Middle East Technical University

Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring 2015-2016

Take Home Exam 2

Due date: 15 April 2016, Friday, 23:55

1 Objectives

This homework aims to familiarize you with more advanced functional programming concepts by implementing operations on a recursive data structure using Haskell.

2 Problem Definition

In this homework, you are going to implement functions that operate on **feature structure**, which is basically a set of nested feature-value pairs prominently utilized in Natural Language Processing domain. You may consider the feature structure as a list of features and values that may happen to be feature lists as well.

Data type of feature structure is defined as follows in Haskell.

```haskell
data FeatureTerm = Simple String | Struct FeatureStruct
data FeatureStruct = FS [(String, FeatureTerm)]
```

Having this recursive definition of data structure in question, feature structures can be employed to store any kind of generic information with layered dependencies, apart from their traditional use in NLP problems.

Consider XML-like representation of information on a student whose name is Paul born in 1996. Let us assume that Paul is taking PL and OS courses this semester in the major programme they are registered to, having a 2.98 CGPA together with other information included in the subsequent data definition.

`FeatureStruct` is going to be used as an Abstract Data Type having its own module, and rather than calling its constructor, it will **only** be initialized or modified through accessor functions, which are to be defined in the next section. Consider the data definition below for illustrative purposes.

```
paul = FS [ ("cgpa", Struct $ FS [ ("cumulative", Simple "2.98"),
                                   ("previous", Simple "3.12")]),
            ("courses", Struct $ FS [ ("OS", Struct $ FS [ ("mt1", Simple "66")]),
                                      ("PL", Struct $ FS [ ("hws", Struct $
                                      FS [ ("hw1", Simple "NA"),("hw2", Simple "78")])
                                                                      ])
                                        ]),
            ("dob", Simple "1996"),
            ("name", Simple "Paul")]
```

Information relevant to Paul's major program is now stored in `paul` variable in a rather free format. Next, let us assume that Paul is also registered to some minor program, in which held information could be slightly different, as the following definition indicates.

```
paulM = FS [ ("cgpa", Struct $ FS [ ("mincum", Simple "3.45"),
                                    ("minprev", Simple "3.55")]),
             ("courses", Struct $ FS [ ("OS", Struct $ FS [ ("fin", Struct $ FS [("bon", ↩
                 Simple "11"),("ques", Simple "77")]),
                                                        ("term", Simple "55")]),
                                       ("SIG", Struct $ FS [("lab", Simple "94")])]),
             ("name", Simple "Paul"),
             ("pcode", Simple "EEX04")]
```

Feature structures defined in `paul` and `paulM` will be used throughout the homework text to exemplify the implementation specifics elaborated in subsequent sections.

Feature structures can also allow backward/cross-dependencies resulting in Directed Acylic Graphs. However, for the sake of simplicity, you should assume the possibility of **only** rooted or unrooted tree structures throughout the tasks.

3 Specifications

In this second homework, you are going to implement and test functionality using `FeatureStruct` to achieve construction, removal, comparison, unification and intersection operations for data manipulation.

3.1 The `HW2` Module

You are going to implement `FeatureStruct` as an Abstract Data Type encapsulated in the module defined below.

```
module HW2(FeatureStruct, FeatureTerm(Simple, Struct), emptyfs, getpath, ↩
    addpath, delpath,
            union, intersect)
where
```

Then, it would be imported an utilized in other files as:

```
import HW2
```

Note that constructor functions will **not** be visible to modules importing `HW2`.

Implementation details for the module are elaborated in the next two subsections.

3.2 `FeatureStruct` and `FeatureTerm` as an Instance of Typeclasses

`FeatureStruct` data type should implement the functionality of the `Show` and `Eq` typeclasses for display and comparison purposes.

3.2.1 `Show` for `FeatureTerm` and `FeatureStruct`

```
instance Show FeatureTerm where
    ...
```

Display of `FeatureTerm` instances should obey the following specifications.

- Each `FeatureTerm` in the form of `(Simple string)` should be display with "`string`" that is present in the `FeatureTerm`.
- Each `FeatureTerm` in the form of `(Struct featurestruct)` should be displayed with "`(show featurestruct)`" that is present in the `FeatureTerm`.

```
instance Show FeatureStruct where
    ...
```

Display of `FeatureStruct` instances should obey the following specifications.

- Empty `FeatureStruct` instances should be displayed as empty list, i.e. "`[]`".
- Each `FeatureStruct` variable should begin and end with square brackets and be shown in dedicated single lines.
- Nested `FeatureStruct` entries should be displayed as nested lists, and they should be placed after their parent node following "`=>`" string.
- List elements should be separated by "`, `" strings (composed of comma and a whitespace character).
- Regarding `(String, Simple String)` pairs, the second constituent should be displayed in quotation marks.

Student `paul` defined previously is shown on GHCI as

```
> paul
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws
=>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
```

Similarly, paulM can be printed-out as follows.

```
> show paulM
"[cgpa=>[mincum=\"3.45\", minprev=\"3.55\"], courses=>[OS=>[final=>[bonus=
\"11\", ques=\"77\"], term=\"55\"], SIG=>[lab=\"94\"]],name=\"Paul\", pcode=
\"EEX04\"]"
```

3.2.2 Eq only for FeatureStruct

```
instance Eq FeatureStruct where
    ...
```

You should implement the required functionality to check whether given two FeatureStruct variables are equal or not.

For two instances to be equal, they should contain the same values located on the same paths.

Checking equality between paul and paulM could be performed as follows.

```
> (==) paul paulM
False
> paul == paul
True
> (/=) paul paulM
True
```

3.3 Functions

Effective utilization of FeatureStruct requires that subsequent functions are defined correctly.

3.3.1 emptyfs

```
emptyfs :: FeatureStruct
```

This function creates an empty FeatureStruct. Example:

```
*HW2> emptyfs
[]
```

3.3.2 addpath

```
addpath :: FeatureStruct -> [String] -> FeatureTerm -> FeatureStruct
```

This function is used to add `FeatureTerm` into the path specified by `[String]` which represents the partial path in depth-first order obeying subsequent specifications.

- Data instances should be constructed from scratch using this function. Direct use of `FS` constructor is **not** allowed.
- All path insertion procedures must take into consideration the **lexicographic** ordering of constituent strings at all depths of `FeatureStruct`.
- If specified path exists on the input `FS`, then the input term will be inserted in the specified location of the tree if applicable.
- If no new data is to be inserted on the already existing specified path of the tree, the input `FS` will be returned as the output.
- If the input path does not exist on the input `FS`, then a new path is created and the feature term is inserted accordingly.

Examples:

```
*HW2> addpath emptyfs ["key"] (Simple "value")
[key="value"]
*HW2> paul
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> addpath paul ["cgpa", "current"] (Simple "2.78")
[cgpa=>[cumulative="2.98", current="2.78", previous="3.12"], courses=>[OS=>[mt1↩
    ="66"], PL=>[hws=>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> addpath paul ["surname"] (Simple "Kent")
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul", surname="Kent"]
*HW2> addpath paul ["cgpa", "previous"] (Simple "3.22")
[cgpa=>[cumulative="2.98", previous="3.22"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> addpath paul ["name"] (Simple "Michael")
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Michael"]
*HW2> addpath paul ["friend"] (Struct paulM)
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", friend=>[cgpa=>[mincum="3.45", minprev↩
    ="3.55"], courses=>[OS=>[fin=>[bon="11", ques="77"], term="55"], SIG=>[lab="↩
    94"]], name="Paul", pcode="EEX04"], name="Paul"]
```

3.3.3 getpath

```
get :: FeatureStruct -> [String] -> Maybe FeatureTerm
```

This function tries to recover information stored in children of the node existing at the deepest level of the path specified in the `[String]` type argument representing this adress in a depth-first manner.

Blueprints related to `getpath` are included successively.

- `getpath` returns an element of type `Maybe FeatureTerm`. In case of success, it returns one `Just FeatureTerm`, while in failure its output would be of type `Nothing`.
- All branches located at deeper levels excluding the parents specified in the path must be returned in case of success.
- Failure cases involve specification of non-existing paths as input to the function.

Examples:

```
*HW2> getpath paul ["name"]
Just "Paul"
*HW2> getpath paul ["surname"]
Nothing
*HW2> getpath paul ["courses"]
Just [OS=>[mt1="66"], PL=>[hws=>[hw1="NA", hw2="78"]]]
*HW2> getpath paul ["courses", "OS", "hws", "hw1"]
Nothing
*HW2> getpath paul ["courses", "PL", "hws", "hw1"]
Just "NA"
*HW2> getpath paulM ["cgpa", "mincum"]
Just "3.45"
*HW2> getpath emptyfs ["name"]
Nothing
*HW2> getpath paul []
Nothing
```

3.3.4 delpath

```
delpath :: FeatureStruct -> [String] -> FeatureStruct
```

This function allows for the removal of the information stored at the address included in the path. It is anticipated to behave as follows.

- If the specified path exists on the tree, then a new tree without all the branches following down the path is returned.
- In case of failure the input tree is returned as output with no modifications performed on the stored data.

Examples:

```
*HW2> paul
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> delpath paul ["cgpa"]
[courses=>[OS=>[mt1="66"], PL=>[hws=>[hw1="NA", hw2="78"]]], dob="1996", name="↩
    Paul"]
*HW2> delpath paul ["cgpa", "cumulative"]
[cgpa=>[previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws=>[hw1="NA", hw2="↩
    78"]]], dob="1996", name="Paul"]
*HW2> delpath paul []
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
delpath paul ["courses", "PL", "hws"]
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[]],↩
     dob="1996", name="Paul"]
*HW2> delpath paul ["courses", "PL", "hws", "hw1"]
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw2="78"]]], dob="1996", name="Paul"]
*HW2> delpath paul ["courses", "PL", "hws", "hw3"]
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> delpath emptyfs ["temp"]
[]
*HW2> delpath paul []
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
```

3.3.5 `union`

```
union :: FeatureStruct -> FeatureStruct -> Maybe FeatureStruct
```

Through this function, two `FeatureStruct` trees will be merged so as to obtain a more specific tree containg all the information from both sources.

- If both trees contain the same information for the same paths, then only one of the equal paths must be inserted on the output tree.
- All distinct information will be inserted to the output tree in order specified by their paths.
- If there is different information on any of the same paths of the FeatureStruct, Nothing should be returned.

Examples:

```
*HW2> paul
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> paulM
[cgpa=>[mincum="3.45", minprev="3.55"], courses=>[OS=>[fin=>[bon="11", ques="77↩
    "], term="55"], SIG=>[lab="94"]], name="Paul", pcode="EEX04"]
*HW2> union paul paulM
Just [cgpa=>[cumulative="2.98", mincum="3.45", minprev="3.55", previous="3.12"↩
    ], courses=>[OS=>[fin=>[bon="11", ques="77"], mt1="66", term="55"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]], SIG=>[lab="94"]], dob="1996", name="Paul", pcode="↩
    EEX04"]
*HW2> let p1 = addpath emptyfs ["a","b"] (Simple "c")
*HW2> let p2 = addpath emptyfs ["a","b"] (Simple "d")
*HW2> let p3 = addpath p1 ["e"] (Simple "d")
*HW2> let p4 = addpath p2 ["e"] (Simple "d")
*HW2> p3
[a=>[b="c"], e="d"]
*HW2> p4
[a=>[b="d"], e="d"]
*HW2> union p3 p4
Nothing
*HW2> union emptyfs emptyfs
Just []
```

3.3.6 intersect

```
intersect :: FeatureStruct -> FeatureStruct -> FeatureStruct
```

This function intends to discover the mutual information content existing in both trees.

- If intersection of the two trees contains nothing then an `emptyfs` must be returned.
- In case of existing mutual information content, output tree must contain only the feature structure holding this content.

Examples:

```
*HW2> paul
[cgpa=>[cumulative="2.98", previous="3.12"], courses=>[OS=>[mt1="66"], PL=>[hws↩
    =>[hw1="NA", hw2="78"]]], dob="1996", name="Paul"]
*HW2> paulM
[cgpa=>[mincum="3.45", minprev="3.55"], courses=>[OS=>[fin=>[bon="11", ques="77↩
    "], term="55"], SIG=>[lab="94"]], name="Paul", pcode="EEX04"]
*HW2> intersect paul paulM
[cgpa=>[], courses=>[OS=>[]], name="Paul"]
```

4 Restrictions and Tips

- Do not modify the given type definitions.
- You are not allowed to import any modules.
- Pay particular attention to the lexicographic order of strings in the data structure as it is of utmost importance in all the functionality you are going to implement.

5 Submission

- You will upload a single Haskell source file named `HW2.hs`. We will provide a template for you to write your functions.
- Submission will be done via COW.
- **Late submission**: Regulations in the course website[1] applies.

6 Grading

- Functions will have different weights in grading based on their relative difficulties.
- We will be using `ghc` on *inek* machines, make sure your code works in them. Otherwise, you will not be able to modify your code.

7 Regulations

- **Programming Language:** Haskell
- **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations.
- Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
- **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.
- Failing to follow the homework submission rules and these regulations will result in grade reduction.

[1] `http://www.ceng.metu.edu.tr/course/ceng242/syllabus#Grading_and_other_policies`