Trees

27 October 2021
Where are we?
Now we can see how lists are defined:

```plaintext
data MyList:
  | my-empty
  | my-link(first, rest :: MyList)
end
```

Self-reference
And just like we did for a List, we use this template to write a function that recursively processes the data:

```ocaml
fun my-list-fun(ml :: MyList) -> ...:
    doc: "Template for a fn that takes a MyList"
    cases (MyList) ml:
        | my-empty => ...
        | my-link(f, r) =>
            ... f ...
            ... my-list-fun(r) ...
    end
    where:
        my-list-fun(...) is ...
end
```
Every data definition has a corresponding template.

The more complex the data definition is – lots of variants, recursion, etc. – the more helpful it is to use the template!
Rumor mills
Ginny controls the rumor mill

You'd think people had better things to gossip about. Three Dementor attacks in a week, and all Romilda Vane does is ask me if it's true you've got a Hippogriff tattooed across your chest.

What did you tell her?

I told her it's a Hungarian Horntail. Much more macho.

Thanks. And what did you tell her Ron's got?

A Pygmy Puff, but I didn't say where.
Tracking rumors

Suppose we want to track gossip in a rumor mill.
Tracking rumors

Suppose we want to track gossip in a rumor mill.

Pansy
Tracking rumors

Suppose we want to track gossip in a rumor mill.

Pansy
Tracking rumors

Suppose we want to track gossip in a rumor mill.
Tracking rumors

Suppose we want to track gossip in a rumor mill.

- Pansy
- Draco
- Cho
- Romilda
- Vincent
Tracking rumors

Suppose we want to track gossip in a rumor mill.
Tracking rumors

Suppose we want to track gossip in a rumor mill.

Simplifying assumption: Each person tells at most two others.
Tracking rumors

Suppose we want to track gossip in a rumor mill.

Simplifying assumption:
Each person tells at most two others
If you ignore my silly Harry Potter example, this is a pretty serious problem.

A lot of research right now is focused on building models of how information – and misinformation! – spreads through social networks, both in person and online.
Representing rumor mills

Is a rumor mill simply a list of people?
Representing rumor mills

Is a rumor mill simply a list of people?
No, because there are relationships among the people.
Representing rumor mills

We could represent these relations with a table, e.g.,

<table>
<thead>
<tr>
<th>name :: String</th>
<th>next1 :: String</th>
<th>next2 :: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Pansy&quot;</td>
<td>&quot;Cho&quot;</td>
<td>&quot;Draco&quot;</td>
</tr>
<tr>
<td>&quot;Cho&quot;</td>
<td>&quot;Pansy&quot;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Representing rumor mills

Using a table doesn’t give us any straightforward way to process the rumor mill.

Could we use something like a list but representing the relations?
Representing rumor mills

data Person:
  | person(name :: String, next1 :: Person, next2 :: Person)
end

How about this?
Representing rumor mills

data Person:
  | person(name :: String, next1 :: Person, next2 :: Person)
end

Some people don’t gossip to anyone else – the red arrows above.
Representing rumor mills

data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

How about this?
Example rumor mills

data RumorMill:
    | no-one
    | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

no-one
Example rumor mills

data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

gossip("Ginny", no-one, no-one)
Example rumor mills

data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

gossip("Romilda",
  no-one,
  gossip("Ginny", no-one, no-one))
gossip("Pansy",
gossip("Cho", no-one, no-one)
gossip("Draco",
gossip("Romilda",
    no-one
    gossip("Ginny", no-one, no-one))
gossip("Vincent", no-one, no-one)))
Example using names for parts:

\[
\textit{GINNY-MILL} = \\
gossip("Ginny", \text{no-one}, \text{no-one})
\]

\[
\textit{ROMILDA-MILL} = \\
gossip("Romilda", \text{no-one}, \text{GINNY-MILL})
\]

\[
\textit{VINCENT-MILL} = \\
gossip("Vincent", \text{no-one}, \text{no-one})
\]

\[
\textit{DRACO-MILL} = \\
gossip("Draco", \text{ROMILDA-MILL}, \text{VINCENT-MILL})
\]

\[
\textit{CHO-MILL} = \\
gossip("Cho", \text{no-one}, \text{no-one})
\]

\[
\textit{PANSY-MILL} = \\
gossip("Pansy", \text{CHO-MILL}, \text{DRACO-MILL})
\]
A *RumorMill* is a type of structure called a *tree*.

Each element in the tree is called a *node*.

The first node in the tree is called the *root*.

A node with no children is called a *leaf*.

Like a list, a tree is recursive: Every subtree is a tree.
Programming with rumors

data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
Programming with rumors

data RumorMill:
  |   no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

Self-reference × 2
Programming with rumors

```plaintext
data RumorMill:
    | no-one
    | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

Self-reference $\times 2$

For each element, there's not just one "next" element; there are two!
Programming with rumors

data RumorMill:
   | no-one
   | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

#|
fun rumor-mill-template(rm :: RumorMill) -> ...:
   doc: "Template for a function with a RumorMill as input"
cases (RumorMill) rm:
   | no-one => ...
   | gossip(name, n1, n2) =>
      ... name
      ... rumor-mill-template(n1)
      ... rumor-mill-template(n2)
end
end
|

Self-reference × 2
Programming with rumors

data RumorMill:
    | no-one
    | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end

# |
fun rumor-mill-template(rm :: RumorMill) -> ...
    doc: "Template for a function with a RumorMill as input"
cases (RumorMill) rm:
    | no-one => ...
    | gossip(name, n1, n2) =>
        ... name
        ... rumor-mill-template(n1)
        ... rumor-mill-template(n2)
end
end

Self-reference × 2

Natural recursion × 2
Starter file:

https://code.pyret.org/editor#share=1f4C2A_I899bWYFtmQIz9N7d9egcENTtt&v=1904b2c
Rumor program examples

Design the function `is-informed` that takes a person’s name and a rumor mill and determines whether the person is part of the rumor mill.
Rumor program examples

Design the function \texttt{rumor-delay} that takes a rumor mill and determines the maximum number of days required for a rumor to reach everyone, assuming that each person waits a day before passing on a rumor.
Solutions:

https://code.pyret.org/editor#share=17JmkvWVeILmobvyXx5qLM_yU19_jnukl&v=1904b2c
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:

- Pansy
- Draco
- Cho
- Romilda
- Vincent
- Ginny
How do we represent an arbitrary number of gossip connections?
How do we represent an arbitrary number of gossip connections?

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

#|
fun gossip-template(g :: Gossip) -> ...:
    ... gossip.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
|#
Starter file:

https://code.pyret.org/editor#share=1LN9PY1sGUoFEFMPzT-WWSKRA1KS9cxlT&v=1904b2c
Design `count-gossips` which takes a gossip and returns the number of people informed by the gossip (including the starting person).
Solutions:

https://code.pyret.org/editor#share=1qbVc9iY-2oGRGAMN20gIJ6GWBaU1opTb?v=1904b2c
Acknowledgments

This lecture incorporates material from:

J. K. Rowling, *Harry Potter and the Half-Blood Prince*

Marc Smith, Vassar College