Core Haskell

ECS713 : Functional Programming
Week 02

Prof. Edmund Robinson
Dr Paulo Oliva

Week 2: Lecture Plan

1. More on lists: Range and comprehension
2. Type declarations (type vs data)
3. Application and composition
4. Declaring functions, patterns and guards
5. If-then-else expressions
6. Case-of expressions / let expressions
7. Offside rule

Week 1 Review

Our First Program

```
-- does line begin with "TEL"?
isphone s = (take 3 s) == "TEL"

-- remove up to colon
strip s = init.tail $ dropWhile notcolon s
   where
      notcolon c = c /= ':'

-- phone list
getPhones card = phones
   where
      all_lines = lines card
      phone_lines = filter isphone all_lines
      phones = map strip phone_lines
```

Haskell Programs

Aim: get you writing Haskell program files

- Haskell program: Sequence of definitions
- We've seen "value definitions"
  - functions count as values
- In a program file (unlike ghci) declarations do not begin with let
  - we'll talk about let later

Expressions and Declarations

- At the value level, Haskell has
  - **expressions**:
    terms that represent values
  - **declarations**:
    binding of identifiers (name) to expressions
- Haskell is a full higher-order language:
  - no distinction between function expressions and other expressions, e.g. Boolean or Int

Lists: Ranges and Comprehension

Lists

- Definitions can run over multiple clauses

```
nlplus []     = 0
nlplus [x]    = x
nlplus (x:y:ys) = x + y
```


three patterns to cover all cases

Note use of same name in different clauses

List Ranges

- Haskell has a number of tricks for writing lists
- These can make life easier
- Includes generating lists by **ranges** (works for types like Integer and Char)
  - [1..4] = [1,2,3,4]
  - [‘a’..’d’] = [‘a’,’b’,’c’,’d’] = “abcd”
- You can have different intervals:
  - [0,2..8] or ['a','c'..'g']

Infinite lists

- You can also (in Haskell, but not F#) have infinite lists:
  - [1,2..]
  - [0,2..]

List Comprehension

And there is a technique for generating lists

```
ghci> [x*x | x <- [1..9]]
[1,4,9,16,25,36,49,64,81]
ghci> [x*x | x <- [1..9], odd x]
[1,9,25,49,81]
ghci> [(x*x)+y | x <- [1..6], odd x, y <- [1..4]]
[2,3,4,5,10,11,12,13,26,27,28,29]
ghci> let vs = ['a','b']
ghci> [ [x,y,z] | x<-vs, y<-vs, z<-vs ]
["aaa","aab","aba","abb","baa","bab","bba","bbb"]
```

Type Declaration

Type Declarations

- Types are critical in Haskell
- So it is important that you can also declare types. From Prelude


```
type String = [Char]
```

- declares String as a synonym for [Char]
- they are effectively the same type (just different names)

Enumerated Types

More interestingly, you can declare **new types**

- The simplest are types that just contain a fixed number of values, e.g. from Prelude


```
data Bool = True | False
```

- This defines the type of booleans
- Similar types useful for e.g. returning flags

Type Declarations

More interestingly, you can declare **new types**

- Use the **data** keyword

- In general, can use **recursion**:


```
data Queue = Head Int Queue | Last Int
```

```
data BTree = Node String BTree BTree
           | Leaf String
           | Empty
```

```
data BTree = Node String BTree BTree
           | Leaf String
           | Empty
```

- This defines a type BTree that contains binary trees whose nodes are labelled by Strings

- There are three “constructors”:
  - Node, Leaf, Empty
- Their names begin with upper case letters (function names can’t start with upper case)

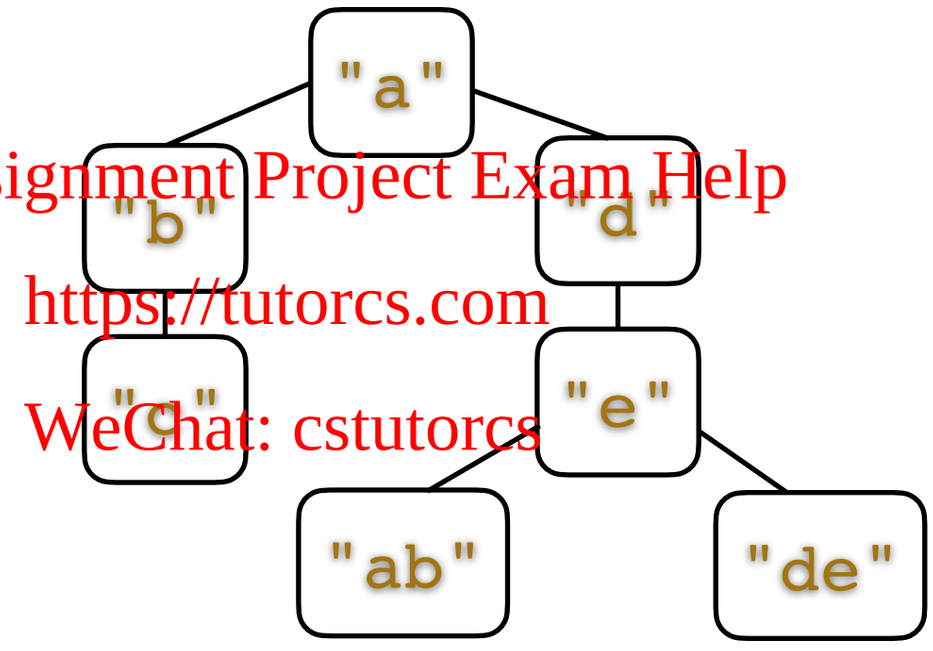
```
data BTree = Node String BTree BTree
           | Leaf String
           | Empty
```

- **Node** is a constructor that requires a string and two other trees, the left and right subtrees

- **Leaf** is a tree consisting of just one node labelled by a string
- **Empty** is just a constant: the empty tree (like the empty list)

```
data BTree = Node String BTree BTree
           | Leaf String
           | Empty
```


A tree:




```
btreeExample = Node "a"
                 (Node "b" (Leaf "c")  Empty)
                 (Node "d"
                    (Node "e" (Leaf "ab") (Leaf "de"))
                     Empty)
```

(Algebraic) Datatypes

```
btreeExample = Node "a"
                 (Node "b" (Leaf "c")  Empty)
                 (Node "d"
                    (Node "e" (Leaf "bb") (Leaf "de"))
                    Empty)
```


- The elements of an algebraic datatype can be thought of as expressions
- They’re like arithmetic expressions: (3+4)*5
- But the operations are data constructors

Parametrised Types

- We can also do this generically (polymorphically)


```
data MyBtree a =
       Node a (MyBtree a) (MyBtree a)
     | Leaf a
     | Empty
```

- Binary tree with values of type **a**

```
type BoolTree = MyBtree Bool
type CharTree = MyBtree Char
```

Function Application and Function Composition

Function Application

Function application is denoted by writing the function next to the argument

```
succ 5
```

```
map even xs
```

```
lines card
```


- Don't need brackets!

- Brackets are for **grouping**, not application

```
f 3 == f (3) == (f 3)
```

```
double 3 + 4 == (double 3) + 4
```

```
double (3+4) == 14
```

```
ghci> let double n = 2 * n
ghci> double(3)
6
ghci> double 3
6
ghci> (double)3
6
ghci> double 3 + 1
7
ghci> double (3 + 1)
8
ghci> (double 3) + 1
7
ghci> (double) 3 + 1
7
ghci> (double 3 + 1)
7
```

Function Application

- Haskell also has explicit syntax for function application ($)
- f $ 3 is f applied to 3
- $ binds very weakly, so that
  - f $ 3+4 is equivalent to f (3+4)
- Useful to avoid writing parenthesis, e.g.


```
strip s = init.tail $ dropWhile notcolon s
```

Function Application

- Also very useful in constructing data pipelines
- Data starts at right and feeds through to the left


```
getPhones card =
      map strip $ filter isphone $ lines card
```


- instead of

```
getPhones card = phones
   where
      all_lines = lines card
      phone_lines = filter isphone all_lines
      phones = map strip phone_lines
```

Function Composition

Haskell also has a syntax for **function composition**

- This means chaining functions together

  - f n = double succ $ n

  - This defines the function that increments n by 1, and then doubles the result

```
getPhones = (map strip).(filter isphone).lines
```

First Program Revisited

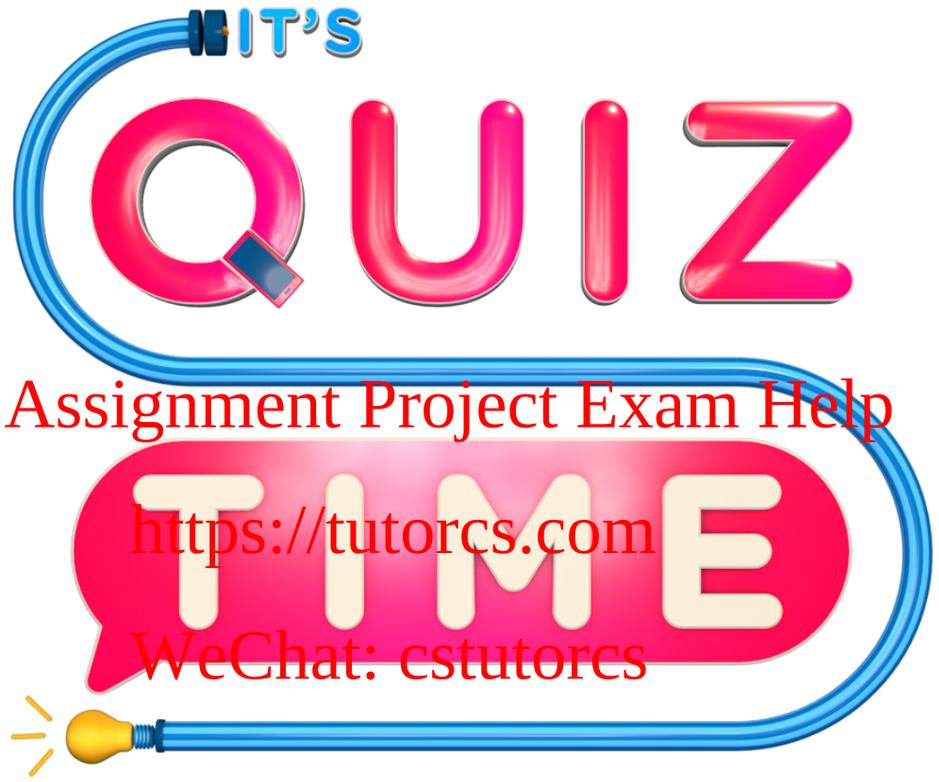
```
-- does line begin with "TEL"?
isphone = (=="TEL").(take 3)

-- remove up to colon
strip = tail.(dropWhile (/=':'))

-- phone list
getPhones = (map strip).(filter isphone).lines
```

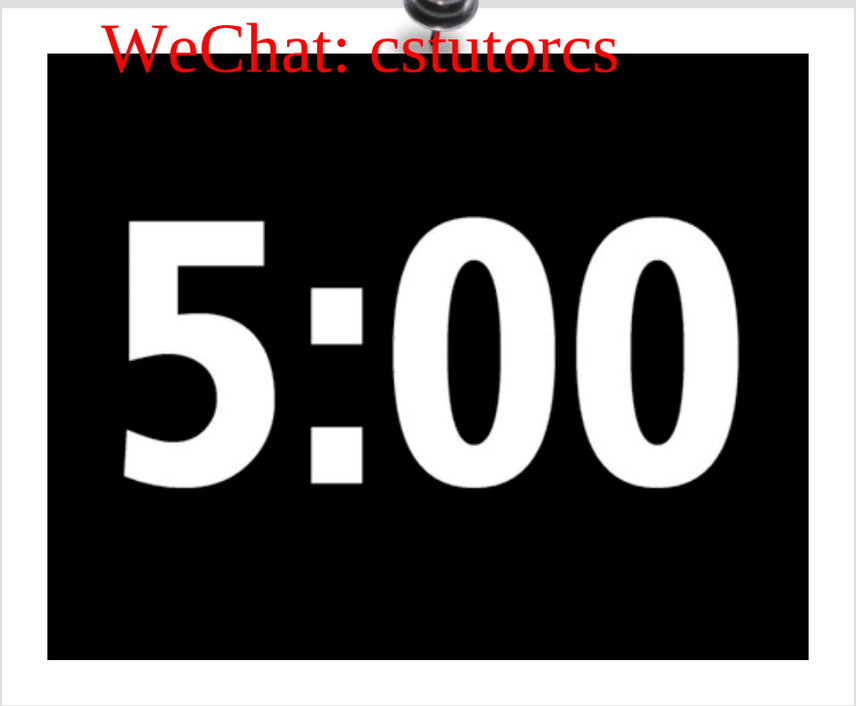

or

```
-- extract phones from vcard
getPhones = (map strip).(filter isphone).lines
  where strip = tail.(dropWhile (/=':'))
        isphone = (=="TEL").(take 3)
```



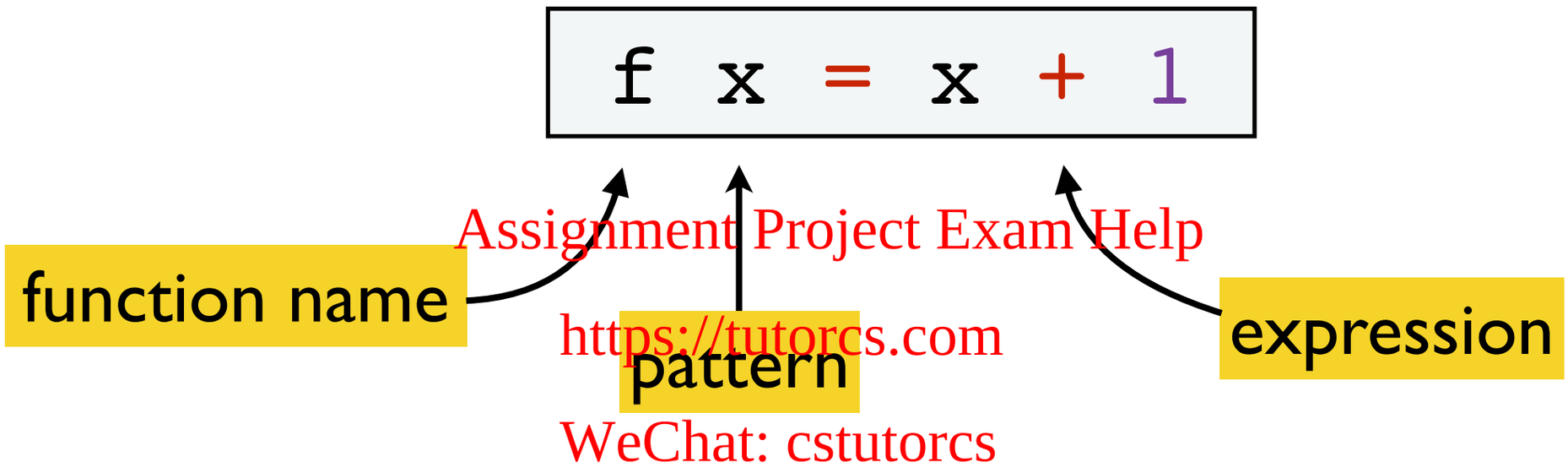
https://cloudsurvey.co.uk

Short Break



Function Declarations

Declaring Functions



- A number of lines, so patterns cover all possible cases
- pattern can be as simple as a single name

Declaring Functions

Function to calculate size of a binary tree

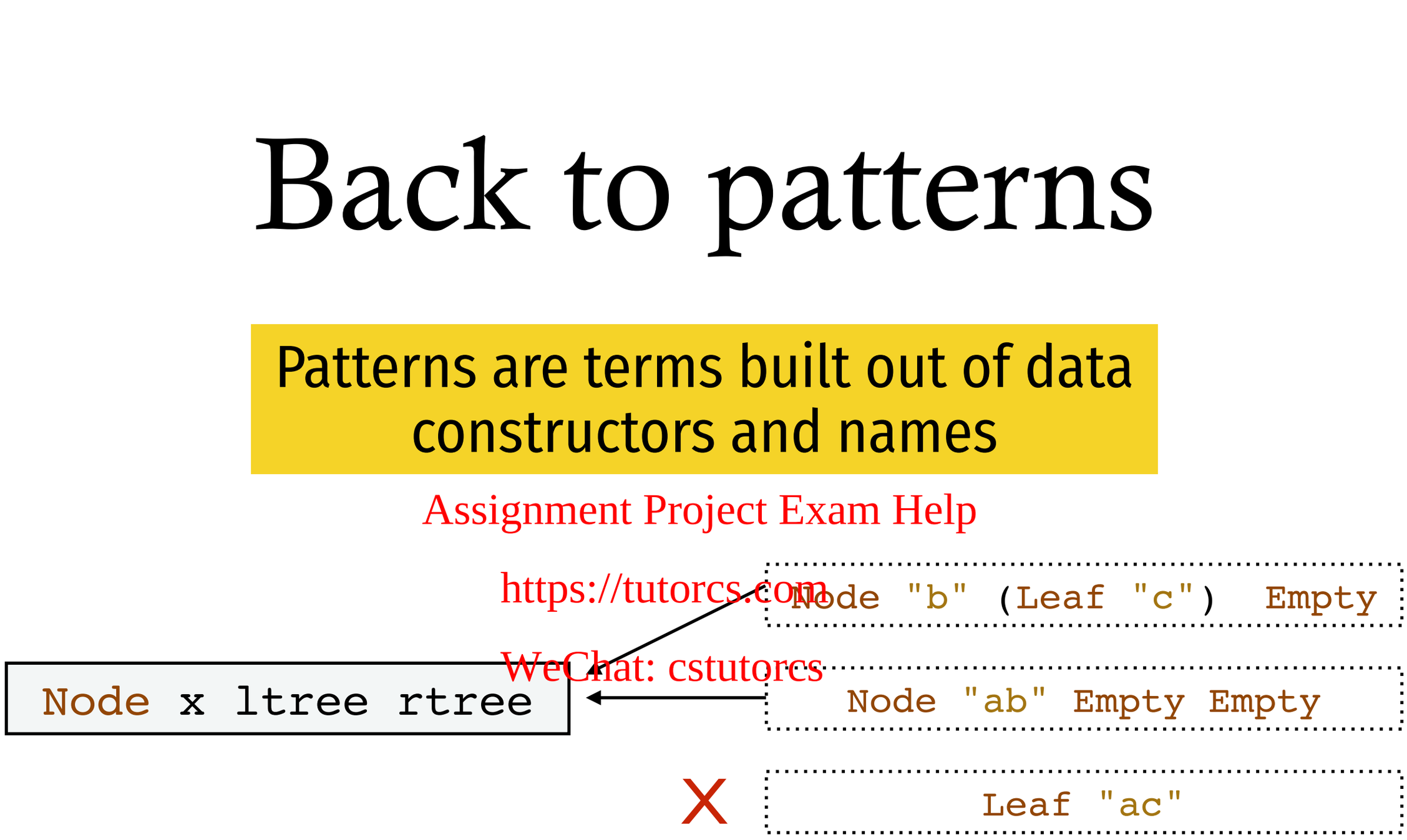
```
data BTree = Node String BTree BTree
           | Leaf String
           | Empty
```


use underscore when actual value is not needed

```
size Empty = 0
size (Leaf _) = 1
size (Node _ ltree rtree) = 1 + l + r
   where
      l = size ltree
      r = size rtree
```

Back to patterns

Patterns are terms built out of data constructors and names



Inbuilt lists and tuples actually fit into this framework: they're just sugar (another Landinism)

Declaring Functions

... or pattern can be nothing at all


```
age = 43
```

function name

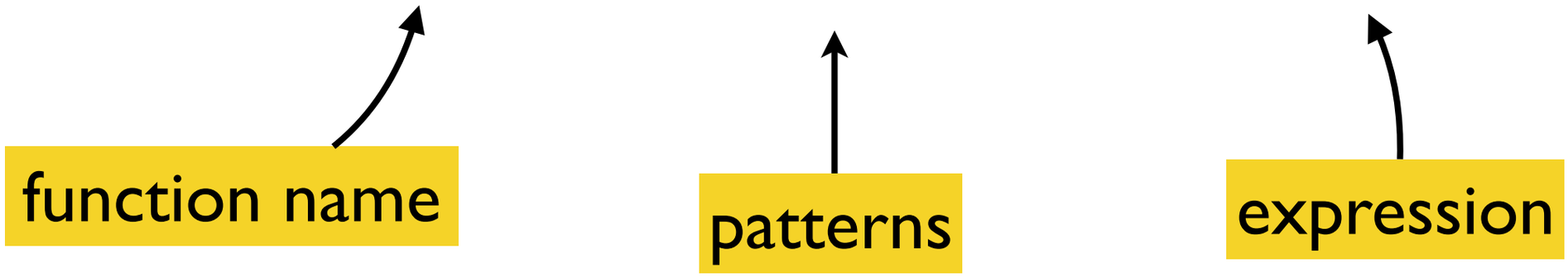
pattern

expression

Declaring Functions

... or multiple patterns


```
threeplus x y z = x + y + z
```

Warning

- Do not start function names (including constants) with uppercase
  - Haskell reserves this types!
- Don’t use the same name twice in the same clause, e.g.
  - f x x = x + x

Guards

- A “guard” is a condition you put on a clause to control when it is executed
- Declarations can be guarded
- This lets you define cases based on more than a pattern


```
fib x | x > 1 = fib (x-1) + fib (x-2)
      | otherwise = 1
```

- **otherwise** not required, but provides a default

Where-Declarations

- Only used in function declarations
- The “where” creates local names for the current pattern


```
strip s = init.tail $ dropWhile notcolon s
   where
      notcolon c = not (c==':')
```

- Can make code look very clear, but it’s less easily manipulable

if-then-else and case-of expressions

Basic Expressions

```
n + 1
```

```
not (c == ':')
```

```
ys ++ (tail xs)
```



```
-- n + 1 is an arithmetical expression
succ n = n + 1

-- not (c == ':') is an boolean expression
notcolon c = not (c == ':')

-- ys ++ (tail xs) is a list expression
replaceHead xs ys = ys ++ (tail xs)
```

if-then-else expressions

```
if b then e1 else e2
```

- Given
  - a boolean expression b :: Bool
  - and two expressions e1, e2 :: a
- We can form an if-expression


```
-- tail [] throws an exception
safeTail xs = if xs == [] then [] else tail xs
```

Declaring Functions

using if-expressions:


```
goo x = if x<0 then 0 else x*x
```


case-of expressions

```
case exp of pattern1 -> result1
            pattern2 -> result2
            ...
            patternN -> resultN
```


- Like if, but can check multiple patterns

```
-- tail [] crashes the program
safeTail xs = case xs of [] -> []
                         (y:ys) -> ys
```

let expressions

```
let x = exp1 in exp2
```

- Creates a local named expression
- Name x can then be used in expression exp2

```
let pi=3.14 in pi*r*r
```

- You can also make multiple local declarations:

```
-- area of a circle
area r = let {pi=3.14; rsq = r*r} in pi*rsq
```

Offside Rule

Offside Rule

- Group definitions using indentation (also called the *layout rule*)


```
isphone s = (take 3 s) == "TEL"
strip s = tail (dropWhile (/=':') s)
getPhones card = phones
   where
      all_lines = lines card
      phone_lines = filter isphone all_lines
      phones = map strip phone_lines
```

Offside Rule

- Also possible to use { } and ;
  (but not recommended)


```
foo = f1.f2
   where
      f1 n = n + 1
      f2 n = 2 * n
```

```
foo = f1.f2
   where {
      f1 n = n + 1;
      f2 n = 2 * n
   }
```

```
foo = f1.f2
   where { f1 n = n + 1 ; f2 n = 2 * n }
```

Offside Rule

- Also for **let**

```
goo x =
    let
        y = x + 1
        z = 2 * y
    in
        x + y + z
```


goo declaration must fit into shaded box

Offside Rule

- Also for **let**

```
goo x =
   let
       y = x + 1
       z = 2 * y
in
   x + y + z
```


…this is fine

Offside Rule

- Also for **let**

```
goo x =
    let
        y = x + 1
        z = 2 * y
in
    x + y + z
```


…this is NOT fine

Offside Rule

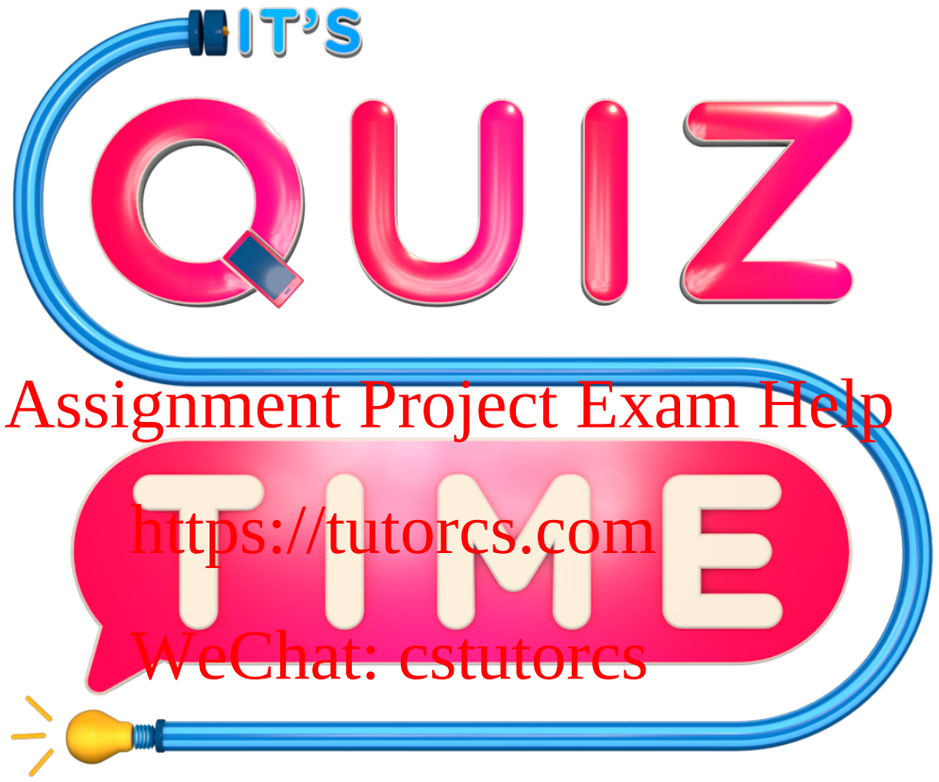
- Also for **case**

this is fine!

```
goo xs = case xs of
   [] -> "Empty list"
   otherwise -> "Non-empty list"
```


this is NOT fine!

```
goo xs = case xs of
   [] -> "Empty list"
  otherwise -> "Non-empty list"
```



https://cloudsurvey.co.uk

Reading

- Learn you a Haskell for Great Good
  Miran Lipovača, Chapters 2 - 4
  http://learnyouahaskell.com/chapters
- Real World Haskell
  B. O'Sullivan et al, Chapters 2 - 4
  http://book.realworldhaskell.org/read/
- Haskell 2010 Language Report (QM+)