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

Assignment 8

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.

A DSL for Sets

In the previous exercise we implemented a DSL for set manipulations. This set-language was composed of 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
```

A limitation of the approach is that sets and elements are both expressions of the same type `Expression`. Hence the type system of the host language *Clean* cannot prevent type errors in the DSL, like in `Insert New (Oper New +. (Union (Integer 7) (Size (Integer 9))))`. Part of these problems can be avoided by introducing two separate types of expressions: a type `Element` for integer expressions and a type `Set` for set expressions.

```
:: Element
    = VariableE    Ident
    | Integer      Int
    | Size         Set
    | Oper         Element Op Element
    | (=.) infixl 2 Ident Expression
:: Op = +. | -. | *.

:: Set
    = New
    | Insert        Element Set
    | Delete        Element Set
    | VariableS     Ident
    | Union         Set    Set
    | Difference    Set    Set
    | Intersection  Set    Set
    | (=..) infixl 2 Ident Expression

:: Ident :== String
```

A drawback of such an approach is that reading a variable from memory and assigning a variable have to be duplicated since they have to work for integers as well as sets of integers. So there are `VariableE` and =. for integers and `VariableS` and =.. for sets. We however want to overload `Variable` and =. to work for both: integers and sets of integers.

In this assignment we do not use any of such data-type, but instead model the language as a set of functions!

1 State

Use a monad type `Sem a` very similar to the one used in the previous assignment. (It is not forbidden to reuse code from the previous assignment.) Note, that there can still be runtime errors, as variables can be uninitialised or the stored value can be of the wrong type. There are several choices for the state used here. Some are: `Map Ident Dynamic`, (`Map Ident Int`, `Map Ident [Int]`) and `Map Ident Val`, where `:: Val = I Int | S Ints`. **Choose one of those types (or another one) to represent the state and motivate your choice.**

2 Integer Expressions

Instead of defining the semantics of all constructs with a value of type `Sem Val`, we now want to use Clean's type system to distinct between integer and set expressions. Use the following type synonyms:

```
:: Element :== Sem Int
:: Set     :== Sem [Int]
```

Define functions with result type `Element` for all alternatives of the `Element` type above. For example:

```
integer ::        Element
size    :: Set -> Element
```

It is not necessary to define an `Oper` that is parametrised by the operator to be applied. You can directly implement instances of the operators `+`, `-` and `*` for `Element`. Write an `eval` function to compute the result of `Sem a` expressions and test `expr1` – `expr3`.

3 Set Expressions

Define functions for the set manipulations in this DSL. Note that you should use the same `variable` and =. functions for integers and sets. Hint: you can use the `delete` function from `Data.List`, but as there will be a name-clash with the `delete` function you will use for your DSL, use it in a qualified way: `'List'.delete`. The same holds for `union`, `difference`, `intersect`.

Test `expr4` and also adapt it to test the other set operations.

4 Statements

In order to make a more serious set manipulation language we add the following language constructs to our DSL.

```
(:.)  infixl 1 :: (Sem a) (Sem b)                   → Sem b    // sequential composition
(==.) infix  4 :: (Sem a) (Sem a)                   → Sem Bool // equality
(<.)  infix  4 :: (Sem a) (Sem a)                   → Sem Bool // less than
IF    :: (Sem Bool) THEN (Sem a) ELSE (Sem a) → Sem a    // conditional expression
WHILE :: (Sem Bool) DO (Sem a)                     → Sem Int  // repetition

:: THEN = THEN
:: ELSE = ELSE
:: DO   = DO
```

Implement the functions above. Most likely you need some class constraints to these DSL-constructs. It is not necessary that the Booleans can be stored, but depending on your choice of how you defined the state, this extension may be very straightforward.

This allows set-programs like:

```
expr =
    z =. integer 7 :.
    x =. new :.
    x =. insert (variable z) (variable x) :.
    y =. union (variable x) (variable x) :.
    WHILE (size (variable x) <. integer 5) DO
        (x =. insert (size (variable x)) (variable x)) :.
    z =. difference (variable x) (intersection (variable x) (insert (variable z) new))
```

Also test the other example programs given in the skeleton. Since this program is a function instead of a data type, it is not possible to make a simulator for our DSL using iTasks in that same simple way as in previous assignment.

5 Printing

Implement another view on expressions, which is a string representing it. Just choose some syntax you find suited. Extend the type to represent the semantics of the language, as in the first part of the lecture. Finally, you have to provide a function `print :: (Sem a) → String`.

You do not have to spend too much time on a perfect printing function. It is okay if there are superfluous brackets and the indentation is not perfect in all cases.

Deadline

The deadline for this assignment is November 16, 13:30h.