Types and Classes in Haskell
Outline

- Data Types
- Class and Instance Declarations
Defining New Types

Three constructs for defining types:

1. **data** - Define a new data type from scratch, describing its constructors

2. **type** - Define a synonym for an existing type (like typedef in C)

3. **newtype** - A restricted form of data that is more efficient when it fits (if the type has exactly one constructor with exactly one field inside it). Used for defining “wrapper” types
Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

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

Bool is a new type, with two new values False and True.

- The two values False and True are called the constructors for the data type Bool.
- Type and constructor names must begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

More examples from standard Prelude:

```hs
data () = () -- unit datatype
data Char = ... | 'a' | 'b' | ...
```
Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes, No, Unknown]

flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```
Another example:

```haskell
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun

next :: Weekday -> Weekday
next Mon = Tue
next Tue = Wed
next Wed = Thu
next Thu = Fri
next Fri = Sat
next Sat = Sun
next Sun = Mon

workDay :: Weekday -> Bool
workDay Sat = False
workDay Sun = False
workDay _ = True
```

Constructors construct values, or serve as patterns
Constructors with Arguments

The constructors in a data declaration can also have parameters, e.g.:

\[
\text{data Shape} = \text{Circle Float} \mid \text{Rect Float Float}
\]

we can define:

\[
\begin{align*}
\text{square} \quad &:: \quad \text{Float} \to \text{Shape} \\
\text{square} \ n \quad &= \quad \text{Rect} \ n \ n \\
\text{area} \quad &:: \quad \text{Shape} \to \text{Float} \\
\text{area} \ (\text{Circle} \ r) \quad &= \quad \pi \times r^2 \\
\text{area} \ (\text{Rect} \ x \ y) \quad &= \quad x \times y
\end{align*}
\]

- Shape has values of the form Circle $r$ where $r$ is a float, and Rect $x$ $y$ where $x$ and $y$ are floats.
- Circle and Rect can be viewed as functions that construct values of type Shape:

\[
\begin{align*}
\text{Circle} \quad &:: \quad \text{Float} \to \text{Shape} \\
\text{Rect} \quad &:: \quad \text{Float} \to \text{Float} \to \text{Shape}
\end{align*}
\]
Another example:

```haskell
data Person = Person Name EyeColor Age
type Name = String
data EyeColor = Brown | Blue | Green
type Age = Int
```

With just one constructor in a data type, often constructor is named the same as the type (cf. Person). Now we can do:

```haskell
let x = Person “Jerry” Green 12
   y = Person “Tom” Blue 16
in ...
```

Quiz: What are the types of the constructors Blue and Person?

```
Blue :: EyeColor
Person :: Name -> EyeColor -> Age -> Person
```
Pattern Matching

name (Person n _ _) = n

oldBlueEyes (Person _ Blue a) | a > 100 = True
oldBlueEyes (Person _ _ _) = False

> let yoda = Person “Yoda” Blue 999
  in oldBlueEyes yoda
True

findPrsn n (p@(Person m _ _):ps)
  | n == m = p
  | otherwise = findPrsn n ps

> findPrsn “Tom”
  [Person “Yoda” Blue 999, Person “Tom” Brown 7]
Person “Tom” Brown 7
Parameterized Data Declarations

Not surprisingly, data declarations themselves can also have parameters. For example, given

```haskell
data Pair a b = Pair a b
```

we can define:

```haskell
x = Pair 1 2
y = Pair "Howdy" 42

first :: Pair a b -> a
first (Pair x _) = x

apply :: (a -> a')->(b -> b') -> Pair a b -> Pair a' b'
apply f g (Pair x y) = Pair (f x) (g y)
```
Another example:
Maybe type holds a value (of any type) or holds nothing

```haskell
data Maybe a = Nothing | Just a
```

`a` is a type parameter, can be bound to any type

```haskell
Just True :: Maybe Bool
Just "x" :: Maybe [Char]
Nothing :: Maybe a
```

we can define:

```haskell
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```
Type Declarations

A new name for an existing type can be defined using a **type** declaration.

```
type String = [Char]
```

String is a synonym for the type [Char].

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define:

```
origin :: Pos
origin   = (0,0)

left :: Pos → Pos
left (x,y) = (x-1,y)
```
Like function definitions, type declarations can also have parameters. For example, given

\begin{center}
\textbf{type Pair a = (a,a)}
\end{center}

we can define:

\begin{align*}
\text{mult} & : \text{Pair Int} \to \text{Int} \\
\text{mult} \ (m,n) & = m*n \\
\text{copy} & : \text{a} \to \text{Pair a} \\
\text{copy} \ x & = (x,x)
\end{align*}
Type declarations can be nested:

```haskell
type Pos   = (Int,Int)

type Trans = Pos -> Pos
```

However, they cannot be recursive:

```haskell
type Tree = (Int,[Tree])
```
Recursive Data Types
New types can be declared in terms of themselves. That is, data types can be recursive.

```haskell
data Nat = Zero | Succ Nat
```

Nat is a new type, with constructors Zero :: Nat and Succ :: Nat -> Nat.

A value of type Nat is either Zero, or of the form Succ n where n :: Nat. That is, Nat contains the following infinite sequence of values:

- Zero
- Succ Zero
- Succ (Succ Zero)
  ...

Example function:

```haskell
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```
Parameterized Recursive Data Types - Lists

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

sum :: List Int -> Int
sum Nil = 0
sum (Cons x xs) = x + sum xs

> sum Nil
0
> sum (Cons 1 (Cons 2 (Cons 2 Nil)))
5
Arithmetic Expressions

Consider a simple form of expressions built up from integers using addition and multiplication.
Using recursion, a suitable new type to represent such expressions can be declared by:

```haskell
data Expr = Val Int
         | Add Expr Expr
         | Mul Expr Expr
```

For example, the expression on the previous slide would be represented as follows:

```
Add (Val 1) (Mul (Val 2) (Val 3))
```
Using recursion, it is now easy to define functions that process expressions. For example:

size        :: Expr → Int
size (Val n) = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval        :: Expr → Int
eval (Val n) = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
Note:

- The three constructors have types:
  
  \[
  \begin{align*}
  \text{Val} & : \text{Int} \rightarrow \text{Expr} \\
  \text{Add} & : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr} \\
  \text{Mul} & : \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr}
  \end{align*}
  \]

- Many functions on expressions can be defined by replacing the constructors by other functions using a suitable `fold` function. For example:

  \[
  \text{eval} = \text{fold \text{id} (+) (\ast)}
  \]
Trees
A binary Tree is either Tnil, or a Node with a value of type a and two subtrees (of type Tree a)

```haskell
data Tree a = Tnil | Node a (Tree a) (Tree a)
```

Find an element:

```haskell
treeElem :: (a -> Bool) -> Tree a -> Maybe a
treeElem p Tnil = Nothing
treeElem p t@(Node v left right)
  | p v = Just v
  | otherwise = treeElem p left `combine` treeElem p right
where combine (Just v) r = Just v
     combine Nothing r = r
```

Compute the depth:

```haskell
depth Tnil = 0
depth (Node _ left right) = 1 + (max (depth left) (depth right))
```
About Folds

A fold operation for Trees:

```haskell
treeFold :: t -> (a -> t -> t -> t) -> Tree a -> t

treeFold f g Tnil = f

treeFold f g (Node x l r)
  = g x (treeFold f g l) (treeFold f g r)
```

How? Replace all \texttt{Tnil} constructors with \texttt{f}, all \texttt{Node} constructors with \texttt{g}.

```haskell
> let tt = Node 1 (Node 2 Tnil Tnil)
    (Node 3 Tnil (Node 4 Tnil Tnil))

> treeFold 1 (\x y z -> 1 + max y z) tt
4

> treeFold 1 (\x y z -> x * y * z) tt
24

> treeFold 0 (\x y z -> x + y + z) tt
10
```
Deriving

- Experimenting with the above definitions will give you many errors
- Data types come with no functionality by default, you cannot, e.g., compare for equality, print (show) values etc.
- Real definition of Bool

```haskell
data Bool = False | True
    deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

- A few standard type classes can be listed in a `deriving` clause
- Implementations for the necessary functions to make a data type an instance of those classes are generated by the compiler
- `deriving` can be considered a shortcut, we will discuss the general mechanism later
Exercises

(1) Using recursion and the function add, define a function that multiplies two natural numbers.

(2) Define a suitable function fold for expressions, and give a few examples of its use.

(3) A binary tree is complete if the two sub-trees of every node are of equal size. Define a function that decides if a binary tree is complete.
Outline

- Data Types
- Class and Instance Declarations
Type Classes

1. A new class can be declared using the class construct
2. Type classes are classes of types, thus not types themselves

Example:

```haskell
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- Minimal complete definition: (==) and (/=)
  x /= y = not (x == y)
  x == y = not (x /= y)
```

- For a type a to be an instance of the class Eq, it must support equality and inequality operators of the specified types
- Definitions are given in an instance declaration
- A class can specify default definitions
Instance Declarations

class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y   = not (x == y)
  x == y   = not (x /= y)

Let us make Bool be a member of Eq

instance Eq Bool where
  (==) False False  = True
  (==) True True    = True
  (==) _ _          = False

- Due to the default definition, (/=) need not be defined
- deriving Eq would generate an equivalent definition
Showable Weekdays

class Show a where
    showsPrec :: Int -> a -> ShowS -- to control parenthesizing
    show :: a -> String

    showsPrec _ x s = show x ++ s
    show x          = showsPrec 0 x ""

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
instance Show Weekday where
    show Mon = "Monday"
    show Tue = "Tuesday"
    ...

> map show [Mon, Tue, Wed]
["Monday", "Tuesday", "Wednesday"]
Parameterized Instance Declarations

Every list is showable if its elements are

instance Show a => Show [a] where
  show [] = "[]"
  show (x:xs) = "[" ++ show x ++ showRest xs
    where showRest [] = "]"
    showRest (x:xs) = "," ++ show x ++ showRest xs

Now this works:

> show [Mon, Tue, Wed]
"[Monday,Tuesday,Wednesday]"
Showable, Readable, and Comparable Weekdays

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
    deriving (Show, Read, Eq, Ord, Bounded, Enum)

*Main> show Wed
"Wed"
*Main> read "Fri" :: Weekday
Fri
*Main> Sat Prelude.== Sun
False
*Main> Sat Prelude.== Sat
True

*Main> Mon < Tue
True
*Main> Tue < Tue
False
*Main> Wed `compare` Thu
LT
Bounded and Enumerable Weekdays

data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
  deriving (Show, Read, Eq, Ord, Bounded, Enum)

*Main> minBound :: Weekday
  Mon
*Main> maxBound :: Weekday
  Sun
*Main> succ Mon
  Tue
*Main> pred Fri
  Thu
*Main> [Fri .. Sun]
  [Fri,Sat,Sun]
*Main> [minBound .. maxBound] ::
  [Weekday]
  [Mon,Tue,Wed,Thu,Fri,Sat,Sun]