Session 3

Structured data types and polymorphism

Haskell types

type	::=	$type_1$ `->` $type_2$		throughout
	|	`Char`		last week
	|	`Int`		”
	|	`Integer`		”
	|	`Float`		”
	|	`Double`		”
	|	`data` type		last week, this week
	|	`(`$type_1$`, ... ,`$type_n$`)`	$n \geq 2$	this week
	|	`()`		later
	|	`[`*type*`]`		next week

More `data` definitions

Last session, we used `data` definitions to define enumerated types:

```
data Direction = North | South | East | West
```

This week we shall see that we can specify alternatives that contain data, as in

```
data Shape
    = Circle Double
    | Rectangle Double Double
```

If there is only one alternative, that yields a form of record type.

But first, we consider built-in record types: pairs, triples, quadruples, quintuples, . . . (tuples).

Tuples

Tuples: pairs, triples, etc

Two or more things, not necessarily of the same type

```
          (1,2) :: (Int,Int)
        (1,2,3) :: (Int,Int,Int)
(1,True,'c',4.0) :: (Int,Bool,Char,Float)
```

Note that the type includes the number and types of the components.

Comparison of tuples:

interpreter

```
compare (1,True,'c') (1,True,'d')
compare (2,'b') (3,'a')
```

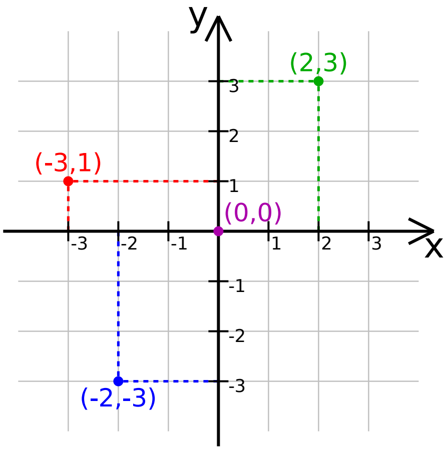
What are the rules?

When comparing tuples, the corresponding components are compared from left to right. So for pairs the rule is that $(x_1, x_2) < (y_1, y_2)$ if either

1. $x_1 < x_2$, or
2. $x_1 = x_2$ and $y_1 < y_2$.

This is called *lexicographical ordering*, because it is the order used in dictionaries.

Representing points in two-dimensional space



We'll only use whole numbers.

```
module Geometry where

-- (Direction stuff)

type Point = (Int, Int)

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

Here **type** defines **Point** as a *synonym* for an existing type.

Functions on points

Many functions on points will need to access their components

```
plusPoint :: Point -> Point -> Point
plusPoint (x1, y1) (x2, y2) = (x1+x2, y1+y2)
```

Here we are able to assign names to the components of the pairs by using *pattern* arguments. If we evaluate

interpreter

```
plusPoint (-3, 1) (2, 3)
```

The variables are defined as **x1** = **−3**, **y1** = **1**, **x2** = **2**, **y2** = **3**, and substituted into the righthand side as usual.

Polymorphic library functions

The standard prelude defines

Prelude

```
fst :: (a, b) -> a
fst (x, y) = x

snd :: (a, b) -> b
snd (x, y) = y
```

We can use these on pairs of any two types:

interpreter

```
fst (1.2, 'b')
snd (True, 5)
```

We could use these in an equivalent definition of **plusPoint**:

```
plusPoint :: Point -> Point -> Point
plusPoint p1 p2 = (fst p1 + fst p2, snd p1 + snd p2)
```

Type synonyms

The keyword **type** introduces a *type synonym*, a name completely interchangeable with the original type:

```
type Point = (Int, Int)
```

Type synonyms can be parameterized, so we could equivalently write:

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

- Because it's the same type, we can use general functions like **fst** and **snd** on it.
- But type system cannot help us keep points separate from other pairs.
- Type synonyms may be expanded in error messages, making them more complicated.

Record types

New types defined with data

The alternative is to define **Point** as a new type using a **data** definition:

```
data Point = MkPoint Int Int
    deriving (Show)
```

This declaration defines:

- a new type **`Point`**
- a new *constructor* **`MkPoint :: Int -> Int -> Point`**.

interpreter

```
:t MkPoint
```

A value of type **`Point`**:

interpreter

```
:t MkPoint 1 3
```

Renaming the constructor

In record types like this, where there is only one constructor, it is common to give it the same name as the type:

```
data Point = Point Int Int
    deriving (Show)
```

There is no ambiguity between the type and constructor both called **`Point`**, because one is found only in types, and the other only in expressions.

```
origin :: Point
origin = Point 0 0
```

Renaming the constructor

Defining **`plusPoint`** on our new **`Point`** type:

```
plusPoint :: Point -> Point -> Point
plusPoint (Point x1 y1) (Point x2 y2) = Point (x1+x2) (y1+y2)
```

Here we see a new kind of pattern argument, but it matches in the same way:

interpreter

```
plusPoint (Point (-3) 1) (Point 2 3)
```

As before, **`x1`** = **`-3`**, **`y1`** = **`1`**, etc.

Note where we need parentheses here to delimit the arguments of the function **`plusPoint`** and the constructor **`Point`**.

Another example

We can define:

```
data PriceTag = Item String Double
```

This declaration defines:

- a new type **`PriceTag`**
- a new *constructor* **`Item :: String -> Double -> PriceTag`**.

interpreter

```
:t Item
```

A value of type **`PriceTag`**: **interpreter**

```
:t Item "Hat" 28.5
```

Functions on product types

We define functions on the **`PriceTag`** type by pattern matching:

```
showPriceTag :: PriceTag -> String
showPriceTag (Item n price) =
    n ++ " -- " ++ show price
```

Functions returning **`PriceTag`** values can build them using the **`Item`** constructor:

```
addVAT :: PriceTag -> PriceTag
addVAT (Item nm price) = Item nm (1.2*price)
```

`type` vs `data`

Both of these define similar types:

```
type Point1 = (Int, Int)
data Point2 = Point Int Int
```

but they have a number of differences:

- `type` defines a synonym for an existing type, with which is completely interchangeable (allowing the use of general functions).
- `data` defines a new type, and a new constructor, which is the only way to build values of that type.
  - More errors will be detected.
  - Error messages are more specific.

This tradeoff is part of designing a representation for your data.

Sum types

Alternatives

A data type may include both alternatives and components:

```
data Shape
    = Circle Double
    | Rectangle Double Double
    deriving (Eq, Show)
```

Some values of this type: **interpreter**

```
Circle 2.5
Rectangle 2.0 4.2
```

Defining functions by cases:

```
area :: Shape -> Double
area (Circle r) = pi*r*r
area (Rectangle w h) = w*h
```

Derived instances

We can make our types into instances of standard classes in the "obvious" way by adding a **deriving** clause to the definition of the **data** type, e.g.:

```
data Shape
    = Circle Double
    | Rectangle Double Double
    deriving (Eq, Show)
```

- Two shapes are equal if they are of the same sort and have the same components.
- This only works for standard classes. (There are other restrictions too, but mostly what you'd expect.)
- It is often a good idea to derive at least **Show**.

More functions

Rotating a shape 90 degrees to the right:

```
rotate :: Shape -> Shape
rotate (Circle r) = Circle r
rotate (Rectangle w h) = Rectangle h w
```

Scaling a shape by a given number:

```
scale :: Double -> Shape -> Shape
scale x (Circle r) = Circle (r*x)
scale x (Rectangle w h) = Rectangle (w*x) (h*x)
```

Pattern matching vs interface inheritance

Function definition by pattern matching is approximately the transpose of interface inheritance:

	kinds of **Shape**	
	Circle	**Rectangle**
area	area of circle	area of rectangle
rotate	rotate circle	rotate rectangle
scale	scale circle	scale rectangle

- pattern matching: easy to add rows (functions)

- interface inheritable: easy to add columns (classes)

In the object-oriented model, one argument is special – it is messy to dispatch on multiple arguments

Parameterized types

Representing errors as values

Recall this function from the exercises:

```
charToNum :: Char -> Int
charToNum c
  | isDigit c = ord c - ord '0'
  | otherwise = 0
```

Here we chose an arbitrary output value for erroneous inputs.

- Sometimes it is useful to flag erroneous inputs.
- In some situations, there might not be a suitable output value.

Solution: define a new type with one more value.
General solution: add an extra value to any base type.

The `Maybe` type constructor

For any type `a`, `Maybe a` is a type with one more value:

Prelude

```
data Maybe a = Nothing | Just a
    deriving (Eq, Ord, Show)
```

Types of the constructors:

interpreter

```
:t Nothing
:t Just
```

Revising our function:

```
charToNum :: Char -> Maybe Int
charToNum c
  | isDigit c = Just (ord c - ord '0')
  | otherwise = Nothing
```

Now errors are just values, which we can handle like other values.

Other uses of `Maybe`

Other functions that might not yield an answer for all inputs:

- some numerical operations
- searching for an element in a container
- parsing a string

- finding a solution to a problem

Data which might not be present:

```
data Person = Person String Date (Maybe Date)
```

In many languages, null pointers are used for such purposes, but they are not flagged by the type, and are a common source of bugs.

Another pre-defined parameterized type

The standard prelude also defines a general type representing the tagged sum of two types:

Prelude
```
data Either a b = Left a | Right b
    deriving (Eq, Ord, Show)
```

interpreter
```
:t Left
:t Right
:t Left True
:t Right 'a'
```

Usually it's better to define a specific type to describe your data.

Next week

Lists in Haskell:

- list comprehensions

interpreter
```
[n*n | n <- [1..10]]
```

- library functions on lists

interpreter
```
reverse [1..10]
zip [0..] "Haskell"
[n | (n, c) <- zip [0..] "science", c == 'e']
```

```haskell
module Geometry where

-- compass points
data Direction = North | South | East | West
    deriving Show

-- the direction immediately to the left
turnLeft :: Direction -> Direction
turnLeft North = West
turnLeft South = East
turnLeft East = North
turnLeft West = South

-- the direction immediately to the right
turnRight :: Direction -> Direction
turnRight North = East
turnRight South = West
turnRight East = South
turnRight West = North

-- x and y coordinates in two-dimensional space
data Point = Point Int Int
    deriving (Eq, Ord, Show)

-- the origin of the two-dimensional space
origin :: Point
origin = Point 0 0

-- add two points
plusPoint :: Point -> Point -> Point
plusPoint (Point x1 y1) (Point x2 y2) = Point (x1+x2) (y1+y2)
```

Figure 3.1: `Geometry.hs`, so far

Exercises

Exercises 1 and 2 ask you to write polymorphic functions on pairs. Exercises 3 and 4 give practice with manipulating user-defined structured types. The last five exercises involve parameterized structured types.

1. Define a function

   ```
   swap :: (a,b) -> (b,a)
   ```

   that takes a pair of values and returns them swapped around.

2. Define a function

   ```
   dup :: a -> (a,a)
   ```

   that returns a pair with two copies of its argument.

3. Expand your **Geometry** module to match the version on the previous page, and add the following functions to it:

   (a) A function

   ```
   minusPoint :: Point -> Point -> Point
   ```

   that subtracts two points.

   (b) A function

   ```
   timesPoint :: Int -> Point -> Point
   ```

   that multiplies both components of a point by a given number.

   (c) A function

   ```
   normPoint :: Point -> Int
   ```

   that computes the sum of the absolute values of the two components. This is called the *Manhattan metric*, because it represents the minimum distance one has to travel to reach the point if one can only move North-South or East-West on the grid.

   (d) A function

   ```
   distance :: Point -> Point -> Int
   ```

   that computes the distance between two points using the Manhattan metric. (Try to do this without accessing the internals of the points, by using two functions you've already written.)

   (e) A function

   ```
   oneStep :: Direction -> Point
   ```

   that maps each direction to the point one unit from the origin in that direction.

4. Start a module **Turtle** that will implement a simplified form of Turtle graphics (to be explained below). Our simplified version will use only whole-number coordinates (*i.e.* **Point**) and horizontal and vertical directions (*i.e.* **Direction**). This module will use the **Geometry** module:

```
module Turtle where

import Geometry
```

However the functions in this module shouldn't need to look inside the `Point` type: they can just use the functions defined above.

(a) A turtle is a simple drawing robot for the two-dimensional plane, whose state consists of

- a position on the two-dimensional grid,
- the direction in which the turtle is facing,
- whether the turtle's pen is down (in the drawing position) or not.

Define a type `Turtle` representing the state of a turtle.

(b) Define a constant

```
startTurtle :: Turtle
```

for the initial configuration of the turtle: at the origin, facing North and with its pen up.

(c) Define a function

```
location :: Turtle -> Point
```

that returns the location of the turtle.

(d) Define a type to represent the commands understood by the turtle, which are:

- turn to the left (by 90 degrees)
- turn to the right (by 90 degrees)
- move some number n steps in the direction the turtle is facing
- lift the pen from the paper (unless already up)
- lower the pen onto the paper (unless already down)

Note that the move command will require an argument for the distance moved.

(e) Define a function

```
action :: Turtle -> Command -> Turtle
```

that returns the new configuration after a turtle in the given configuration executes the given command. (This doesn't do any drawing; we'll get to that in later weeks.)

5. As noted in session 1, the function `div` gives a runtime error if its second argument is `0`. Write a function

```
safeDiv :: Int -> Int -> Maybe Int
```

that reports the error case as `Nothing` and wraps the success case with `Just`.

6. Define a function

```
pairMaybe :: Maybe a -> Maybe b -> Maybe (a,b)
```

that produces a **`Just`** result only if both arguments are **`Just`**, and a **`Nothing`** if either argument is **`Nothing`**.

7. Write your own implementation of the function

```
fromMaybe :: a -> Maybe a -> a
```

which extracts the value inside a **`Maybe`** value, returning a default value (supplied as the first argument) if the second argument is **`Nothing`**.

8. Define a function

```
whatever :: Either a a -> a
```

that returns the value inside an **`Either`** value, ignoring how it is tagged.

9. A generalization of the **`Maybe`** type allows the failing part to include an error message:

```
data Err a = OK a | Error String
```

Define a function

```
both :: Err a -> Err b -> Err (a,b)
```

that produces an **`OK`** result only if both arguments are **`OK`**, and otherwise an **`Error`** (but including both messages if both parts failed). You can concatenate strings with the **`++`** operator|.