Programming Paradigms 2024
Session 5 : Recursion

Hans Hüttel

15 October 2024

Our plan for today

1. The learning goals
2. Presentations of the discussion problems
3. Problem no. 1
4. Discussion
5. Problem no. 2
6. Break
7. Problem no. 3
8. Discussion
9. Problem no. 4
10. Problem no. 5
11. If time allows: More problems at your own pace.
12. We evaluate today's session – please stay until the end!

Learning goals

- To understand the structure of a recursive function definition
- To be able to read and write recursive function definitions on lists
- To be able to read and write recursion definitions that make use of multiple recursion or mutual recursion
- To be able to write recursive function definitions in a structured fashion

Discussion problem – replicate

Define the function `replicate` – and *use pattern matching* in your solution. This function takes an integer n and and an element x and gives us a list with n elements where x has been repeated exactly n times.

As an example, `replicate 3 5` should give us `[5,5,5]`. What should the type of `replicate` be?

Discussion problem – improve

Define the function improve – and *use pattern matching* in your solution. It takes a list xs and, if xs contains at least two elements, it gives us a list where every other element has been removed.

As an example, `improve [1,2,3,4,5,6,7]` should give us `[1,3,5,7]`. What should the type of improve be?

About the problems

- When you work on the problems in this problem set (and in the future), use the approach outlined in Section 6.6 of the book – learning this strategy is a way to reach the learning goals of this part of the course.
- **Avoid "headtailery"!** (using head, tail etc. for taking lists apart) Pattern matching is your friend here. Use it instead.
- **Avoid "intboolery"!**, that is, wrongly assuming that whole numbers and truth values are all there is. Polymorphism is all over the place! Try having your type specification as a comment at first and use type inference to find the actual type. Types are often more general than one assumes.

Problem 1 – reverse (10 minutes)

The function reverse appears in the Haskell prelude. It will reverse a list such that e.g. reverse [1,2,3] evaluates to [3,2,1].

Now it is your task to define your own version of this function, rev.

Discussion – Descending lists

A list $[a_1, a_2, \ldots, a_n]$ is *descending* if $a_1 \geq a_2 \geq \ldots \geq a_n$. The list descending [6,5,5,1] is descending. The list descending ["plip","pli","ppp"] is not.

A famous stand-up comedian has defined a function descending that will return True if a list is descending and False otherwise.

Here is what the stand-up comedian wrote.

```
descending (x:y:xs) = if (x >= y)
                      then True
                      else False
```

Will this work ?

Problem 2 – descending (15 minutes)

Now it is your turn!

Write a function descending that will return True if a list is descending and False otherwise.

Discussion – isolate

The function isolate takes a list l and an element x and returns a pair of two new lists (l1 , l2). The first list l1 is a list that contains all elements in l that are not equal to x. The second list l2 is a list that contains all occurrences of x in l.

- isolate [4,5,4,6,7,4] 4 evaluates to ([5,6,7],[4,4,4]) .
- isolate ['g','a','k','a'] 'a' evaluates to (['g','k'], ['a','a']) .

A very confident Java programmer wrote the following definition of isolate (also found on the Moodle page as `isolate.hs`) using their Java skills.

```
isolate ys x = if (head ys == x) == False
                    then ([(head ys)]
                           ++ fst (isolate (tail
                               ys) x),
                                snd (isolate (
                                   tail ys) x)
                                   )
               else
                            (fst (isolate (tail
                               ys) x),[(head ys
                               )]
                           ++ (snd (isolate (
                              tail ys) x)))
```

The Java programmer exclaimed: Why does everything have to be so clumsy in Haskell?

Comments, please!

Problem 3 – Your own isolate (20 minutes)

Now it is your turn: Define isolate in Haskell using recursion[1] but *without* using fst, snd, head or tail.

What should the type of isolate be?

Major hint: Place the recursive call in a where-clause and use pattern matching to find the components in the result of that call.

[1] No list comprehension – that was last week!

Problem 4 – Wrapping up (20 minutes)

The function wrapup is a function that takes a list and returns a list of lists. Each list in this list contains the successive elements from the original list that are identical.

For instance,

wrapup [1,1,1,2,3,3,2] should give us [[1,1,1],[2],[3,3],[2]]

wrapup [True,True,False,False,False,True] should give us [[True,True],[False,False,False],[True]].

Define wrapup in Haskell using recursion[2] but *without* using fst, snd, head or tail. What is the type of wrapup?

[2] No list comprehension – that was last week!

Problem 5 – Triples (20 minutes)

A former minister of science and education has decided to get a university degree and is now trying to define a Haskell function triples that takes a list of tuples (each tuple has exactly 3 elements) and converts that list of tuples into a tuple of lists.

triples [(1,2,3) , (4, 5, 6), (7, 8, 9)] should produce ([1,4,7], [2, 5, 8], [3, 6, 9]).

The minister wrote the following but ran into problems. What seems to be wrong?

```
triples :: Num a => [(a,a,a)] -> ([a],[a],[a])

triples [] = ()
triples [(a,b,c)]= ([a],[b],[c])
triples (x:xs,y:ys,z:zs) = [x,y,z] : Triples [(xs,
    ys,zs)]
```

Can you fix these issues? How can Section 6.6 help you here?

Evaluation

- What did you find difficult?
- What surprised you?
- What went well?
- What could be improved? How does the setup work this time?
- Is there a particular problem that we should follow up on with a short video?