Recursion (continued)

Recursion (continued)

Recursion (continued)

Recursion (continued)

Recursion (continued)

Recursion (continued)

Recursion (continued)

Recursion (continued)

Recursion (continued)
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"]`
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
```python
>>> countries = ["austria", "germany", "yemen"]

>>> map(stripped-flag, countries)
[[list: ], [ ], [ ]]
A complication
• What if we have a different number of stripes?
• Consider Ukraine:
  >>> ukraine = [list: "blue", "yellow"]
  >>> striped-flag(ukraine)

  ![Flag of Ukraine](image)

• Wrong dimensions!
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
```python
>>> ukraine = ["blue", "yellow"]
>>> striped_flag(ukraine)

---

>>> germany = ["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
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?
  •三级 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"

  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

The result doesn’t depend on the next smallest case – it depends on the one after that!
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) => ...

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]
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]
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]
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
fun alternating(List :: 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
In case previous slide doesn’t look quite right.

fun alternating((List :: 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])

→ 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
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
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]
What if we want the biggest number in a list?

```python
>>> max([-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
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
#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 $\text{avg}$($\text{lst}$ :: 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
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

Case in point: computing average, just in case you need this functionality for some reason
Meanwhile, back in Ukraine...
Building a better striped-flag

\[\text{FLAG-WIDTH} = 120\]
\[\text{FLAG-HEIGHT} = 90\]

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

  cases (List) colors:
  | empty => empty-image
  | link(color, rest) =>
    \textit{height} = FLAG-HEIGHT / length(colors)
    \textit{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:

\[ \text{FLAG-WIDTH} = 120 \]
\[ \text{FLAG-HEIGHT} = 90 \]

fun \text{striped-flag}(colors :: List<String>) -> Image:

doc: "Produce a flag with horizontal stripes"

\[ \text{height} = \text{FLAG-HEIGHT} / \text{length(colors)} \] # non-recursive calculation

fun \text{stripe-helper}(lst :: List<String>) -> Image:

cases (List) colors:
| empty => empty-image
| link(color, rest) =>
    \[ \text{stripe} = \text{rectangle}(\text{FLAG-WIDTH}, \text{height}, \"solid\", \text{color}) \]
    above(stripe, stripe-helper(rest))
end
end

\text{stripe-helper(colors)} # simply call the \text{stripe-helper!}
end
Output of better striped-flag:

- `map(stripped-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