Advanced Programming (I00032)
Deep Embedding of a DSL for Sets

Assignment 7

Preparation

The skeleton comes with a project file, you can use if you work on the console. In case you use the IDE, please create a new project and choose the iTasks environment.

1 A DSL for Sets

In this exercise you will develop an evaluator for a special purpose language for sets. Using iTasks expressions in this language can be created and evaluated. Runtime errors in the evaluation should result in (somewhat) appropriate error messages.

1.1 Set Language

As explained in the lecture, in deep embeddings the language is represented by a family of data structures. These data structures represent the abstract syntax tree of the language. Usually they are a direct representation of the syntax. Our set language has expressions of type `Expression`.

```
:: Expression
   = New
   | Insert       Element Set
   | Delete       Element Set
   | Variable     Ident
   | Union        Set     Set
   | Difference   Set     Set
   | Intersection Set     Set
   | Integer      Int
   | Size         Set
   | Oper         Element Op  Element
   | (=.) infixl 2 Ident Expression
:: Op      =  +. | -. | *.
:: Set     :== Expression
:: Element :== Expression
:: Ident   :== String
```

The `Expression` type shows that there are two kind of values: integers and sets. The type cannot be checked at compile time, but has to be encoded by the data structure. Define a type `Val`, which contains either integers or sets of integers.

1.2 State

The state of the evaluator for expressions is a binding of variables to values. In contrast to the state used in the lecture (which was just a function), the state used in this assignment

must be a data structure that can be easily inspected and changed. For instance, you can use the `Map` type used in the previous assignment or a list of pairs of a name and a value. Define a type (or synonym) `State` containing such binding.

1.3 State Manipulations

Define a type `Sem a` that can be used as a monad, used to transform the state defined above and producing results. As runtime errors can occur, as in:

`Insert New (Oper New +. (Union (Integer 7) (Size (Integer 9))))`,

the result can also be an error with an error string. Note that in this assignment the result will always be of type `Val`, but the monad class nevertheless requires a type parameter.

Define instances for `Functor`, `Applicative` and `Monad` for `Sem`. Implement further following operations to store and read variable values:

```
store :: Ident Val → Sem Val
read  :: Ident     → Sem Val
```

Further, add a function throwing an error:

`fail :: String → Sem a`

This should be used in case of type-errors.

1.4 Evaluator

Implement the evaluator in monadic form:

`eval :: Expression → Sem Val`

assigning a semantics to expressions. Also you finally need a function to run the monad on a given expression and state:

`evalExpr :: Expression State → ?`

2 Priting

Implement another view on expressions, which is a string representing it. Just choose some syntax you find suited. Wrint the print function in continuation style. To efficiently concatenate the list of strings together to a single string use:

`'Text'.concat :: [String] → String`

3 Simulation

Write a simulator for the language implemented above using the iTask system. The iTask library uses a special monad class, so there are name clashes between the monad module and the iTask library. To work around this in the skeleton, new names are introduced for some iTask functions:

```
≫>=      >>=
≫>|      >>|
treturn   return
```

So you can write for instance `enterInformation "input" []` `≫>=` `viewInformation "output" []`. You can just use the combinators `≫*`, `-||-`, `||-` and `-||`. If you need more constructs from the iTask library, you can add them in the same way as:

```
ActionOk :== 'iTasks'.ActionOk
```

Write a simulator for the evaluation of expressions. The user should see an expression that can be edited and evaluated (which updates the state) by a click on the button. Furthermore, the user should get a view on the contents of the variable-value binding and the result of the expression (or an error message). Also show a string representing the expression, using your print-implementation of the previous part. Add the possibility to reset the state to its initial value and the option to quit. You do not have to use shared stores, it is fine that all views are updated when the user clicks a button.

Deadline

The deadline for this assignment is November 9, 13:30h.