Further Recursion

12 October 2022
Warm up
Exercise

Define a *recursive* function `add-prev` that goes through a list of numbers and adds the previous number to the current one, e.g.,

```python
>>> add-prev([list: 1, 2, 3])
[list: 1, 3, 5]
```
fun add-prev(lst :: List<Number>) -> List<Number>:
    doc: "Make a list where each number is the number in the original list plus the previous number in the original list"

fun add-prev-helper(l :: List<Number>, prev :: Number) -> List<Number>:
    doc: "Make a list where prev is added to the first number, which is then added to the next number, and so on"
    cases (List) l:
        | empty => empty
        | link(f, r) => link(f + prev, add-prev-helper(r, f))
    end
end

add-prev-helper(lst, 0)

where:
    add-prev([list: ]) is [list: ]
    add-prev([list: 1]) is [list: 1]
    add-prev([list: 1, 2]) is [list: 1, 3]
    add-prev([list: 1, 2, 3]) is [list: 1, 3, 5]
fun add-prev(lst :: List<Number>) -> List<Number>:
  doc: "Make a list where each number is the number in the original list plus the previous number in the original list"

  fun add-prev-helper(l :: List<Number>, prev :: Number) -> List<Number>:
    doc: "Make a list where prev is added to the first number, which is then added to the next number, and so on"
    cases (List) l:
      | empty => empty
      | link(f, r) => link(f + prev, add-prev-helper(r, f))
    end
  end

  add-prev-helper(lst, 0)

where:
  add-prev([list: ]) is [list: ]
  add-prev([list: 1]) is [list: 1]
  add-prev([list: 1, 2]) is [list: 1, 3]
  add-prev([list: 1, 2, 3]) is [list: 1, 3, 5]
end

This is a wrapper function. It only exists to pass the starting “previous number” value to the function that does the work.
fun add-prev(lst :: List<Number>) -> List<Number>:
    doc: "Make a list where each number is the number in the original list plus the previous number in the original list"

fun add-prev-helper(l :: List<Number>, prev :: Number) -> List<Number>:
    doc: "Make a list where prev is added to the first number, which is then added to the next number, and so on"
    cases (List) l:
        | empty => empty
        | link(f, r) => link(f + prev, add-prev-helper(r, f))
    end
end

add-prev-helper(lst, 0)

where:
add-prev([list: ]) is [list: ]
add-prev([list: 1]) is [list: 1]
add-prev([list: 1, 2]) is [list: 1, 3]
add-prev([list: 1, 2, 3]) is [list: 1, 3, 5]
end

This helper function recursively goes through the list and it has a second argument, which is whatever the last number it saw was.
fun add-prev(lst :: List<Number>) -> List<Number>:
  doc: "Make a list where each number is the number in the original list plus the previous number in the original list"

fun add-prev-helper(l :: List<Number>, prev :: Number) -> List<Number>:
  doc: "Make a list where prev is added to the first number, which is then added to the next number, and so on"
cases (List) l:
  | empty => empty
  | link(f, r) => link(f + prev, add-prev-helper(r, f))
end
end

add-prev-helper(lst, 0)

where:
  add-prev([list: ]) is [list: ]
  add-prev([list: 1]) is [list: 1]
  add-prev([list: 1, 2]) is [list: 1, 3]
  add-prev([list: 1, 2, 3]) is [list: 1, 3, 5]
fun add-prev(lst :: List<Number>) -> List<Number>:
  doc: "Make a list where each number is the number in the original list plus
the previous number in the original list"
  add-prev-helper(lst, 0)
where:
  add-prev([list: ]) is [list: ]
  add-prev([list: 1]) is [list: 1]
  add-prev([list: 1, 2]) is [list: 1, 3]
  add-prev([list: 1, 2, 3]) is [list: 1, 3, 5]
end

fun add-prev-helper(l :: List<Number>, prev :: Number) -> List<Number>:
  doc: "Make a list where prev is added to the first number, which is then
added to the next number, and so on"
cases (List) l:
  | empty => empty
  | link(f, r) => link(f + prev, add-prev-helper(r, f))
end
end

The helper function doesn’t need to be nested, though it can be nice to “bundle” things that way.
...and lists!
Flags that are just stripes can be represented as lists of colors, e.g.,

```python
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 = ['austria', 'germany', 'yemen']

>>> map(stripped_flag, countries)

[flag, flag, flag]
```
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 = ["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

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
g>>> 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"
    ...

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 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]
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

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

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) =>
                    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

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

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

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([list: -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: -10, 0, 4]) is 4
  max([list: ]) raises "The list is empty"
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 every function that takes a list as input will employ explicit 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) / 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 / 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 / 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
```python
>>> map(striped_flag, [list: germany, ukraine])

[list: Germany, Ukraine]
```
Code from class:

https://tinyurl.com/101-2022-10-12
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Marc Smith, Vassar College