Session 8

Recursive data types

Introduction

We have seen:

- User-defined data types, with functions defined by pattern matching (session 3)
- Recursively defined functions over lists (session 7)

This session:

- Recursively defined types (of which lists are a special case with built-in syntactic support) with which we can express tree-like structures
- Functions over these types naturally use recursion
- We can bundle some common recursive forms
- Case study of manipulating abstract syntax

Trees

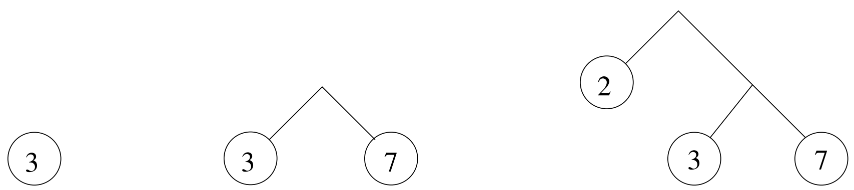
Trees with data in their leaves

A tree of integers:

```
data LTree = Leaf Int | Branch LTree LTree
    deriving Show
```

Some examples: **interpreter**

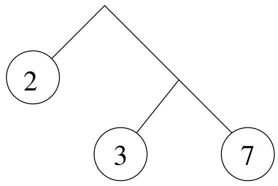
```
Leaf 3
Branch (Leaf 3) (Leaf 7)
Branch (Leaf 2) (Branch (Leaf 3) (Leaf 7))
```



Defining functions on trees

```
data LTree = Leaf Int | Branch LTree LTree
    deriving Show

sumLTree :: LTree -> Int
sumLTree (Leaf x) = x
sumLTree (Branch l r) = sumLTree l + sumLTree r
```



Polymorphism and recursion

Trees with anything in their leaves:

```
data LTree a = Leaf a | Branch (LTree a) (LTree a)
    deriving Show
```

A general function on trees:

```
sumLTree :: Num a => LTree a -> a
sumLTree (Leaf x) = x
sumLTree (Branch l r) = sumLTree l + sumLTree r
```

Another list type:

```
data List a = Nil | Cons a (List a)
```

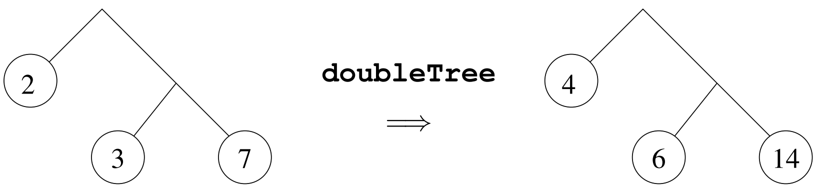
Another recursive function on trees

Recall general trees:

```
data LTree a = Leaf a | Branch (LTree a) (LTree a)
    deriving Show
```

Doubling each element in a tree of numbers:

```
doubleTree :: LTree Int -> LTree Int
doubleTree (Leaf x) = Leaf (2*x)
doubleTree (Branch l r) = Branch (doubleTree l) (doubleTree r)
```



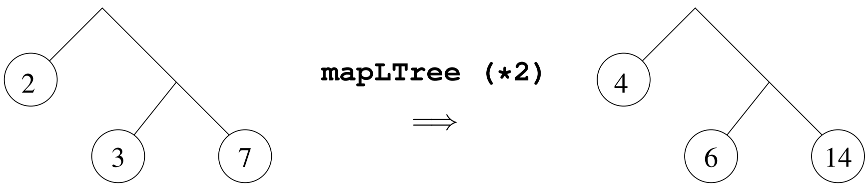
Generalizing to a higher-order function on trees

Recall general trees:

```
data LTree a = Leaf a | Branch (LTree a) (LTree a)
    deriving Show
```

Applying an arbitrary function to each value in a tree:

```
mapLTree :: (a -> b) -> LTree a -> LTree b
mapLTree f (Leaf x) = Leaf (f x)
mapLTree f (Branch l r) = Branch (mapLTree f l) (mapLTree f r)
```



Other tree types

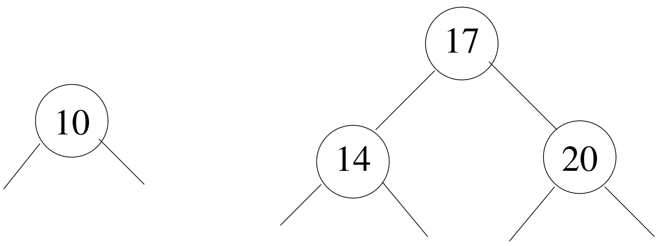
Trees with data in the nodes

A tree parameterized by the data in branched (nodes):

```
data NTree a = Empty | Node a (NTree a) (NTree a)
    deriving Show
```

Some examples of trees of integers: interpreter

```
Node 10 Empty Empty
Node 17 (Node 14 Empty Empty) (Node 20 Empty Empty)
```

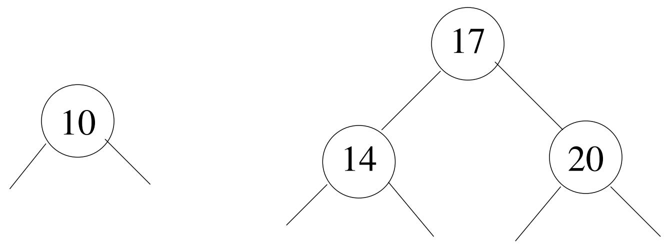


Search trees

If we keep these trees ordered, we can use them as search trees:

```
member :: Ord a => a -> NTree a -> Bool
member x Empty = False
member x (Node k l r)
  | x < k = member x l
```

```
  | x > k = member x r
  | otherwise = True
```



Various kinds of trees

Trees with data in the leaves:

```
data LTree a = Leaf a | Branch (LTree a) (LTree a)
```

Trees with data in the nodes:

```
data NTree a = Empty | Node a (NTree a) (NTree a)
```

Trees with both:

```
data LNTree a b
    = Empty
    | Leaf a
    | Node (LNTree a b) b (LNTree a b)
```

Multiway trees (“Rose trees”):

```
data RTree a = RNode a [RTree a]
```

Other hierarchical data

XML documents

XML (including XHTML, SVG, etc) can be internally represented with:

```
data Element = Element Name [Attribute] [Content]
type Name = String
type Attribute = (Name, String)
data Content = Text String | Child Element
```

For example, an XHTML fragment

```
<p><img src="warning.png"/> A paragraph with <em>emphasis</em>.</p>
```

could be represented by

interpreter

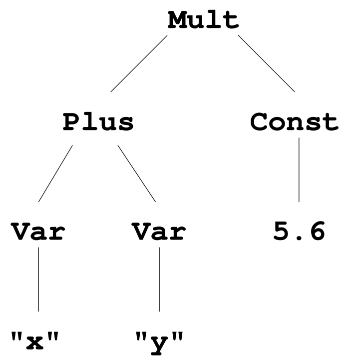
```
Element "p" [] [
    Child (Element "img" [("src", "warning.png")] []),
    Text " A paragraph with ",
    Child (Element "em" [] [Text "emphasis"]),
    Text "."]
```

Abstract syntax

```
module Expr where

type Variable = String

data Expr
    = Const Double
    | Var Variable
    | Minus Expr Expr
    | Plus Expr Expr
    | Mult Expr Expr
    deriving (Eq, Show)
```



For example, an expression like "($x + y$) * 5.6" is represented by the value

interpreter

```
Mult (Plus (Var "x") (Var "y")) (Const 5.6)
```

Example: tautology testing

Propositions

Consider propositional formulae, such as:

- $A \wedge \neg A$
- $(A \wedge B) \Rightarrow A$
- $A \Rightarrow (A \wedge B)$
- $(A \wedge (A \Rightarrow B)) \Rightarrow B$

Task

Test whether a propositional formula is a *tautology*, that is, is true for any possible substitution of Boolean values for the variables A, B, etc.

Truth tables

A formula is a tautology if all entries in its truth table are true:

A	$A \land \neg A$
F	F
T	F

A	B	$(A \land B) \Rightarrow A$
F	F	T
F	T	T
T	F	T
T	T	T

A	B	$A \Rightarrow (A \land B)$
F	F	T
F	T	T
T	F	F
T	T	T

A	B	$(A \land (A \Rightarrow B)) \Rightarrow B$
F	F	T
F	T	T
T	F	T
T	T	T

Representing propositional formulae

We define a new type to represent propositional formulae, with a constructor for variables and another for each of the logical connectives $\neg$, $\land$ and $\Rightarrow$:

```
data Prop a
    = Var a
    | Not (Prop a)
    | And (Prop a) (Prop a)
    | Imply (Prop a) (Prop a)
    deriving (Show)
```

Our type is parameterized by the type of variables, so we can use whatever variable type we want. (This also allows us to define a counterpart of `map` for this type, which will be useful.)

Sample `Prop` values

The above example propositional formulae

1. $A \land \neg A$
2. $(A \land B) \Rightarrow A$
3. $A \Rightarrow (A \land B)$
4. $(A \land (A \Rightarrow B)) \Rightarrow B$

Can be represented by the following values, using characters for variables:

```
p1, p2, p3, p4 :: Prop Char
p1 = And (Var 'A') (Not (Var 'A'))
p2 = Imply (And (Var 'A') (Var 'B')) (Var 'A')
p3 = Imply (Var 'A') (And (Var 'A') (Var 'B'))
p4 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

Testing

To test the function we are about to write, we give a list of inputs and expected outputs:

```
tautologyTests :: [(Prop Char, Bool)]
tautologyTests = [
     (p1, False), (p2, True),
     (p3, False), (p4, True)]
```

This general-purpose function will report inputs for which the function's output does not match what we expected:

```
failures :: Eq b => (a -> b) -> [(a, b)] -> [(a, b, b)]
failures f xys = [(x, y, f x) | (x, y) <- xys, f x /= y]
```

It this returns `[]`, all the tests passed.

Plan

1. Substitutions of Booleans for variables

   ```
   type Subst a = [(a, Bool)]
   ```

2. Evaluate a proposition with a given substitution

   ```
   eval :: Ord a => Subst a -> Prop a -> Bool
   ```

3. All possible substitutions for a proposition

   ```
   substs :: Ord a => Prop a -> [Subst a]
   ```

4. A proposition is a tautology if it evaluates to **`True`** for all substitutions:

   ```
   tautology p = and [eval s p | s <- substs p]
   ```

Evaluating a formula with a substitution

We break this down into two steps:

1. Replace each variable in the formula with the corresponding Boolean value. This is a **`map`**-like operation.

2. Evaluate the formula, now with Boolean values instead of variables.

$$\begin{tikzcd} \texttt{Prop a} \arrow[d, "\texttt{mapProp (value s)}"] & \texttt{a} \arrow[d, "\texttt{value s}"] \\ \texttt{Prop Bool} \arrow[d, "\texttt{evalBool}"] & \texttt{Bool} \\ \texttt{Bool} & \end{tikzcd}$$

Replacing variables with values

Replacing variables is a general **map**-like operation:

```
mapProp :: (a -> b) -> Prop a -> Prop b
mapProp f (Var v) = Var (f v)
mapProp f (Not p) = Not (mapProp f p)
mapProp f (And p q) = And (mapProp f p) (mapProp f q)
mapProp f (Imply p q) = Imply (mapProp f p) (mapProp f q)
```

It remains to say how we replace a single variable:

```
value :: Eq a => Subst a -> a -> Bool
value s v = fromMaybe False (lookup v s)
```

Here we have used the **fromMaybe** function (developed in the session 3 exercises) from **Data.Maybe** and the standard **lookup** function.

Evaluating propositions

A formula with variables replaced with Booleans is simple to evaluate:

```
evalBool :: Prop Bool -> Bool
evalBool (Var b) = b
evalBool (Not p) = not (evalBool p)
evalBool (And p q) = evalBool p && evalBool q
evalBool (Imply p q) =
    not (evalBool p) || evalBool q
```

Combining this with replacement of variables yields evaluation with respect to a substitution:

```
eval :: Ord a => Subst a -> Prop a -> Bool
eval s p = evalBool (mapProp (value s) p)
```

Getting all the substitutions

We also break this down into steps:

1. Get the set of variables found in formula

   ```
   vars :: Ord a => Prop a -> Set a
   ```

2. Get the list of elements

   Data.Set

   ```
   Set.elems :: Set a -> [a]
   ```

3. get all the Boolean substitutions for a list

   ```
   bools :: [a] -> [Subst a]
   ```

Together, these yield all the substitutions for a formula:

```
substs :: Ord a => Prop a -> [Subst a]
substs p = bools (Set.elems (vars p))
```

The set of variables in a formula

To extract the set of variables in a formula, we use recursion over the structure of the formula.

```
vars :: Ord a => Prop a -> Set a
vars (Var v) = Set.singleton v
vars (Not p) = vars p
vars (And p q) = Set.union (vars p) (vars q)
vars (Imply p q) = Set.union (vars p) (vars q)
```

- A variable yields a singleton set.
- For the binary connectives, we take the union of the sets of variables in the two arguments.

Boolean substitutions

The final part takes a list and returns all the possible Boolean substitutions of that list.

- If the input list has n elements, there will be 2^n substitutions.
  - For 0 elements there will be 1 substitution.
  - For non-empty lists, we combine each possible replacement for the first value with each substitutions for the rest.
- Each substitution with have length n, with the initial value of each pair matching the corresponding value in the input list.

```
bools :: [a] -> [Subst a]
bools [] = [[]]
bools (x:xs) = [(x, b):s | b <- [False, True], s <- bools xs]
```

Exercises

Several of these are optional extras, and can be skipped unless you are looking for extra challenges.

1. Consider the type of "leaf trees":

   ```
   data LTree a = Leaf a | Branch (LTree a) (LTree a)
       deriving (Show)
   ```

   Define some values of this type to use in testing functions on it.

   Write functions to

   (a) return the size (number of leaves) of a leaf tree.
   (b) return the depth of a leaf tree.
   (c) return the list of leaf values of a tree.
   (d) return the mirror image of an input leaf tree.

2. The functions in the previous question all have a similar structure, which we can express with a pair of higher-order functions.

(a) Generalize the **sumLTree** function from the lecture to obtain a higher-order function

```
foldLTree :: (a -> a -> a) -> LTree a -> a
```

that reduces a tree to a summary value by combining the values of branches using the supplied function.

(b) Redefine **sumLTree** using **foldLTree**.

(c) Define the function returning the size of a leaf tree as a composition of the form **sumLTree . mapLTree** g for some suitable function g. (You might also find the standard function **const** useful here.)

(d) Redefine each of your functions from the first question as a composition of the form **foldLTree** f **. mapLTree** g for some functions f and g (different in each case). (You might also find the standard function **flip** useful here.)

3. Write a function

```
printElement :: Element -> String
```

to convert a value of the XML **Element** type from the lecture to its string representation. (optional extra challenge: use XML escaped characters to handle characters that cause problems in strings.)

4. Extend the tautology checker to support the use of logical disjunction ($\vee$) and equivalence ($\Leftrightarrow$) in propositions.

5. (optional extra challenge) The following type can be used to describe a path from the root of a leaf tree to one of its leaves:

```
type Path = [Step]
data Step = L | R
    deriving Show
```

Write a function

```
paths :: LTree Bool -> [Path]
```

That returns the list of paths from the root to leaves containing the value **True**.

6. (optional extra challenge)

(a) Write a function

```
foldProp :: (a -> a -> a) -> Prop a -> a
```

to reduce a proposition to a summary value, using the supplied function to combine the values of binary operations (**And**, etc).

(b) Use **foldProp** and **mapProp** to write a function to count the number of variables in a proposition, including repetitions. For example, $A \wedge \neg A$ has 2 variables.

(c) Use **foldProp** and **mapProp** to obtain a one-line definition of the **vars** function on propositions.