Preparation and administration

- There are three files available via Canvas: `GameOfLife.hs`, and `simple.level`, and `Practicum3.hs`.
- Submit your files via Canvas as a zip file.

Goal

The goal of the third set of Haskell exercises is to play with an implementation of the Game of Life, and to practice with equational specifications and Combinatory Logic.

Exercise Game of Life

The starting point for this exercise is the file `GameOfLife.hs`, and in addition we use the file `simple.level`.

We consider Game of Life, as defined by Conway. Game of Life is a zero-player game where a grid of cells is considered. A cell can be alive or dead. The amount of neighbours of a cell, and its current state (alive or dead) determine its state in the next iteration. Please see the wiki about Game of Life:
wiki Game of Life
which also refers to the wiki about John Conway:
wiki John Conway
The starting point of the exercise is the file `GameOfLife.hs`. The question is to complete the file by defining the function `nextGeneration` that implements the rules of Game of Life.

In addition, you may wish to use other initial configurations, and to see variations, with other rules for nextGeneration. (This is not obligatory.)

We get an executable GameOfLife.o by applying ghc to GameOfLife.hs. We can execute by `./GameOfLife`.

Exercises Arithmetic Expressions

The starting point for this exercise is the file `Practicum3.hs`. We define the following:

```
data IntExp  = Lit Int | Add IntExp IntExp | Mul IntExp IntExp
  deriving Show
```

Intuitively this defines a data type for integers with addition and multiplication. The `deriving Show` makes that we can print on the screen.

1. Define a function

   ```
   showintexp :: IntExp -> [Char]
   ```

   that takes an input of type `IntExp`, and prints the 'interpretation' of the input on the screen. Example:

   ```
   *Main> showintexp (Lit 5)
   "5"
   *Main> showintexp (Add (Lit 7) (Lit 5))
   "(7+5)"
   ```

   You may wish to use `++` to concatenate strings. Please note that strings are written using double quotes. In the clause for `Lit x` you can use `show x`, for example:

   ```
   Prelude> show 6
   "6"
   Prelude>
   ```

2. Define a function

   ```
   evalintexp :: IntExp -> Int
   ```

   that takes an input from `IntExp`, and calculates what we intuitively expect. Example:

   ```
   *Main> evalintexp (Lit 5)
   5
   *Main> evalintexp (Add (Lit 7) (Lit 5))
   12
   ```

Exercises Combinatory Logic

We work with the file `Practicum3.hs`.

Combinatory Logic (CL) is a system closely related to λ-calculus, but without bound variables. Terms in CL are built using three constants: S, K, and I that have a behaviour as the terms with the same names we know from λ-calculus. For this exercise, we restrict attention to closed CL-terms, so without variables. They are inductively defined using the following grammar:

$$M ::== \mathsf{S} \,|\, \mathsf{K} \,|\, \mathsf{I} \,|\, M\,M'$$

So a closed CL-term is either a constant or an application. Some examples of CL-terms: S, $\mathsf{K}\,\mathsf{I}$, S , $\mathsf{S}(\mathsf{K}\,\mathsf{K})$. As in λ-calculus, application is left-associative. The dynamics of CL comes from three reduction rules, one for every combinator:

$$\begin{array}{rcl} \mathsf{I}\,P & \to & P \\ \mathsf{K}\,P\,Q & \to & P \\ \mathsf{S}\,P\,Q\,R & \to & (P\,R)\,(Q\,R) \end{array}$$

where P, Q, R stand for arbirary CL-terms. Note that I is in fact not necessary because it can be defined as $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$. The following page may be useful: Haskell wiki about CL. We will consider CL in Haskell.

1. The grammar for closed CL-terms in Haskell is given as follows:

   ```
   data Term =
     S | K | I | App Term Term
   ```

   In order to print the terms on the screen, we use the following:

   ```
   instance Show Term where
    show a = showterm a
   ```

   Define the function `showterm`, such that we for example have the following:

   ```
   *Main> showterm (App (App (App S K) I) (App I I ))
   "(((SK)I)(II))"
   ```

2. Define a function `isredex` that indicates whether a term is a redex. A term is a redex if it is in the shape of the left-hand side of one of the three reduction rules. Note the difference between 'is a redex' and 'contains a redex'. For example, the term $\mathsf{I}\,x$ is a redex and contains a redex, and the term $x\,(\mathsf{I}\,x)$ contains a redex but is not a redex. We should for example have the following:

   ```
   *Main> isredex (App (App (App S I) K ) S)
   True
   *Main> isredex (App (App I K) (App (App K I) K))
   False
   ```

   Note that Haskell tries to apply the clauses of a definition in order, so we can use the last clause for the 'otherwise case'.

3. Define a function `hasredex` that indicates whether a term contains a redex. A special case is when the term is a redex itself. The other case is that the term is of the shape $P\,Q$, so an application, and either P or Q or both has (have) a redex.

4. Define a function `isnormalform` that indicates whether a term is a normal form, which means: no rule applies. So the term does not have a redex.

   Note: the following are normal forms in Combinatory Logic:

   - I
   - K
   - $\mathsf{K}\,N$ with N a normal form

- S
- $\mathsf{S}\,N$ with N a normal form
- $\mathsf{S}\,N\,N'$ with N and N' normal forms

The following are not normal forms:

- $\mathsf{I}\,P$ with P an arbitrary term
- $\mathsf{K}\,P\,P'$ with P and P' arbitrary terms
- $\mathsf{S}\,P\,P'\,P''$ with P, P', P'' arbitrary terms.

We should for example have the following:

```
*Main> isnormalform I
True
*Main> isnormalform K
True
*Main> isnormalform S
True
*Main> isnormalform (App K I)
True
*Main> isnormalform (App (App I I) K)
False
*Main> isnormalform (App K (App I I))
False
```

5. Define a function `headstep` that reduces a term one step in case the term is a redex, and that does not change the input in the other case. We should for example have the following:

```
*Main> headstep (App (App (App S I) K ) S)
((IS)(KS))
*Main> headstep (App (App I K) (App (App K I) K))
((IK)((KI)K))
```

6. Define a function `doall` that reduces all redexes that are present in a term in one go. We should for example have the following:

```
*Main> doall (App (App I I) (App (App K K ) K ) )
(IK)
*Main> doall (App I (App I I ) )
I
*Main> doall (App ( App (App K I ) K ) K)
(IK)
doall ( App I (App (App K K) K ) )
K
```

We can think of this "doall" function as marking all redexes of a term, and contracting them intuitively in parallel. We can also imagine such as step as marking all redexes of a term, and reducing them for example in an inside-out order, for example $\mathsf{I}\,(\mathsf{K}\,\mathsf{K}\,\mathsf{K}) \to \mathsf{I}\,\mathsf{K} \to \mathsf{K}$,

Exercises Equational Specifications

The equational specifications from this section are discussed in the lectures. We work with the file `Practicum3.hs`.

1. Implement in Haskell the initial model of the specification `Toy` of the slides, and of Example 6.4 of the course notes. Take care in defining the syntax:

   ```
   data Thing =
     deriving Show
   ```

   Complete the line after `Thing`. The data type `Thing` should contain at least the interpretation of `c` and the interpretation of `d`. Does the initial model have more elements than these two? If so, add it/them to the data type `Thing`.

   Make sure that your implementation indeed satisfies the two equations.

2. Implement in Haskell the first model ($\mathcal{K}$) from Exercise 6.3.

   Make sure that your implementation indeed satisfies the two equations.