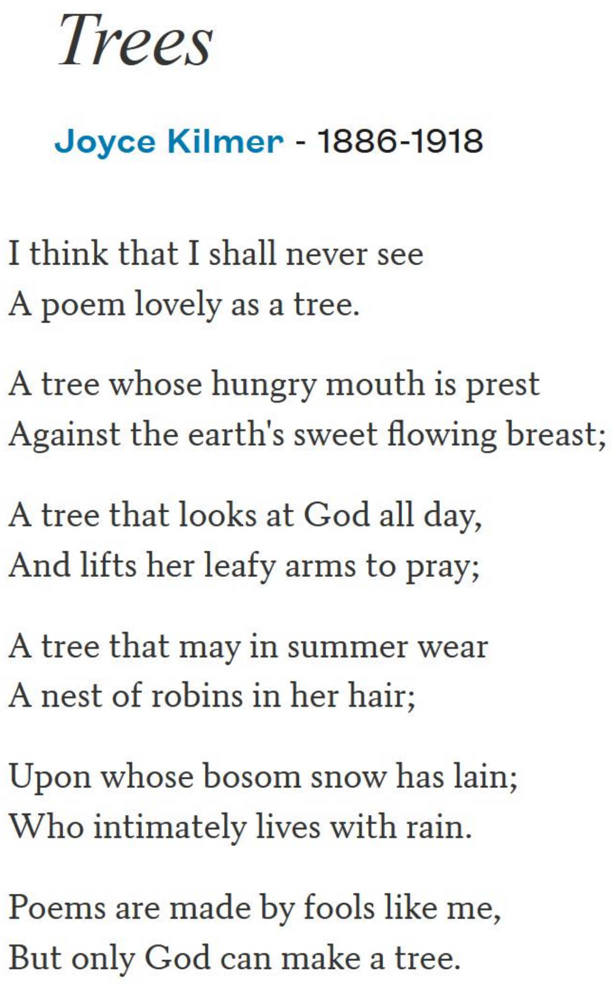
Trees



Trees

Joyce Kilmer - 1886-1918

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

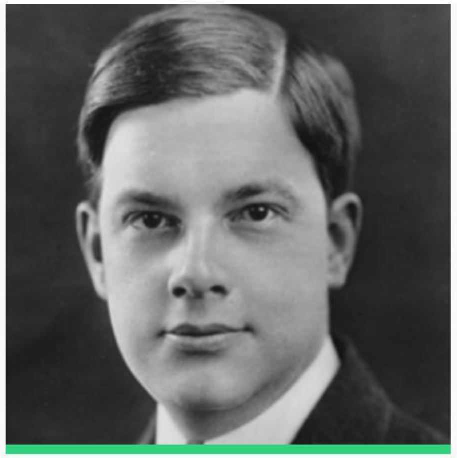
A tree whose hungry mouth is prest
Against the earth's sweet flowing breast;

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

```
my-empty

my-link(1,
  my-link(2,
    my-link(3,
      my-empty)))
```

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!

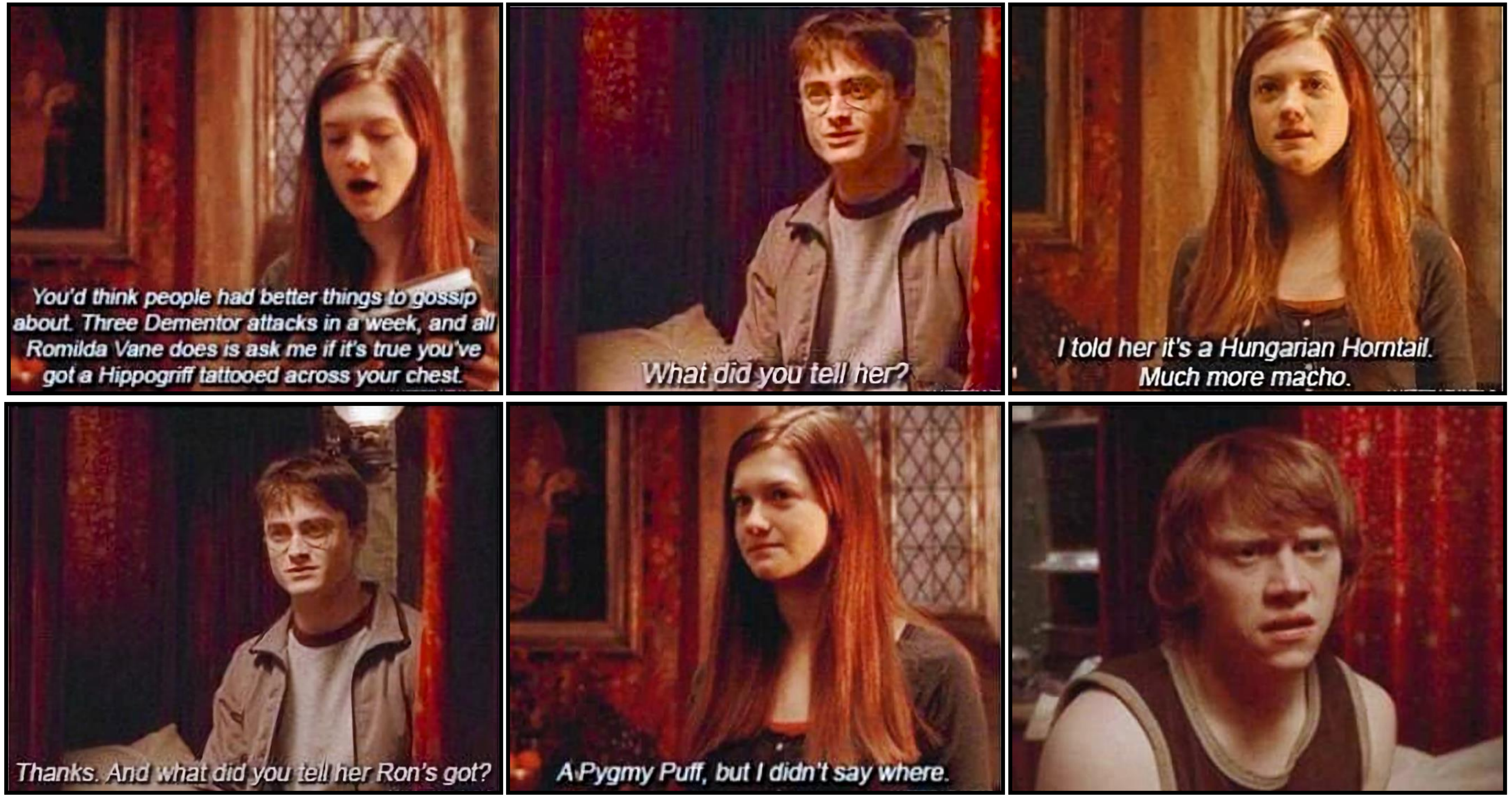
Using my-list Data Template

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

```
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 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.

Ginny controls the rumor mill

Tracking rumors

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

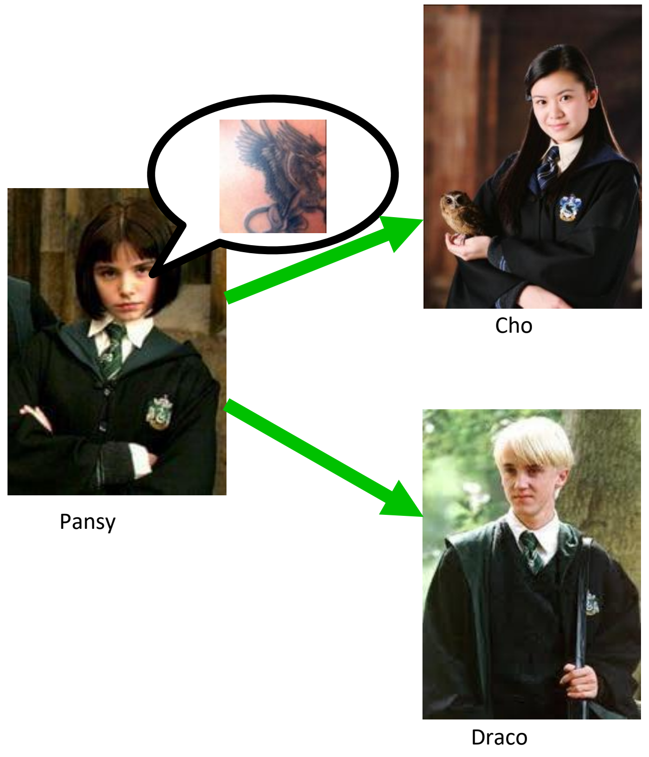
Pansy

Tracking rumors

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

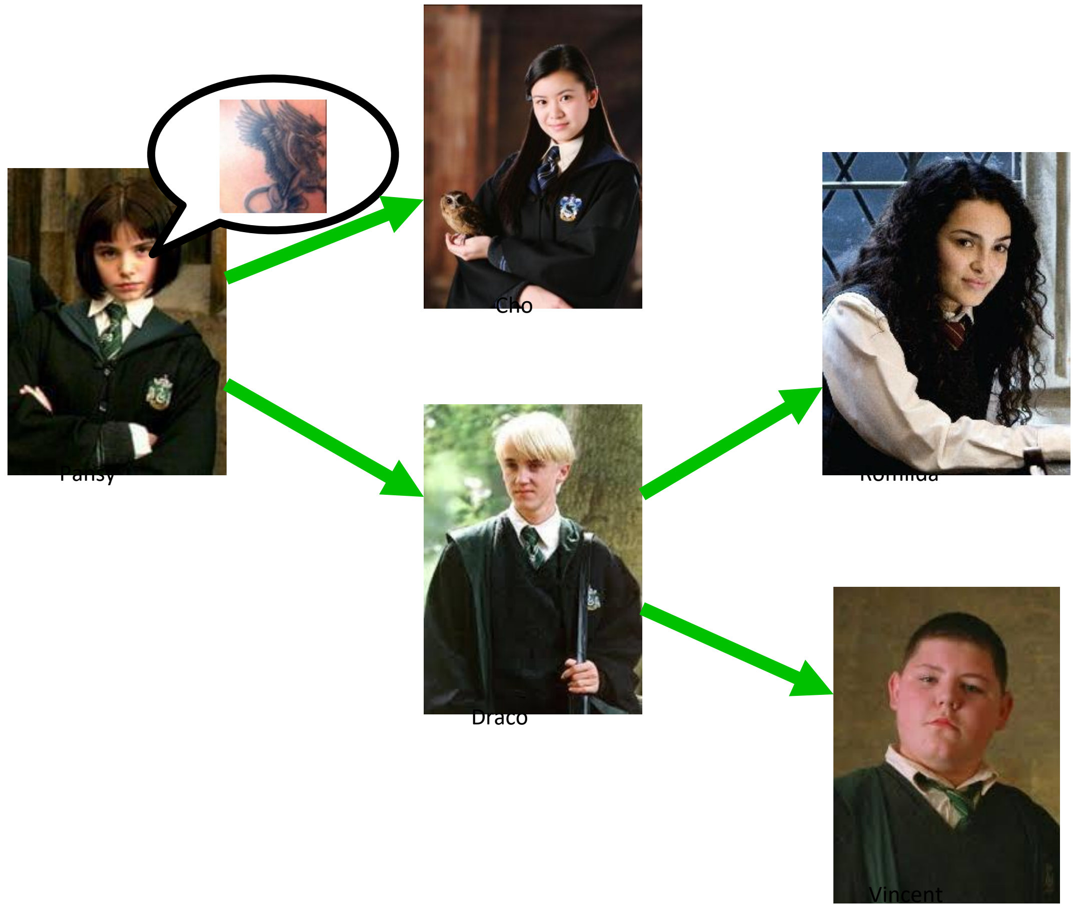
Tracking rumors

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



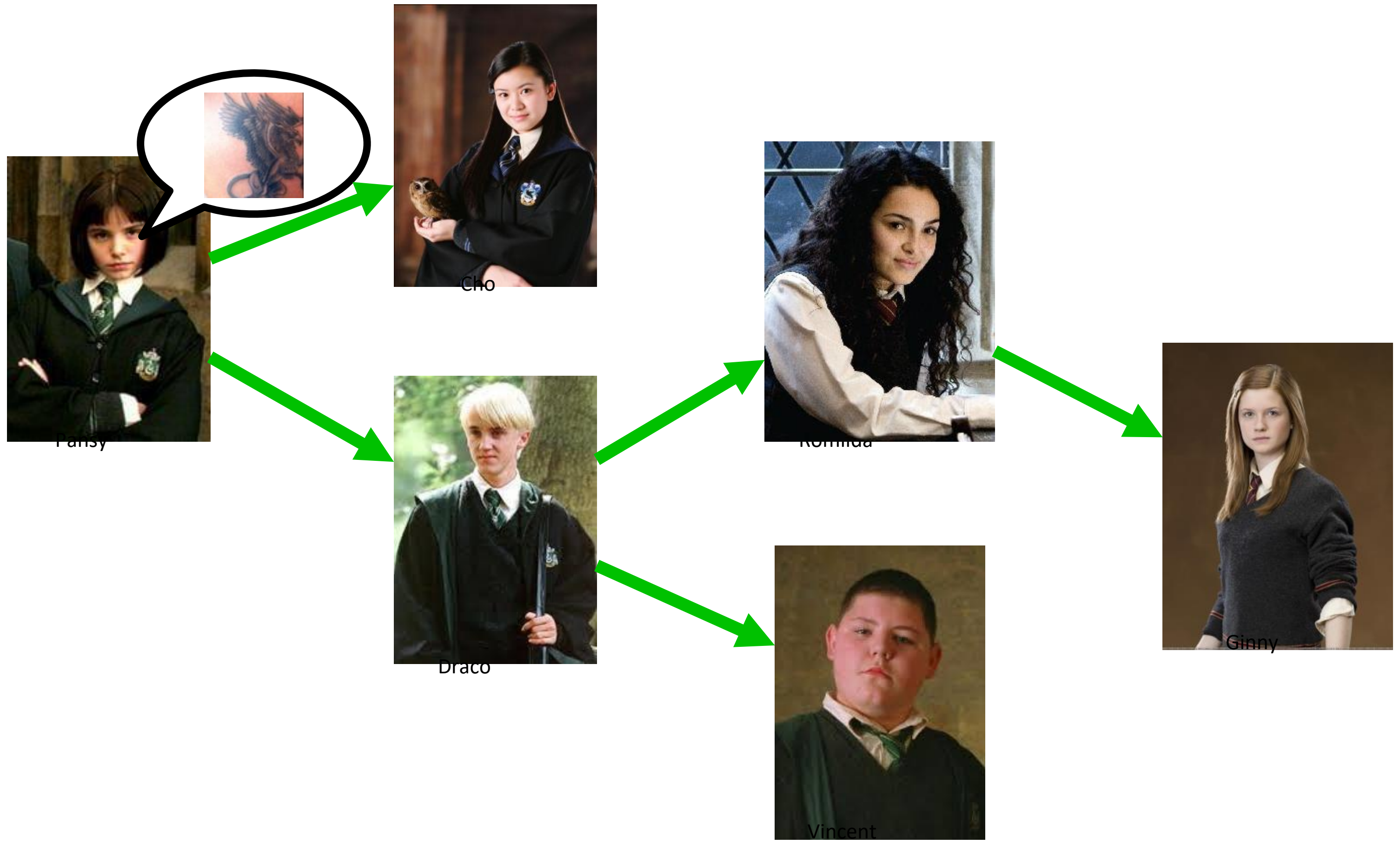
Tracking rumors

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



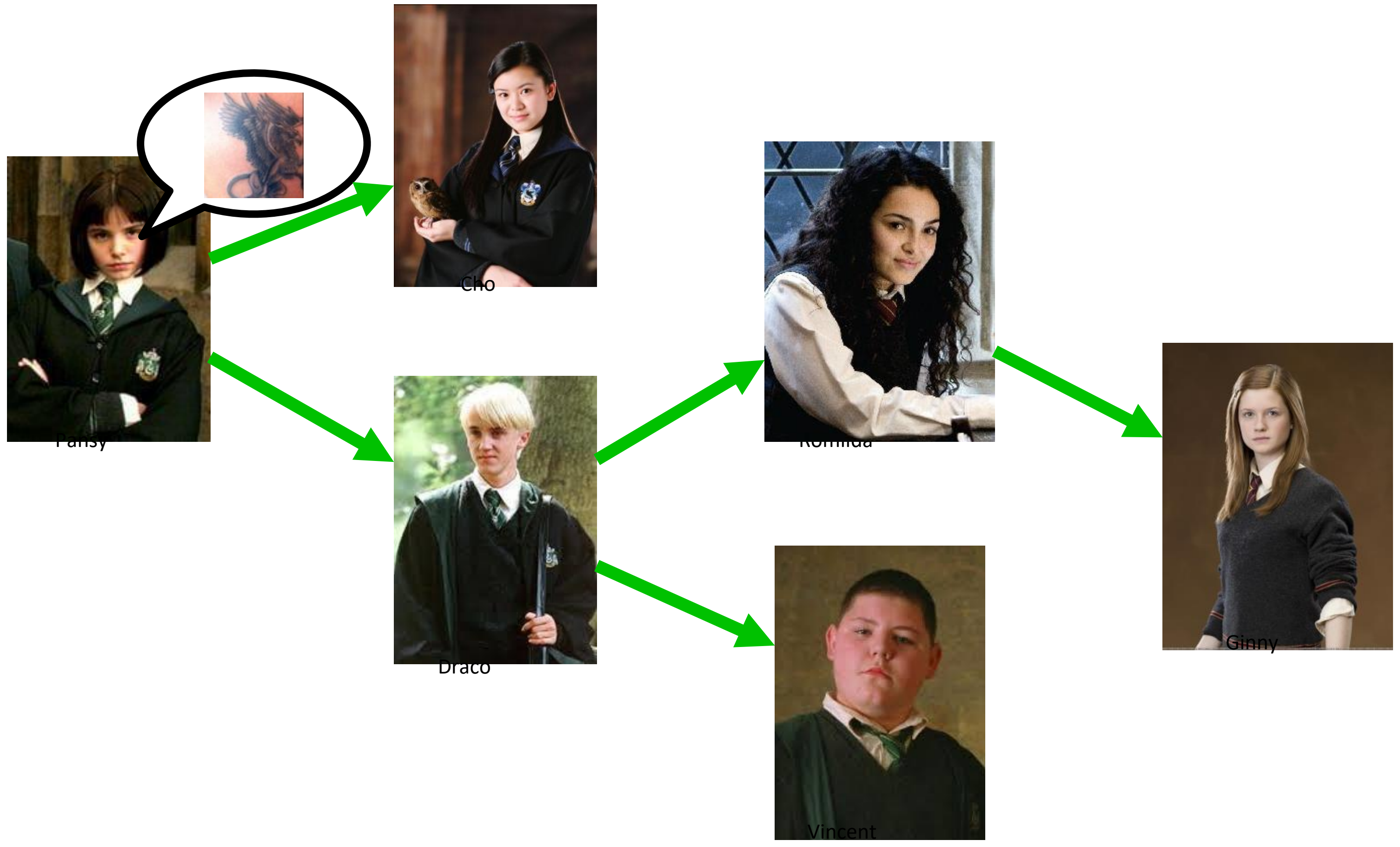
Tracking rumors

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



Tracking rumors

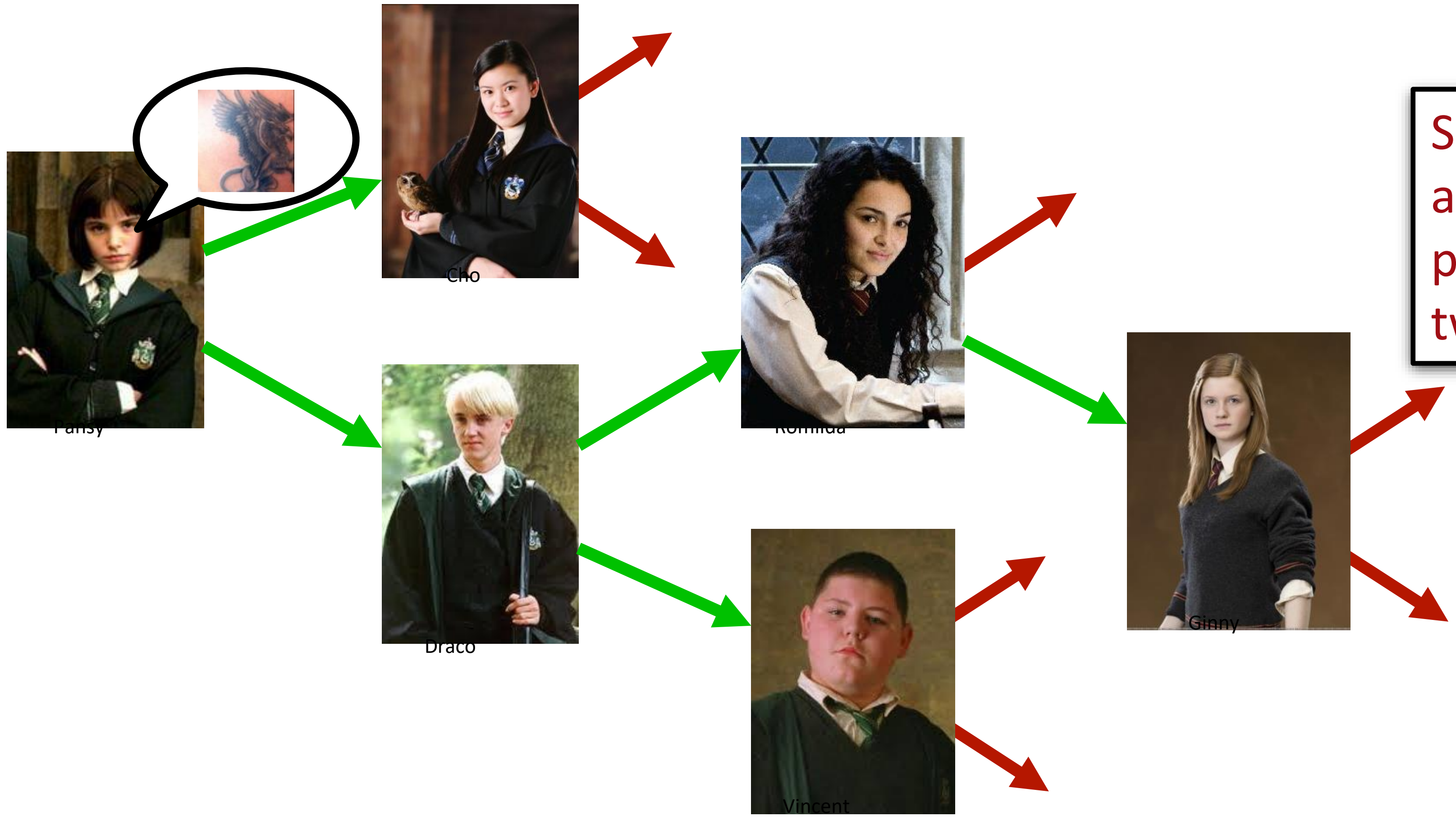
- Suppose we want to track gossip in a rumor mill.



Simplifying assumption: Each person tells at most two others

Tracking rumors

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

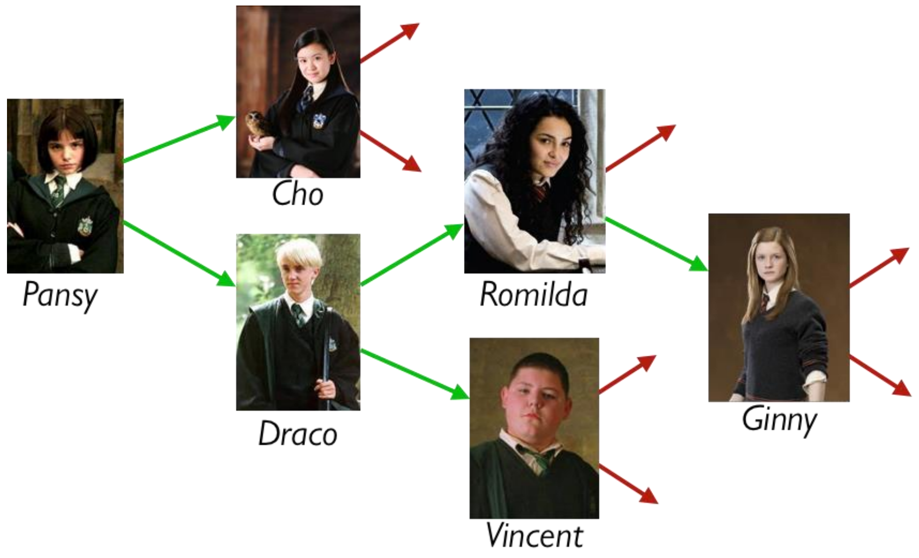


Simplifying assumption: Each person tells at most two others

Tracking rumors

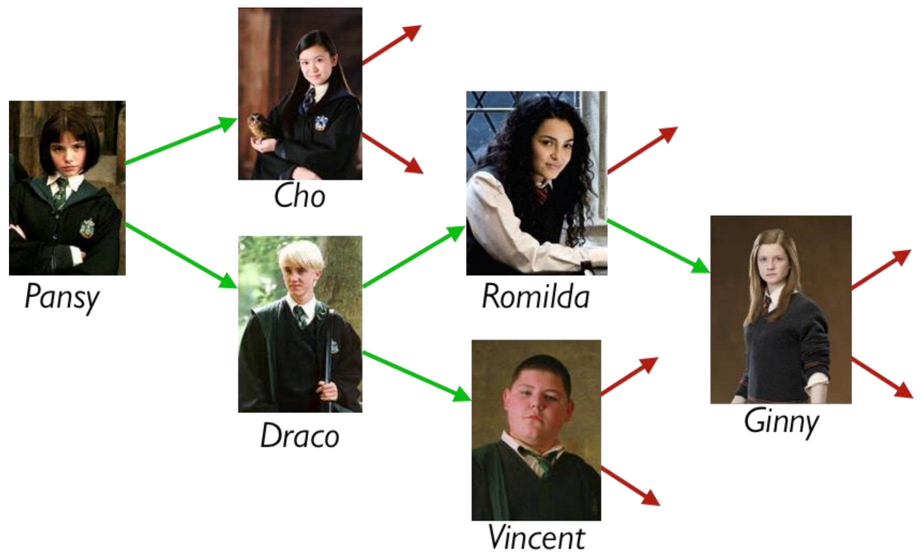
- If you ignore my silly Harry Potter example, this is a pretty serious problem.
- A lot of research right now is focused on building models of how information – and misinformation! – spreads through social networks, both in person and online.

Representing rumor mills



Is a rumor mill simply a list of people?

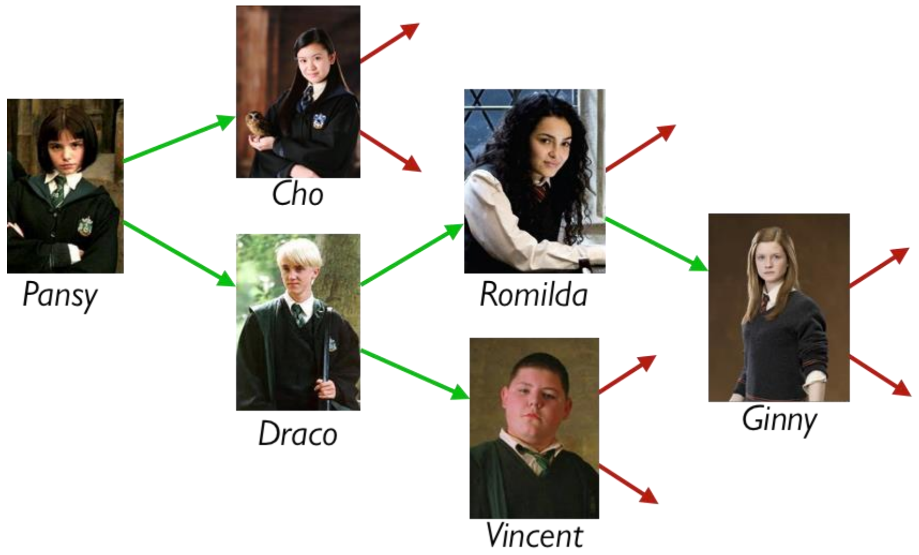
Representing rumor mills



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

Answer: No, because there are *relationships* among the people.

Representing rumor mills

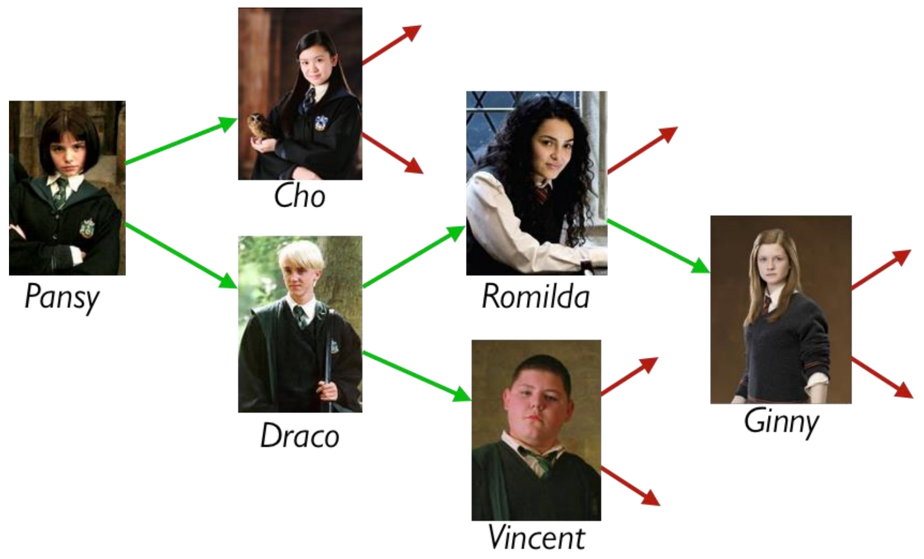


We *could* represent these

relations with a table, e.g.,

name :: String	next1 :: String	next2 :: String
"Pansy"	"Cho"	"Draco"
"Cho"		
…	…	…

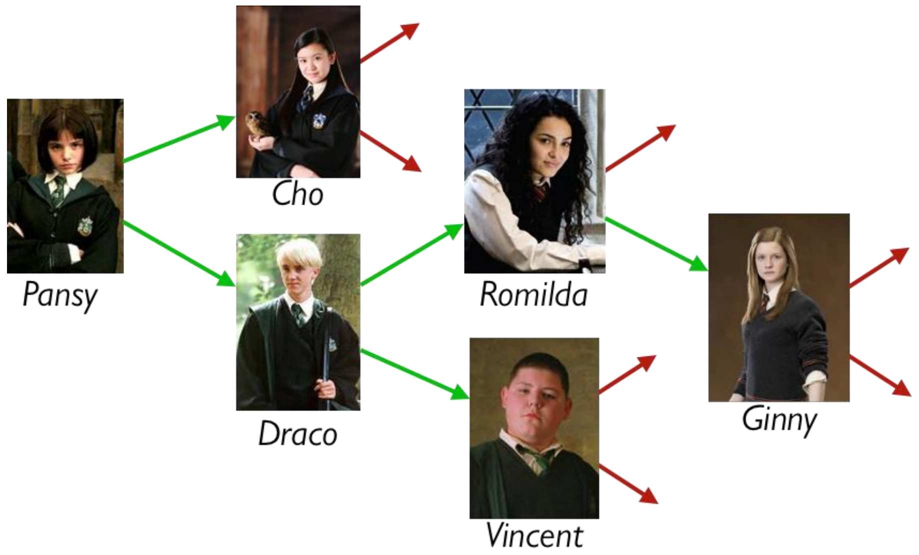
Representing rumor mills



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?

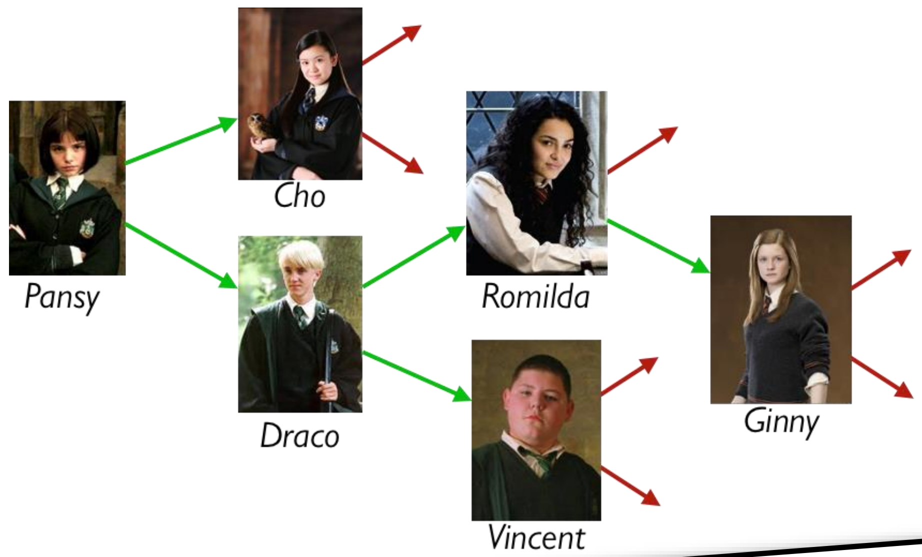
Representing rumor mills



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

How about this?

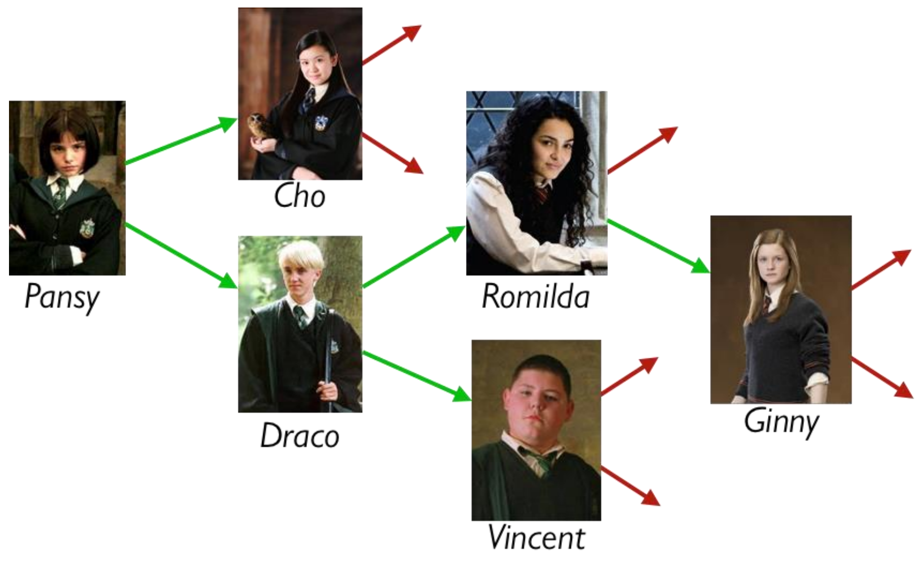
Representing rumor mills



Some people don't gossip to anyone else – see the red arrows above.

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

Representing rumor mills



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

How about this?

Example rumor mills

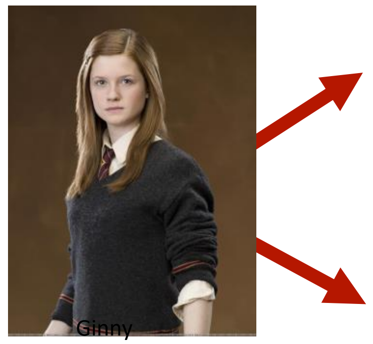
```
data RumorMill:
  | no-one #at the start there is… no-one in the rumor mill!
  | gossip(name :: String, next1 :: RumorMill, next2 :: RumorMill)
end
```

no-one

Example rumor mills

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

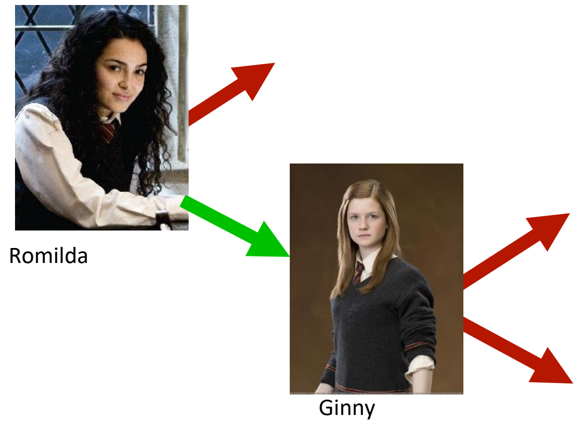
```
gossip("Ginny", no-one, no-one)
```



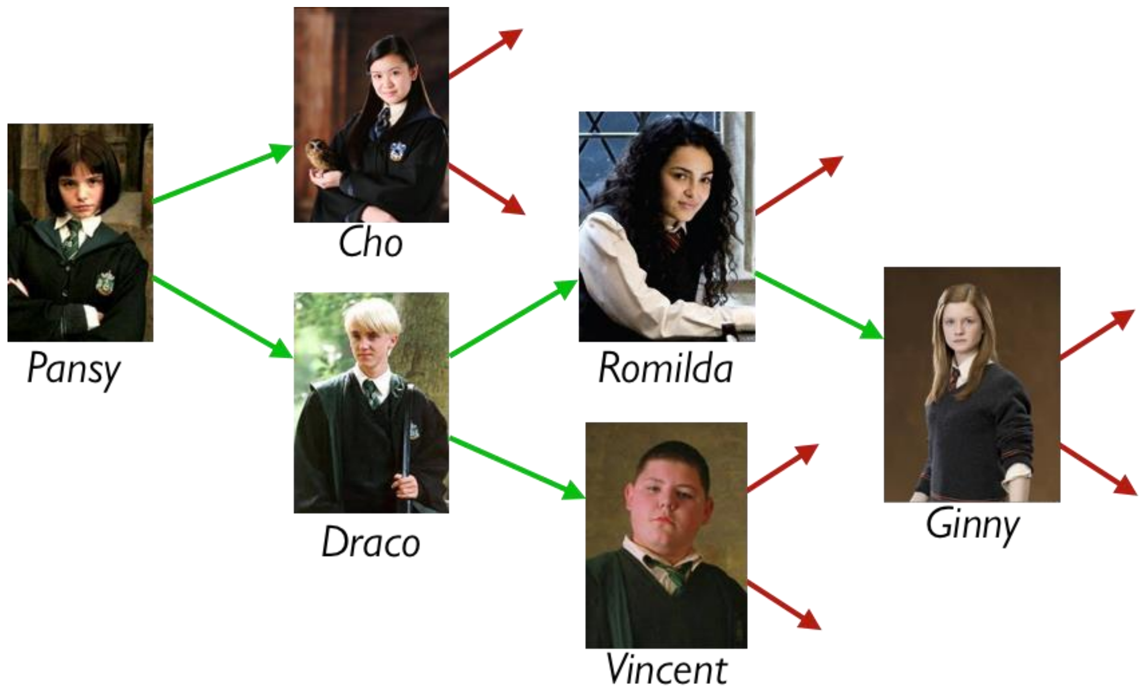
Example rumor mills

```
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

```
GINNY-MILL =
  gossip("Ginny", no-one, no-one)

ROMILDA-MILL =
  gossip("Romilda", no-one, GINNY-MILL)

VINCENT-MILL =
  gossip("Vincent", no-one, no-one)

DRACO-MILL =
  gossip("Draco", ROMILDA-MILL, VINCENT-MILL)

CHO-MILL =
  gossip("Cho", no-one, no-one)

PANSY-MILL =
  gossip("Pansy", CHO-MILL, DRACO-MILL)
```

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

Self-reference × 2

```
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!

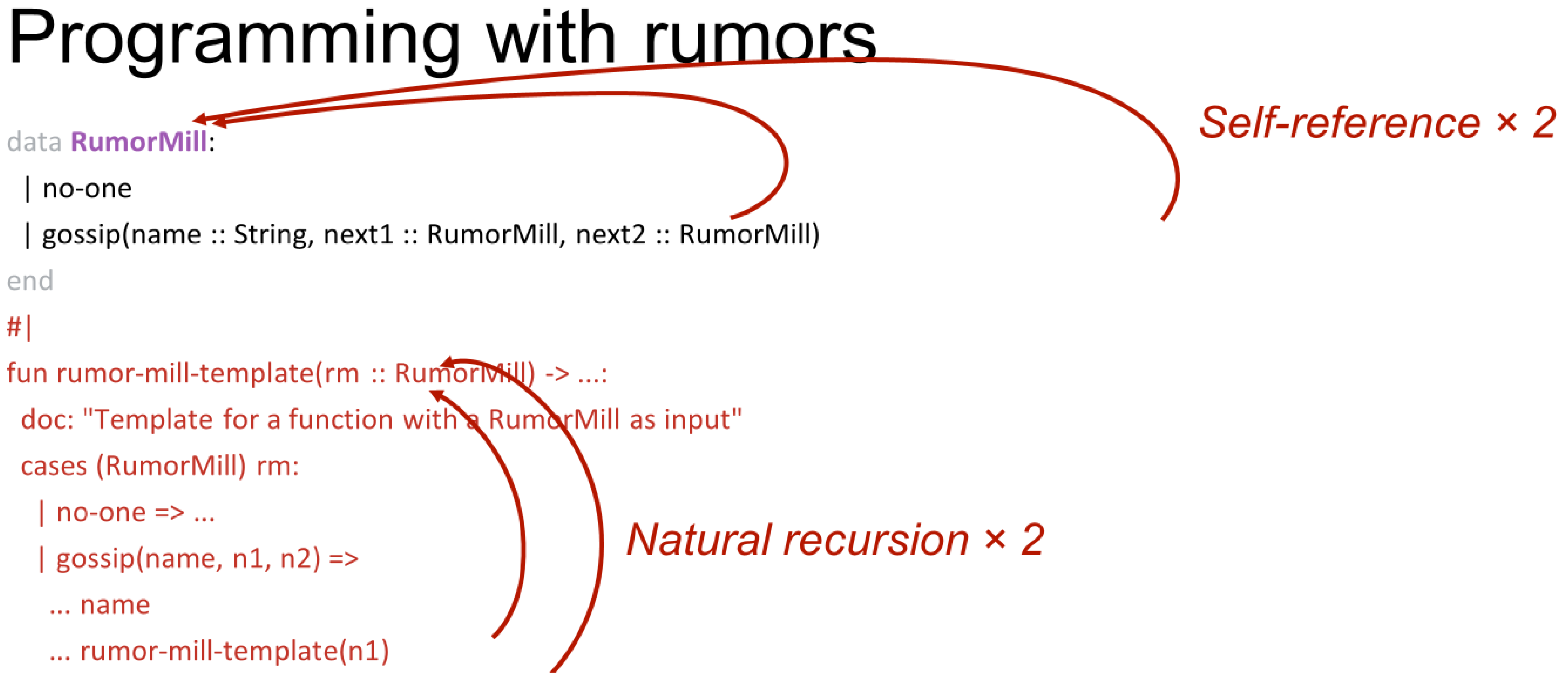
Rumor Mill Template

Programming with rumors

Self-reference × 2

```
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
|#
```

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)
```

Link to code

- 14-new-data-types.arr

Acknowledgements

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