Problem Solving and Abstraction (CMPU 101)

Tom Ellman
Lecture 15
Defining Recursive Data

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

my-list = my-link(1,
  my-link(2,
    my-link(3, my-empty))))

#[my-list: 1, 2, 3]

Here we see how we could have defined the list data type ourselves.
fun my-list-length(ml :: MyList) -> Number:
  doc: "Returns length of ml."
  cases (MyList) ml:
    | my-empty => 0
    | my-link(f, r) => 1 + my-list-length(r)
  end
where:
  my-list-length(my-empty) is 0
  my-list-length(my-list) is 3
end

Here we use a cases expression with pattern matching to implement a function on my-list.
Template for List-Processing Functions

fun my-list-fun(ml :: MyList) -> <data-type>

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

  cases (MyList) ml:
    | my-empty => <base-value>
    | my-link(f, r) => <expression(f, my-list-fun(r))>

  end

where:
  my-list-fun(...) is ... <test-value>

end

|
Data Definitions & Function Templates

• Every data definition has a corresponding template.

• The recursive structure of the template matches the recursive structure of the data.

• We will see this correspondence later today.
Rumor Mill

• Let’s track gossip in a rumor mill.

• A gossip event is when a person passes a rumor to one or more other people.

• Collect and store data about each gossip event.
  – Person sending the rumor.
  – People receiving the rumor.
  – Not the rumor itself. (That would be illegal, ha ha!)

• Acknowledgment: This research is funded by the NSA.
Participants in the Rumor Mill

Pansy

Cho

Romilda

Draco

Ginny

Vincent
“Harry Got a Hippogryph Tattoo”

- Pansy
- Cho
- Draco
- Romilda
- Vincent
- Ginny
Rumor Mill Data Type

Simplifying Assumption: Each person sends a rumor to at most two other people.

data RumorMill:
    | no-one
    | gossip(name :: String,
            next1 :: RumorMill,
            next2 :: RumorMill)
end
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)))
Building a Tree Bottom-Up
(Define Receiver before Sender)

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)
Tree Terminology

• Each element of a tree is called a “node”.
• Arrows go from “parent” to “child”.
• The “root” is the node with no parent.
• A node with no children is a “leaf”.
• A tree in which each node has at most two children is called a “binary tree”.
Recursive Data Structure
Trees and Subtrees

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

- Each child of a node represents a sub-tree.
- Each node is the root of a tree or sub-tree.
- Thus a leaf is a tree.
Programming with Rumors

“I heard we need to use recursion.”

“I heard we should use map.”

“I heard we should use filter.”

Ooops! That’s not what I meant.
Programming with RumorMill

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

#|
fun rumor-mill-template(rm :: RumorMill) -> <data-type>
doc: "Template for a function with a RumorMill as input"
cases (RumorMill) rm:
  | no-one       => <base-value>
  | gossip(n, g1, g2) => <expression(n,
                                     rumor-mill-template(g1),
                                     rumor-mill-template(g2))>
end
end
|#
Programming Example 1

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.
Design the function `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.
Some gossips talk to lots of other gossips. We must generalize our design.
A Gossip is the root node of a tree. Each node in the tree may have any number: 0, 1, 2, … n, … children.

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

Each Gossip has a list of next Gossip(s).
One template takes a single Gossip as parameter.

```haskell
fun gossip-template(g :: Gossip) -> <Any>
  ... gossip.name
  ... log-template(g.next)
End
```

Another template takes a list of Gossip(s) as parameter.

```haskell
fun log-template(l :: List<Gossip>) -> <Any>
cases (List) l:
  | empty => ...
  | link(f, r) =>
    ... gossip-template(f)
    ... log-template(r)
end
end
```
Design **count-gossips** which takes a Gossip and returns the number of people informed by the gossip (including the starting person).
Acknowledgments

This lecture incorporates material from: J. K. Rowling, Harry Potter and the Half-Blood Prince, Marc Smith, Vassar College and Jonathan Gordon, Vassar College