# Exercises week 14

**Exercise 14.1** (The purpose of this question is to review Haskell programming. If you are confident with Haskell, skip to the next question.)

Define the following functions in Haskell:

- A function `max2 :: Int -> Int -> Int` that returns the largest of its two integer arguments. For instance, `max2 99 3` should give 99.
- A function `max3 :: Int -> Int -> Int -> Int` that returns the largest of its three integer arguments.
- A function `isPositive :: [Int] -> Bool` so that `isPositive xs` returns true if all elements of `xs` are greater than 0, and false otherwise.
- A function `isSorted :: [Int] -> Bool` so that `isSorted xs` returns true if the elements of `xs` appear sorted in non-decreasing order, and false otherwise. For instance, the list `[11, 12, 12]` is sorted, but `[12, 11, 12]` is not. Note that the empty list `[]` and any one-element list such as `[23]` are sorted.
- A function `count :: IntTree -> Int` that counts the number of internal nodes (`Br` constructors) in an `IntTree`, where the type `IntTree` is defined by:

  ```
  data IntTree = Br Int IntTree IntTree | Lf
  ```

  That is, `count (Br 37 (Br 117 Lf Lf) (Br 42 Lf Lf))` should give 3, and `count Lf` should give 0.
- A function `depth :: IntTree -> Int` that measures the depth of an `IntTree`, that is, the maximal number of internal nodes (`Br` constructors) on a path from the root to a leaf. For instance, `depth (Br 37 (Br 117 Lf Lf) (Br 42 Lf Lf))` should give 2, and `depth Lf` should give 0.

**Exercise 14.2** (i) File `Intro2.hs` on the course homepage contains a definition of the lecture's `expr` expression language and an evaluation function `eval`. Extend the `eval` function to handle three additional operators: `"max"`, `"min"`, and `"=="`. Like the existing operators, they take two argument expressions. The equals operator should return 1 when true and 0 when false.

(ii) Write some example expressions in this extended expression language, using abstract syntax, and evaluate them using your new `eval` function.

(iii) Rewrite one of the `eval` functions to evaluate the arguments of a primitive before branching out on the operator, in this style:

```
eval :: Expr -> [(String, Int)] -> Int
eval (CstI i) env   = ...
eval (Var x)  env   = ...
eval (Prim op e1 e2) env
    = let i1 = ...
          i2 = ...
      in case op of
           "+" -> i1 + i2
           ...
```

(iv) Extend the expression language with conditional expressions `If e1 e2 e3` corresponding to Haskell's conditional expression `if e1 then e2 else e3`.

You need to extend the `Expr` datatype with a new constructor `If` that takes three `Expr` arguments.

(v) Extend the interpreter function `eval` correspondingly. It should evaluate `e1`, and if `e1` is non-zero, then evaluate `e2`, else evaluate `e3`. You should be able to evaluate this expression `If (Var "a") (CstI 11) (CstI 22)` in an environment that binds variable `a`.

Note that various strange and non-standard interpretations of the conditional expression are possible. For instance, the interpreter might start by testing whether expressions `e2` and `e3` are syntactically identical, in which case there is no need to evaluate `e1`, only `e2` (or `e3`). Although possible, this is rarely useful.

**Exercise 14.3** (i) Declare an alternative datatype `AExpr` for a representation of arithmetic expressions without let-bindings. The datatype should have constructors `CstI`, `Var`, `Add`, `Mul`, `Sub`, for constants, variables, addition, multiplication, and subtraction.

The idea is that we can represent $x * (y + 3)$ as `Mul (Var "x") (Add (Var "y") (CstI 3))` instead of `Prim "*" (Var "x") (Prim "+" (Var "y") (CstI 3))`.

(ii) Write the representation of the expressions $v - (w + z)$ and $2 * (v - (w + z))$ and $x + y + z + v$.

(iii) Write a Haskell function `fmt :: AExpr -> String` to format expressions as strings. For instance, it may format `Sub (Var "x") (CstI 34)` as the string `"(x - 34)"`. It has very much the same structure as an `eval` function, but takes no environment argument (because the *name* of a variable is independent of its *value*).

(iv) Write a Haskell function `simplify :: AExpr -> AExpr` to perform expression simplification. For instance, it should simplify $(x + 0)$ to $x$, and simplify $(1 + 0)$ to 1. The more ambitious student may want to simplify $(1 + 0) * (x + 0)$ to $x$. Hint 1: Pattern matching is your friend. Hint 2: Don't forget the case where you cannot simplify anything.

You might consider the following simplifications, plus any others you find useful and correct:

| | | |
|---|---|---|
| $0 + e$ | $\longrightarrow$ | $e$ |
| $e + 0$ | $\longrightarrow$ | $e$ |
| $e - 0$ | $\longrightarrow$ | $e$ |
| $1 * e$ | $\longrightarrow$ | $e$ |
| $e * 1$ | $\longrightarrow$ | $e$ |
| $0 * e$ | $\longrightarrow$ | $0$ |
| $e * 0$ | $\longrightarrow$ | $0$ |
| $e - e$ | $\longrightarrow$ | $0$ |

**Exercise 14.4** Write a version of the formatting function `fmt` from the preceding exercise that avoids producing excess parentheses.

For instance,

```
Mul (Sub (Var "a") (Var "b")) (Var "c")
```

should be formatted as `"(a-b)*c"` instead of `"((a-b)*c)"`, whereas

```
Sub (Mul (Var "a") (Var "b")) (Var "c")
```

should be formatted as `"a*b-c"` instead of `"((a*b)-c)"`.

Also, it should be taken into account that operators associate to the left, so that

```
Sub (Sub (Var "a") (Var "b")) (Var "c")
```

is formatted as `"a-b-c"` whereas

```
Sub (Var "a") (Sub (Var "b") (Var "c"))
```

is formatted as `"a-(b-c)"`.

Hint: This can be achieved by declaring the formatting function to take an extra parameter `pre` that indicates the precedence or binding strength of the context. The new formatting function then has type `fmt :: AExpr -> Int -> String`.

Higher precedence means stronger binding. When the top-most operator of an expression to be formatted has higher precedence than the context, there is no need for parentheses around the expression. A left associative operator of precedence 6, such as minus (−), provides context precedence 5 to its left argument, and context precedence 6 to its right argument.

As a consequence, `Sub (Var "a") (Sub (Var "b") (Var "c"))` will be parenthesized `a - (b - c)` but `Sub (Sub (Var "a") (Var "b")) (Var "c")` will be parenthesized `a - b - c`.