Further Recursion

20 February 2023
THE 8TH ANNUAL

DATAFEST 2023

@ VASSAR

RETURNS FULLY IN-PERSON!

FRIDAY MAR 31 - SUNDAY APR 2, 2023

5PM FRIDAY TO 5PM SUNDAY

Your challenge for the weekend: ask questions and draw insights from the data. But the data will remain a secret until Friday's opening ceremony! To participate, you only need an enthusiasm for data and friendly competition. Prizes for the winners: food and swag for everyone. Registration is FREE but limited to the first 80 students. Register by Friday, March 3, 2023 (before Spring Break).

To register and for more information, visit https://pages.vassar.edu/datafest/ or scan the QR code:
Where are we?
Recursive definition of a list
The list is: [list: "A", "A", "C", "B"]
first  rest

```python
list: ['A', 'C', 'B']
```
list: ["C", "B"]
\textit{first} \quad \textit{rest}

\texttt{[list: "B"]} \quad ?
data \textbf{List}:
  | empty
  | \textbf{link}(first :: Any, rest :: List)
end
[list: "A", "A", "C", "B"]

link("A",
    link("A",
        link("C",
            link("B",
                empty)))))
data List:
    | empty
    | link(first :: Any, rest :: List)
end
fun list-fun(lst :: List) -> ...

doc: "Template for a fn that takes a List"
cases (List) lst:
  | empty => ...
  | link(f, r) =>
  ... f ...
  ... list-fun(r) ...
end

where:
  list-fun(...) is ...
end

Structural (natural) recursion:
the shape of the function follows
the shape of the data
Recursive definition of a binary tree
I said yes!
data RumorMill:
  | no-one
  | gossip(name :: String,
           next1 :: RumorMill,
           next2 :: RumorMill)
end
gossip("Emma",
gossip("Mr Woodhouse", no-one, no-one),
gossip("Mrs Weston",
gossip("Mr Weston",
gossip("Jane", no-one, no-one),
gossip("Miss Bates",
gossip("Mrs Cole", no-one, no-one),
gossip("Mrs Elton", no-one, no-one)),
no-one)))
data RumorMill:
    | no-one
    | gossip(name :: String,
               next1 :: RumorMill,
               next2 :: RumorMill)
end
fun rumor-mill-fun(rm :: RumorMill) -> ...

doc: "Template for a fn that takes a RumorMill"

cases (RumorMill) rm:
    | no-one => ...
    | gossip(name, next1, next2) =>
        ... name
        ... rumor-mill-fun(next1)
        ... rumor-mill-fun(next2)
end
end

Natural recursion × 2
Warm up
Design the function *gossip-length* that takes a rumor mill and determines the length of the longest sequence of people transmitting the rumor.
A more realistic rumor mill
In our rumor mill, we restricted each person to spread gossip to at most two other people.

This isn’t very realistic; some gossips talk to lots of people!
Let each gossip talk to any number of people:

- Emma talks to Mrs Weston
- Mrs Weston talks to Mr Woodhouse
- Mr Woodhouse talks to Mrs Cole
- Mrs Weston talks to Jane
- Jane talks to Mrs Perry
- Miss Bates talks to Mrs Elton

Names:
- Emma
- Mrs Weston
- Mr Woodhouse
- Mr Weston
- Miss Bates
- Mrs Cole
- Mrs Elton
- Mrs Perry
How do we represent an arbitrary number of gossip connections?
How do we represent an arbitrary number of gossip connections?

This is really two mutually recursive data definitions:

```haskell
data Gossip:
    | gossip(name :: String, next :: List<Gossip>)
end
```
How do we represent an arbitrary number of gossip connections?

```plaintext
data Gossip:
  | gossip(name :: String, next :: List<Gossip>)
end

data List<Gossip>:
  | empty
  | link(first :: Gossip, rest :: List<Gossip>)
end
```

This is really two mutually recursive data definitions:
data Gossip:
    | gossip(name :: String, next :: List<Gossip>)
end

#|
fun gossip-template(g :: Gossip) -> ...
    ... g.name
    ... log-template(g.next)
end

fun log-template(l :: List<Gossip>) -> ...
    cases (List) l:
        | empty => ...
        | link(f, r) =>
            ... gossip-template(f)
            ... log-template(r)
end
end

|#

Mutually recursive functions: (they call each other)
Starter file:

tinyurl.com/101-2023-02-20-starter
Design **count-gossips** which takes a gossip and returns the total number of people informed by the gossip (including the starting person).
Solutions:

tinyurl.com/101-2023-02-20
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(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 you need to design a recursive function.
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) / 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
Sheldon Cooper presents... fun with flags... and lists!
Flags that are just stripes can be represented as lists of colors, e.g.,

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

>>> 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 / length(colors)
        stripe = rectangle(FLAG-WIDTH, height, "solid", color)
        above(stripe, striped-flag(rest))
    end
end
>>> ukraine = [list: "blue", "yellow"]
>>> striped_flag(ukraine)

[Image of Ukraine flag]

>>> germany = [list: "black", "red", "yellow"]
>>> striped_flag(germany)

[Image of Germany flag]
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?
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 / 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!
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)

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)
```python
>>> map(stripped_flag, [list: germany, ukraine])

[ list: Germany, Ukraine ]
```