universität innsbruck

Functional Programming	WS 2023/2024	LVA 703025
Exercise Sheet 3, 5 points		Deadline: Tuesday, October 31, 2023, 8pm

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs file that is provided on the proseminar page.
- Upload your modified .hs file in OLAT.
- Your .hs file must be compilable with ghci.

Exercise 1 *Programming with Pattern Matching and Recursion* **5 p.**

In this exercise we will manipulate expressions in various ways. To this end we consider the datatype `Expr` of expressions of Lecture 2. We additionally represent variable assignments i.e., mappings from variables to integers by type `Assign`.

```
data Expr = Variable String | Number Integer | Add Expr Expr | Negate Expr deriving Show
data Assign = Empty | Assign String Integer Assign deriving Show
```

For instance, `exampleExpr` represents $-(x + (-(y + 3)))$, and `exampleAssign` corresponds to the assignment $\{x \mapsto 5, y \mapsto 12\}$ in the following code:

```
exampleExpr = Negate (Add (Variable "x") (Negate (Add (Variable "y") (Number 3))))
exampleAssign = Assign "x" 5 (Assign "y" 12 Empty)
```

For some of the following tasks the if-then-else function `ite` might be useful: given a Boolean and two arguments `x` and `y` of the same type, it selects one of the arguments depending on the truth value of the Boolean.

```
ite True  x y = x
ite False x y = y
```

1. Define a function `value :: String -> Assign -> Integer` that returns the value of a variable in an assignment. If a variable does not occur in the assignment, you can you indicate an error via `error "message"` in Haskell. You may assume that assignments contain each variable at most once. (1 point)

   For example, `value "x" exampleAssign` results in 5, and `value "z" exampleAssign` results in an error.

2. Define a function `eval :: Assign -> Expr -> Integer` that evaluates an expression w.r.t. a given assignment. You may assume that all variables in the expression occur in the assignment and that `value` is available, even if you did not solve the first task. (1 point)

   Example: `eval exampleAssign exampleExpr` results in $-(5 + (-(12 + 3)))$, i.e., 10.

3. Define a function `containsVar :: String -> Assign -> Bool` that determines whether a variable occurs in an assignment. (1 point)

   Example: `containsVar "x" exampleAssign` results in `True`, whereas `containsVar "z" exampleAssign` results in `False`.

4. Define a function `substitute :: Assign -> Expr -> Expr` that replaces all variables of the expression, that also occur in the assignment by the corresponding value. You may assume that both `containsVar` and `value` are available, even if you did not solve these tasks. (1 point)

   Example: `substitute exampleAssign exampleExpr` results in the Haskell encoding of $-(5+(-(12+3)))$, whereas the result of `substitute (Assign "x" 7 Empty) exampleExpr` represents $-(7 + (-(y + 3)))$.

5. We want to normalize expressions by moving negations as far inside the expression as possible. We say that an expression `e` is normalized if the negations only occur in front of variables or numbers. In Haskell this means that whenever `Negate e1` is a subexpression of `e`, then `e1` is of the form `Variable x`. Note that if `e1` represents a number, then one can eliminate the `Negate` constructor by just negating the number.

   Define a function `normalize :: Expr -> Expr` that computes a normalized expression. (1 point)

   Example: `normalize exampleExpr` results in the Haskell encoding of $-x + (y + 3)$.

Solution 1

```haskell
data Expr = Variable String | Number Integer | Add Expr Expr | Negate Expr deriving Show
data Assign = Empty | Assign String Integer Assign deriving Show

exampleExpr = Negate (Add (Variable "x") (Negate (Add (Variable "y") (Number 3))))
exampleAssign = Assign "x" 5 (Assign "y" 12 Empty)

ite True x y = x
ite False x y = y

value :: String -> Assign -> Integer
value x Empty = error (x ++ " does not occur in assignment")
value x (Assign y i a) = ite (x == y) i (value x a)

eval :: Assign -> Expr -> Integer
eval a (Number i) = i
eval a (Add e1 e2) = eval a e1 + eval a e2
eval a (Negate e) = - eval a e
eval a (Variable x) = value x a

containsVar :: String -> Assign -> Bool
containsVar x Empty = False
containsVar x (Assign y i a) = x == y || containsVar x a

substitute :: Assign -> Expr -> Expr
substitute a v@(Variable x) = ite (containsVar x a) (Number (value x a)) v
substitute a (Add e1 e2) = Add (substitute a e1) (substitute a e2)
substitute a (Negate e) = Negate (substitute a e)
substitute a n@(Number _) = n

normalize :: Expr -> Expr
normalize (Add e1 e2) = Add (normalize e1) (normalize e2)
normalize v@(Variable _) = v
normalize n@(Number _) = n
normalize (Negate (Number i)) = Number (- i)
normalize (Negate (Negate e)) = normalize e
normalize (Negate (Add e1 e2)) = Add (normalize (Negate e1)) (normalize (Negate e2))
normalize e@(Negate (Variable _)) = e
```