Monad-Based Programming WS 2021

Assignment 2

Deadline for solutions: 24.11.2021

Exercise 1 Not Quite Contextual Equivalence (8 Points)

Within this exercise we stick to PCF under the call-by-name semantics.

The simplest implementation of Fibonacci numbers, corresponding to the following Haskell definition

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

can be extremely inefficient, for every step recursively calls `fib` twice, so the number of recursive calls grows exponentially. Because of that, it makes sense to use the following variant of `fib`, which simultaneously calculates the n-th and the $(n+1)$-th Fibonacci number:

```
fib2 0 = (1 , 1)
fib2 n = let (fn_1, fn) = fib2 (n - 1) in (fn, fn_1 + fn)
```

1. Encode both examples in PCF.

2. Using the operational semantics rules (small-step or big step, chosen at pleasure), show that for every input natural number n, both `fib` and `fst $ fib2` applied to n reduce to the same value. **Hint:** Use induction over n; for the second function, consider strengthening the induction invariant.

Exercise 2 Stateful Calculator (10 Points)

1. Write a Haskell function for parsing arithmetic expressions given by the following grammar:

$$e_1, e_2 = n \mid -e_1 \mid (e_1) \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 \,/\, e_2$$

where n ranges over positive natural numbers represented as nonempty strings of decimal digits.

Hint: Reuse the example code from the lecture: https://www8.cs.fau.de/ext/teaching/wise2021-22/mbprog/parsing.hs.

2. Write a Haskell function evaluating the parse trees obtained in the previous clause to the corresponding integer values or to an error message if division by zero occurs. In any case record the evaluation history in the following style:

```
Prelude> eval $ calc $ "12 + 5 * 6"
Prelude> Result 42, ["5 * 6 -> 30", "12 + 30 -> 42"]

Prelude> eval $ (calc $ "12 + 5 / (0*6)"
Prelude> Error, ["0 * 6 -> 0", "5 / 0 -> error"]
```

To this end, use the *writer monad* `Writer` from `Control.Monad.Trans.Writer` for recording the calculation history as a list of strings. Use an instance of the *exception monad transformer* from `Control.Monad.Except` to implement exceptions

```
newtype ExceptT e (m :: * -> *) a
```

with an exception monad as the argument `m`.

Alternatively: Implement your own monad from scratch that combines writing calculation history with exceptions i.e. the monad $TX = (X + E) \times S$ where S is the type of lists of strings (histories) and E is the type of strings (error messages).

Exercise 3 First-Order Logic (12 Points)

Analogously to the previous exercise, implement (also in Haskell!) a parser for first-order formulas

$$\phi, \psi ::= \texttt{T} \mid \texttt{F} \mid \sim \phi \mid \phi \texttt{/\textbackslash} \psi \mid \phi \texttt{\textbackslash/} \psi \mid \phi \texttt{->} \psi \mid \texttt{forall}\ x.\, \phi \mid \texttt{exists}\ x.\, \phi$$

Implement the following transformations of formulas:

1. Recursive replacement of the implication under the rule $\phi\texttt{->}\psi \rightsquigarrow (\sim \phi)\texttt{\textbackslash/}\psi$.
2. Negation normal form, as described in GLoIn (p.42 of https://www8.cs.fau.de/ext/teaching/wise2021-22/gloin/skript.pdf).
3. Prenex normal form, as described in GLoIn (p.43 of https://www8.cs.fau.de/ext/teaching/wise2021-22/gloin/skript.pdf). **Attention:** you will need to ensure capture avoidance, e.g. $(\texttt{forall}\ x.\, \phi)\texttt{\textbackslash/}\psi$ must be transformed to $\texttt{forall}\ y.\, (\phi[y/x]\texttt{\textbackslash/}\psi)$ for a suitable y if x occurs freely in ψ (Circumventing this challenge, e.g. via de Bruijn indices will not be considered legit). Avoid unnecessary renamings!