Week 3

Ch 3: Syntax in Functions
Ch 4: Hello, Recursion!

University of the Fraser Valley

Dr. Russell Campbell

Russell.Campbell@ufv.ca

COMP 481: Functional and Logic Programming

1

Overview

2

— Pattern Matching —

3

Pattern Matching (1)

```
Int -> String
 "LUCKY NUMBER SEVEN!"
 "Sorry, you're out of luck, pal!"
```

4

Pattern Matching (2)

- order of cases matters, as having a pattern with variable `x` first would match any input:

```
sayMe :: Int -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe x = "Not between 1 and 3!"
```

- notice the last case does not use argument `x`
- we could replace unused variables with underscore
- `_` is known as a temporary variable

5

Pattern Matching (3)

```
Integer -> Integer
```

6

— Pattern Matching with Tuples —

7

Pattern Matching with Tuples (1)

```
(Double, Double) -> (Double, Double) -> (Double, Double)

(Double, Double) -> (Double, Double) -> (Double, Double)
```

8

Pattern Matching with Tuples (2)

- we can make our own functions for pulling elements out of triples, similar to `fst` and `snd` for pairs:

```
first :: (a,b,c) -> a
first (x, _, _) = x

second :: (a,b,c) -> b
second (_, y, _) = y

third :: (a,b,c) -> c
third (_, _, z) = z
```


9

— Pattern Matching with Lists —

10

Pattern Matching with List Comprehensions

```
let xs = [ (1,3),(4,3),(2,4),(5,3),(5,6),(3,1) ]
```

```
[ a+b | (a, b) <- xs ]
```

- using pattern matching in the above list comprehension gives the result:

```
[ 4, 7, 6, 8, 11, 4 ]
```

- if one of the tuples in the list does not match the pattern, the list comprehension moves to the next tuple

11

Pattern Matching with Lists (1)

```
(Show a) => [a] -> String
  "List is empty!"
      "List has one element: "
        "List has two elements: "              " and "
       "Long list; 1st two items:                " and "
```

12

Pattern Matching with Lists (2)

- a common pattern is `x:xs`, especially in recursive functions
  - the above will match a singleton with the one head value as `x` and the empty list as `xs`
  - otherwise, `x` is the first element, and `xs` the tail
- `[]` can have elements added to the front of the list with `:`
  - e.g.: re-implementation of the `head` function:
    - `head' (x:_) = x`

13

As-Patterns (2)

```
@
@
        @

   String -> String
""   "Empty string, whoops!"
          "The first letter of "          " is "
```

14

— Guards —

15

Guards (1)

```
(Ord a) => a -> a -> a
```

16

Guards (2)

- more complex cases can be used to define a function with the Sheffer stroke `|` as a "guard"
- a guard begins successive lines and must be indented with at least one space
- each guard is followed by a Boolean expression
- if the expression result is `False`, the next guard will be tested
- the expression among many guards that evaluates to `True` will be executed for the function
- the last guard can take care of remaining cases with keyword `otherwise` in place of the Boolean expression
- if no guards or patterns match, then an exception is thrown

17

— `where` Clauses —

18

`where` Clauses (1)

- guards can use variables defined in a final block of code starting with keyword `where`
- these variables have a scope only inside the where block, so that any variable names do not pollute the global namespace

```
tellRatio :: Double -> Double -> String
tellRatio x y
    | r < zero  = "That is a negative ratio."
    | r < small  = "That is a fractional ratio."
    | r < substantial = "That is a substantial ratio."
    | r < large = "That is a large ratio!"
    | True = "Whatever, that ratio is ridiculously huge!"
    where
{
    r = x / y;
    zero = 0;
    small = 1;
    substantial = 10;
    large = 100;
}
```


19

`where` Clauses (2)

```
{ }
;
where
```

20

— Local vs Global Scope —

21

Local vs Global Scope

```
                                    where

                                    "Hello"
                                    "Bye"

                                    String -> String
"Juan"                                      ", Juan!"
"Fernando"                                  ", Fernando!"
                                                "!"
```

22

— Pattern Matching with `where` —

23

Pattern Matching with `where`

```
Double -> Double -> String

     "That is a negative ratio."
      "That is a fractional ratio."
           "That is a substantial ratio."
     "That is a large ratio!"
"Whatever, that ratio is ridiculously huge!"
```

24

Pattern Matching with `where`

- another example (but it could be done shorter with pattern matching in the function definition)

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


25

— Functions in `where` Blocks —

26

Functions in `where` Blocks

- we may want to define a function in a `where` block to make use of applying it to each element in a list

```
calcRatios :: [(Double, Double)] -> [Double]
calcRatios xs = [ratio x y | (x, y) <- xs]
    where
    ratio x y = x / y
```


27

— Keyword `let` —

28

Keyword `let`

- the keyword `let` begins bindings to define variables you can use elsewhere within another expression following `in` keyword
- the syntax is `let <bindings> in <expression>`

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


29

Comparing `where` and `let`

`let`

`where`

`let`

- `let square x = x * x in (square 5, square 3, square 2)`
- `let a = 100; b = 200; c = 300 in a*b*c`
- `(let (a,b,c) = (1,2,3) in a+b+c) * 100`

`let`

`where`

30

Replacing `where` with `let`

- going back to see the `tellRatio` example and we will replace the `where` clause with a `let` expression:

```
calcLetRatios :: [(Double, Double)] -> [Double]
calcLetRatios xs = [ratio | (x, y) <- xs, let ratio = x / y]
```

- we can use the `let` expression everywhere but in the generator part of the list comprehension, i.e.: `(x, y) <- xs`
- it is also possible to specify further filters using the `let` expression:

```
calcLetRatios :: [(Double, Double)] -> [Double]
calcLetRatios xs = [ratio | (x, y) <- xs, let ratio = x / y, ratio > 0.25]
```


31

— `case` Expressions —

32

`case` Expressions (1)

- `case` keyword begins expressions much like the `let` keyword

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

- expressions such as `case` can be used many places
- the first set of braces makes layout syntax unavailable
  - so, lines are completed with semicolons
- OR just use layout syntax without braces for the whole expression, but then we must use proper indentation

33

`case` Expressions (2)

```
case

[a] -> String
  "The list is "    case    of
"empty."
 "a singleton list."
"a longer list."
```

```
[a] -> String
  "The list is "
where
"empty."
 "has one element."
"has many elements."
```

34

— Chapter 4: Hello, Recursion! —

35

Recursive Functions (1)

```
(Ord a) => [a] -> a
        "There is no maximum for an empty list!"
```

36

Recursive Functions (2)

```
replicate' :: (Eq b) => Int -> b -> [b]
replicate' x y
    | x <= 0 = []
    | True = y:(replicate' (x - 1) y)
```


37

Recursive Functions (3)

```
(Integral a, Eq b) => a -> [b] -> [b]
```

```
otherwise       True
```

38

Recursive Functions (4)

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = (reverse' xs) ++ [x]
```


39

Recursive Functions (5)

```
:: a -> [a]
```

40

Recursive Functions (6)

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```


41

Recursive Functions (7)

```
(Eq a) => a -> [a] -> Bool
```

42

— Quicksort —

43

Quicksort

```
(Ord a) => [a] -> [a]
```

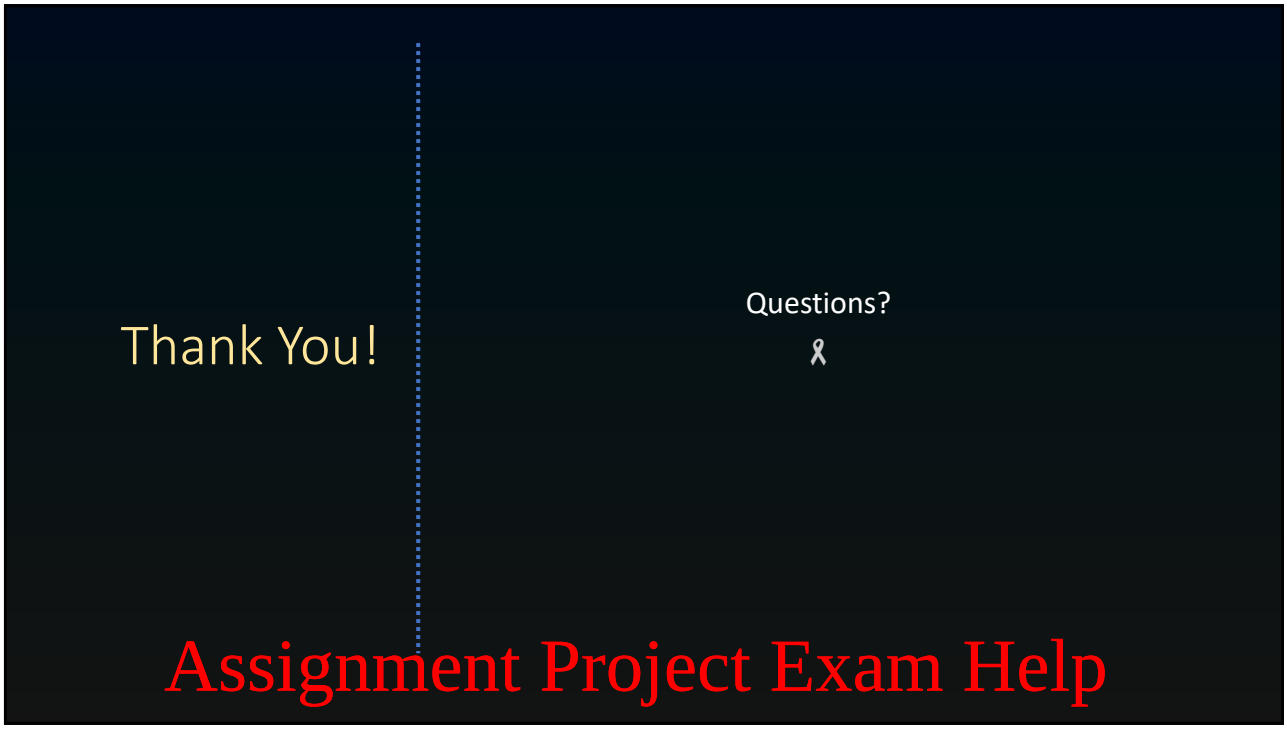
44

— Designing with Recursion —

45

Designing with Recursion

46



47