EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer Term 2025
Prof. Torsten Grust, Björn Bamberg
WSI — Database Systems Research Group

Assignment 7

Hand in this assignment until **Tuesday, July 08, 12pm** at the latest.

> 🤔 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—maybe you'll find just the help you need.

> Ⓔxam-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.

Task 1: Pattern-Matching DSL

The library `PatternMatching.hs` defines a shallowly embedded DSL for *string pattern matching*. Patterns are defined as:

```
1 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:

```
1 expr → num
2       | (expr op expr)
3
4 op   → + | - | * | /
5
6 num → [0 - 9]+
```

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

A Ⓔ Define algebraic data types `Expr`, and `Op` to represent the language defined by the grammar.

B Use the pattern matching functions in module `PatternMatching` to construct a parser for expressions described by the grammar:

```
1 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 $(\frac{a}{b})$, exponentiation (a^b), square root $(\sqrt{a})$ and factorial of numbers $(4!)$.

Example:

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

A Ⓔ Define a data type `ExprTree` to represent such mathematical expressions.

B Ⓔ 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 (*e.g.*, division by zero) return `Nothing`, instead.

C 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.)

```
trees [4,4,4,4] => [ (4+4+4+4), …
                   , (4*4+4+4), …
                   , (4*(4+4)+4), …
                   , ((4/4)^4)^4, …
                   , ((4!+4)/4+4), …
                   ]
```

> **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])]`

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

> **Hint**
> To test whether an evaluated floating point number is equal to an integer value you might use the following function:
>
> ```
> isInt :: Float -> Bool
> isInt x = x == fromInteger (round x)
> ```

Task 3: Find the Middle Element

Implement function `middle :: [a] -> a` such that `middle xs` returns the element in the middle of non-empty list `xs`:

```
middle [1]       = 1
middle [1,2]     = 1
middle [1,2,3]   = 2
middle [1..1000] = 500
```

Puzzle: Your implementation may not compute the number of elements in `xs` to do its thing (using `length` or any other related/similar built-in or user-defined function).

Task 4: Deep Embedding of a While-Language

In the video lecture we have implemented a deeply embedded, typed expression language over Integers and Booleans. To ensure that only well-typed expressions can be built, the module makes use of Haskell's *Generalized Algebraic Data Types* (`GADTs`). GADTs allow to check the types of expressions of our embedded language at compile time. In this exercise we are going to extend this DSL to a simple *While-Language*.

Inspect your repository to find the module `DeepWhile` which contains a slightly modified version of the lecture's code to start with.

The given module exports the AST data type `Expr a`, together with its constructors and an observer function `eval :: Expr a -> a` which evaluates DSL expressions.

A First, extend the expression language with a relational operator `Lt` which takes two expressions e_1, e_2 of type `Expr Int` and – when evaluated – returns whether $e_1 < e_2$ or not. Complete the `Show` instance and `eval` observer accordingly.

Example:

```
eval (Lt (ValI 41) (ValI 42)) ≡ True
```

B Next, we are going to add support for DSL expressions of *non-empty lists* of `Int`s or `Bool`s. Provide constructors for the following operations and complete `Show` and `eval` accordingly:

- A `Singleton` constructor builts an expression of type `Expr [a]` from an expression of type `Expr a` which represents the list containing a single given element of type `a`.
- `Append` takes two list expressions and appends the list on evaluation.
- `Length` takes a list expression and returns its *length* when evaluated.

Example:

```
eval (Length (Append (Singleton (ValI 41)) (Singleton (ValI 42)))) ≡ 2
```

C Extend the expression language with support for a *variable environment* (see type `Env b` in `DeepWhile`[1]) which binds values of a particular type `b` to variable names (type `String`) that can be used in the DSL expression:

- Equip the data type `Expr` with a second type variable `b` that determines the type of environment values (`Expr a b`: evaluates to a value of type `a`, based on an environment of variables of type `b`).
- Adapt `eval` to additionally take an environment of type `Env b`:

  ```
  eval :: Env b -> Expr a b -> a
  ```

- Provide an additional constructor `Var` which takes a variable name (`String`). `Var` represents a lookup of the variable name in the environment on evaluation. If the variable is not bound in the environment an `error "Variable not bound!"` is thrown at runtime. Complete `Show` and `eval` accordingly.

  Example:

  ```
  eval (M.singleton "a" 42) (Var "a") ≡ 42
  ```

D Define a new algebraic data type `Stmt b` for While-Language statements operating on an environment of type `b`. Use GADT syntax to define the following constructors:

- `Assign` – takes a variable name (`String`) and an expression (`Expr`) that evaluates to a value of type `b`. It represents the assignment of the expression result to a variable with the given name.
- `While` – takes a Boolean expression and a statement (`Stmt b`) which is to be repeated while the condition holds. Both, the expression and the loop statement are evaluated based on an environment of type `b`.
- `Seq` – takes two statements (`Stmt b`) and represents their sequential execution.

Define a `Show` instance for `(Stmt b)`.

E Write a function

```
runS :: Env b -> Stmt b -> Env b
```

[1] The environment is implemented as a dictionary using the module `Data.Map`.
See https://hackage-content.haskell.org/package/containers-0.8/docs/Data-Map-Lazy.html for further information.

which runs a given statement based on a given environment. The – possibly modified – environment is returned. Whenever statements are run sequentially (or looped) the modifications in the environment of the previous statement are visible in the execution of the subsequent.

Example:

```
let p = Seq (Assign "a" $ ValI (21)) $
            Seq (Assign "r" $ ValI 0) $
                While (Lt (Var "r") (Var "a")) $
                    Seq (Assign "a" $ Add (Var "a") (ValI 1)) $
                        Assign "r" $ Add (Var "r") (ValI 2)
```

```
runS M.empty p ≡ M.fromList [("a",42),("r",42)]
```

F Finally, define a function

```
    run :: String -> Stmt b -> Maybe b
```

which takes a variable name v and a statement s. This function should run s with an empty environment and returns the value bound to v after evaluation (or `Nothing` if v was not bound by the program). Add `run` together with the data type `Stmt` and its constructors to the export list of module `DeepWhile`.

Example:

```
run "r" p ≡ 42
```