Further Recursion

2 March 2022
Warm up
Flags that are just stripes can be represented as lists of colors, e.g.,

```python
austria = ["red", "white", "red"]
germany = ["black", "red", "yellow"]
yemen = ["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
>>> countries = [list: austria, germany, yemen]

>>> L.map(stripped-flag, countries)

[ list: , , ]
A complication
What if we have a different number of stripes?

Consider Ukraine:

```python
>>> ukraine = ["blue", "yellow"]
>>> striped_flag(ukraine)
```

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 / L.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)
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 / L.length(colors)
            stripe = rectangle(FLAG-WIDTH, height, "solid", color)
            above(stripe, striped-flag(rest))
    end
end

What’s wrong with this code?
We'll fix this – but not yet!

First, we’re going deeper into how we can design functions using lists and recursion.
Going further
Alternating elements
What if we want to select every other element of a list?

```python
>>> 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 \texttt{L.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"
    ...

where:
    ...

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
fun \texttt{alternating}(\texttt{lst} :: \texttt{List<Number>}) \rightarrow \texttt{List<Number>}:
  \texttt{doc}: "Select every other element of the list"
...

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

\texttt{where:}
  \texttt{alternating}([\texttt{list: 1, 2, 3, 4, 5, 6}]) \texttt{is} [\texttt{list: 1, 3, 5}]
  \texttt{alternating}([\texttt{list: 2, 3, 4, 5, 6}]) \texttt{is} [\texttt{list: 2, 4, 6}]
  \texttt{alternating}([\texttt{list: 3, 4, 5, 6}]) \texttt{is} [\texttt{list: 3, 5}]
  \texttt{alternating}([\texttt{list: 4, 5, 6}]) \texttt{is} [\texttt{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
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
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
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

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]
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])
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
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 ]))
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

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
What if we want the biggest number in a list?

```python
>>> max([-10, 0, 8, 4])
8
```
This function is provided by Pyret:

```python
>>> import math as M
>>> M.max([-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: 1, 2, 3]) is 3
    max([list: 3, 1, 2]) is 3
    max([list: 1, 3, 2]) is 3
    max([list: 1, 2, 3]) is 3
Recursion is all you need?
fun sum-of-squares(lst :: List<Number>) -> Number:
  doc: "Add up the square of each number in the list"
  cases (List) lst:
    | empty => 0
    | link(f, r) =>
      (f * f) + sum-of-squares(r)
  end
where:
  sum-of-squares([list: ]) is 0
  sum-of-squares([list: 1, 2]) is 5
end

fun sum-of-squares(lst :: List<Number>) -> Number:
  doc: "Add up the square of each number in the list"
  M.sum(L.map(lam(x): x * x end, lst))
where:
  sum-of-squares([list: ]) is 0
  sum-of-squares([list: 1, 2]) is 5
end
Just because lists are structurally recursive data doesn’t mean every function that takes a list as input will employ recursion.
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
fun avg(lst :: List<Number>) -> Number:
    doc: "Compute the average of the numbers in lst"
    M.sum(lst) / L.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
Resolution
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 / L.length(colors)
    | `stripe` = rectangle(FLAG-WIDTH, height, "solid", color)
    above(stripe, striped-flag(rest))
  end
end

*This is like the denominator for computing the average!*
fun striped-flag(colors :: List<String>) -> Image:
    doc: "Produce a flag with horizontal stripes"
    
    height = FLAG-HEIGHT / L.length(colors)
    
    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)
end

FLAG-WIDTH = 120
FLAG-HEIGHT = 90
L.map(striped-flag, [list: germany, ukraine])

[ list:  , ]
Code from class:

https://code.pyret.org/editor#share=1pi1Tsxn\MLI52NYL5Y50Cx0dKsMMvTtW&v=1904b2c
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Marc Smith, Vassar College