Recursion (continued)

Recursion (continued)
Recursion (continued)
Recursion (continued)
Recurs
(continu

CMPU 101 – Problem Solving and Abstraction

Peter Lemieszewski


SHELDON COOPER
presents
FUN WITH FLAGS
and lists!

Recursion and flags revisited

- Flags that are just stripes can be represented as lists of colors, e.g.,
  - *austria* = [list: "red", "white", "red"]
  - *germany* = [list: "black", "red", "yellow"]
  - *yemen* = [list: "red", "white", "black"]

Recursive striped-flag

```
fun striped-flag(colors :: List<String>) -> Image:
  doc: "Produce a flag with horizontal stripes"
  cases (List) colors:
    | empty => empty-image
    | link(color, rest) =>
      stripe = rectangle(120, 30, "solid", color)
      above(stripe, striped-flag(rest))
  end
end
```

››› *countries* = [list: austria, germany, yemen]

››› map(striped-flag, countries)

[list: , ,]

A complication

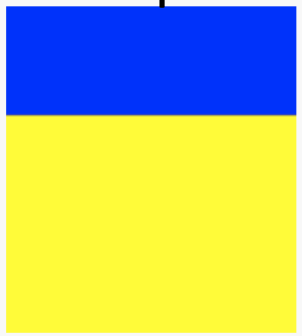
- What if we have a different number of stripes?
- Consider Ukraine:

```
››› ukraine = [list: "blue", "yellow"]
››› striped-flag(ukraine)
```

- Wrong dimensions!

```
FLAG-WIDTH = 120
FLAG-HEIGHT = 90

fun striped-flag(colors :: List<String>) -> Image:
  doc: "Produce a flag with horizontal stripes"

  cases (List) colors:
    | empty => empty-image
    | link(color, rest) =>
      height = FLAG-HEIGHT / length(colors)
      stripe = rectangle(FLAG-WIDTH, height, "solid", color)
      above(stripe, striped-flag(rest))
  end
end
```

```
››› ukraine = [list: "blue", "yellow"]
››› striped-flag(ukraine)
```




```
››› germany = [list: "black", "red", "yellow"]
››› striped-flag(germany)
```

```
FLAG-WIDTH = 120
FLAG-HEIGHT = 90

fun striped-flag(colors :: List<String>) -> Image:
  doc: "Produce a flag with horizontal stripes"

  cases (List) colors:
    | empty => empty-image
    | link(color, rest) =>
      height = FLAG-HEIGHT / length(colors)
      stripe = rectangle(FLAG-WIDTH, height, "solid", color)
      above(stripe, striped-flag(rest))
  end
end
```

What's wrong with this code?

Going further

Alternating elements

Alternating Elements

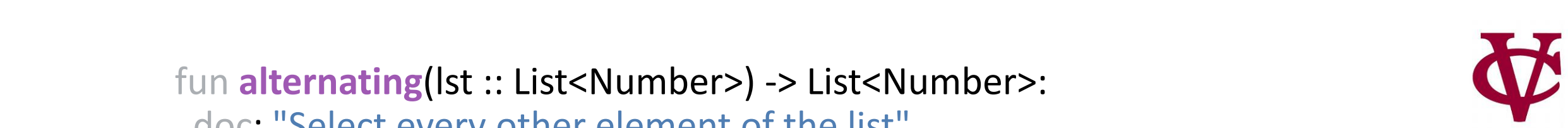
- This should serve as a “demonstration”

- What if we want to select every other element of a list?
  - ››› **alternating([list: "a", "b", "c", "d"])**
  - [list: "a", "c"]

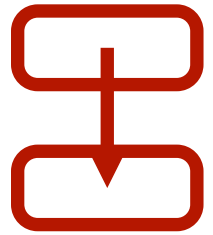
- Usually when we want to get just some of the elements of a list, we use **filter**, but it's hard to think how we could do that for this problem.
- In this case, it's easier to use explicit recursion – though we'll see there's an interesting difference from the recursive functions we've written so far.

```
fun alternating(lst :: List<Number>) ->
List<Number>:
  doc: "Select every other element of the list"
  #not without tests (first)...




where:
  ...



end
```

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  ...
```



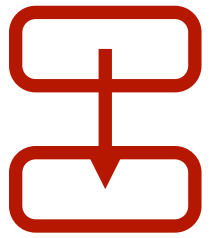
```
where: #easy to see what we want here…
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```



```
fun alternating(lst :: List<Number>) ->
List<Number>:
  doc: "Select every other element of the list"
  ...
```

The result doesn't depend on the next smallest case – it depends on the one after that!

```
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```



```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => ...
    | link(f, r) => ...

  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) => ...

  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty => ...

        | link(fr, rr) => ...

      end
  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) => ...

      end
  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

In this case, the list has an odd number of elements!

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) => ...

      end
  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

fr = first of the rest. Skip this!

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) => ...

      end
  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

Need to check that rest of the list is not empty here…

rr = rest of the rest. This is where we keep going!

```
fun alternating(lst :: List<Number>) -> List<Number>:
  doc: "Select every other element of the list"
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
where:
  alternating([list: 1, 2, 3, 4, 5, 6]) is [list: 1, 3, 5]
  alternating([list:    2, 3, 4, 5, 6]) is [list: 2, 4, 6]
  alternating([list:       3, 4, 5, 6]) is [list:    3, 5]
  alternating([list:          4, 5, 6]) is [list:    4, 6]
end
```

```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

- alternating([list: 1, 2, 3, 4, 5])

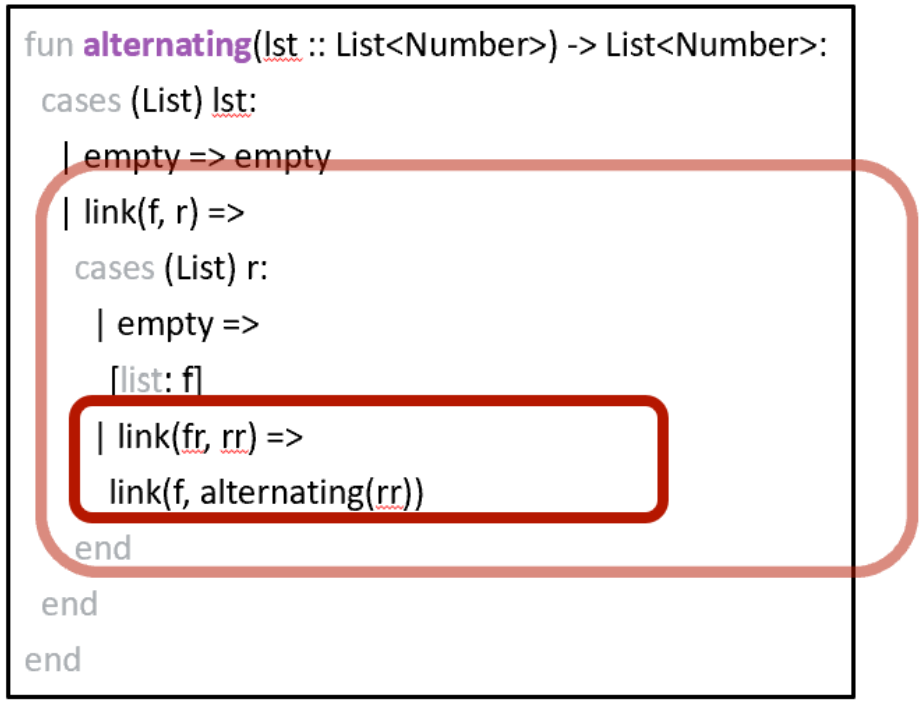
-
-
-
-
-
-
-

```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

- alternating([list: 1, 2, 3, 4, 5])

f *r* *fr* *rr*

In case previous slide doesn't look quite righet



```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

- alternating([list: 1, 2, 3, 4, 5])
  - *f*: 1
  - *r*: 2, 3, 4, 5
  - *fr*: 2
  - *rr*: 3, 4, 5
-
-
-
-
-
-
-

```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

alternating([list: 1, 2, 3, 4, 5])

→ link(1, alternating([list: 3, 4, 5]))

```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

alternating([list: 1, 2, 3, 4, 5])

→ link(1, alternating([list: 3, 4, 5]))

→ link(1, link(3, alternating([list: 5])))

```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

alternating([list: 1, 2, 3, 4, 5])

→ link(1, alternating([list: 3, 4, 5]))

→ link(1, link(3, alternating([list: 5])))

→ link(1, link(3, [list: 5]))

How the list develops:

```
fun alternating(lst :: List<Number>) -> List<Number>:
  cases (List) lst:
    | empty => empty
    | link(f, r) =>
      cases (List) r:
        | empty =>
          [list: f]
        | link(fr, rr) =>
          link(f, alternating(rr))
      end
  end
end
```

alternating([list: 1, 2, 3, 4, 5])

→ link(1,
alternating([list: 3, 4, 5]))

→ link(1,
link(3, alternating([list: 5])))

→ link(1,
link(3, [list: 5]))

→ [list: 1, 3, 5]

Max: Largest element in a list

- What if we want the biggest number in a list?

```
››› max([list: -10, 0, 8, 4])

8
```

Max: Largest element in a list

- This function is provided by Pyret:

```
››› import math as M
››› M.max([list: -10, 0, 8, 4])
8
```

- But let’s try writing it ourselves!

```
fun max(lst :: List<Number>) -> Number:
  doc: "Return the max number in the list"
  cases (List) lst:
    | empty => raise("The list is empty")
    | link(f, r) =>
      cases (List) r:
        | empty => f
        | else  => num-max(f, max(r))
      end
  end
where:
  max([list: 3, 2, 1]) is 3
  max([list: 3, 1, 2]) is 3
  max([list: 1, 3, 2]) is 3
  max([list: 1, 2, 3]) is 3
end
```

Who wore it better? (-a)

```
fun sum-of-squares-a(lst :: List<Number>) -> Number:
  doc: “Recursively add up the square of each number in the list"
  cases (List) lst:
    | empty => 0
    | link(f, r) =>
     (f * f) + sum-of-squares-a(r)
  end
where:
  sum-of-squares-a([list: ]) is 0
  sum-of-squares-a([list: 1, 2]) is 5
end
```

Who wore it better? (-b)

```
#Use the math library, specifically sum()
import math as M
fun sum-of-squares-b(lst :: List<Number>) -> Number:
  doc: "Add up the square of each number in the list, NOT recursively!"
  M.sum(map(lam(x): x * x end, lst))
where:
  sum-of-squares-b([list: ]) is 0
  sum-of-squares-b([list: 1, 2]) is 5
end
```

Who wore it better?

- Lists are structurally recursive data
- Every function that uses a list as a parameter need not (!) be recursive!

Case in point: computing average

```
fun avg(lst :: List<Number>) -> Number:
  doc: "Compute the average of the numbers in lst"
  ...
where:
  avg([list: 1, 2, 3, 4]) is 10/4
  avg([list:    2, 3, 4]) is  9/3
  avg([list:       3, 4]) is  7/2
  avg([list:          4]) is  4/1
end
```

Case in point: computing average, just in case you need this functionality for some reason

```
fun avg(lst :: List<Number>) -> Number:
  doc: "Compute the average of the numbers in lst"
   M.sum(lst) / length(lst)
where:
  avg([list: 1, 2, 3, 4]) is 10/4
  avg([list:    2, 3, 4]) is  9/3
  avg([list:       3, 4]) is  7/2
  avg([list:          4]) is  4/1
end
```

Meanwhile, back in Ukraine…

Building a better striped-flag

```
FLAG-WIDTH = 120
FLAG-HEIGHT = 90

fun striped-flag(colors :: List<String>) -> Image:
  doc: "Produce a flag with horizontal stripes"

  cases (List) colors:
    | empty => empty-image
    | link(color, rest) =>
      height = FLAG-HEIGHT / length(colors)
      stripe = rectangle(FLAG-WIDTH, height, "solid", color)
      above(stripe, striped-flag(rest))
  end
end
```

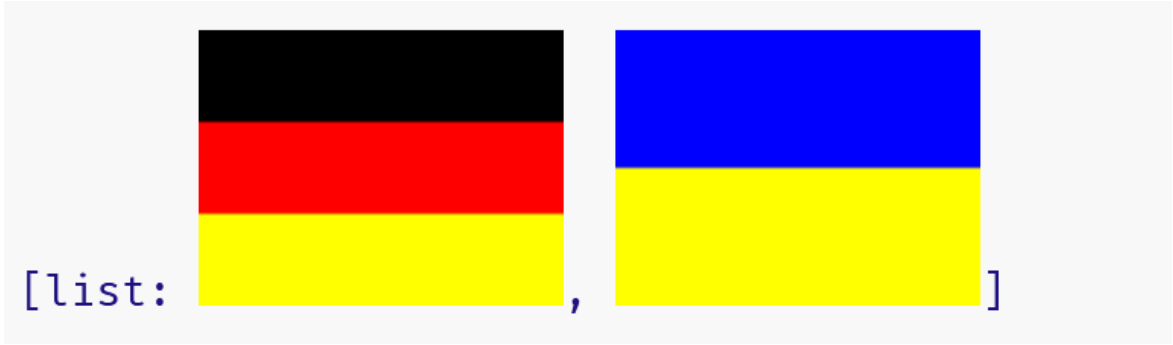
This is like computing the average!

Building a better striped-flag:

```
FLAG-WIDTH = 120
FLAG-HEIGHT = 90
fun striped-flag(colors :: List<String>) -> Image:
  doc: "Produce a flag with horizontal stripes"

  height = FLAG-HEIGHT / length(colors) # non-recursive calculation

  fun stripe-helper(lst :: List<String>) -> Image:
    cases (List) colors:
      | empty => empty-image
      | link(color, rest) =>
        stripe = rectangle(FLAG-WIDTH, height, "solid", color)
        above(stripe, stripe-helper(rest))
    end
  end
  stripe-helper(colors) # simply call the stripe-helper!
end
```

Output of better striped-flag:

- map(striped-flag, [list: germany, ukraine])
- #list of images built with proper proportions



Link to code

- 13_flags-ukraine.arr

Acknowledgements

- This lecture incorporates material from:
- Kathi Fisler, Brown University,
- Marc Smith, Vassar College
- And, Jonathan Gordon, Vassar College