Guards

Often we want to have different equations for different equations

$$signum(x) - \begin{cases} 0, & \text{if } x = 0 \\ -1, & \text{if} x < 0 \\ 1, & \text{if} x > 0 \end{cases}$$

To write this in Haskell:

```
signum x | x < 0  = -1
         | x == 0 =  0
         | x > 0  =  1
```

Each guard is tested iin turn, adn the first one to match selects an alternative. This means its OK to have a guard that would always be true, as long as it is the last alternative.

So the previous definition could have been written like this:

```
signum x | x < 0   =  -1
         | x == 0  =   0
         | True    =   1
```

We can use guards to select special cases in functions. This function is true when the year number is a leap year:

```
leapyear :: Int -> Bool
leapyear y | mod y 400 == 0  =  True   -- 2000 was
           | mod y 100 == 0  =  False  -- 1900 wasn't
           | mod y 4 == 0    =  True   -- 2020 was
           | otherwise       =  False  -- 2021 isn't
```

Patterns and Guards

Guards and Patterns can be combined:

```
startswith _ [] = False
startswith c (x:xs) | x == c     = True
                    | otherwise = False
```

First the patterns are matched; when an equation is found the guards are evaluated in order in the usual ways.

If no guard matches then we return to the pattern matching stage and try to find another equation.

The Behaviour of startswith

The second argument of startswith is a list of things, while the first argument must have the same type as the things in the list.

The first line of the startswith code above matches the case when the second list argument is empty, and ignores the first argument, returning False.

> startswith 1 [ ]

False

If the second list argument is not empty, then the second pattern match succeeds, and we proceed to compare the first list element (x) with the first argument (c), thus we do the second line of the startswith code above.

> startswith 42 [42, 41, 40]

True

Factorial as Patterns

```
factorial 0          =  1
factorial n | n > 0  =  n * factorial (n-1)
```

Function Notation

$f(x) = x + 1$

In Haskell, we could write:

f1(x) = x+1

But we usually write:

f2 x = x+1

$g(x, y, z) = x + y + z$

In Haskell, we could write:

g1(x,y,z) = x+y+z

But we usually write:

g1 x y z = x+y+z

As far as Haskell is concerned, f1(x) and f2 x are the same.

However, g1(x,y,z) and g2 x y z are not:

Their types are different:

g1 :: Num a =¿ (a,a,a) -¿ a

g2 :: Num a =¿ a-¿ a -¿ a-¿ a

Function g1 takes a triple of numbers and returns a number, whereas g2 takes a number, another number and another number, and returns a number.

Pattern Matching

Pattern ('c':zs)

matches expression 'c':('a':('t':[ ]))

with zs bound to 'a':('t':[ ]

- We can build patterns from atomic values, variables and certain kinds of constructions.
- An atomic value, such as 3, 'a' or "abc" can only match itself
- A variable or the wildcard _will match anything
- A construction is either:

  a tuple such as (a.b) or (a.b.c) etc

  a list built using [ ] or :

or a user-defined datatype

A construction pattern matches if all its sub-components match

Function definition equations may have a sequence of patterns. Each pattern is matched against the corresponding expression, and all such matches must succeed. One binding is returned for all of the matches. Any given variable may only occur once in any pattern sequence.

Pattern Examples

myfun x y z = whatever –using x y z

- If we want the first two arguments to be the same, then we must use a conditional:

  myfun x x z = whatever –cant use x twice here, illegal

  myfun x y z — x==y = whatever –this works!

- First argument must be zero, second is arbitrary and third is a non-empty list

  myfun 0 y (z:zs) = whatever

- First argument must be zero, second is arbitrary and third is a non-empty list, whose first element is character 'c'

  myfun 0 y ('c':zs) = whatever

- First argument must be zero, second is arbitrary and third is a non-empty list, whose tail is a singleton

  myfun 0 y (z:[z']) = whatever

Patterns	Values	Outcome
x (y:ys) 3	99 [] 3	Fail
x (y:ys) 3	99 [1,2,3] 3	Ok, $x \mapsto 99, y \mapsto 1, ys \mapsto [2,3]$
x (3:ys) 3	99 [3,2,1] 3	Ok, $x \mapsto 99, ys \mapsto [2,1]$

The pattern is x (y:ys) 3

- First fails because the pattern has a non-empty list for (y:ys) but the values have an empty list [ ]

- Second passes because x = 99, y = 1, ys = [2,3]

- Third passes because x = 99, ys = [2,1]

Also, the binding x = 99, y = 1, ys = [2,3] can be written as a simultaneous substitution: [99,1,[2,3]/x,y,ys]

More Function Notation

We can define and use functions whose names are either variable identifiers (varid) or variable operators (varsym)

For varid names, the function def uses prefix notation, where the function names appears before the arguments:

myfun x y = x+y+Y

For varsym names, it uses infix, where the function jas two arguments and the name appears between the arguments:

x +++ y = x+y+y

We can use infix for varid by enclosing the name in backticks

x 'plus2' y = x+y+y

We can use prefix for varsym using parentheses

(+++) x y = x+y+y

Writing Functions using Other Functions

- Function even returns true if its integer argument is even

  even n = n 'mod' 2 == 0

  We use the Prelude modulo function

- Function recip calculates the reciprocal of its argument

  recip n = 1/n

  This uses the division function from Prelude

- Function call splitAt n xs returns two lists, the first with the first n elements of xs, the second with the rest of the elements

  splitAt n xs = (take n xs, drop n xs)

  Here we use the list functions take and drop from Prelude

Higher Order Functions

What is the difference between these two functions?

add x y = x+y

add2 (x, y) = x+y

We can see it in the types; add takes one argument at a time, returning a function that looks for the next argument. This concept is known as Currying.

add :: Integer -¿ (Integer -¿ Integer)

add2 :: (Integer -¿ Integer) -¿ Integer

Functions are first class, meaning they occupy the same status as values; you can pass them as arguments, make them part of data structures etc

increment :: Integer -¿ Integer

increment = add 1

If the type of add is Integer -¿ Integer -¿ Integer and the type of add 1 2 is Integer, then the type of add 1 is Integer -¿ Integer

This is the notion of partial application.

Another example of this would be that an infix operator can be partially applied by taking a "section"

increment = (1 +)

addnewline = (++"/n"

double :: Integer -¿ Integer

double = (*2)
> [double x — x ¡- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
Functions can be taken as parameters as well.
twice :: (a -¿ a) -¿ a -¿ a
twice f x = f (f x)
addtwo = twice increment
Here we see functions being defined as functions of other functions.
In fact, we can define function composition using this technique.
compose :: (b -¿ c) -¿ (a -¿ b) -¿ a -¿ c
compose f g x = f (g x)
twice2 f = f 'compose' f
We can define functions without naming their inputs, using composition:
second :: [a] -¿ a
second = head . tail
> second [1,2,3]
2

Writing Functions using Recursion
take

- take :: Int -¿ [a] -¿ [a] Let xs1 = take n xs below. Then xs1 is the first n elements of xs. If n¡= 0, then xs1 = [ ]. If n¿= length xs, then xs1 = xs
- take n _— n ¡= 0 = [ ]

  take _[ ] = [ ]

  take n (x:xs) = x : take (n-1) xs

drop

- drop :: Int -¿ [a] -¿ [a]. Let (xs1,xs2) = splitAt n xs below. Then xs1 is the first n elements of xs. Then xs2 is xs with the first n elements removed. If n¿= length xs then (xs1,xs2) = (xs,[ ]). If n¡= 0, then (xs1,xs2) = ([ ],xs)
- splitAt n xs — n ¡= 0 = ([ ],xs)

  splitAt _[ ] = ([ ],[ ])

  splitAt n (x:xs) = let (xs1,xs2) = splitAt (n-1) xs in (x:xs1,xs2)