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

Chapter 3:

- pattern matching
  - with tuples
  - with list comprehensions
- as-patterns
- guards
- where clauses
- local vs global scopes
- pattern matching with `where`
- functions in `where` blocks
- keyword `let`
- `case` expressions

Chapter 4:

- recursion
- recursive functions
  - `replicate`
  - `take`
  - `reverse`
  - `repeat`
  - `zip`
  - `elem`
- quicksort
- designing with recursion

2

— Pattern Matching —

3

Pattern Matching (1)

- functions pattern match in a similar way conditional code executes different branches or cases
- the following example defines a function to return different strings:

```
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

- calling the `lucky` function with input 7 will match the first pattern describe
- calling `lucky` with any other input value will match with the last pattern description

Consider how much more code it would take to write the following function when required in a language that uses `if`-statements.

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)

- consider the factorial function but defined using recursion:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

- If you try the above and call `factorial 50`, what do you notice about the output?

- to call a function where the input does not match any of the patterns will cause an exception
- then try to always describe a last pattern that will take care of all other possible inputs

6

— Pattern Matching with Tuples —

7

Pattern Matching with Tuples (1)

- consider defining a function in the two ways below:

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

let addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
    addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

- the second version clearly has input tuples, increasing readability

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)

- we need to use parentheses around a pattern matched argument with parts
- an example of returning a value regardless of what kind of list might be input:

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

- keep in mind that `_` matches with any length list, even an empty list,
- the 2nd pattern matches for a list with exactly two elements
  - so the last pattern only matches with lists of longer length.

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)

- we can give an alternative pattern to simplify references
- use the `@`
- prefix `@` with a name you want to reference the whole pattern
- an example of using `@`:

```
firstLetter :: String -> String
firstLetter "" = "Empty string, whoops!"
firstLetter all@(x:_) = "The first letter of " ++ all ++ " is " ++ [x]
```

14

— Guards —

15

Guards (1)

- an example of using guards:

```
max' :: (Ord a) => a -> a -> a
max’ x y
    | x <= y        = y
    | otherwise     = x
```

- the keyword `otherwise` can be replaced with Boolean value `True`

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)

- note that you can leave out the braces `{ }`
- and the semicolons `;`
- but then the variables will need to be indented at least as far as the indentation of the `where` keyword

20

— Local vs Global Scope —

21

Local vs Global Scope

- beware that the `where` block has local scope
  - only for its immediately preceding guards,
  - and no previous function definitions or patterns

```
messageHi  = "Hello"
messageBye = "Bye"

greet :: String -> String
greet "Juan" = messageHi ++ ", Juan!"
greet "Fernando" = messageHi ++ ", Fernando!"
greet name = messageBye ++ ", " ++ name ++ "!"
```

22

— Pattern Matching with `where` —

23

Pattern Matching with `where`

- we could rewrite the ratio example to be more concise with pattern matching used in the `where` clause

```
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, zero, small, substantial, large) = (x / y, 0, 1, 10, 100)
```

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

- because `let` is an expression, you can use it anywhere an expression can be used
- `where` must be used at the end of a function definition
- `let` expressions can define functions in a local scope
  - `let square x = x * x in (square 5, square 3, square 2)`
- more than one binding can be included by separating them with semicolons
  - `let a = 100; b = 200; c = 300 in a*b*c`
- tuples make binding more concise
  - `(let (a,b,c) = (1,2,3) in a+b+c) * 100`
- unfortunately, `let` expressions cannot be used across guards due to their local scope
- some prefer `where` clauses to keep the function body closer to its referenced name

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)

- another example where a `case` is given further nested within the description:

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

- equivalently:

```
describeList :: [a] -> String
describeList ls = "The list is " ++ what ls
    where
    what [] = "empty."
    what [x] = "has one element."
    what xs = "has many elements."
```

34

— Chapter 4: Hello, Recursion! —

35

Recursive Functions (1)

The following are recursive functions to help practice all the concepts learned so far.

```
max' :: (Ord a) => [a] -> a
max' [] = error "There is no maximum for an empty list!"
max' [x] = x
max' (x:xs) = max x (max' xs)
```

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)

```
take' :: (Integral a, Eq b) => a -> [b] -> [b]
take' n _
 | n <= 0 = []
take' n (x:xs) = x : take' (n-1) xs
```

- note in the above that there is no `otherwise` or last `True` guard so that matching will move on to test the next pattern

38

Recursive Functions (4)

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


39

Recursive Functions (5)

```haskell
repeat' :: a -> [a]
repeat' x = x : repeat' x
```

- the above creates an infinite list of an element we pass in
- use in combination with another function that will cut off an infinite number of the elements in some way
- we would really only want to use it together with `take`,
  - for example, `take 5 $ repeat' 3`

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)

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x  = True
    | True    = elem' a xs
```

42

— Quicksort —

43

Quicksort

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) =
    let
        left = [a | a <- xs, a <= x]
        right = [a | a <- xs, a > x]
    in
    (qsort left) ++ [x] ++ (qsort right)
```

44

— Designing with Recursion —

45

Designing with Recursion

How could smaller subproblem solutions be used toward building up a larger problem solution?

- nothing to stop you from thinking in the reverse
  - larger problem solution split to smaller subproblem solutions
- neither will work unless there is a true base case(s)
- tends to work well with data structures that can be constructed with recursion
- general theorems could be reduced in scope by restricting to data structures or subproblems to a more specific kind that are then solved with recursion

46

Thank You!

Questions?

47