Advanced Programming (I00032)
A Deep Embedded DSL for Sets

Assignment 8

Goal

The purpose of this exercise is to make a deep embedded DSL and see how we can make different views of such a DSL.

Preparation

This exercise comes with a skeleton for the `iTasks` environment. It contains some complicated import statements to avoid name conflicts. At some places we use qualified names, like `'iTasks'.return`, to select the desired operation.

1 A DSL for Sets

In this exercise you will develop an evaluator for a special purpose language for sets of integers. 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

In deep embedding the DSL is represented by a family of data structures. These data structures represent the abstract syntax tree of the language. Our set language has expressions of type `Expression`.

```
:: Expression
   = New
   | Insert       Element Set
   | Delete       Element Set
   | Variable     Ident
   | Intersection Set     Set
   | Integer      Int
   | Size         Set
   | Oper         Expression Op Expression
   | (=.) 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 by the host language, but has to be encoded by the data structure. The operators `+.` and `-.` are overloaded. For integers they are addition and subtraction, while for sets these operators indicate union and difference.

1.2 State

Define a type `Val`, which contains either integers or sets of integers. The state of the evaluator for expressions is a binding of variables to values. In contrast to the state used in the lecture (which is just a function), the state in this assignment must be a data structure that can be easily inspected and changed. The `Data.Map` type from `iTasks` is a convenient choice, but any type that associates names and values will do. 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. Runtime errors can occur during evaluation. For instance `Insert New (Oper New +. (Size (Integer 9)))` is a valid instance of `Expression`, but has no proper value for the evaluator. Hence, the monad produces a *result* or a *fail* with some message. 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
```

Add a function to indicate an error:

```
fail :: String → Sem a
```

1.4 Evaluator

Implement the evaluator in monadic form:

```
eval :: Expression → Sem Val
```

assigning a semantics to valid expressions.

2 Printing

Implement another view on expressions, which is a string representing it. Just choose some syntax you find suited. Write the print function in continuation style.

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:

```
(>>>=)      :== 'iTasks'.tbind
(>>|) a b   :== 'iTasks'.tbind a (λ_ → b)
treturn     :== 'iTasks'.return
```

So you can write for instance `enterInformation "input" [] >>>= viewInformation "output" []`. You can just use the combinators $\gg\!*$, `-||-`, `||-` and `-||`.

In the simulator the user sees 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 state (i.e. the value of all defined variables) and the result of the expression (a value or a fail 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.

Bonus: Type Checker

This part is optional, make it only when you like some additional challenge.

Add a view that checks whether the expression contains a valid statement in the DSL given the current state. Evaluating only correct expression should prevent runtime evaluation errors.

Deadline

The deadline for this assignment is November 15, 13:30h.