Programming Paradigms Core - Functional Programming

Course: B-CS-MOD08A-2B-202001039 B-CS Programming Paradigms Core 202001039

Number of questions: 9

Generated on: Jun 1, 2021

Contents: Pages:

Programming Paradigms Core - Functional Programming

Course: B-CS Programming Paradigms Core 202001039

- You have on your table:
  - Materials supplied by us (Chromebook, scrap paper, possibly instructions provided by the Remindo team)
  - Your student card, put it at the front-left side of your table now (then we can easily check it)
  - Pens or pencils for writing on scrap paper
  - If you have it: the book "Learn You a Haskell for Great Good!" in English without written notes (marking is ok)
- Calculators, laptops, mobile phones, smart watches, etc. are not allowed! *Put them in your bag (switched off*!)
- If you have questions about the content of the exam (e.g., if you think you found a mistake), use the chat. We cannot answer questions that are related to your understanding of the material (e.g., "what is the function 'head'?").
- This is an open book exam. During the exam you may use the online book "Learn You A Haskell for Great Good!" [clickable]
- Some questions have specific requirements for the solution (e.g., "use recursion"). You can only receive points for a solution if these requirements are met precisely.
- You may use predefined Haskell functions and operators from the packages Prelude, Data.List, Data.Char, Data.Maybe, Data.Either, System.IO, and Control.Applicative (unless a question explicitly does not allow the use of functions, of course). See See attachment: HaskellOperatorsAndFunctions.pdf for examples of operators and functions that you may use.
- It is not allowed to use Monads and do-notation.

Number of questions: 9

1 The function **scanl** has the type:

5 pt.

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

scanl is similar to **foldl**, but outputs also the intermediate results.

For example:

```
scanl (+) 0 [1,2,3]
```

gives

```
[0,1,3,6]
```

Define **scanl** using **recursion**

2 The function **cycle** takes a list and creates an infinite repetition of that list. For example

10 pt.

```
take 8 (cycle [3,2,5])
```

gives

```
[3,2,5,3,2,5,3,2]
```

Provide the **type signature** of **cycle** and define it using *only* **list comprehension** (no helper functions, use of other functions, recursion, etc)

3 The function **find** gives the first value in a *list* for which a predicate holds. It results in the value **Nothing** when no such value can be found.

10 pt.

For example

```
find (\(x,y) -> x+y == 5) [(1,2),(4,2),(2,3),(4,4),(5,0)]
```

results in

```
Just (2,3)
```

Give the **type signature** of **find** and define it using only **foldr**. Feel free to use a helper function or lambda abstraction for the function supplied to **foldr**.

4 10 pt.

The function **zipFiles** receives two file names and defines an IO action that:

1. Reads the two files
2. Creates a list of tuples of type **(String,String)**, where the n-th tuple in the list contains the n-th line from the first file and the n-th line from the second file.

If one file contains less lines than the other, this file determines the number of resulting tuples.

Important: The function **lines :: String -> [String]** gives all the lines within a multiline **String**

Give the **type signature** of **zipFiles** and define it using the **applicative style**.

5 5 pt.

The function **maximum** gives the highest value in a list. For example:

```
maximum [4,2,5,1]
```

results in **5**.

Give the **type signature** of **maximum**, and then use the **point-free style** to define **maximum.** You *must use* **sort** (recall that it sorts a list in *ascending* order!) and **head** in this definition.

6 5 pt.

The function **mzip :: [Maybe a] -> [Maybe b] -> [(a,b)]** is similar to **zip**, but combines two values pairwise only when *both* are a **Just**.

For example:

```
mzip [Just 1, Just 2, Nothing, Just 4] [Just 'a', Nothing, Just 'b', Just 'c']
```

Gives **[(1,'a'),(4,'c')]**

Define **mzip** using **recursion**.

7 This question is about sorted binary search tree. For this question these trees meet the following requirements:

- All nodes contain a value and two subtrees
- Leaves contain no value
- For each node with a value x, all values (in the nodes) of the left subtree are smaller than x
- For each node with a value x, all values (in the nodes) of the right subtree are larger than x
- There *never* no two nodes with the same value.

A Haskell data type for sorted binary search trees is:

```
data Tree a = Node a (Tree a) (Tree a)
            | Leaf
```

5 pt. **a.** Define the function **treeins** that inserts a value in a sorted binary tree by replacing one of the leaves, and give the **type signature**. You may assume this value does not occur in the tree. For example:

treeins 2 (Node 3 (Node 1 Leaf Leaf) Leaf) = Node 3 (Node 1 Leaf (Node 2 Leaf Leaf)) Leaf)

5 pt. **b.** Use **foldl/foldr** (choose the most suitable) to define a function **list2tree :: Ord a => [a] -> Tree a** that produces a sorted binary tree from a list.

You may assume that **treeins** is properly defined (even if you skipped the previous question).

3 pt. **c.** Define a function **tree2list** that creates a sorted list from a sorted binary tree.

2 pt. **d.** The function **treesort** that sorts a list by using sorted binary trees. Give the **type signature** of **treesort** and define it using **point-free style**

8 In this question we work with an abstract data type **Stream** that describes an infinite sequence of values of *any type*. Both its type constructor and data constructor are called **Stream**.

Example: the follow definition is a **Stream** with an infinite number of 0's:

```
zeros :: Stream Integer
zeros = Stream 0 zeros
```

4 pt. **a.** Define the abstract data type **Stream**, which should be compatible with the definition of **zeros**.

1 pt. **b.** Define a function **toList** that produces a list from a **Stream** and give its **type signature**.

5 pt. **c.** Make **Stream** an instance of **Functor**

5 pt. **d.** Is it possible to define a different **Functor** for **Stream**? If "yes", provide such an instance, if "no" explain why. Mention the **functor laws** in your answer.

15 pt. **e.** Define an **Applicative** instance for **Stream** that can be used to combine values from multiple **Stream**s (combine values with the same offset within the streams).

5 pt. **f.** The function **zipstream :: Stream a -> Stream b -> Stream (a,b)** combines two **Stream**s to one **Stream** of tuples. Define this function using the **applicative style**.

9 5 pt. The mathematical sequence $a_i = 1$ can be described in Haskell as the recursive definition

```
a = 1 : a
```

For this question we would like a recursive definition of the sequence $r_i = i$

Use a **higher-order function** to provide a recursive definition of **r :: [Integer]** that makes use of the function **ones**

Name:				Signature:
Date:	/ /	Birth date:	/ /	

Course: B-CS-MOD08A-2B-202001039 B-CS Programming Paradigms Core 202001039 - Questions: Programming Paradigms Core - Functional Programming

- You have on your table:
  - Materials supplied by us (Chromebook, scrap paper, possibly instructions provided by the Remindo team)
  - Your student card, put it at the front-left side of your table now (then we can easily check it)
  - Pens or pencils for writing on scrap paper
  - If you have it: the book "Learn You a Haskell for Great Good!" in English without written notes (marking is ok)
- Calculators, laptops, mobile phones, smart watches, etc. are not allowed!
  Put them in your bag (switched off!)
- If you have questions about the content of the exam (e.g., if you think you found a mistake), use the chat. We cannot answer questions that are related to your understanding of the material (e.g., "what is the function 'head'?").
- This is an open book exam. During the exam you may use the online book "Learn You A Haskell for Great Good!" [clickable]
- Some questions have specific requirements for the solution (e.g., "use recursion"). You can only receive points for a solution if these requirements are met precisely.
- You may use predefined Haskell functions and operators from the packages Prelude, Data.List, Data.Char, Data.Maybe, Data.Either, System.IO, and Control.Applicative (unless a question explicitly does not allow the use of functions, of course). See See attachment: HaskellOperatorsAndFunctions.pdf for examples of operators and functions that you may use.
- It is not allowed to use Monads and do-notation.

1
5 pt.

Answer:

2
10 pt.

Answer:

3
10 pt.

Answer:

4
10 pt.

Answer:

2

4

5
5 pt.

Answer:

2

6
5 pt.

Answer:

2

4

6

8

10

12

7
15 pt.

a.

Answer:

b.

Answer:

c.

Answer:

d. Answer:

2

4

6

8

8
35 pt.

a.

Answer:

2

b.

Answer:

2

4

6

8

c.

Answer:

2

4

6

8

d.

Answer:

e.

Answer:

f.

Answer:

9
5 pt.

Answer:

2

Correction model

1.
5 pt.

Correction criterion	Points
Base case correct	1 point
Correct definition of the recursive case. Uses recursion correctly (arguments, pattern matching, apply function in recursion)	2 points
Well-typed recursive code	1 point
Works correctly as required by question and uses recursion	1 point
Total points:	*5 points*

2.
10 pt.

Correction criterion	Points
Correct type	2 points
Generators in the correct order (or other relevant solution)	3 points
Use of two generators	3 points
Correct answer	2 points
Total points:	*10 points*

3.
10 pt.

Correction criterion	Points
Correct type signature	2 points
Correct use of foldr (order of arguments for find and foldr	2 points
Correct definition of the function/lambda abstraction provided to foldr. This includes order of the arguments.	3 points
Nothing is used correctly	2 points
Completely correct and elegant solution	1 point
Total points:	*10 points*

4.
10 pt.

Correction criterion	Points
Correct type signature	2 points
lines <$> readFile (elegancy)	3 points
Elegancy of solution	3 points
Correctness of answer	2 points
Total points:	*10 points*

5.
5 pt.

Correction criterion	Points
Correct type signature	1 point
Gives the maximum value (not the minimum); reverse needs to be used	1 point
Functions in function composition are in the correct order and match types	3 points
Total points:	*5 points*

6.
5 pt.

Correction criterion	Points
Base case []	1 point
Other base cases	1 point
Correctness of recursive case	3 points
Total points:	*5 points*

7.
15 pt.

a.

Correction criterion	Points
Correct type	1 point
Correct equation for Leaf	1 point
Correct equation for Node	3 points
Total points:	*5 points*

b.

Correction criterion	Points
Correct use of foldr with treeins	1 point
Starting value and list supplied	1 point
Correct answer	3 points
Total points:	*5 points*

c.

Correction criterion	Points
Equation for Leaf	1 point
Equation for Node	2 points
Total points:	*3 points*

d.

Correction criterion	Points
Correct implementation without arguments (only one correct answer)	2 points
Total points:	*2 points*

8.
35 pt.

a.

Correction criterion	**Points**
Correct type constructor	2 points
Correct data constructor	2 points
Total points:	*4 points*

b.

Correction criterion	**Points**
Correct definition of toList (0 or 3 points)	1 point
Total points:	*1 point*

c.

Correction criterion	**Points**
Correct "instance" line	2 points
Correct implementation	3 points
Total points:	*5 points*

d.

Correction criterion	**Points**
Points for correct motivation for first "No". No points for incorrect/missing motivation.	2 points
Points for correct motivation for second "No". No points for incorrect/missing motivation.	3 points
Total points:	*5 points*

e.

Correction criterion	**Points**
Instance line correct	2 points
Pure correct	5 points
<*> correct	8 points
Total points:	*15 points*

f.

Correction criterion	**Points**
Correct definition with applicative style (0 or 5)	5 points
Total points:	*5 points*

9.
5 pt.

Correction criterion	Points
Use of zipWith	2 points
Correct definition	3 points
Total points:	*5 points*

Caesura

Points scored	Grade
100	10
99	9.9
98	9.8
97	9.7
96	9.6
95	9.5
94	9.4
93	9.3
92	9.2
91	9.1
90	9.0
89	8.9
88	8.8
87	8.7
86	8.6
85	8.5
84	8.4
83	8.3
82	8.2
81	8.1
80	8.0
79	7.9
78	7.8
77	7.7
76	7.6
75	7.5
74	7.4
73	7.3
72	7.2
71	7.1
70	7.0
69	6.9
68	6.8
67	6.7
66	6.6
65	6.5
64	6.4
63	6.3
62	6.2
61	6.1
60	6.0
59	5.9
58	5.8
57	5.7
56	5.6
55	5.5
54	5.4
53	5.3
52	5.3
51	5.2
50	5.1
49	5.0
48	4.9
47	4.8
46	4.8
45	4.7
44	4.6
43	4.5
42	4.4
41	4.4
40	4.3

39	4.2
38	4.1
37	4.0
36	3.9
35	3.9
34	3.8
33	3.7
32	3.6
31	3.5
30	3.5
29	3.4
28	3.3
27	3.2
26	3.1
25	3.0
24	3.0
23	2.9
22	2.8
21	2.7
20	2.6
19	2.6
18	2.5
17	2.4
16	2.3
15	2.2
14	2.1
13	2.1
12	2.0
11	1.9
10	1.8
9	1.7
8	1.7
7	1.6
6	1.5
5	1.4
4	1.3
3	1.2
2	1.2
1	1.1
0	1.0

Question identifiers

These identifiers can be used to track the exact origin of the question. Use these identifiers together with the identifier of this document when sending in comments about the questions, so that your comment can be connected precisely with the question you are referring to.

Document identifier: 5552-7035

Question number	Question identifier	Version identifier
1	59022	f0e0f818-5a63-d81a-0776-80794aa51b87
2	59027	a05dfa76-563a-ebe2-3d52-47c6fa7672b9
3	59032	345292e3-b0f1-9267-56a0-a331d80da047
4	59042	499f1f7c-ea11-52df-351a-af9f7e86ef75
5	59037	3a66139d-7642-6f5c-14d8-4634c56d69b1
6	59047	80816cef-192f-9969-f817-eaddbc3e2430
7	59147	75da5f5a-27af-9f2c-2dde-117cf5d8c345
8	59052	b28974db-c754-0934-67bf-99e33cfb586b
9	59057	5288a52f-42ca-dccf-00cd-af86de35413e