EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 6 due: 11:59 PM, Monday, November 14
- Assignment 7 due: 11:59 PM, Wednesday, December 7

Any Questions?

In-Class Problem Solution

- 32-(11-9) In-Class Problem Solution.pptx

Any Questions?

Type Declarations

In Haskell, 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

Type declarations can be used to make other types easier to read. For example, declare a type for a position on a 2D grid:

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

we can define:

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

left :: Pos -> Pos
left (x,y) = (x-1,y)
```

Type Declaration Parameters

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int -> Int
mult (m,n) = m*n

copy :: a -> Pair a
copy x = (x,x)
```

Nested Type Declarations

Type declarations can be nested:

```
type Pos = (Int,Int)

type Trans = Pos -> Pos
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

Any Questions?

Data Declarations

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

```
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 type Bool.
- Type and constructor names must always begin with an upper-case letter.

Data Declarations

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]
```

creates a list of Answers

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

flips an Answer

Any Questions?

Data Declaration Constructor Parameters

The constructors in a data declaration can also have <u>parameters</u>. For example, given:

```
data Shape = Circle Float
           | Rect Float Float
```

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

```
Circle :: Float -> Shape

Rect :: Float -> Float -> Shape
```

Data Declaration Constructor Parameters

```
data Shape = Circle Float
           | Rect Float Float
```

With this Data Declaration we can define a function that takes a Float and makes a Rectangle with equal sides, i.e., a square:

```
square :: Float -> Shape
square n = Rect n n
```

Data Declaration Constructor Parameters

```
data Shape = Circle Float
           | Rect Float Float
```

Here the area is calculated with different formulas depending on the type of Shape (Circle or Rect):

```
area :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Any Questions?

Data Declaration Parameters

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

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

- What do “Nothing” and “Just” mean?
- We are creating a new Data Type called Maybe, so we are defining its values, Nothing and Just.
- Same as we did for Answer, except this time we have a parameter, a.

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

Maybe is a built-in type that represents values of type a that may either fail or succeed.

Data Declaration Parameters

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

We can use Maybe like this:

```
safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

```
> safediv 4 2
Just 2

> safediv 4 0
Nothing
```

Data Declaration Parameters

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

Or like this:

```
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

```
> safehead [1,2,3]
Just 1

> safehead []
Nothing
```

Any Questions?

Recursive Data Declarations (e.g., List)

- In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.
- Here is a data structure like a list:

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

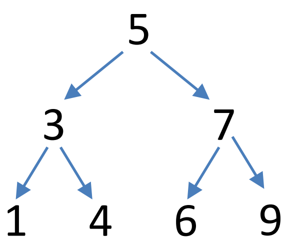
- We can then define length, using List:

```
len :: List a -> Int
len Empty = 0
len (Cons _ xs) = 1 + len xs
```

Recursive Data Declarations (e.g., Tree)

- Here is a data structure for a binary tree:

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



- We can then define size and flatten, using Tree:

```
size :: Tree a -> Int
size Empty = 0
size (Node lhs _ rhs) = size lhs + 1 + size rhs

flatten :: Tree a -> [a]
flatten Empty = []
flatten (Node lhs a rhs) = flatten lhs ++ [a] ++ flatten rhs
```

Any Questions?

Summary

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

```
type Pair a = (a,a)
```

```
type Pos = (Int,Int)
type Trans = Pos -> Pos
```

type declarations can:

- re-name existing types
- have parameters
- be nested
- not be recursive

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

```
data Shape = Circle Float
           | Rect Float Float
```

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

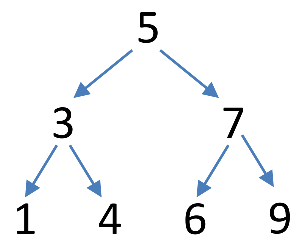
```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

data declarations can:

- create completely new types
- have parameters
- be recursive

Any Questions?

In-Class Problem



- Consider the following type of binary trees:

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

- Let us say that such a tree is balanced if the number of leaves in the left and right subtree of every node differs by at most one.

1. Define a function that returns the number leaves in a tree:

```
leaves :: Tree a -> Int
```

2. Use the leaves function, to define a function:

```
balanced :: Tree a -> Bool
```

   That decides if a binary tree is balanced or not.