PATTERN MATCHING

PATTERN MATCHING IN HASKELL

- Pattern matching consists of specifying patterns to which some data should conform and then..
- checking to see if it does and…
- deconstructing the data according to those patterns.
- When defining functions, separate function bodies can be defined for different patterns.
- This leads to really neat code that's simple and readable.
- Pattern matching can be used on any data type — numbers, characters, lists, tuples, etc.

EXAMPLE TO MATCH A NUMBER

```
--sayMe.hs
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5“
```

```
ghci> :l sayMe.hs
[1 of 1] Compiling Main             ( sayMe.hs, interpreted )
Ok, one module loaded.
```

```
ghci> sayMe 1
"One!"
ghci> sayMe 5
"Five!"
ghci> sayMe 7
"Not between 1 and 5"
```

PATTERN MATCHING

- Pattern Matching is process of matching specific type of expressions.
- Can be considered as a variant of dynamic polymorphism where at runtime, different methods can be executed depending on their argument list.
- Example:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )

main = do
   putStrLn "The factorial of 5 is:"
   print (fact 5)

ghci> main
The factorial of 5 is:
120
```

- The compiler will start searching for a function called "fact" with an argument.
- If the argument is not equal to 0, then the number will keep on calling the same function with 1 less than that of the actual argument.
- When the pattern of the argument exactly matches with 0, it will call our pattern which is "fact 0 = 1".

A FACTORIAL EXAMPLE

```
--factorial using pattern matching factorialp.hs
factorialp :: (Integral a) => a -> a
factorialp 0 = 1
factorialp n = n * factorialp (n - 1)

ghci> :l factorialp.hs
[1 of 1] Compiling Main             ( factorialp.hs, interpreted )
Ok, one module loaded.
ghci> factorialp 0
1
ghci> factorialp 5
120
```

- Order is important !
- Specify the most specific ones first and then the more general ones later

FACTORIAL EXECUTION FOR N=3

factorial 3

3* factorial 2

3*(2 * factorial 1)

3 * (2 * (1 * factorial 0))

3 * (2 * (1 * 1))

3 * (2 * 1)

3 * 2

6

factorial 0 = 1
pattern is matched
here

Had we written the second pattern (factorial n) before the first one (factorial 0), it would catch all numbers, including 0 and our calculation would never terminate.

PATTERN MATCHING COULD FAIL!!

```
--charName.hs
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil“

ghci> charName 'a'
"Albert"
ghci> charName 'c'
"Cecil"
ghci> charName ‘h'
"*** Exception: charName.hs:(3,1)-(5,22): Non-exhaustive patterns in function charName
```

When making patterns, we should always include a catch-all pattern so that our program doesn't crash if we get some unexpected input.

FIX !!

```
--charName1.hs
charName1 :: Char -> String
charName1 'a' = "Albert"
charName1 'b' = "Broseph"
charName1 'c' = "Cecil"
charName1  x  = "String not defined“

ghci> :l charName1.hs
[1 of 1] Compiling Main             ( charName1.hs, interpreted )
Ok, one module loaded.
ghci> charName1 'b'
"Broseph"
ghci> charName1 'h'
"String not defined"
```

PATTERN MATCHING ON TUPLES

- To make a function that takes two vectors in a 2D space (that are in the form of pairs) and adds them together

- Without pattern matching

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors a b = (fst a + fst b, snd a + snd b)
```

```
ghci> :l addVectors.hs
[1 of 1] Compiling Main             ( addVectors.hs, interpreted )
Ok, one module loaded.
ghci> addVectors (1,2) (3,4)
(4,6)
```

ADD VECTORS USING PATTERN MATCHING

```
--addVectors1.hs - with pattern matching
addVectors1 :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors1 (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)


ghci> :l addVectors1.hs
[1 of 1] Compiling Main             ( addVectors1.hs, interpreted )
Ok, one module loaded.
ghci> addVectors1 (1,2) (3,4)
(4,6)
```

Note that this is already a catch-all pattern. The type of addVectors (in both cases) is addVectors :: (Num a) => (a, a) -> (a, a) - > (a, a), so we are guaranteed to get two pairs as parameters.

DEFINING OUR OWN FUNCTIONS FOR TRIPLES

```
first :: (a, b, c) -> a

first (x, _, _) = x

second :: (a, b, c) -> b

29

second (_, y, _) = y

third :: (a, b, c) -> c

third (_, _, z) = z
```

_ means don't care

PATTERN MATCH IN LIST COMPREHENSIONS

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

- Should a pattern match fail, it will just move on to the next element.

LISTS AND PATTERN MATCHING

- Lists themselves can also be used in pattern matching
- You can match with the empty list [ ] or any pattern that involves : and the empty list
- A pattern like x:xs will bind the head of the list to x and the rest of it to xs, even if there's only one element so xs ends up being an empty list
- The x:xs pattern is used a lot, especially with recursive functions
- Patterns that have : in them only match against lists of length 1 or more
- If you want to bind, say, the first three elements to variables and the rest of the list to another variable, you can use something like x:y:z:zs
- It will only match against lists that have three elements or more

OUR OWN IMPLEMENTATION OF THE HEAD FUNCTION

```
--myhead.hs
myhead :: [a] -> a
myhead [] = error "Can't call head on an empty list, dummy!"
myhead (x:_) = x


ghci> :l myhead.hs
[1 of 1] Compiling Main             ( myhead.hs, interpreted )
Ok, one module loaded.
ghci> myhead [4,5,6]
4
ghci> myhead "Hello"
'H'
```

A SAFE LIST TELLER FUNCTION !

```
--tell.hs

tell :: (Show a) => [a] -> String

tell [] = "The list is empty"

tell (x:[]) = "The list has one element: " ++ show x

tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y

tell (x:y:_) = "This list is long. The first two elements are: " ++ show x ++ " and " ++ show y
```

```
ghci> :l tell.hs

[1 of 1] Compiling Main             ( tell.hs, interpreted )

Ok, one module loaded.

ghci> tell []

"The list is empty"

ghci> tell [1]

"The list has one element: 1"

ghci> tell [1,2]

"The list has two elements: 1 and 2"

ghci> tell [1,2,3]

"This list is long. The first two elements are: 1 and 2"
```

- This function is safe because it takes care of the empty list, a singleton list, a list with two elements and a list with more than two elements.
- Note that (x: [ ]) and (x:y: [ ]) could be rewriten as [x] and [x,y](because its syntatic sugar, we don't need the parentheses). We can't rewrite (x:y:_) with square brackets because it matches any list of length 2 or more.

LENGTH USING PATTERN MATCHING

```
--mylength.hs
mylength :: (Num b) => [a] -> b
mylength [] = 0                      -- length of empty list
mylength (_:xs) = 1 + mylength xs   -- recursive call to mylength


ghci> :l mylength.hs
[1 of 1] Compiling Main
Ok, one module loaded.
ghci> mylength []
0
ghci> mylength [1,2,3,4,5]
5
```

- This is similar to the factorial function.
- First define the result of a known input — the empty list, also known as the edge condition.
- In the second pattern, take the list apart by splitting it into a head and a tail.
- We say that the length is equal to 1 plus the length of the tail.
- _ is used to match the head because we don't actually care what it is.
- Also note that all possible patterns of a list are taken care of. The first pattern matches an empty list and the second one matches anything that isn't an empty list.

EXECUTION OF MYLENGTH

- Let's see what happens if we call length' on "ham".
- First, it will check if it's an empty list. Because it isn't, it falls through to the second pattern.
- It matches on the second pattern and there it says that the length is 1 + length' "am", because we broke it into a head and a tail and discarded the head.
- The length' of "am" is, similarly, 1 + length' "m".
- So right now we have 1 + (1 + length' "m").
- length' "m" is 1 + length' "" (could also be written as 1 + length'[]).
- And we've defined length' [] to be 0.
- So in the end we have 1 + (1 + (1 + 0)).

SUM OF A LIST

```
--mysum.hs
mysum :: (Num a) => [a] -> a
mysum [] = 0
mysum (x:xs) = x + sum xs


ghci> :l mysum.hs
[1 of 1] Compiling Main             ( mysum.hs, interpreted )
Ok, one module loaded.
ghci> mysum [1,2,3,4,5]
15
```

GUARDS

- Patterns are a way of making sure a value conforms to some form and deconstructing it
- Guards are a way of testing whether some property of a value (or several of them) are true or false.
- Guards are similar to if statements
- Guards are lot more readable to match several conditions

A BMI EXAMPLE USING GUARDS

```
--bmiTell.hs
bmiTell :: (RealFloat a) => a -> String
bmiTell bmi
 | bmi <= 18.5 = "You're underweight, you emo, you!"
 | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet yo
 | bmi <= 30.0 = "You're fat! Lo                     weight, fatty!
 | otherwise = "You're a whale, congratulations!“

ghci> :l bmiTell.hs
[1 of 1] Compiling Main             ( bmi
Ok, one module loaded.
ghci> bmiTell 24.3
"You're supposedly normal. Pffft, I be
```

Reminiscent of a big if else tree in imperative languages but more readable

Many times, the last guard is otherwise. otherwise is defined simply as otherwise = True and catches everything.

- Guards are indicated by pipes that follow a function's name and its parameters.
- Usually, they're indented a bit to the right and lined up.
- A guard is basically a boolean expression.
- If it evaluates to True, then the corresponding function body is used.
- If it evaluates to False, checking drops through to the next guard and so on.

BMI WITH MORE PARAMETERS

There's no = right after the function name and its parameters, before the first guard.

```
--bmiTell1.hs

bmiTell1 :: (RealFloat a) => a -> a -> String

bmiTell1 weight height

 | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"

 | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"

 | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"

 | otherwise = "You're a whale, congratulations!“


ghci> bmiTell1 85 1.90

"You're supposedly normal. Pffft, I bet you're ugly!"
```

OUR OWN MAX FUNCTION

```
--mymax.hs
mymax :: (Ord a) => a -> a -> a
mymax a b
 | a > b = a
 | otherwise = b
ghci> mymax 3 2
3
```

<u>Inline Guards:</u>

```
--maxI.hs
maxI :: (Ord a) => a -> a -> a
maxI a b | a > b = a | otherwise = b
ghci> maxI 2 3
3
```

- The Ord class is used for types that have an ordering. Ord covers all the standard comparing functions such as >, <, >= and <=.

- The compare function takes two Ord members of
- the same type and returns an ordering.

- Ordering is a type that can be GT, LT or EQ, meaning
- greater than, lesser than and equal, respectively.

Guards can also be written inline, although its not advisable as it's less readable, even for very short functions.

OUR OWN COMPARE FUNCTION

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
| a > b = GT
| a == b = EQ
| otherwise = LT
ghci> 3 `myCompare` 2
GT

ghci> 3 `myCompare` 2
GT
```

Back ticks (` `) (check tilde ~ key on your keyboard) are used to call and define functions as infix to make it easy for readability ! ! !

WHERE CLAUSE

```
roots :: (Float, Float, Float) -> (Float, Float)
roots (a,b,c) = (x1, x2) where
 x1 = e + sqrt d / (2 * a)
 x2 = e - sqrt d / (2 * a)
 d = b * b - 4 * a * c
 e = - b / (2 * a)
main = do
 putStrLn "The roots of our Polynomial equation
 print (roots(1,-8,6))
```

- **Where** is a keyword or inbuilt function that can be used at runtime to generate a desired output.
- It can be very helpful when function calculation becomes complex.
- Consider a scenario where your input is a complex expression with multiple parameters.
- In such cases, you can break the entire expression into small parts using the "where" clause

```
ghci> :l whereroots.hs
[1 of 1] Compiling Main             ( whereroots.hs, interpreted )
Ok, one module loaded.
ghci> main
The roots of our Polynomial equation are:
(7.1622777,0.8377223)
```

THE BMI EXAMPLE MODIFIED

```
bmiTell :: (RealFloat a) => a -> a -> String    -- Earlier Version
bmiTell weight height
 | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
 | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
 | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
 | otherwise = "You're a whale, congratulations!"
```

The where keyword is put after the guards , indented and several names or functions are defined

Same expression repeated thrice ! ! !

```
bmiTell :: (RealFloat a) => a -> a -> String    --Revised using Where
bmiTell weight height
| bmi <= 18.5 = "You're underweight, you emo, you!"
| bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
| bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
| otherwise = "You're a whale, congratulations!"
where bmi = weight / height ^ 2
```

BMI – ANOTHER VERSION…

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
| bmi <= skinny = "You're underweight, you emo, you!"
| bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
| bmi <= fat = "You're fat! Lose some weight, fatty!"
| otherwise = "You're a whale, congratulations!"
where bmi = weight / height ^ 2
skinny = 18.5
normal = 25.0
fat = 30.0
```

- The names defined after where are visible across the Guards
- They give us the advantage of not having to repeat ourselves
- If we decide that we want to calculate BMI a bit differently, we only have to change it once.
- It also improves readability by giving names to things
- It can make our programs faster since bmi variable here is calculated only once
- The names we define in the where section of a function are only visible to that function
- All the names are aligned at a single column if they are a part of the same block
- Bindings aren't shared across function bodies of different patterns
- If you want several patterns of one function to access some shared name, you have to define it globally.

USING WHERE BINDINGS TO PATTERN MATCH!

- Rewrite the where section of the previous function as

```
...
where bmi = weight / height ^ 2
(skinny, normal, fat) = (18.5, 25.0, 30.0)
```

A TRIVIAL FUNCTION TO GET A FIRST AND A LAST NAME AND BACK INITIALS.

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
 where (f:_) = firstname
       (l:_) = lastname

ghci> initials "Abraham" "Lincoln"
"A. L."
```

LET BINDINGS

- Let Bindings are similar to where bindings

Syntax: let <bindings> in <expression>

- Normally, bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards.
- Let bindings let you bind to variables anywhere and are expressions themselves, but are very local,

  so they don't span across guards.
- Let bindings can also be used for pattern matching.
- Example: Function to calculate surface area of a cylinder

```
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
let sideArea = 2 * pi * r * h
     topArea = pi * r ^2
in  sideArea + 2 * topArea
```

Indentation is important !

```
ghci> :l surfarea.hs
[1 of 1] Compiling Main             ( surfarea.hs, interpreted )
Ok, one module loaded.
ghci> cylinder 20 30
6283.185307179587
```

DIFFERENCE BETWEEN LET AND WHERE BINDINGS

Let Binding	Where Binding
let bindings are expressions themselves	where bindings are just syntactic constructs

- Uses of Let Bindings
  - In replacing Expressions

```
ghci> 4 * (if 10 > 5 then 10 else 0) + 2
42
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

  - To introduce functions in a local scope:

```
ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]
```

VARIANTS OF LET

- Separate several inline variables with semicolons:

  ghci> (let a = 100; b = 200; c = 300 in a*b*c, let foo="Hey "; bar = "there!" in foo ++ bar)

  (6000000,"Hey there!")

- Pattern matching with let bindings:

  ghci> (let (a,b,c) = (1,2,3) in a+b+c) * 100

  600

- Let bindings inside list comprehensions:

  calcBmis :: (RealFloat a) => [(a, a)] -> [a]

  calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]

No need for a semicolon after the last binding but you can if you want !!

Let inside a comprehension is possible only if it doesn't filter the list and it only binds to names

VARIANTS OF LET

```
calcBmis :: (RealFloat a) => [(a, a)] -> [a]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi >= 25.0]
```

We can't use the bmi name in the (w, h) <- xs part because it's defined prior to the let binding.

The names defined in a let inside a list comprehension are visible to the output function (the part before the |) and all predicates and sections that come after of the binding. So we could make our function return only the BMIs of fat people

- The in part of the let binding is omitted in list comprehensions because the visibility of the names is already predefined there.
- However, we could use a let in binding in a predicate and the names defined would only be visible to that predicate.

VARIANTS OF LET

ghci> let zoot x y z = x * y + z

ghci> zoot 3 9 2

29

ghci> let boot x y z = x * y + z in boot 3 4 2

14

ghci> boot

<interactive>:1:0: Not in scope: `boot'

- The in part can also be omitted when defining functions and constants directly in GHCi.
- If we do that, then the names will be visible throughout the entire interactive session.

CASE EXPRESSIONS

- Case expressions are much similar to if else expressions and let bindings.
- Not only can expressions be evaluated based on the possible cases of the value of a variable, but also can do pattern matching.
- The syntax for case expressions is pretty simple:

```
case expression of pattern -> result
pattern -> result
pattern -> result
...
```

Expression is matched against the patterns.

- The first pattern that matches the expression is used.
- If it falls through the whole case expression and no suitable pattern is found, a runtime error occurs.

- These two pieces of code do the same thing and are interchangeable:

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x
```

```
head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

EXAMPLE USAGE OF CASE EXPRESSIONS

- While Pattern matching on function parameters can only be done when defining functions, case expressions can be used pretty much anywhere.
- They are useful for pattern matching against something in the middle of an expression.

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

Because pattern matching in function definitions is syntactic sugar for case expressions, this is also possible !

```
describeList :: [a] -> String
describeList xs = "The list is " ++ what xs
    where what [] = "empty."
    what [x] = "a singleton list."
    what xs = "a longer list."
```