currying

Built-in curry and uncurry:

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
prompt> member
member :: Eq t => (t, [t]) -> Bool
prompt> curry member
```

```
curry member :: Eq t => t -> [t] -> Bool
prompt> memberC = curry member
prompt> memberC 1 [1, 2, 3]
True
```

currying

Built-in curry and uncurry:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
prompt> sum'
sum' :: Num a => a -> a -> a
prompt> uncurry sum'
```

```
uncurry sum' :: Num c => (c, c) -> c
prompt> sum'' = uncurry sum'
prompt> sum'' (2, 3)
5
```

functions and operators

Any two-parameter curried function can be used as an operator:

```
prompt> elem 3 [2,3,4]
True
prompt> 3 `elem` [2,3,4]
True
```

Any operator can be used as a function:

```
prompt> (*) 2 3
6
prompt> (:) 2 [3,4]
[2,3,4]
```

sections

```
prompt> map (+2) [1,2,3]

[3,4,5]
prompt> map (2+) [1,2,3]

[3,4,5]
prompt> map (/2) [1,2,3]

[0.5,1.0,1.5]
prompt> map (2/) [1,2,3]

[2.0,1.0,0.6666666666666666]
prompt> map (:[42]) [1,2,3]

[[1,42],[2,42],[3,42]]
prompt> map (42:) [[1],[2,3],[4,5]]

[[42,1],[42,2,3],[42,4,5]]
```

type synonyms

We can give existing types new names. Syntax:

$$\texttt{type}\ \mathit{NewType} = \mathit{OldType}$$

NewType becomes an alias (a synonym) for the **existing** type *OldType*.

```
type String = [Char]


type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

user defined datatypes

General Syntax:

```
data NewType =
        Cons1 Type1
      | Cons2 Type2
        ...
      | ConsN TypeN
```

- Defines a **new** type called `NewType`.
- `Type1,...,TypeN` are previously defined types.
- `Cons1,...,ConsN` are constructors. They are used to create a value of `NewType` type.
- `Type` is omitted if a constructor does not need any argument (such constructors are called constants).

enumerated types

All constructors are constants (no argument).
Example:

```
data Colour = Red | Green | Blue
c = Red

colorName Red = "red"
colorName Green = "green"
colorName Blue = "blue"


prompt> :t c

c :: Colour
prompt> :t colorName

colorName :: Colour -> [Char]
prompt> colorName Blue

"blue"
```

variant types

Create union of different types:

```
data Text = Letter Char | Word [Char]

textLen (Letter _) = 1
textLen (Word w) = length w


prompt> :t textLen

textLen :: Text -> Int
```

recursive types

A datatype can be recursive, of course: e.g. a linked list (ignore the deriving business for now)

```
data LList = Nil | Node (Int, LList) deriving Show

llist = Node (1, Node (2, Node(3, Nil)))

llistLen Nil = 0
llistLen (Node (_,rest)) = 1 + llistLen rest

llistLen :: LList -> Int
```

recursive types

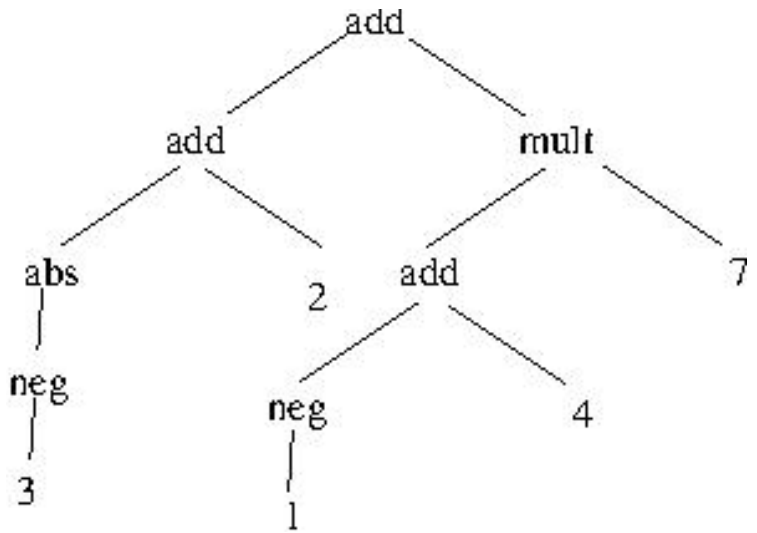
What about a polymorphic linked list?

```
data LList a = Nil | Node (a, LList a) deriving Show

llist1 = Node (1, Node (2, Node(3, Nil)))
llist2 = Node ('1', Node ('2', Node ('3', Nil)))

llistLen Nil = 0
llistLen (Node (_,rest)) = 1 + llistLen rest

llistLen :: LList t -> Int
```

recursive types

Example: Tree representation of simple mathematical expressions.

$$(|-3|+2)+((-1)+4)*7)$$



What is the datatype we need?

```
data MathExpr =
    Leaf Int
  | Unary (Int -> Int, MathExpr)
  | Binary (Int -> Int -> Int, MathExpr, MathExpr)
```

recursive types

The tree in the figure:

```
t = Binary((+),
           Binary((+),
                  Unary(abs,
                        Unary ((0-),
                               Leaf 3)),
                  Leaf 2),
           Binary((*),
                  Binary((+),
                         Unary((0-),
                               Leaf 1),
                         Leaf 4),
                  Leaf 7))
```

recursive types

Evaluating the tree:

```
eval (Leaf v) = v
eval (Unary (f,t)) = f (eval t)
eval (Binary (f,l,r)) = f (eval l) (eval r)
```

Type of eval?

curried type/value constructors

Type/value constructors are simply functions. And so they may be curried.

Value constructor:

```
data BTree a = Empty | Node (a, BTree a, BTree a)
```

`BTree` is a type constructor. `Empty` and `Node` are value constructors.

Curried value constructor:

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

curried type/value constructors

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

Example:

```
t = (Node 1 (Node 2 (Node 3 Empty Empty)
                    (Node 4 Empty Empty))
            (Node 5 (Node 6 Empty Empty)
                    (Node 7 Empty
                            (Node 8 Empty Empty))))
prompt> :type t

t :: BTree Integer
```

curried value constructors

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

prompt> :t Node "a"

Node "a" :: BTree [Char] -> BTree [Char] -> BTree [Char]

prompt> :t Node "a" Empty

Node "a" Empty :: BTree [Char] -> BTree [Char]

prompt> :t Node "a" Empty (Node "a" Empty Empty)

Node "a" Empty (Node "a" Empty Empty) :: BTree [Char]
```

curried value constructors

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

tree2list :: (BTree a) -> [a]
tree2list Empty = []
tree2list (Node v l r) =
    (tree2list l) ++ [v] ++ (tree2list r)
```

A better solution?

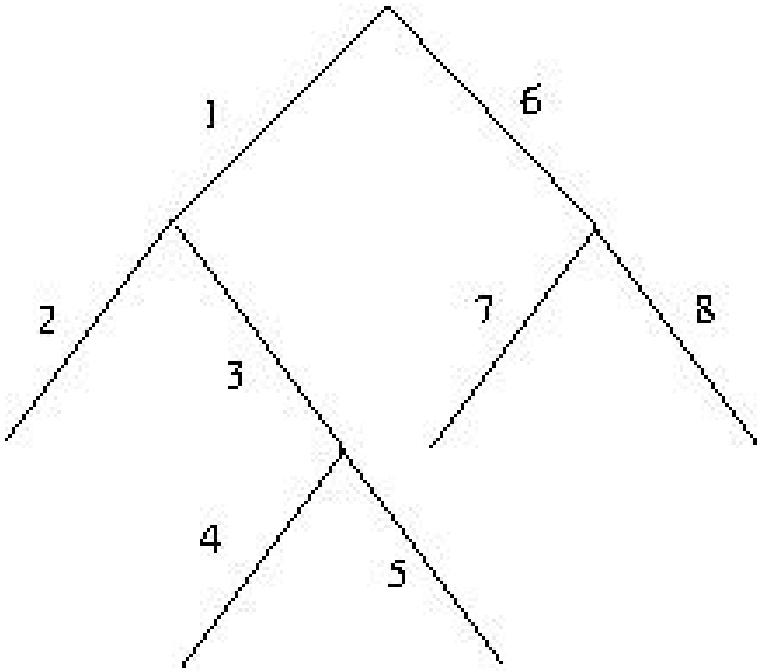
curried value constructors

(Ignore the "deriving" business for now).

```
data Tree a = Leaf a |
              Internal (Tree a) (Tree a) a deriving Show

makeLeafForest = map Leaf
makeForest = map $ Internal (Leaf 42) (Leaf 24)

prompt> makeLeafForest [1,2,3,4,5]
[Leaf 1,Leaf 2,Leaf 3,Leaf 4,Leaf 5]
Prelide> makeForest [1,2,3,4,5]
[Internal (Leaf 42) (Leaf 24) 1,
 Internal (Leaf 42) (Leaf 24) 2,
 Internal (Leaf 42) (Leaf 24) 3,
 Internal (Leaf 42) (Leaf 24) 4,
 Internal (Leaf 42) (Leaf 24) 5]
```

mutually recursive types

Example: a tree with labeled branches:




```
data Tree a = Empty
            | Node (Branch a) (Branch a)
data Branch a = Branch a (Tree a)
```

mutually recursive types

The tree in the figure:

```
lt =
   Node(Branch 1
                (Node (Branch 2 Empty)
                      (Branch 3 (Node (Branch 4 Empty)
                                      (Branch 5 Empty)))))
        (Branch 6
                (Node (Branch 7 Empty)
                      (Branch 8 Empty)))
```

mutually recursive types

```
data Tree a = Empty
            | Node (Branch a) (Branch a)
data Branch a = Branch a (Tree a)
```

Return the list of branch labels, in order:

```
listTree Empty = []
listTree (Node l r) = (listBranch l) ++ (listBranch r)
listBranch (Branch b t) = b : listTree t
```

```
listTree :: Tree a -> [a]
```

```
listBranch :: Branch a -> [a]
```

recursive types

1. A powerful tool for constructing new types.
2. The structure of the datatype suggests the structure of the recursive function on the datatype.

infix value constructors

```
data Rational = Integer :/ Integer deriving Show

r1 = 1 :/ 2
r2 = 1 :/ 3

addRat (x0 :/ y0) (x1 :/ y1) =
    (x0 * y1 + x1 * y0) :/ (y0 * y1)

prompt> r1 `addRat` r2
5 :/ 6
```

Thus `(:) :: a -> [a] -> [a]` is no different.

curried type constructors

```
data Either a b = Left a | Right b
```

```
prompt> :kind Either
Either :: * -> * -> *
```

But let's not go there...