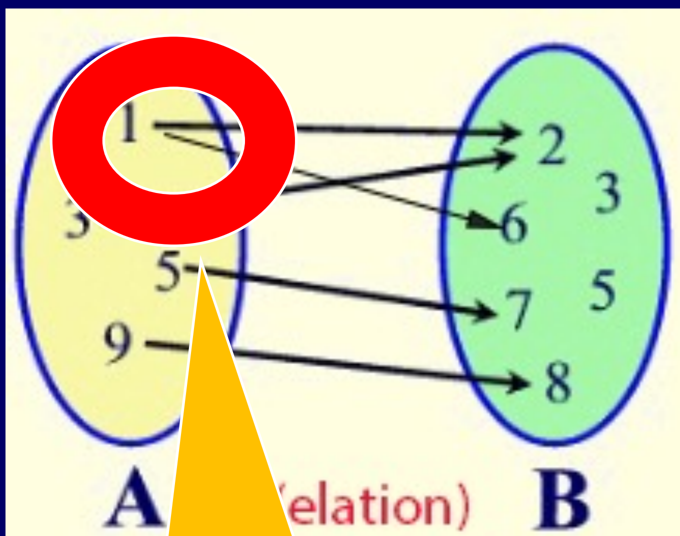
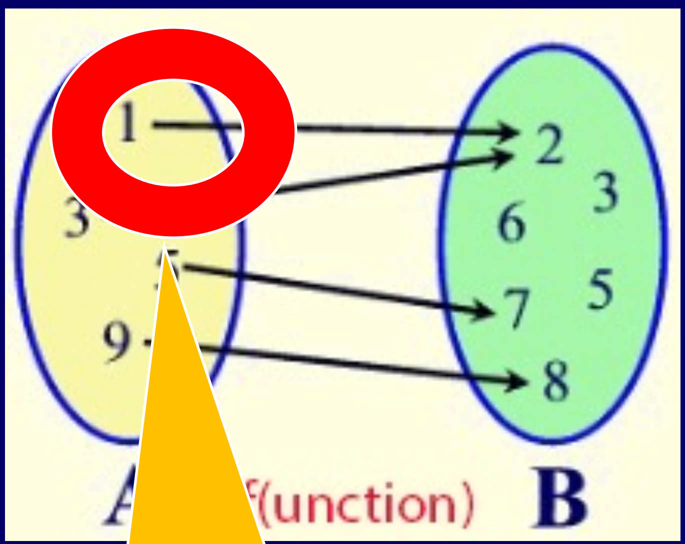
PROGRAMMING IN HASKELL

Chapter 4 - Defining Functions

The nature of functions

You may remember …..



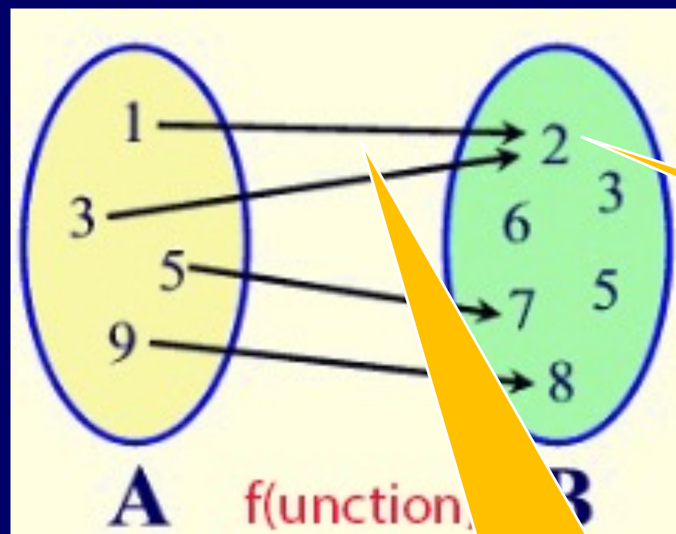


R = { .. (1,2), (1, 6), ..}
A relation may have many mappings from the domain.

f = { .. (1,2), (3, 6), ..}
A function has one mapping from each element in the domain

The nature of functions

So, in mathematical terms, we apply a function to a value of type A and it returns a value of type B.



One unique value returned

f: A -> B
f = {..(1,2) .. }

f (1) = 2

The nature of functions.. maths

- So, 2 being returned from the application of f to 1 is the ***effect*** of the function f.

- In mathematical functions, nothing else happens when f is called/applied.

- We say ‘there are no side-effects’

The nature of functions.. Programming, e.g. Java

- We use the term **methods**.
- Methods can be
  - Accessors/read (e.g. getters)
  - Mutators/ read/write (e.g. setters)

The nature of functions.. accessors and mutators

Simple class with two fields!

```
class Spot{
  float xCoord, yCoord;

  // constructors…
  // display method…
  // colour methods…
  // move methods…
}
```

The nature of functions.. accessors

This changes no state and simply returns a value

```
public float getXCoord(){
      return xCoord;
   }
```

This is the **effect** of the function

This function has no **side-effects**. It is ***pure***

The nature of functions.. mutators

This only changes state and returns no value

```
public void  setXCoord (float xCoord){
    this.xCoord = xCoord;
 }
```

This function has no **effect**

This function has only **side-effects**.

The nature of functions.. mutators ++

This changes state and returns a value

```
public float  setXCoord (float xCoord){
    this.xCoord = xCoord;
    return this.xCoord;
 }
```

This function also has **side-effects**.

This function has an **effect**

Purity in Haskell

In Haskell, functions are pure. This means that functions have only effects, no side-effects.

Thus

- We do not deal with state.
- Functions simply take arguments and return a value. The application or running of a function does not change the **outside world** in any way.

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
myAbs :: Int → Int
myAbs n = if n ≥ 0 then n else -n
```

myAbs takes an integer n and returns n if it is non-negative and -n otherwise.

When calling this on a negative number we need to parenthesise e.g. myAbs (-7)

Conditional expressions can be nested:

```
mySignum :: Int → Int
mySignum n = if n < 0 then -1 else
                if n == 0 then 0 else 1
```

In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity problems with nested conditionals.

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

```
myAbs n | n ≥ 0     = n
        | otherwise = -n
```

As previously, but using guarded equations.

Guarded equations can be used to make definitions involving multiple conditions easier to read:

```
mySignum n | n < 0     = -1
           | n == 0    = 0
           | otherwise = 1
```

The catch all condition <u>otherwise</u> is defined in the prelude by otherwise = True.

Case statement

As an alternative to conditionals, functions can also be defined using case statements

```
addOneIfOdd n = case odd n of
        True -> f n
        False -> n
        where f n = n+1
```

Use if this will return one of small number of possible values.

Pattern Matching

Many functions have a particularly clear definition using pattern matching on their arguments.

```
not :: Bool → Bool
not False = True
not True  = False
```

not maps False to True, and True to False.

Pattern Matching

Functions can often be defined in many different ways using pattern matching. For example

```
(&&) :: Bool → Bool → Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

can be defined more compactly by

```
True && True = True
_    && _    = False
```

Using wildcard _

Pattern Matching

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True  && b = b
False && _ = False
```

The underscore symbol _ is a <u>wildcard</u> pattern that matches any argument value.

Pattern Matching

- Patterns are matched <u>in order</u>. For example, the following definition always returns False:

```
_    && _    = False
True && True = True
```

- Patterns may not <u>repeat</u> variables. For example, the following definition gives an error:

```
b && b = b
_ && _ = False
```

Use of where with Guards

- Want to avoid calculating the same value over and over.
- Calculate this intermediate value once, store and use often
- Use the where clause
- The scope of the variables defined in the where section of a function is the function itself. (clean)
- We can also use where bindings to pattern match

Use of where with Guards(2)

Look at a function to ‘calculate’ your annual salary

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
   | hourlyRate * (weekHoursOfWork * 52) <= 40000 = "Poor child, try to get another job"
   | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "Money, Money, Money!"
   | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
   | otherwise = "Hello Elon Musk!"
```

Would be useful to name the

`hourlyRate* weekHoursOfWork * 52`

value

Use of where with Guards and patterns (3)

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
    | annualSalary <= smallSalary = "Poor child, try to get another job"
    | annualSalary <= mediumSalary = "Money, Money, Money!”
    | annualSalary <= highSalary = "Ri ¢ hie Ri ¢ h"
    | otherwise = "Hello Elon Musk!"
    where
      annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary) = (40000, 120000, 200000)
```

The let expression

Let expressions are similar to where bindings

```
cylinder :: Double -> Double -> Double
cylinder r h =
   let  sideArea = 2 * pi * r * h
        topArea = pi * r ^ 2
   in sideArea + 2 * topArea
```

Example using let

```
cylinder :: Double -> Double -> Double
cylinder r h =
    sideArea + 2 * topArea
    where sideArea = 2 * pi * r * h
          topArea = pi * r ^ 2
```

Example using where

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called "cons" that adds an element to the start of a list.

```
[1,2,3,4]
```

Means 1:(2:(3:(4:[]))).

Functions on lists can be defined using x:xs patterns.

```
head :: [a] → a
head (x:_) = x

tail :: [a] → [a]
tail (_:xs) = xs
```

head and tail map any non-empty list to its first and remaining elements.

Note:

- x:xs patterns match <u>non-empty</u> lists:

```
> head []
*** Exception: No head for empty lists!
```

- This can be effected by writing as part of the function def:

```
head :: [a] → a
head[] = error “No head for empty lists!”
head (x:_) = x
```

Note:

- x:xs patterns must be <u>parenthesised</u>, because application has priority over (:). For example, the following definition gives an error:

```
head x:_ = x
```

Operator Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

For example:

```
> 1+2
3

> (+) 1 2
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```
> (1+) 2
3

> (+2) 1
3
```

In general, if ⊕ is an operator then functions of the form (⊕), (x⊕) and (⊕y) are called <u>sections</u>.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

`(1+)`	-	successor function
`(1/)`	-	reciprocation function
`(*2)`	-	doubling function
`(/2)`	-	halving function


ANY
QUESTIONS?