Trees

Joyce Kilmer - 1886-1918

I think that I shall never see
A poem lovely as a tree.

A tree whose hungry mouth is prest
Against the earth's sweet flowing breast;

A tree that looks at God all day,
And lifts her leafy arms to pray;

A tree that may in summer wear
A nest of robins in her hair;

Upon whose bosom snow has lain;
Who intimately lives with rain.

Poems are made by fools like me,
But only God can make a tree.

Joyce Kilmer was born on December 6, 1886, in New Brunswick, New Jersey. The author of Main Street and Other Poems (George H. Doran Company, 1917), he was killed while fighting in World War I.

Themes

nature
plants

About Joyce Kilmer >
Steps to write a generic template

• Given a (recursive) data definition, you write a generic template by:
  1. Creating a function header,
  2. Using cases to break the data input into its variants,
     • In each case, list each of the fields as part of the answer
  3. And, calling the function itself on any recursive fields.
Data Definition: Start With A Template

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

Self-reference Definition!
Debrief: lists and recursion

\[
data \text{MyList}: \\
| \text{my-empty} \\
| \text{my-link(first, rest :: MyList)} \\
end
\]

What's different here?

1. We have a case that's just a special keyword rather than a constructor.
2. Part of the second case" is of the same type we're defining.
   - A recursive definition!
Using my-list Data Template

We use this template to write a function that recursively processes the data:

```plaintext
fun my-fun(ml :: MyList) -> ...:
  doc: "Template for a function that takes a MyList"
  cases (MyList) ml:
    | my-empty => ...
    | my-link(f, r) =>
      ... f ...
      ... my-fun(r) ...
  end
where:
  my-fun(...) is ...
end
```
Tracking rumors

• Suppose we want to track gossip in a 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 Hippogrift 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 Ron’s got?

A Pygmy Puff, but I didn’t say where.

Ginny controls the rumor mill
Tracking rumors

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

• Suppose we want to track gossip in a rumor mill.
• Suppose we want to track gossip in a rumor mill.
• 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
• 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

**Question:** Is a rumor mill simply a list of people?

**Answer:** 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</th>
<th>next1</th>
<th>next2</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></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?
data Person:

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

Representing rumor mills

Some people don’t gossip to anyone else – see the red arrows above.
data RumorMill:
  | no-one
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
Example rumor mills

data RumorMill:
    | no-one #at the start there is... no-one in the rumor mill!
    | 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 the parts

\[
\begin{align*}
\text{GINNY-MILL} &= \text{gossip("Ginny", no-one, no-one)} \\
\text{ROMILDA-MILL} &= \text{gossip("Romilda", no-one, GINNY-MILL)} \\
\text{VINCENT-MILL} &= \text{gossip("Vincent", no-one, no-one)} \\
\text{DRACO-MILL} &= \text{gossip("Draco", ROMILDA-MILL, VINCENT-MILL)} \\
\text{CHO-MILL} &= \text{gossip("Cho", no-one, no-one)} \\
\text{PANSY-MILL} &= \text{gossip("Pansy", CHO-MILL, DRACO-MILL)}
\end{align*}
\]
Computer Science concepts wrung from a rumor 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

For each element, there’s not just one “next” element; there are two!
Rumor Mill Template

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
| #
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 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 Mill Examples

• Design the function \texttt{add-gossip} that takes a rumor mill and two names – one new and one old – and adds the new person to the rumor mill, receiving rumors from the old person.
  • (You can assume the “old person” does not already have two next persons!)
Realism

• 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 (and lots…) of people
• Let’s relax this restriction.
  • Let each gossip talk to any number of people:
Realism

- `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`
Realism

• With the more realistic template, we can...
• Design `count-gossips` which takes a gossip and returns the number of people informed by the gossip (including the starting person)
Link to code Template

• 14-new-data-types.arr
Acknowledgements

• This lecture incorporates material from:
• Kathi Fisler, Brown University,
• Marc Smith, Vassar College
• And, Jonathan Gordon, Vassar College