Problem Solving and Abstraction (CMPU 101)

Tom Ellman
Lecture 15
Defining Recursive Data

data MyList:
    | my-empty
    | 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

### Template for List-Processing Functions

```haskell
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
```

```haskell
#|
```
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
  | rMill(name :: String,
         next1 :: RumorMill,
         next2 :: RumorMill)
end
Each red arrow represents a transmission of the rumor from one person to another.
Building a Tree Bottom-Up
(Define Receiver before Sender. Why?)

GINNY-MILL =
  rMill("Ginny", no-one, no-one)

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

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

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

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

PANSY-MILL =
  rMill("Pansy", CHO-MILL, DRACO-MILL)
Tree Terminology

- Each element of a tree is called a “node”.
- Each arrow goes from a “parent” to a “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
  | rMill(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.”

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

data RumorMill:
  | no-one
  | rMill(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>
    | rMill(n, g1, g2) => <expression(n,
                                      rumor-mill-template(g1),
                                      rumor-mill-template(g2))>
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.
fun is-informed(rm :: RumorMill, person :: String) -> Boolean:

doc: "True if and only if person is informed of rm rumor rm"
cases (RumorMill) rm:
  | no-one => false
  | rMill(name, next1, next2) =>
    (person == name) or
    is-informed(next1, person) or
    is-informed(next2, person)
end

where:
  is-informed(no-one, "Cho") is false
  is-informed(ROMILDA-MILL, "Draco") is false
  is-informed(PANSY-MILL, "Ginny") is true
  is-informed(GINNY-MILL, "Ginny") is true
  # No one tells Dobby anything. :-(
  is-informed(PANSY-MILL, "Dobby") is false
end
Programming Example 2

Design the function `gossip-length` that takes a rumor mill and determines the length of the longest sequence of people who are transmitting the rumor.
fun gossip-length(rm :: RumorMill) -> Number:
   doc: "Determine the length of the longest sequence of people who are transmitting the rumor"
   cases (RumorMill) rm:
      | no-one => 0
      | rMill(name, next1, next2)
         => 1 + num-max(gossip-length(next1), gossip-length(next2))
   end
where:
gossip-length(no-one) is 0
      gossip-length(CHO-MILL) is 1
      gossip-length(ROMILDA-MILL) is 2
      gossip-length(DRACO-MILL) is 3
      gossip-length(PANSY-MILL) is 4
end
Some gossips talk to lots of other gossips. We must generalize our design.
gossip

name: "Pansy",
next: [list:

  Item 0: gossip
    name: "Romilda",
    next: [list: gossip
        name: "Ginny",
        next: [list: ]
      ]

  Item 1: gossip
    name: "Cho",
    next: [list: ]

  Item 2: gossip
    name: "Draco",
    next: [list: gossip
        name: "Vincient",
        next: [list: ]
      ]
    ]]
A Gossip is the root node of a tree. Each node in the tree may have any number: 0, 1, 2, ... n, ... children.

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.

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

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

```
#|
fun log-template(l :: List<Gossip>) -> <Any>
cases (List) l:
  | empty => ...
  | link(f, r) =>
    ... gossip-template(f)
    ... log-template(r)
  end
end
end
|#
```
Programming Example 3

Design count-gossips which takes a Gossip and returns the number of people informed by the gossip (including the starting person).
fun count-gossip(g : Gossip) -> Number:
  1 + count-gossip-list(g.next)
where:
  count-gossip(gPansy) is 6
  count-gossip(gDraco) is 4
end

fun count-gossip-list(glst : List<Gossip>) -> Number:
  cases (List) glst:
    | empty => 0
    | link(f,s) => count-gossip(f) + count-gossip-list(s)
end
end
Sorting Lists of Numbers

[5, 2, 7, 3, 8, 0, 9]  ➡️  [0, 2, 3, 5, 7, 8, 9]

[5, 2, 7, 3, 8, 0, 9]  ➡️  [0, 2, 3, 5, 7, 8, 9]

Binary Search Tree
(Binary Sort Tree)
Structure of a Binary Sort Tree

Root

- Numbers Less than \( a \)
- Numbers Greater than \( a \)

Left Subtree

Right Subtree

Numbers Less than \( a \)

Numbers Greater than \( a \)
data BSTNode:
   | emptyBST
   | bstNode(n :: Number, left :: BSTNode, right :: BSTNode)
end
Binary Sort Tree

- Store the numbers in a tree structure.
- The root of the tree holds a number $n$.
- The left subtree holds numbers less than $n$.
- The right subtree holds numbers greater than $n$.
- Each subtree stores numbers in the same way as the whole tree.
Example: [list: 0, 2, 3, 5, 7, 8, 9]
Inserting the number $N$ into BST

- If BST is empty, then return a new tree containing only the number $N$.
- If $N < R$ then insert $N$ into the left subtree.
- If $N > R$ then insert $N$ into the right subtree.
- If the root $R$ of BST is $N$, then return BST.
fun bstInsert(n :: Number, node :: BSTNode) -> BSTNode:
  cases (BSTNode) node:
    | emptyBST => bstNode(n, emptyBST, emptyBST)
    | bstNode(m, left, right) =>
      if   (n < m): bstNode(m, bstInsert(n, left), right)
      else if (n > m): bstNode(m, left, bstInsert(n,right))
      else:           bstNode(m, left,right)
  end
end
end
bsTreeSort

fun bsTreeSort(dat :: List<Number>) -> List<Number>:
  bstToList(listToBST(dat))
end

[ list: 5, 2, 7, 3, 8, 0, 9 ]  ───>  [ list: 0, 2, 3, 5, 7, 8, 9 ]
fun listToBST(lst :: List<Number>) -> BSTNode:
  cases (List) lst:
    | empty => emptyBST
    | link(f,r) => bstInsert(f, listToBST(r))
  end
end

lst = [list: 3, 7, 2, 6, 4, 1, 0, 5]
bst = listToBST(lst)
lst = [3, 7, 2, 6, 4, 1, 0, 5]
bst = listToBST(lst)
fun bstToList(node :: BSTNode) -> List<Number>:
  cases (BSTNode) node:
  | emptyBST  => [list: ]
  | bstNode(m, left, right)
    =>
      sLeft  = bstToList(left)
      sRight = bstToList(right)
      append(sLeft, link(m, right))
  end
end

```python
>>> sorted1 = bstToList(bst)
```
```bash
>>> sorted1
[list: 0, 1, 2, 3, 4, 5, 6, 7]
```
fun bsTreeSort(dat :: List<Number>) -> List<Number>:
    bstToList(listToBST(dat))
end

>>> sorted2 = bsTreeSort(lst)

>>> sorted2
[list: 0, 1, 2, 3, 4, 5, 6, 7]
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