



Assignment 7

Hand in this assignment until **Friday, 23. June 2023, 10:00** at the latest.

⚠ Lecture Evaluation

In the upcoming days, you will have the opportunity to **evaluate lectures** you are attending. With this in mind, we kindly ask you to keep an eye on your inbox and to provide us with **your** valuable feedback. Thank you!

📄 Exam-style Exercises

Exercises marked with 📄 are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Running out of ideas?

Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the [#forum](#) channel on our Discord server and check the tag for this assignment—maybe you'll find just the help you need.

Task 1: Pattern-Matching DSL

The library `PatternMatching.hs` defines a shallowly embedded DSL for *string pattern matching*. Patterns are defined as `type Pattern a = String -> [(a, String)]`. Thus, a pattern is a function with the following properties:

1. Given an input string, the function returns a list of pattern matches. If matching fails, it returns the empty list.
2. Each match is a tuple consisting of:
 - A value of type `a` that is described by the matched substring (e.g. the matched characters, token, or parse tree).
 - The residual input string left after the matched substring.

The following grammar defines a language of fully parenthesized expressions over integers:

```
expr → num
      | (expr op expr)
op   → + | - | * | /
num  → [0 - 9]+
```

Example expression: `"((4*10)+2)"`

1. Define algebraic data types `Expr`, and `Op` to represent the language defined by the grammar.
2. Use the pattern matching functions in module `PatternMatching` to construct a parser for expressions described by the grammar:

```
parse :: String -> Expr
```

Hint

We advise you to first build simpler parsers for the individual alternatives of the grammar, e.g., a parser that can only accept operators `+`, `-`, `*`, `/` (this individual parser will have type `Pattern Op`) and then assemble function `parse` from these pieces.



Task 2: Mathematical Expressions with Only Four 4s

Solve the following puzzle:

For each number n between 0 and 20, find at least one mathematical expression which evaluates to n and is an arbitrary combination of exactly four numbers 4 using the following arithmetic operations: addition ($a+b$), subtraction ($a-b$), multiplication ($a*b$), division (a/b), exponentiation (a^b), square root (\sqrt{a}) and factorial of numbers ($4!$).

Example:

$$0 = \frac{4}{4} * 4 - 4 \qquad 1 = \left(\frac{4}{4}\right)^{4^4} \qquad \dots$$

1.  Define a data type **ExprTree** to represent such mathematical expressions.
2.  Write a function `eval :: ExprTree -> Maybe Int` which evaluates an expression to an integer number if possible; if the result is not an integer or undefined (division by zero) return **Nothing**, instead.
3. Write a function `trees :: [ExprTree] -> [ExprTree]` which takes a list of leaf nodes and returns a list of all expression trees that can be built in combination of these leaf nodes.

Important!

Assume that a sequential application of the unary square-root-operation (e.g. $\sqrt{\sqrt{a}}$) is **not** allowed, while factorial is only applied to leaf nodes ($4!$), but not to other expressions (e.g. $(a+b)!$).

Example: (Note: This example uses a *human-digestable* notation for **ExprTree** values.)

```
1 trees [4,4,4,4] => [ (4+4+4+4), ...
2                       , (4*4+4+4), ...
3                       , (4*(4+4)+4), ...
4                       , ((4/4)^4)^4, ...
5                       , ((4!+4)/4+4), ...
6                       ]
```

Hint

It might be useful to write a helper function `splits :: [a] -> [[a], [a]]` which returns all combinations of a list split in two parts.

Example: `splits [1..4] => [(1],[2,3,4]),(1,2],[3,4]),(1,2,3],[4])]`

4. Finally, write a function `solution :: Int -> Maybe ExprTree` that returns one arbitrary “expression of four 4s” which evaluates to a given number $n \in \{1, \dots, 20\}$. Return **Nothing** if no such expression was found.