Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming

WS 2019/20
Benjamin Dietrich, Denis Hirn

Assignment #6

Submission Deadline: Thu, 05.12.2019

Exercise 1: Integer Sets (8 Points)

In the lecture we discussed different approaches to a small DSL for integer sets. You can find the files `SetLanguageDeepCard.hs` and `SetLanguageShallowCard.hs` providing the deep and shallow embedding approaches in your repository. Add your solutions to the respective modules.

1. Both the shallow and deep embedding of integer sets lack important set operations. Add new constructors

   ```
   union :: IntegerSet -> IntegerSet -> IntegerSet
   difference :: IntegerSet -> IntegerSet -> IntegerSet
   ```

   that implement set union and set difference in both modules. Extend the export list of the modules.

2. Add a new observer

   ```
   maximum :: IntegerSet -> Integer
   ```

   to the deeply embedded DSL that computes the maximum element of a set. Can you add an equivalent observer to the shallowly embedded DSL based on *characteristic functions* in a reasonable way? If "yes", do it. If "no", explain why not.

3. Construct a `Show` instance for the deep embedding variant that pretty-prints integer sets in the usual set notation $\{x_1, x_2, \dots\}$ (order arbitrary, no duplicates). You will have to remove the `deriving` clause from the `IntegerSet` data type.

Exercise 2: Pattern-Matching DSL (8 Points)

The library `PatternMatching.hs` from this week's lecture defines a shallowly embedded DSL for *string pattern matching.* Patterns are defined as `type Pattern a = String -> [(a, String)]`. Thus, a pattern is a function with the following properties:

1. Given an input string, the function returns a list of pattern matches. If matching fails, it returns the empty list.
2. Each match is a tuple of
   - a value of type `a` that is described by the matched substring (e.g. the matched characters, token or parse tree)
   - the residual input string left after the matched substring.

The following grammar defines a language of fully parenthesized expressions over integers:

$$\begin{array}{lcl} \mathit{expr} & \rightarrow & \mathit{num} \\ & | & (\ \mathit{expr}\ ) \\ & | & \mathit{expr} + \mathit{expr} \\ & | & \mathit{expr} * \mathit{expr} \\ & & \\ \mathit{num} & \rightarrow & [0-9]+ \end{array}$$

Example expression: `"((4*10)+2)"`

1. Define an algebraic data type `Expr` to represent the language defined by the grammar above.
2. Use the pattern matching functions in module `PatternMatching` to construct a parser for expressions described by the above grammar:

   ```
   parse :: String -> Expr
   ```

   We advise you to first build simpler parsers for the individual alternatives of the grammar (e.g., a parser that can only accept multiplicative expressions `e * e`) and then assemble function `parse` from these pieces. These individual parsers will have type `Pattern Expr`.

Exercise 3: Matrices (4 Points)

Using lists, a *matrix* might be represented as a list of lists such that each inner list represents a row of the matrix.

For example, the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

is represented as follows

```
type Matrix = [[Integer]]

m1 :: Matrix
m1 = [ [1, 2, 3],
       [4, 5, 6],
       [7, 8, 9] ]
```

Write a function `trace :: Matrix -> Integer` that computes the *trace* of a *quadratic* matrix such as `m1`. The trace of a quadratic matrix is defined as the sum of the elements on the main diagonal:

$$\text{trace}(a_{ij})_{1 \leq i,j \leq n} = \sum_{l=1}^{n} a_{ll}$$

Example: `trace m1` $\equiv$ `15`

Note: You may assume that the input of your function is a well-formed quadratic matrix.