Programming Paradigms 2022
Session 4: Recursion

Problems for solving and discussing

Hans Hüttel

4 October 2022

How you should approach this

- Some problems are meant to be solved in groups. Please work together at the table for those. Screen sharing at your table is best.
- Other problems are meant to be solved in pairs. Please work together two by two, not in groups. Screen sharing at your table is not a good idea for these problems.
- We set a limit on how much time you are going to spend on each of the numbered problems. *If you do not manage to solve a problem* within the time set aside, do not worry about that. We will discuss solutions afterwards. *If you finish early*, work on one of the "lettered problems" in the problem set instead of working on the next numbered problem.
- 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.
- Pattern matching is your friend. Use it.

Problems that we will definitely talk about

1. ***(Everyone at the table together – 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. First try to find out what the type of rev should be and follow the overall approach described in Section 6.6.

2. ***(Solve in pairs – 10 minutes)***

   In an earlier session we used other functions from the prelude to define the function last that finds the last element of a list.

   Now it is your task to give a recursive definition of mylast (which you should call it, since last exists in the prelude). First try to find out what the type of mylast should be and follow the overall approach described in Section 6.6.

3. ***(Everyone at the table together – 15 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 the list `[[1,1,1],[2],[3,3],[2]]` and

   wrapup `[True,True,False,False,False,True]` should give us the list `[[True,True],[False,False,False],[True]]`.

   Define wrapup in Haskell. First try to find out what the type of wrapup should be and follow the overall approach described in Section 6.6.

4. ***(Solve in pairs – 10 minutes)***

   The function rle is a function that, when given a list xs produces a list of pairs of elements of xs and integers[1]. This list of pairs has its elements appears in the order that they appeared originally and contains (x, n) if there are n successive occurrences of x in the list. For instance

   ```
   rle ['a','a','a','g','g','b','a','a']
   ```

   should give us the list [('a',3) ,('g',2) ,('b',1) ,('a',2)] and

   ```
   rle [1,1,1,2,2,1,3,3]
   ```

   should give us [(1,3) ,(2,2) ,(1,1) ,(3,2)].

   Define rle in Haskell. First try to find out what the type of rle should be and follow the overall approach described in Section 6.6.

5. ***(Everyone at the table together – 15 minutes)***

   A former minister of science and education now wants to get a masters degree and is learning Haskell. The minister is trying to construct a 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 piece of code and a type specification 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?

More problems to solve at your own pace

Here, too, Section 6.6 is helpful.

a) 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']).

   Define isolate in Haskell. What should the type of isolate be?

b) Define a function amy that will tell us if any elements of a list satisfy a given predicate.

   For instance, if

   ```
   odd x = ((x `mod` 2) == 1)
   ```

   then

   ```
   amy odd [2,5,8,3,7,4]
   ```

   should return True, whereas

   ```
   amy odd [2,8,42]
   ```

   should return False.

c) Create a function frequencies that, given a string s, creates a list of pairs [(x1,f1) ,....(xk,fk)] such that if the character xi occurs a total number of fi times throughout the list s, then the list of pairs will contain the pair (xi, fi).

   As an example of this,

   ```
   frequencies "regninger"
   ```

[1] This function computes what is called a run-length encoding, thus its name.

should return the list

```
[('r',2),('e',2),('g',2),('n',2),('i',1)]
```

First find out what the type of the function should be.

d) A theorem in number theory states that every non-zero real number x can be written as a *continued fraction*. This is a potentially infinite expression of the form

$$x = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cfrac{1}{\cdots \cfrac{1}{a_n}}}}}} \tag{1}$$

For rational numbers, the a_i's will eventually all be 0, so the continued fraction is finite; for irrational numbers, the continued fraction will be infinite. See e.g. [1] for more.

The goal of this problem is to write a Haskell function cfrac that will, given a real number r and a natural number n, finds the list of the first n numbers in the continued fraction expansion of r. What should the type of cfrac be?

Bibliography

[1] Wikipedia. Continued fractions. `https://en.wikipedia.org/wiki/Continued_fraction`.