

Prof. Dr. Jürgen Giesl  
René Thiemann

## Exercises *Functional Programming* – Sheet 5

Solutions will be collected until Thursday, Nov 21, 2002 in the exercise course.

### Exercise 1 ( 1.5 + 2.5 points)

- (a) Define a function `appendTwiceFile :: String -> String -> IO ()` that takes a filename `fn` and a string `s` as input and appends `s` to the contents of `fn` twice. You should only use the predefined functions `readFile` and `writeFile` for I/O. In this part, you are not allowed to use the `do`-notation.
- (b) Define a function `showNthLine :: IO ()` that asks the user for a filename and a number `n`. Then the function should show the `n`-th line of the file on the screen. You can use the predefined function `putStr :: String -> IO ()` to show a given string on the screen. You should then mark the parts of your program that are free of I/O-actions.

### Exercise 2 ( 3.5 + 1 + 2.5 + 1 points)

The error-monad `Maybe` was given in the lecture as follows:

```
data Maybe a = Just a | Nothing
```

```
instance Monad Maybe where  
  return      = Just  
  Nothing >>= q = Nothing  
  Just x >>= q = q x
```

- (a) Show that the three monad-rules hold in the `Maybe`-monad.
- (b) In the file `Monad.hs` the `Maybe`-monad is extended to the `ErrMonad` which has information about the type of error and a description of the error as a `String`.

```
data ErrMonad a = Value a | Error ErrType String
```

Make `ErrMonad` a member of class `Monad` by implementing the corresponding `instance`-declaration.

- (c) We deal with the evaluation of terms once again and want to evaluate these terms, but do not want to get negative values. So, there are two error types: Division by zero and negative value. Therefore, we use the following implementation of `ErrType`:

```
data ErrType = DivByZero | NegValue
```

There is an evaluator `eval :: Term -> ErrMonad Float` given in the file *Monad.hs*. There is also a new type `Handler a = ErrType -> Maybe a`. Given an error, a handler can probably compute a default value (`Just x`), where `x` is the result that should be returned in case of such an error. If the handler cannot deal with this error, it returns `Nothing`. Furthermore, a function `try :: ErrMonad a -> [Handler a] -> ErrMonad a` is given. With these constructs it is possible to catch some errors using a list of handlers and return an alternative value in case of an error. So `try exp handlerList` tries to evaluate the expression `exp`. If this does not lead to an error, the resulting value is returned as the result. Otherwise, the handlers in `handlerList` have the chance to catch the error and return their result. Please look at the implementation for further details.

You should extend the given evaluator such that the multiplication of zero with an expression that results in a negative-value-error should return a zero. To do this, modify the `eval` function by using the `try` function and write an appropriate handler. Your solution should be able to deal with both `0 * negValue` and `negValue * 0`.

- (d) The expression `0 * divByZero` should not be evaluated to 0 because in contrast to the `negValues`, `divByZero` does not always represent a finite value.

Is there a bug in the above sentence?

### Exercise 3 ( 1.5 + 1.5 + 3 points)

This part deals with the state-monad. The idea is to keep a certain state inside this monad. Here, we use a simplified version where the type of the state is fixed as a string. We will use the monad to write a scanner, a program that transforms a given string into a token-list. For the scanner, we will keep the input-string inside the monad. With some special functions we will be able to access and modify the value of the encapsulated string.

The implementation is given as follows:

```
data ScanMonad a = MakeSM (String -> (String,a))

instance Monad ScanMonad where
  return a = MakeSM (\s -> (s,a))
  (MakeSM s_to_sa) >>= a_to_stb =
    MakeSM ( \s -> let (s',a) = s_to_sa s
                    (MakeSM s_to_sb) = a_to_stb a
                    in s_to_sb s' )
```

So, in the `bind` function the initial string `s` is passed to the first argument of `bind` that contains a mapping from `String` to `(String,a)`. The resulting string and value are `s'` and `a`. The second argument of `bind` (of type `a -> ScanMonad b`) is then applied to value `a`. So we will get a new string transformer of type `String -> (String,b)`. Into this we put the `String s'` and take the output as the final result.

We can now define some access-functions like

```
putString :: String -> ScanMonad ()
putState str = MakeSM ( \s -> (str, ()) )

getString :: ScanMonad String
getString = MakeSM ( \s -> (s,s) )
```

There is a scanner `scan :: ScanMonad [Token]` given in the file `Monad.hs`. It uses the string inside the monad as input. This string should end with a dot (`“.”`). The scanner should transform this input that represents a polynomial into a token-list. The data-type `Token` is defined as follows:

```
data Token = Id String | Number Int | Plus | Times | ParenL | ParenR
```

So, `"x*(fun+12)." --scan--> [Id "x",Times,ParenL,Id "fun",Plus,Number 12,ParenR]`.

Write the following functions:

(a) `lookChar :: ScanMonad Char`

`lookChar` should return the first character of the string in the state, but should not change the internal string. If the string is empty it should result in an error.

(b) `dropChar :: ScanMonad ()`

`dropChar` should modify the string inside the monad such that the first character of the string is dropped. If this is not possible the function should result in an error.

(c) `scanWhile :: String -> ScanMonad String`

`scanWhile` takes a set of characters represented as a string. The result of `scanWhile set` has a value and it transforms the state. Given a state `s` (a string), the resulting value is the beginning prefix of `s` which only contains characters from `set`. The resulting state is `s` without the prefix.

So, if the initial state is `"hellas"` then `scanWhile "abcdefghi"` should return the value `"he"` and the state should be changed to `"llas"`.

Hint: If you want to test your routines you have to call `run "my*(test+poly)." scan` or `run "someString" (scanWhile someSet)`