Haskell lets us create new datatypes:
\[
data \text{Datatype } a_1 \ldots a_n = \text{constr}_1 \mid \ldots \mid \text{constr}_m
\]
where
1. \text{Datatype} is the name of a new type constructor
2. \(a_1, \ldots, a_n\) are type variables representing the arguments of \text{Datatype}
3. \(\text{constr}_1, \ldots, \text{constr}_m\) are the different ways in which we can create new elements of the new datatype.

Each \text{constr} is of the form
\[
\text{Name} \ \text{type}_1 \ldots \text{type}_r
\]
where \text{Name} is a new name beginning with a capital letter.

The following definition introduces a new type \text{Day} with elements \text{Sun}, \text{Mon}, \text{Tue},\ldots:
\[
data \text{Day} = \text{Sun}\mid \text{Mon}\mid \text{Tue}\mid \text{Wed}\mid \text{Thu}\mid \text{Fri}\mid \text{Sat}
\]

Simple functions manipulating elements of type \text{Day} can be defined using pattern matching:
\[
\begin{align*}
\text{what\_shall\_I\_do } \text{Sun} &= \text{"relax"} \\
\text{what\_shall\_I\_do } \text{Sat} &= \text{"go shopping"} \\
\text{what\_shall\_I\_do } \_ &= \text{"go to work"}
\end{align*}
\]

We can represent temperatures either using centigrade or fahrenheit:
\[
data \text{Temp} = \text{Centigrade Float} \mid \text{Fahrenheit Float}
\]
deriving Show
\[
\begin{align*}
\text{freezing} & \quad :: \text{Temp} \rightarrow \text{Bool} \\
\text{freezing (Centigrade temp)} &= \text{temp} \leq 0.0 \\
\text{freezing (Fahrenheit temp)} &= \text{temp} \leq 32.0
\end{align*}
\]

We add the syntax deriving Show so that we can print out elements of the datatype:
\[
\begin{align*}
> \text{Centigrade 66} \\
\text{Centigrade 66.0}
\end{align*}
\]
Recursive Datatypes

- We can define recursive datatypes.
- In fact, we can use datatypes to define our own kind of lists!
- Here’s a list of integers:
  ```haskell
data IntList =
    IntCons Int IntList |
    IntNil
deriving Show
```
- As usual, a list is either Nil or a Cons cell consisting of an integer and the rest of the list.
- Here’s the list \([5,6]\) in our new representation:
  ```haskell
  IntCons 5 (IntCons 6 IntNil)
  ```

Polymorphic Recursive Datatypes

- Here’s a recursive definition of a polymorphic list:
  ```haskell
data List a =
    Cons a (List a) |
    Nil
deriving Show
```
- We can define our own versions of head and tail:
  ```haskell
  hd Nil = error "Head of Nil"
  hd (Cons a _) = a
  tl Nil = error "Tail of Nil"
  tl (Cons _ b) = b
  ```
- And we can construct lists of arbitrary types and take them apart:
  ```plaintext
  > hd (tl (Cons 1 (Cons 2 Nil)))
  2
  > hd (tl (Cons "hello" (Cons "bye" Nil)))
  "bye"
  ```

Polymorphic Binary Tree

- Here’s the definition of a binary tree with data in each leaf and internal node:
  ```haskell
data Tree a = Leaf a |
    Node (Tree a) a (Tree a)
deriving Show
```
- For example, here’s a binary search tree with the elements \(f, 10, 12, 15, 16\):
  ```haskell
  Node
    (Leaf 5)
  10
    (Node
      (Leaf 12)
    15
    (Leaf 16)
  )
```

Polymorphic Binary Search Tree

- Here’s a function that looks up a value in a tree:
  ```haskell
treemem :: Ord a => Tree a -> a -> Bool
  treemem (Leaf v) x = x == v
  treemem (Node l v r) x
    | x == v = True
    | x < v = treemem l x
    | x > v = treemem r x
```
- Examples:
  ```plaintext
  > let t = Node (Leaf 5) 10 (Node (Leaf 12) 15 (Leaf 16))
  > treemem t 16
  True
  > treemem t 5
  True
  > treemem t 1
  False
  ```
Homework 1

- Write the function `depth` which calculates the depth of a tree, `leaves` which returns the leaves of a tree, and `inorder` which returns a list of the nodes of the tree in inorder:

  ```
  depth :: Tree a -> Int
  leaves :: Tree a -> [a]
  inorder :: Tree a -> [a]
  ```

Examples:

````
> let t1 = Node (Leaf 5) 10 (Leaf 15)
> let t2 = Node (Leaf 5) 10 (Node (Leaf 12) 15 (Leaf 16))
> depth t1
2
> depth t2
3
> leaves t1
[5,15]
> leaves t2
[5,12,16]
> inorder t1
[5,10,15]
> inorder t2
[5,10,12,15,16]
```

Homework 2

- Here’s a datatype for arithmetic expressions:

  ```
  data Expr = Val Int
  | Add Expr Expr
  | Sub Expr Expr
  | Mul Expr Expr
  | Div Expr Expr
  | Neg Expr
  deriving Show
  ```

- Write a function `eval e` which evaluates an arithmetic expression `e`:

  ```
  eval :: Expr -> Int
  ```

Examples:

````
> eval (Val 5)
5
> eval (Add (Val 6) (Val 5))
11
> eval (Add (Mul (Val 7) (Val 5)) (Val 7))
42
```