TUTORIAL 2 · [COMPUTATION AND LOGIC]

OBJECTIVES

In this tutorial you will:

- learn more about *types* and *predicates*;
- bring a small universe into Haskell and play with it;
- use quantifiers and compare natural language with Haskell.

TASKS

Exercises 1–6 are mandatory. Exercise 7 is optional.

DEADLINE

Saturday, 3rd of October, 4 PM UK time

Good Scholarly Practice

Please remember the good scholarly practice requirements of the University regarding work for credit.

You can find guidance at the School page

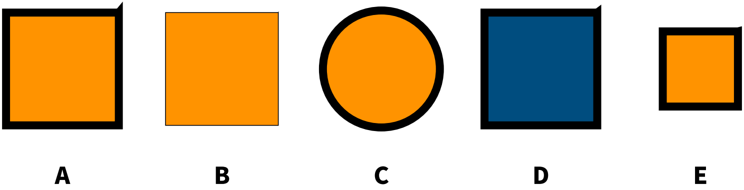
http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct.

This also has links to the relevant University pages. Please do not publish solutions to these exercises on the internet or elsewhere, to avoid others copying your solutions.

EXERCISE 1

MANDATORY | BEFORE TUTORIAL SESSION

Which of the things below is the odd one out? Why?



SOLUTION TO EXERCISE 1

Finding the odd one out in a set of things usually requires identifying some common features that all but one element in a set share.

Let's try to count the features that the 5 things have in common.

- Thing A has 3 features in common with each of the other things:
    - *colour*, *shape*, and *size* with B,
    - *colour*, *border*, and *size* with C,
    - *shape*, *size*, and *border* with D,
    - *colour*, *shape* and *border* with E.
- B, C, D, E on the other hand, share only 2 features with every other thing in our universe.

So it goes that the thing having the most features in common with the other things is the odd one out.

EXERCISE 2

MANDATORY | BEFORE TUTORIAL SESSION

Read Chapter 6 (*Features and Predicates*) from the textbook up to *Sequents*, page 53.

Define a Haskell type of things and a list of all the things in the universe from Exercise 1, similarly to the way it is done in the textbook on page 49. Save your code in a file named `cl-tutorial-2.hs`

Identify the 4 features that characterize the things in Exercise 1.

Now create a type for each of the 4 features, and also write functions that say which things have those features.

SOLUTION TO EXERCISE 2

We define the type of things and a list of all the things in the universe:

```
data Thing = A | B | C | D | E deriving Show

things :: [Thing]
things = [ A, B, C, D, E ]
```

The four features that characterize the things in our universe are: colour, shape, size, and border.

```
data Colour = Amber | Blue

colour :: Thing -> Colour
colour A = Amber
colour B = Amber
colour C = Amber
colour D = Blue
colour E = Amber
```

SOLUTION TO EXERCISE 2 (CONT.)

```
data Shape = Square | Disc

shape :: Thing -> Shape
shape A = Square
shape B = Square
shape C = Disc
shape D = Square
shape E = Square

data Size = Big | Small

size :: Thing -> Size
size A = Big
size B = Big
size C = Big
size D = Big
size E = Small
```

SOLUTION TO EXERCISE 2 (CONT.)

```
data Border = Thin | Thick

border :: Thing -> Border
border A = Thick
border B = Thin
border C = Thick
border D = Thick
border E = Thick

type Predicate u = u -> Bool

isBlue :: Predicate Thing

isBlue A = False
isBlue B = False
isBlue C = False
isBlue D = True
isBlue E = False

...
```

EXERCISE 3

MANDATORY | BEFORE TUTORIAL SESSION

In the textbook, you have seen how `deriving Show` can be used to allow the values of type `Thing` to be printed. Similarly, you can use `deriving Eq` to allow values to be compared for equality or inequality, or `deriving (Eq,Show)` to allow both printing and testing for (in)equality.

When a type derives `Eq`, you can write:

- `x == y` to check if `x` and `y` are equal, and
- `x /= y` to check if they are different.

Modify your code from Exercise 2 such that all data types derive `Eq`. Can you see how using `==` can simplify the definitions of the functions saying which things have which features? (Hint: Try to evaluate, for example, in `ghci`, `colour x == Blue` for every thing `x` in $\{A, B, C, D, E\}$, and compare the Boolean values you get with the output of `isBlue x`.) Update your definitions so that they use equalities.

SOLUTION TO EXERCISE 3

We modify our code such that all data types derive `Eq`

```
data Thing = A | B | C | D | E deriving (Eq,Show)
```

and simplify the definitions of the functions saying which things have which values of the 4 features:

```
data Colour = Amber | Blue deriving Eq
isBlue :: Predicate Thing
isBlue x = colour x == Blue
isAmber :: Predicate Thing
isAmber x = colour x == Amber
```

Note that `colour x == Blue` evaluates to `True` if and only if `x` is `D` – and to `False` for any other thing. So this definition of the function `isBlue` and the one given on slide 8 are equivalent.

SOLUTION TO EXERCISE 3 (CONT.)

```haskell
data Shape = Square | Disc deriving Eq
isSquare :: Predicate Thing
isSquare x = shape x == Square
isDisc :: Predicate Thing
isDisc x = shape x == Disc

data Size = Big | Small deriving Eq
isBig :: Predicate Thing
isBig x = size x == Big
isSmall :: Predicate Thing
isSmall x = size x == Small
```

SOLUTION TO EXERCISE 3 (CONT.)

```
data Border = Thin | Thick deriving Eq

hasThinBorder :: Predicate Thing
hasThinBorder x = border x == Thin

hasThickBorder :: Predicate Thing
hasThickBorder x = border x == Thick
```

EXERCISE 4

MANDATORY | BEFORE TUTORIAL SESSION

Write a function `thingsOtherThan` that returns, for every input `x`, the list of all remaining 4 things that are different from `x`.

```
thingsOtherThan :: Thing -> [Thing]
```

Complete the list of properties of things. It should have 8 elements.

```
properties :: [Predicate Thing]
properties = [isBlue, ...]
```

SOLUTION TO EXERCISE 4

```
thingsOtherThan :: Thing -> [Thing]
thingsOtherThan x = [ y | y <- things, y /= x ]
```

```
properties :: [Predicate Thing]
properties = [isBlue, isAmber, isSquare, isDisc, isBig,
              isSmall, hasThinBorder, hasThickBorder]
```

EXERCISE 4 (CONT.)

Write a function `propertiesOf` that returns, for every input `x`, the list of all properties of `x`. Note that every thing has exactly 4 properties.

```
propertiesOf :: Thing -> [Predicate Thing]
```

Write a function `isPropertyOfAnotherThing` that checks, for every predicate `p` and thing `x`, if `p` is a property of a thing different from `x`.

```
isPropertyOfAnotherThing :: Predicate Thing -> Thing -> Bool
```

SOLUTION TO EXERCISE 4 (CONT.)

```
propertiesOf :: Thing -> [Predicate Thing]
propertiesOf x = [ p | p <- properties, p x ]
```

```
isPropertyOfAnotherThing :: Predicate Thing -> Thing -> Bool
isPropertyOfAnotherThing p x = or [ p y | y <- thingsOtherThan x ]
```

EXERCISE 4 (CONT.)

Write a function `propertiesOnlyOf` that returns, for every input thing `x`, the list of all properties that are unique to `x`.

```
propertiesOnlyOf :: Thing -> [Predicate Thing]
```

Write a function `rank` that returns, for every input thing `x`, how many properties are unique to `x`. We call this number the *rank* of `x`.

```
rank :: Thing -> Int
```

Now look at the ranks of all the things in our universe.
Which one stands out?

SOLUTION TO EXERCISE 4 (CONT.)

```
propertiesOnlyOf :: Thing -> [Predicate Thing]
propertiesOnlyOf x = [ p | p <- propertiesOf x,
                           not (isPropertyOfAnotherThing p x)]
```

```
rank :: Thing -> Int
rank x = length (propertiesOnlyOf x)
```

EXERCISE 5

MANDATORY | BEFORE TUTORIAL SESSION

Express the following statements in Haskell using the five logical operations `&&`, `||`, `and`, `or`, `not` to combine the predicates defined above. Give the values of the Haskell expressions and check that they are correct according to the list of things in Exercise 1.

1. Every blue square has a thin border.
2. Some amber things are not squares.
3. Every big square is either amber or has a thick border.
4. Some amber disc is not big.

SOLUTION TO EXERCISE 5

1. Every blue square has a thin border.

```
and [ hasThinBorder x | x <- things, isBlue x && isSquare x]
```

2. Some amber things are not squares.

```
or [ not (isSquare x) | x <- things, isAmber x ]
```

3. Every big square is either amber or has a thick border.

```
and [ isAmber x || hasThickBorder x | x <- things, isBig x
                                     && isSquare x ]
```

4. Some amber disc is not big.

```
or [ not (isBig x) | x <- things, isAmber x && isDisc x ]
```

EXERCISE 6

MANDATORY | BEFORE TUTORIAL SESSION

The statement “No square is blue” doesn’t fit either of the patterns “Every X is Y” or “Some X is Y”. But it is equivalent to a statement of the form “It is not the case that some X is Y” and also to a statement of the form “Every X is not Y”.

Give those two equivalent statements and express them in Haskell using the five logical operations `&&`, `||`, `and`, `or`, `not`.

SOLUTION TO EXERCISE 6

The statement “No square is blue” is equivalent to:

- “It is not the case that some square is blue”:

```
not (or [ isBlue x | x <- things, isSquare x ])
```

- “Every square is not blue”:

```
and [ not (isBlue x) | x <- things, isSquare x ]
```

EXERCISE 7

OPTIONAL | DURING TUTORIAL SESSION

The words “all”, “every”, “each”, “any”, and “some” are indicators of quantity. Generally they fall into two categories:

- indicators that refer to *all* things with a given property, or
- indicators that refer to *some* things with a given property.

“all”, “every” and “each” belong to the first category, while “some” belongs to the second category. But “any” may be ambiguous.

Discuss with your colleagues the meaning of the word “any” in the following sentences.

1. Is there any amber square in our universe?
2. Any thing in our universe is amber.
3. We say that the universe is warm if any thing in it is amber.

EXERCISE 7 (CONT.)

OPTIONAL | DURING TUTORIAL SESSION

Natural language is imprecise, but some languages are more precise than others. Can you think of similar examples of ambiguity related to “every”, “any”, and “some” from other languages?

Explain them to your group.

Use the universe in Exercise 1 to exemplify these ambiguities. Write the sentences in natural language, then disambiguate them by translation into Haskell code.