Inductive Data Types

Tjark Weber
tjark.weber@it.uu.se

Program Design and Data Structures

Labs

Fall labs:

- 11 graded labs in Fall. Of these, you must pass **at least 8** in order to complete the Fall lab part of the course. At most two of the 8 labs can be made up in Spring.

Spring labs:

- 8 graded labs in Spring. Of these, you must pass **at least 5** in order to complete the Spring lab part of the course.
- If you pass more than 5 labs in Spring, up to two supernumerary passed labs will be transferred to your Fall lab component to compensate for missed labs there.

Today and Next Lecture

- Recap: Haskell's Type System
- New Names for Old Types
- New Types
    - Enumeration Types
    - Tagged Unions
    - Polymorphic Data Types
    - Inductive Data Types
        - Trees

Recap: Haskell's Type System

Types and Type Annotations

Every Haskell expression has a **type**.

```
> :t length "hello"
length "hello" :: Int
```

Type annotations can be used to indicate the type of expressions.

```
> (length :: String -> Int) ("hello" :: String) :: Int
5
```

However, since GHC automatically infers the type of expressions, type annotations are usually (except in special situations) redundant.

Strongly Typed, Static Type-Checking

Haskell is a **strongly typed** language.

```
> 1 + "1"

<interactive>:2:3:
    No instance for (Num [Char]) arising from a use of `+'
    ...
```

Type checking is **static**, i.e., performed at compile time.

```
> let f x = x + length x

<interactive>:2:22:
    Couldn't match expected type `[a0]' with actual type `Int'
    ...
```

Basic Types and Type Constructors

Types are built starting from **basic types** and using **type constructors**.

Basic Types:

- `Int`, `Integer`, `Char`, `Bool`, `Double`, `()`, ...

Type constructors:

- `->`/2 e.g., `String -> Int`
- `(,)`/2 e.g., `(Char, Bool)`
- `[]`/1 e.g., `[Double]`
- ...

New Names for Old Types

Type Synonyms

The keyword `type` gives a new name to an existing type.

The simplest form is

```
type Identifier = TypeExpression
```

For instance,

```haskell
type Real = Double

type String = [Char]
```

Note that type names must begin with an uppercase character.

Type Synonyms (cont.)

The new name becomes **synonymous** with the type expression. Both can be used interchangeably.

For instance,

```
> "hello" :: [Char]
"hello"

> "hello" :: String
"hello"

> ("hello" :: [Char]) == ("hello" :: String)
True
```

Parametric Type Synonyms

Types can be **polymorphic**, i.e., contain type variables. Therefore, type synonym declarations can take type variables as parameters.

The general form of a type synonym declaration is

```
type Identifier tyvar_1 ... tyvar_k = TypeExpression
```

For instance,

```
type Predicate a = a -> Bool

type AssocList a b = [(a,b)]
```

Parametric Type Synonyms (cont.)

All type variables that appear in the type expression (i.e., on the right) must be mentioned as a parameter (i.e., on the left).

```
> type Predicate = a -> Bool

<interactive>:2:18: Not in scope: type variable `a'
```

Type variables may be instantiated with types (as usual).

```
> (even :: Predicate Int) 0
True
```

New Types

New Types: Motivation

Types prevent programming errors by ensuring that operations can only be applied to appropriate data.

```
> 1 + "1"

<interactive>:2:3:
    No instance for (Num [Char]) arising from a use of `+'
```

Here, GHC detects an error because `1` and `"1"` have different types. (In particular, `"1"` is not a number.)

New Types: Motivation (cont.)

Suppose we want to represent cardinal directions (North, South, East, West) in our program. If we declare

```
> type Direction = Int
> let north = 1; south = 2; east = 3; west = 4
```

then there is no difference between the type of directions and the type of integers:

```
> 1 + north  -- adding directions to integers!?
2
```

Enumeration Types

The keyword `data` declares a **new** type.

Here, we declare an **enumeration type**, i.e., a type that has a finite number of values C1, . . . , Cn:

```
data Identifier = C1 | ... | Cn
```

For instance,

```
data Direction = North | South | East | West
```

Now, `North` is a value of type `Direction`:

```
> :t North
North :: Direction
```

Enumeration Types: Example

The type `Bool` has a finite number of values. It could (in principle) be declared as an enumeration type as follows:

```
data Bool = True | False
```

Other types that could (in principle) be declared as enumeration types are `()`, `Char`, even `Int`.

Enumeration Types: Exercise

If we compare two integers a and b, then either (i) $a < b$, (ii) $a = b$, or (iii) $a > b$.

1. Declare an enumeration type `Ordering` that has three values: `LT`, `EQ`, `GT`.
2. Write a function `compare :: Int -> Int -> Ordering` such that `compare a b` returns `LT` if `a < b`, `EQ` if `a == b`, and `GT` if `a > b`.

Enumeration Types: Exercise (cont.)

Solution:

```
data Ordering = LT | EQ | GT

{- compare a b
   PRE: true POST: ... EXAMPLES: ...
 -}
compare :: Int -> Int -> Ordering
compare a b
    | a < b     = LT
    | a == b    = EQ
    | otherwise = GT
```

`Ordering` and `compare` are already declared in the Haskell Prelude.

Constructors

In a datatype declaration

```
data Identifier = C1 | ... | Cn
```

C1, . . . , Cn are called **(value) constructors**.

Note that value constructors must begin with an uppercase character.

Constructor Patterns

Constructors can be used in patterns. A constructor matches only itself.

For instance,

```
{- opposite d
   PRE: true
   POST: the direction opposite d
   EXAMPLE: opposite North = South
 -}
opposite :: Direction -> Direction
opposite North = South
opposite South = North
opposite East = West
opposite West = East
```

Constructor Patterns (cont.)

Constructors can be used in patterns. A constructor matches only itself.

For instance,

```
{- signum n
   PRE: true
   POST: -1 if n<0, 0 if n=0, 1 if n>0
   EXAMPLE: signum 42 = 1
 -}
signum :: Int -> Int
signum n =
  case compare n 0 of
    LT -> -1
    EQ -> 0
    GT -> 1
```

Beyond Enumeration Types

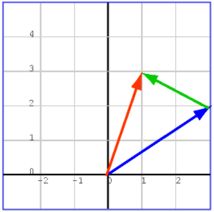
Beyond Enumeration Types: Motivation

Suppose you want to implement a calculator for rational numbers. You could model fractions as pairs of integers, `(Integer,Integer)`.

```
qadd (a,b) (c,d) = (a*d + b*c, b*d)
```

But pairs of integers could also denote vectors in $\mathbb{Z}^2$.

```
vadd (a,b) (c,d) = (a+c, b+d)
```



We would like rational numbers to have a separate type (so that GHC can detect a type error when we accidentally try to, e.g., add a rational number to a vector).

Constructors with Arguments

Constructors in a datatype declaration may take arguments. The type of each argument is specified after the constructor.

```
data Identifier = C1 Type_1_1 Type_1_2 ...
                | ...
                | Cn Type_n_1 Type_n_2 ...
```

For instance,

```
data MyRational = Rat Int Int
```

```
Rat 2 3
```

Constructors with Arguments: Pattern Matching

Constructors that take arguments can be used in patterns (together with a pattern for each argument).

For instance,

```
{- qadd x y
   PRE: x and y are rational numbers (with denominator <> 0)
   POST: the sum of x and y
   EXAMPLE: qadd (Rat 1 2) (Rat 1 3) = Rat 5 6
 -}
qadd :: MyRational -> MyRational -> MyRational
qadd (Rat a b) (Rat c d) =
  Rat (a*d + b*c) (b*d)
```

Example: Geometric Shapes

A type that models circles (of a given radius), squares (of a given side length) and triangles (of three given side lengths):

```
data Shape = Circle Double
           | Square Double
           | Triangle Double Double Double
```

Constructing values of type `Shape`:

```
Circle 1.0
```

```
Square (1.0 + 1.0)
```

```
Triangle 3.0 4.0 5.0
```

Example: Geometric Shapes (cont.)

A function that computes the area of a geometric shape:

```
{- area s
   PRE: true
   POST: the area of s
   EXAMPLE: area (Circle 1.0) = 3.141592654
 -}
area :: Shape -> Double

area (Circle r) = pi * r * r
area (Square l) = l * l
area (Triangle a b c) =
  let s = (a + b + c) / 2.0
  in -- Heron's formula
    sqrt (s * (s-a) * (s-b) * (s-c))
```

The Type of Constructors

A constructor that takes no arguments has the declared type.

```
> :t North
North :: Direction
```

A constructor that takes arguments has a function type.

```
> :t Circle
Circle :: Double -> Shape

> :t Square
Square :: Double -> Shape

> :t Triangle
Triangle :: Double -> Double -> Double -> Shape
```

However, constructors (unlike functions) may be used in patterns!

Tagged Unions/Sum Types

Types declared via

```
data Identifier = C1 Type1 | ... | Cn TypeN
```

are also known as **tagged unions** or **sum types**.

We can think of values of this type as either being a value of type 1 or ... or a value of type N, that is *tagged* with a constructor that indicates which type the value has.

Example: Computation with Integers and Reals

Handling numbers, whether they are integers or reals. (This is not really practical, but still an interesting example of a tagged union.)

```
data Number = NumberInt Integer | NumberDouble Double

{- toReal x
   PRE: true
   POST: the real number that corresponds to x
   EXAMPLE: toReal (NumberInt 1) = NumberDouble 1.0
 -}
toReal :: Number -> Number
toReal (NumberInt i) = NumberDouble (fromInteger i)
toReal (NumberDouble d) = NumberDouble d
```

Example: Computation with Integers and Reals (cont.)

```
{- addNumbers x y
   PRE: true
   POST: x+y (the result is a real number if x or y are
       real numbers)
   EXAMPLE: addNumbers (NumberDouble 3.1) (NumberInt 1) =
       NumberDouble 4.1
 -}
addNumbers :: Number -> Number -> Number
addNumbers (NumberInt a) (NumberInt b) = NumberInt (a+b)
addNumbers x y =
  let
    NumberDouble rx = toReal x
    NumberDouble ry = toReal y
  in
    NumberDouble (rx+ry)
```

Polymorphic Data Types

The argument types of constructors can be **polymorphic**, i.e., contain type variables. Therefore, datatype declarations can take type variables as parameters.

The general form of a datatype declaration is

```
data Identifier tyvar_1 ... tyvar_k = C1 Type_1_1 Type_1_2 ...
                                     | ...
                                     | Cn Type_n_1 Type_n_2 ...
```

Example: Maybe a

For instance,

```
data Maybe a = Just a | Nothing

> :t Nothing
Nothing :: Maybe a

> :t Just
Just :: a -> Maybe a

> :t Just 'a'
Just 'a' :: Maybe Char

> :t Just True
Just True :: Maybe Bool

> :t Just []
Just [] :: Maybe [a]
```

The Maybe Type

We can think of values of type `Maybe a` as representing either a single value or zero values of type `a`.

Type `Maybe a` is useful to model partial functions (i.e., functions that normally return a value of type `a`, but may not do so for some argument values).

For instance, not every key is present in an association list:

```
> lookup 1 [(0,"x"),(1,"y")]
Just "y"
```

```
> lookup 2 [(0,"x"),(1,"y")]
Nothing
```

`Maybe a` is more explicit than exceptions. (It requires callers to deal with the `Nothing` case explicitly.) This may or may not be desirable.

`Maybe a` and related functions are declared in the Haskell Prelude.

Polymorphic Data Types (cont.)

All type variables that appear in an argument type (i.e., on the right) must be mentioned as a parameter (i.e., on the left).

```
> data Maybe = Just a | Nothing

<interactive>:2:19: Not in scope: type variable `a'
```

Type variables may be instantiated with types (as usual).

```
> Nothing :: Maybe Int
Nothing
```

Inductive Data Types

Inductive Data Types

The argument types of constructors may refer to (instances of) the data type that is being declared.

That is, in a datatype declaration

```
data Identifier tyvar_1 ... tyvar_k = C1 Type_1_1 Type_1_2 ...
                                    | ...
                                    | Cn Type_n_1 Type_n_2 ...
```

the type expressions `Type_i_j` may contain `Identifier` (applied to k type parameters).

Example: [a]

The type `[a]` of lists (with elements from `a`) is an inductive type.

1. Base case:

   `[] :: [a]`

2. Inductive step:

   If `x :: a` and `xs :: [a]`, then `x:xs :: [a]`.

Type `[a]` could (in principle) be declared as follows:

```
data [] a = []
          | a : [a]
```

Example: Peano's Natural Numbers

Giuseppe Peano (1889) gave an axiomatization of natural numbers $\mathbb{N}$, based on a successor function S.

1. Base case:
   $0 \in \mathbb{N}$.
2. Inductive step:
   If $x \in \mathbb{N}$ then $S(x) \in \mathbb{N}$.

For instance, the natural number 3 is written as $S(S(S(0)))$.

Peano's numbers as an inductive data type in Haskell:

Example: Peano's Natural Numbers

Giuseppe Peano (1889) gave an axiomatization of natural numbers $\mathbb{N}$, based on a successor function S.

1. Base case:
   $0 \in \mathbb{N}$.
2. Inductive step:
   If $x \in \mathbb{N}$ then $S(x) \in \mathbb{N}$.

For instance, the natural number 3 is written as $S(S(S(0)))$.

Peano's numbers as an inductive data type in Haskell:

```
data Nat = Zero | S Nat
```

Example: Peano's Natural Numbers (cont.)

```
data Nat = Zero | S Nat

{- addNat x y
   PRE: true
   POST: x+y
   EXAMPLE: addNat (S Zero, S Zero) = S (S Zero)
 -}
addNat :: Nat -> Nat -> Nat
```

Example: Peano's Natural Numbers (cont.)

```haskell
data Nat = Zero | S Nat

{- addNat x y
   PRE: true
   POST: x+y
   EXAMPLE: addNat (S Zero, S Zero) = S (S Zero)
 -}
addNat :: Nat -> Nat -> Nat

-- VARIANT: size of x
addNat Zero y = y
addNat (S x) y = S (addNat x y)
```

Example: Arithmetic Expressions

For instance, $1 + 2$, $3 \cdot 4 + 5$, ...

Let us define arithmetic expressions (that involve addition and multiplication over integers) inductively:

1. Base case:
    - Each integer is an arithmetic expression (AExp).
2. Inductive step:
    - If e_1 and e_2 are AExps, then $e_1 + e_2$ is an AExp.
    - If e_1 and e_2 are AExps, then $e_1 \cdot e_2$ is an AExp.

Example: Arithmetic Expressions

For instance, $1 + 2$, $3 \cdot 4 + 5$, ...

Let us define arithmetic expressions (that involve addition and multiplication over integers) inductively:

1. Base case:
   - Each integer is an arithmetic expression (AExp).
2. Inductive step:
   - If e_1 and e_2 are AExps, then $e_1 + e_2$ is an AExp.
   - If e_1 and e_2 are AExps, then $e_1 \cdot e_2$ is an AExp.

```
data AExp = Atom Int
          | Plus AExp AExp
          | Times AExp AExp
```

Example: Evaluation of Arithmetic Expressions

```
data AExp = Atom Int
          | Plus AExp AExp
          | Times AExp AExp

{- eval e
   PRE: true
   POST: the value of e
   EXAMPLE: eval (Plus (Atom 1) (Atom 2)) = 3
 -}
eval :: AExp -> Int
```

Example: Evaluation of Arithmetic Expressions

```
data AExp = Atom Int
          | Plus AExp AExp
          | Times AExp AExp

{- eval e
   PRE: true
   POST: the value of e
   EXAMPLE: eval (Plus (Atom 1) (Atom 2)) = 3
 -}
eval :: AExp -> Int

-- VARIANT: size of e
eval (Atom i) = i
eval (Plus x y) = eval x + eval y
eval (Times x y) = eval x * eval y
```

Functions for Data Types

In Haskell, the declaration of types is separate from the declaration of functions for those types.

For instance, so far we cannot compare two `AExp`s with equality:

```
> Atom 1 == Atom 2

<interactive>:...:8:
    No instance for (Eq AExp) arising from a use of `=='
    ...
```

Haskell doesn't even know how to print an `AExp`:

```
> Atom 1

<interactive>:...:1:
    No instance for (Show AExp) arising from a use of `print'
    ...
```

Data Types: deriving

You can tell GHC to print your own data type "in the standard way" by adding a

```
deriving (Show)
```

clause immediately after the datatype declaration. For instance,

```
data AExp = Atom Int
          | Plus AExp AExp
          | Times AExp AExp deriving (Show)
```

Now,

```
> Atom 1
Atom 1
```

We'll talk more about `deriving` later, in the context of type classes.

Trees

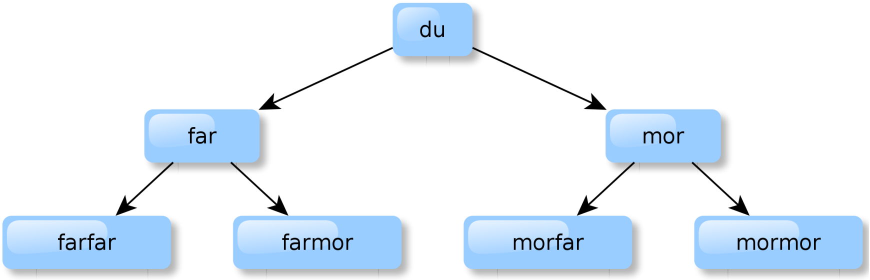
Trees: Motivation

So far, the only *container* type that we have seen are lists. Why should one (sometimes) use trees?

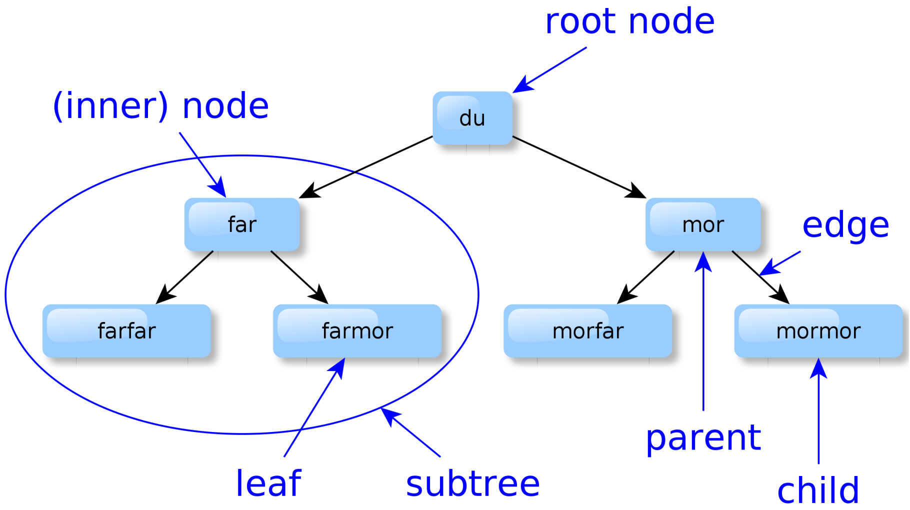
Hierarchical data:
The internal structure of a list is linear (i.e., there is a first, second, . . . element). Trees allow to model hierarchical data.

Complexity:
Algorithms that use trees may have different complexity than algorithms that use lists.

Example: Family Tree



Trees: Terminology



Full Binary Trees

A tree where each inner node has exactly two children is called a **full binary** tree. Full binary trees (with values of type `a`) can be defined inductively:

1. Base case:
   Each value of type `a` is a full binary tree, namely a leaf.
2. Inductive step:
   If x is of type `a` and t_1 and t_2 are full binary trees, then Node t_1 x t_2 is a full binary tree.

In Haskell:

Full Binary Trees

A tree where each inner node has exactly two children is called a **full binary** tree. Full binary trees (with values of type `a`) can be defined inductively:

1. Base case:
   Each value of type `a` is a full binary tree, namely a leaf.
2. Inductive step:
   If x is of type `a` and t_1 and t_2 are full binary trees, then Node t_1 x t_2 is a full binary tree.

In Haskell:

```
data FBTree a = Leaf a
              | Node (FBTree a) a (FBTree a)
```

Full Binary Trees: Examples

```
data FBTree a = Leaf a
              | Node (FBTree a) a (FBTree a)
```

Some full binary trees in Haskell:

```
> Leaf "farfar"

> Node (Leaf "farfar") "far" (Leaf "farmor")
```

Pattern Matching: Example

Functions over trees—like functions over inductive data types in general—are usually defined using pattern matching and recursion.

For instance,

```
{- rootValue t
   PRE: true
   POST: the value at t's root node
   EXAMPLE: rootValue (Leaf "foo") = "foo"
 -}
rootValue :: FBTree a -> a
```

Pattern Matching: Example

Functions over trees—like functions over inductive data types in general—are usually defined using pattern matching and recursion.

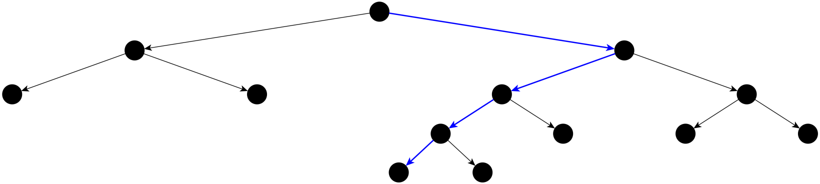
For instance,

```
{- rootValue t
   PRE: true
   POST: the value at t's root node
   EXAMPLE: rootValue (Leaf "foo") = "foo"
 -}
rootValue :: FBTree a -> a

rootValue (Leaf x) = x
rootValue (Node _ x _) = x
```

Tree Height

The **height** of a tree is the length of the *longest* path from the root to a leaf.

For instance, the following tree has height 4:



Tree Height in Haskell

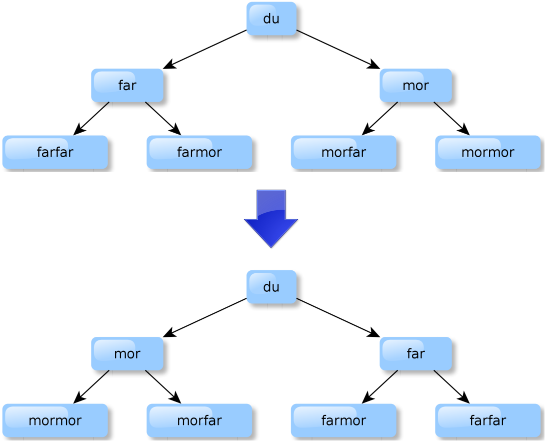
```haskell
{- height t
   PRE: true
   POST: the height of t
   EXAMPLE: height (Leaf "foo") = 0
 -}
height :: FBTree a -> Int
```

Tree Height in Haskell

```haskell
{- height t
   PRE: true
   POST: the height of t
   EXAMPLE: height (Leaf "foo") = 0
 -}
height :: FBTree a -> Int

-- VARIANT: size of t
height (Leaf _) = 0
height (Node l _ r) = 1 + max (height l) (height r)
```

Mirror Image

Let's write a function that "mirrors" a full binary tree by (recursively) exchanging left and right subtrees. For instance,



Mirror Image in Haskell

```
{- mirror t
   PRE: true
   POST: the mirror image of t
   EXAMPLE: mirror (Node (Leaf 1) 2 (Leaf 3)) =
                Node (Leaf 3) 2 (Leaf 1)
 -}
mirror :: FBTree a -> FBTree a
```

Mirror Image in Haskell

```haskell
{- mirror t
   PRE: true
   POST: the mirror image of t
   EXAMPLE: mirror (Node (Leaf 1) 2 (Leaf 3)) =
                Node (Leaf 3) 2 (Leaf 1)
 -}
mirror :: FBTree a -> FBTree a

-- VARIANT: size of t
mirror (Leaf x) = Leaf x
mirror (Node l x r) = Node (mirror r) x (mirror l)
```

Pre-, In-, Post-Order Tree Traversal

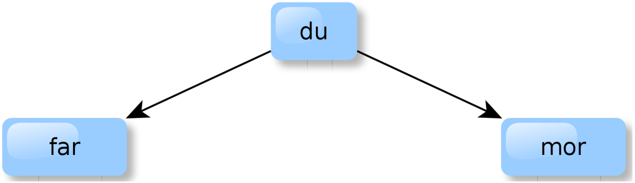
A **traversal** of a data structure is a way of processing each element in the data structure exactly once.

For binary trees, we distinguish:

- **Pre-order traversal**: process the root value, then traverse the left subtree, then traverse the right subtree
- **In-order traversal**: traverse the left subtree, then process the root value, then traverse the right subtree
- **Post-order traversal**: traverse the left subtree, then traverse the right subtree, then process the root value

Pre-, In-, Post-Order Tree Traversal: Example

Suppose we want to list the elements of a `FBTree`. For instance,



- Pre-order traversal: "du", "far", "mor"
- In-order traversal: "far", "du", "mor"
- Post-order traversal: "far", "mor", "du"

In-Order Traversal in Haskell

```
{- listInOrder t
   PRE: true
   POST: the list of elements in t, in order
   EXAMPLE: listInOrder (Node (Leaf 'a') 'b' (Leaf 'c')) =
               "abc"
 -}
listInOrder :: FBTree a -> [a]
```

In-Order Traversal in Haskell

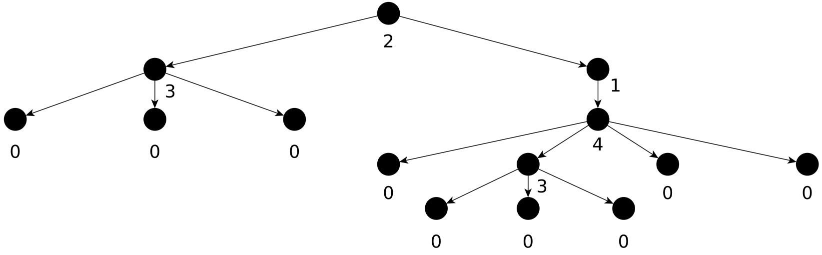
```
{- listInOrder t
   PRE: true
   POST: the list of elements in t, in order
   EXAMPLE: listInOrder (Node (Leaf 'a') 'b' (Leaf 'c')) =
               "abc"
 -}
listInOrder :: FBTree a -> [a]

-- VARIANT: size of t
listInOrder (Leaf x) = [x]
listInOrder (Node l x r) = listInOrder l ++ x : listInOrder r
```

Out-Degree

The **out-degree** of a node is the number of children that the node has.

Thus, in a full binary tree, all nodes have out-degree 0 (leafs) or 2 (inner nodes). However, in general, nodes in a tree may have varying out-degrees.

For instance,



General Trees in Haskell

We already know a container type that can hold a variable (arbitrary but finite) number of elements: lists. Let's use lists to define general trees:

```
data Tree a = Node a [Tree a]
```

Note that no separate constructor for leafs is required. Leafs simply have an empty list of subtrees.

For instance,

```
> Node 1 []
```

```
> Node 1 [Node 2 [], Node 3 [], Node 4 []]
```

Recursion over General Trees: Examples

```
{- treeSum t
   PRE: true
   POST: the sum of elements in t
   EXAMPLE: treeSum (Node 1 [Node 2 []]) = 3
 -}
treeSum :: Tree Int -> Int
```

Recursion over General Trees: Examples

```
{- treeSum t
   PRE: true
   POST: the sum of elements in t
   EXAMPLE: treeSum (Node 1 [Node 2 []]) = 3
 -}
treeSum :: Tree Int -> Int
treeSum (Node x ts) = x + treeListSum ts

{- treeListSum ts
   PRE: true
   POST: the sum of elements of trees in ts
   EXAMPLE: treeListSum [Node 1 [], Node 2 []] = 3
 -}
treeListSum :: [Tree Int] -> Int
treeListSum [] = 0
treeListSum (t:ts) = treeSum t + treeListSum ts
```

Lists Are (a Special Case of) Trees

A list is (essentially) a tree where each inner node has out-degree 1.