- When we get to the end, we need some specific value (doesn't have to be an empty value) so that we can go back and compute the rest of the steps

First: make the contract and purpose statement, then follow these steps...

Strategy: Developing Functions Over Lists

Leverage the structure of lists and the power of concrete examples to develop list-processing functions.

- Pick a concrete list with (at least) three elements. Write a sequence of examples for each of the entire list and each suffix of the list (including the empty list).
- Rewrite each example to express its expected answer in terms of the `first` and `rest` data of its input list. You don't have to use the `first` and `rest` operators in the new answers, but you should see the `first` and `rest` *values* represented explicitly in the answer.
- Look for a pattern across the answers in the examples. Use these to develop the code: write a **`cases`** expression, filling in the right side of each `=>` based on your examples.

This strategy applies to structured data in general, leveraging components of each datum rather than specifically `first` and `rest` as presented so far.

Stuff that is due tomorrow (at the end of class):

Design recipe handout: draw-cell
Design recipe handout: draw-row - include examples
Design recipe handout: draw-row - include examples

Nested Flowchart - draw-row
Nested Flowchart - draw table

Tomorrow we are FINALLY getting back to our games, but you can work on your handouts with any extra time.