Recursion

ECS713 : Functional Programming
Week 03

Prof. Edmund Robinson
Dr Paulo Oliva

Recap

Week 01

- Introduced Haskell
- Program to get phone numbers from vcard


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


-- phone list
getPhones vcard = map strip phonelines
        where phonelines = filter isphone (lines vcard)
```

Week 02

- Function declaration

```
f x = x + 1
```



```
g (x:xs) = x
```


- Anonymous function

```
\x -> x + 1
```

(Haskell)

```
lambda x: x + 1
```

(Python)

Week 02

- "let" versus "where"

```
\x -> let {sq=x*x; pi=3.14} in
          pi * sq
```


```
\x -> pi * sq
          where {sq=x*x; pi=3.14}
```


```
$ ghci
GHCi, version 7.10: http://www.haskell.org/ghc/
Loading packages ...
Prelude> (\x -> let {sq=x*x;pi=3.14} in pi*sq) 10
314.0
Prelude> (\x -> pi*sq where {sq=x*x;pi=3.14}) 10
<interactive:1:15 parse error on input 'where'
```

Week 02

- conditionals

```
empty xs = if xs==[] then "yes" else "no"
```

if-then-else


```
empty [] = "yes"
empty (_:_) = "no"
```


pattern matching


```
empty xs = case xs of
             [] -> "yes"
             _:_ -> "no"
```

case expression

```
empty xs | xs==[] = "yes"
         | otherwise = "no"
```

guards

Week 02

- data types, data constructors
- lists:


```
-- list range, xs = [1,2,3,4,5]
xs = [1..5]

-- infinite lists, ys = [1,2,3,4,5...
ys = [1..]

-- list comprehension, zs = [1,4,9,16,25]
zs = [x*x | x<-[1..5]]
ws = [x*x | x<-[1..5], odd x]
```

Week 3: Lecture Plan

1. Recursive Function Definitions
2. Recursive Type Definitions
3. Recursive List Definitions
4. Multiple Recursion
5. Mutual Recursion
6. Multiple Arguments

Recursive Definitions

Recursive Functions

- Definition of a function that uses the function itself, e.g.


```
f 0 = 1
f n = 5 * f (n-1)
```


- Basic mechanism for looping in Haskell
- Most common example: factorial function

```
fact 1 = 1
fact n = n * fact (n-1)
```

If you don't know what recursion is, read this sentence.

Google

recursion

Search

About 1,560,000 results (0.23 seconds)

Web

Images

Maps

Videos

News

Shopping

More

London, UK

Change location

The web

Pages from the UK

More search tools

Did you mean: ***recursion***

Recursion - Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/**Recursion**

Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...

Formal definitions of recursion - Recursion in language - Recursion in mathematics

Recursion (computer science) - Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/Recursion_(computer_science)

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. The approach can ...

ThinkGeek :: **Recursion**

www.thinkgeek.com › ... › Unisex Shirts › IT Department

Recursion - 10 print. ... **Recursion** Main Image. Buy this and earn. 350Geek Points Geek Points is a program to reward you, our incredibly cool customers, ...

Google Helps You Understand **Recursion**

googlesystem.blogspot.com/.../google-helps-you-un...

by Alex Chitu

23 Jul 2009 – Google uses the "did you mean" feature, which normally corrects misspellings, to illustrate a nerdy joke: defining the word "**recursion**" using ...

Recursion

introcs.cs.princeton.edu/23**recursion**

2 Mar 2012 – **Recursion** is a powerful general-purpose programming technique, and is ... The HelloWorld for **recursion** is to implement the factorial function, ...

Unfolding

- Given a recursively defined function

```
f 0 = 1

f n = 5 * f (n-1)
```


- We can evaluate f at a given value by "unfolding" the definition. E.g.


```
unfolding f 3

f 3 = 5 * f 2
    = 5 * 5 * f 1
    = 5 * 5 * 5 * f 0
    = 5 * 5 * 5 * 1
    = 125
```

Edge Conditions


```
-- this function diverges (does not terminate)
f x = 3 * f (x-1)

-- we need edge conditions (base cases)
f x | x<0  = error "illegal argument"
    | x==0 = 1
    | otherwise = 3 * f (x-1)
```

Non-termination

- What about the function


```
f n = 5 * f (n+1)
```


- Non-terminating loops correspond to ill-defined recursive functions

Recursion on Integers

Recursion on Integers

```
-- file: factorial.hs
factorial 1 = 1
factorial n = n * factorial (n-1)
```



```
$ ghci
GHCi, version 7.10: http://www.haskell.org/ghc/
Loading packages ...
Prelude> :load factorial.hs
[1 of 1] Compiling Main     ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> factorial 10
3628800
```

Choice of Definitions

```
factorial n = if n==1 then 1 else n * factorial (n-1)
```

```
factorial 1 = 1
factorial n = n * factorial (n-1)
```



```
*Main> factorial (-5)
```



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

```
*Main> factorial (-5)
** Exception: Non-exhaustive patterns in function factorial
```

Deal with Illegal Arguments

```
factorial(-3)
 = -3 * factorial(-4)
 = -3 * -4 * factorial(-5)
 = ...
```



```
factorial n
      | n<1  = error "illegal argument"
      | n==1 = 1
      | n>1  = n * factorial (n-1)
```

Recursion on Lists

Recursion on Lists

```
-- file: list-product.hs
lproduct :: [Int] -> Int
lproduct [] = 1
lproduct (n:ns) = n * lproduct ns
```



```
$ ghci
GHCi, version 7.10: http://www.haskell.org/ghc/
Loading packages ...
Prelude> :load list-product.hs
[1 of 1] Compiling Main     ( factorial.hs, interpreted )
Ok, modules loaded: Main.
*Main> lproduct [2,1,4]
8
```

Recursion on Lists

```
-- calculate length of a list
length :: [a] -> Int
length [] = 0
length (_:ns) = 1 + length ns
```



```
-- concatenate two lists
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```



```
-- reverse a list
reverse :: [a] -> [a]
reverse [] = []
reverse (n:ns) = reverse ns ++ [n]
```

Recursive Data Types

```haskell
-- File: tree.hs
data Tree = Leaf Int | Node String Tree Tree

tree1 = Leaf 4

tree2 = Leaf 7

tree3 = Node "Very" tree1 (Node "Nice" tree1 tree2)

addLeaves (Leaf n) = n
addLeaves (Node _ t1 t2) = (addLeaves t1) + (addLeaves t2)
```



```
Prelude> :load tree.hs
[1 of 1] Compiling Main     ( tree.hs, interpreted )
Ok, modules loaded: Main.
*Main> :type addLeaves
addLeafs :: Tree -> Int
*Main> addLeaves tree3
15
```

Recursively Defined Lists

Recursively Defined Lists

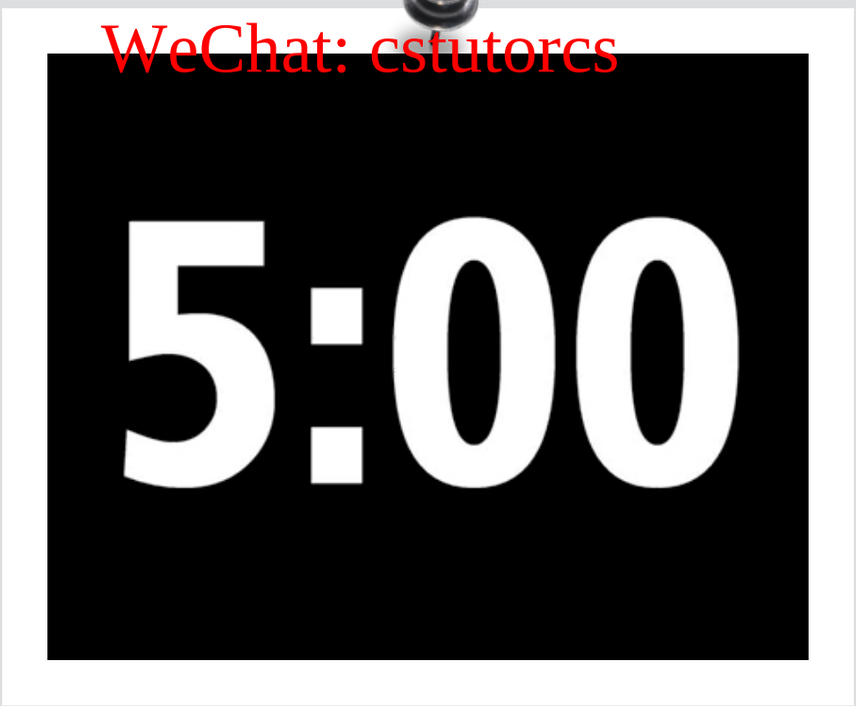
```
$ ghci
GHCi, version 7.10: http://www.haskell.org/ghc/
Loading packages ...
Prelude> let xs = 0:xs
Prelude> :type xs
xs :: Num a => [a]
Prelude> take 10 xs
[0,0,0,0,0,0,0,0,0,0]
Prelude> let ys = 0:1:ys
Prelude> :type ys
ys :: Num a => [a]
Prelude> take 20 ys
[0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]
```

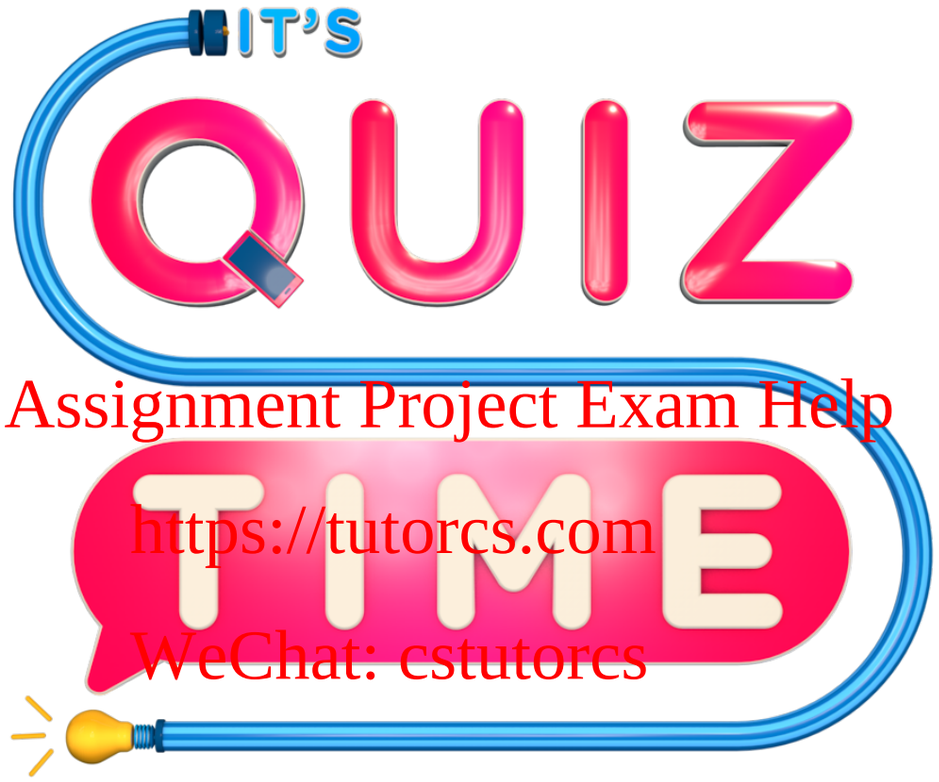
Recursively Defined Lists

```
Prelude> let power = 1:(map (*2) power)
Prelude> take 10 power
[1,2,4,8,16,32,64,128,256,512]

Prelude> let fib = 1:1:(zipWith (+) fib (tail fib))
Prelude> :type fib
fib :: Num a => [a]
Prelude> take 10 fib
[1,1,2,3,5,8,13,21,34,55]
```

Short Break





https://cloudsurvey.co.uk

Multiple Recursion

Multiple Recursion

```
-- fibonacci function


fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n-1) + fibonacci (n-2)
```



```
unfolding fibonacci 4
fibonacci 2 = fibonacci 0 + fibonacci 1
            = 1 + 1 = 2
fibonacci 3 = fibonacci 1 + fibonacci 2
            = 1 + 2 = 3
fibonacci 4 = fibonacci 2 + fibonacci 3
            = 2 + 3 = 5
```

Mutual Recursion

Mutual Recursion

```
-- evens and odds

even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

Multiple Arguments

Multiple Arguments

- The function
  `fact :: Int -> Int`
  takes a **single** argument

- The function
  `max : Int -> Int -> Int`
  takes **two** arguments

- We can define a function that takes multiple arguments recursively on both arguments

Multiple Arguments

```
-- drop first n elements from a list
drop 0 xs = xs
drop n [] = []
drop n (_:xs) = drop (n-1) xs
```



```
-- add two lists
-- e.g. addLists [1,2] [10,20] = [11,22]
addLists [] _ = []
addLists _ [] = []
addLists (x:xs) (y:ys) = (x+y) : addLists xs ys
```

Final Comments

Recursion Five Steps

1. Define the type
2. Enumerate the cases
3. Define the simple cases
4. Define other cases
5. Generalise and simplify


```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop n [] = []
drop n (_:xs) = drop (n-1) xs
```

References

- Programming in Haskell
  Graham Hutton, Chapter 6
- Learn you a Haskell for Great Good
  Miran Lipovača, Chapter 5