T-501-FMAL Programming languages, Practice class 4
Spring 2021

1. (i) Recall the discriminated union type `expr` of arithmetic expressions with free variables (but no let-binding)

   ```
   type expr =
       | Var of string
       | Num of int
       | Op of string * expr * expr
   ```

   from the file Expressions2.fs. Modify it so that there are individual cases with their own tags for operations $+$, $-$, $*$ and also $/$, $\%$ and unary $-$ (negation of a number). Don't allow any other operations.

   E.g., instead of the value `Op ("+", Var "x", Num 3)` the type `expr` should rather contain a value `Plus (Var "x", Num 3)`.

   (ii) Modify `prettyprint` and `eval` accordingly.

2. Write a better version of prettyprint that minimizes the number of parentheses based on a precedence and associativity mode for every operation.

   $+$, $-$ should have precedence 4; $*$, $/$, $\%$ should have precedence 7, the unary $-$ should have precedence 10. (Higher precedence means stronger binding.)

   $+$, $-$, $*$, $/$, $\%$ are left-associative. The unary $-$ is right-associative.

   The prettyprinter should work with an accumulator for the precedence level of the enclosing context of a subexpession.

   If the principal operation of a subexpression has higher precendence than its context, the expression does not need parentheses around it.

   The empty enclosing context of the main expression has precedence 0.

   A non-associative operation of precedence n provides its arguments a context of precedence n.

   A left-associative operation of precedence n provides its left argument a context of precedence $n-1$ and the right argument a context of precedence n.

3. (i) Define a type `xexpr` of extended expressions where you also have constants *true*, *false* and operations $=$, $<$, &&, ||, *not*, *if-then-else*.

   (ii) Extend evaluation. A value is now either an integer or a boolean, with the two cases discriminated.

   ```
   type value =
       | I of int
       | B of bool
   ```

   Let us agree that variables can only take integer values. Evaluation should fail when an operation is applied to a argument of an unsuitable type.

   if-then-else should be lazy, i.e., only evaluate the correct branch. && and || could be lazy too, i.e., only evaluate the second argument when the first argument does not determine the return value ("short-circuit" conjunction and disjunction).

4. Write a compiler from extended expressions back to extended expressions (a "desugarer") that compiles away &&, || and *not* by rewriting them in terms of *if-then-else*.

5. Write a type inferrer, ie a function `infer : xexpr -> typ` where

   ```
   type typ =
       | Int
       | Bool
   ```

   You will find that the type inferrer can be defined very similarly to the evaluator, except that, instead of concrete values of the form `I i` and `B b`, it has to work with their abstracted versions `Int` and `Bool`.

   When the type inferrer encounters an *if-then-else* with branches of different types, it should fail.