Further Recursion

2 October 2023
Quiz 1

Quizzes are graded on Gradescope.

We want you to learn from how you did on the quiz before we get to the first exam, so you can correct your answers and submit them for partial credit.
Problem 1, Part b

if 2.1 > running-version:
    string-append(
        "New version available:",
        num-to-string(latest-version))
else:
    "Software is up-to-date."
end

Identify the problem you’re correcting and then include the code below. If it’s not a function (e.g., Problem 1), put it in a comment.
Template you can use for your corrections:
tinyurl.com/101-q1-corr
Exam 1

Friday, in lab!

Preparing:

We’ll give you a practice exam and go over problems during a review session outside of class. We’ll share example solutions afterward.

You should prepare one (double-sided) sheet of handwritten notes, which you can refer to during the exam.
Academic integrity

Labs

- Collaboration allowed!
- Coaches and I are here to help!

Assignments (including quiz corrections)

- No collaboration!
- Get help:
  - Coaches during coaching hours
  - Me during office hours or by email

Quizzes and exam

- No collaboration!
- No outside resources!
Where are we?
Recursive definition of a list
[list: "A", "A", "C", "B"]
first

list: "A", "A", "C", "B"]
first

rest

list: ['A', 'A', 'C', 'B']
[list: "A", "C", "B"]
first

["A", "C", "B"]
first: "A"

rest: "C", "B"
[list: "C", "B"]
first

`[list: "C", "B"]`
First rest

\[\text{list:} \quad "C", \quad "B"\]
[list: "B"]
first

[ list: "B" ]
list: ['B']

first rest

?
data List:
  | empty
  | 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

*Self-reference*
fun list-fun(lst :: List) -> ...

doc: "Template for a function that takes a List"

cases (List) lst:
  | empty => ...
  | link(f, r) =>
  |   ... f ...
  |   ... list-fun(r) ...
end

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

Self-reference × 2
fun rumor-mill-fun(rm :: RumorMill) -> ...

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

cases (RumorMill) rm:
| no-one => ...
| gossip(name, next1, next2) =>
  ... name
  ... rumor-mill-fun(next1)
  ... rumor-mill-fun(next2)
end
end
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!
But we’re a bit tired of gossip.

Let’s take that idea – a tree where each node can have an arbitrary number of branches – and see how it plays out for a different kind of data.
Files and directories
# A
Pittsburgh
road is
...

hugo

snacks

4 8 15 16
23 42

kate

puppy.jpg

map.pdf

escape.arr
A file system is a tree structure where each directory can have an arbitrary number of children.

Both a file system and a more realistic model of a rumor mill are kinds of *n-ary trees*. 
A file has a name and contents, such as code or an image.

# A Pittsburgh road is...

```
4 8 15 16
23 42
```
data File:
  | file(name :: String, size :: Number)
end

For simplicity, we’ll ignore the actual file contents and just keep track of its size – how much disk space it takes up.
A directory has a name and zero or more files or directories that it contains.
data `File`:
   | file(name :: String, size :: Number)
end

data `Directory`:
   | dir(name :: String, contents :: ...)
end
data File:
  | file(name :: String, size :: Number)
end

data Directory:
  | dir(name :: String, contents :: List)
end
data File:
  | file(name :: String, size :: Number)
end

data Directory:
  | dir(name :: String, contents :: List<File>)
end
data File:
  | file(name :: String, size :: Number)
end

data Directory:
  | dir(name :: String, contents :: List<File>)
end

This is a specialization of the general definition of a list.
data **File**:  
  | file(name :: String, size :: Number)  
end

data **Directory**:  
  | dir(name :: String, contents :: List<FileOrDirectory>)  
end
data FileOrDirectory:
  | file(name :: String, size :: Number)
  | dir(name :: String, contents :: List<FileOrDirectory>)
end
data FileOrDirectory:
    | file(name :: String, size :: Number)
    | dir(name :: String, contents :: List<FileOrDirectory>)
end

data List<FileOrDirectory>:
    | empty
    | link(first :: FileOrDirectory,
          rest :: List<FileOrDirectory>)
end
data FileOrDirectory:
  | file(name :: String, size :: Number)
  | dir(name :: String, contents :: List<FileOrDirectory>)
end

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

You don’t need to give this second definition to Pyret; it already knows how to make a List<X> for any X.
data FileOrDirectory:
    | file(name :: String, size :: Number)
    | dir(name :: String, contents :: List<FileOrDirectory>)
end

data List<FileOrDirectory>:
    | empty
    | link(first :: FileOrDirectory, rest :: List<FileOrDirectory>)
end
data **FileOrDirectory**:
   | file(name :: String, size :: Number)
   | dir(name :: String, contents :: List<FileOrDirectory>)
end

data **List<FileOrDirectory>**:
   | empty
   | link(first :: FileOrDirectory,
       rest :: List<FileOrDirectory>)
end
data FileOrDirectory:
    | file(name :: String, size :: Number)
    | dir(name :: String, contents :: List<FileOrDirectory>)
end

data List<FileOrDirectory>:
    | empty
    | link(first :: FileOrDirectory, rest :: List<FileOrDirectory>)
end
data `FileOrDirectory`:
   | file(name :: String, size :: Number)
   | dir(name :: String, contents :: List<FileOrDirectory>)
end

data List<FileOrDirectory>:
   | empty
   | link(first :: FileOrDirectory,
          rest :: List<FileOrDirectory>)
end
fun fod-fun(fod :: FileOrDirectory) -> ...:
cases (FileOrDirectory) fod:
  | file(name, size) => ... name ... size
  | dir(name, contents) =>
    ... name
    ... lofd-fun(contents)
end
end

fun lofd-fun(lofd :: List<FileOrDirectory>) -> ...:
cases (List) lofd:
  | empty => ...
  | link(f, r) =>
    ... fod-fun(f)
    ... lofd-fun(r)
end
end

 Mutually recursive functions – they call each other!
Starter file:

tinyurl.com/101-2023-10-02-starter
Design a function that goes through a directory hierarchy looking for a specified name.
Design a function to compute how much space is used on disk.
Solutions:

tinyurl.com/101-2023-10-02
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"
    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 always need to design a recursive function to work with one.
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
include math

fun avg(lst :: List<Number>) -> Number:
  doc: "Compute the average of the numbers in lst"
  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.,

\[
\text{austria} = [\text{list: } \text{"red"}, \text{"white"}, \text{"red"}]
\]
\[
\text{germany} = [\text{list: } \text{"black"}, \text{"red"}, \text{"yellow"}]
\]
\[
\text{yemen} = [\text{list: } \text{"red"}, \text{"white"}, \text{"black"}]
\]
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)

["minster", "germany", "yemen"]
```
What if we have a different number of stripes?

Consider Ukraine:

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

Wrong dimensions!
fun stripped-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)
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?
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 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])

[Flag: Germany, Flag: Ukraine]
```
Code:

tinyurl.com/101-2023-10-02-further
Acknowledgments

This lecture incorporates material from:

Kathi Fisler, Brown University
Marc Smith, Vassar College