Assignment 4: Functors and Monads in Haskell

SWEN 431

Due date: November 2

Your task for this assignment is to implement the lambda calculus and various extensions in Haskell using monads. Your submission should be a Literate Haskell file compiled with LaTeX in the polycode format, with accompanying explanations of your code. Examine the source of this document to find extra formatting for some binary operators. Your submission should form a coherent essay.

You *may not* import anything defined in the `transformers` or `mtl` packages. You must define the relevant classes and functions yourself. When defining *Monad* instances, you must also include *Functor* and *Applicative* instances.

1 Implementing the Lambda Calculus (50 marks)

Consider the following syntax for the pure, untyped lambda calculus:

$$e \ ::= \ x \mid \lambda x.e \mid e \ e \mid (e)$$

This question requires you to implement a data representation of a lambda calculus program, and a parser and interpreter for that representation.

1.1 The Parser

We begin by defining a data structure and accompanying functions for parsing simple context-free programs. The parser will not bother with lexing, and will just produce an AST directly from text.

Note that while the parser does form a monad, we do not define it here. *You may only use the applicative functor interface* to form your parser.

```
module Parser
  (Parser, parse,
   parsers, parens,
   symbol, variable, keyword, number, string
  ) where
import Control.Applicative
import Data.Bifunctor
import Data.Char
import Data.Foldable
import Control.Monad
```

The parser will be fairly simple wrapper around a function that takes the text to parse, and either fails with an error message, or produces the result of the parse and the remaining text.

```
newtype Parser a = Parser {runParser :: String → Either String (a, String)}
```

This parser forms a *Functor* over its single type argument, with *fmap* transforming the first element of the resulting pair if a parse step succeeds.

```
instance Functor Parser where
  fmap f = Parser ∘ (fmap ∘ fmap) (first f) ∘ runParser
```

It also forms an *Applicative* functor, with *pure* maintaining the remaining text, and the <*> operator running the parser from left to right, halting on the first failure.

```
instance Applicative Parser where
  pure a = Parser (Right ∘ (,) a)
  f <*> a = Parser (runParser f >>= λ(g, s) → first g <$> runParser a s)
```

We also need to be able to combine parsers such that if one fails, we can try another one. The *Parser* type forms an *Alternative* functor that provides this functionality.

```
instance Alternative Parser where
  empty = Parser (const (Left "Empty parser"))
  a <|> b = Parser (λs → onLeft (runParser b s) (runParser a s))
    where onLeft = flip either Right ∘ const
```

Now two parsers can be combined with the <|> operator, which will use the right parser if the left one fails.

The top-level parser definition can be run by applying the string to parse to the wrapped function, with a successful parse producing an empty string for the remaining content.

```
parse :: Parser a → String → Either String a
parse p = either Left leftovers ∘ runParser p
  where
    leftovers (a, "") = Right a
    leftovers (_, l)  = Left ("Unexpected leftovers: " ++ l)
```

Before defining our parsing operations, we'll define an auxiliary function to construct a parser that ignores all leading whitespace and raises an exception if it encounters the end of the input before it manages to run the given parsing function, otherwise splitting the head and tail of the input for lookahead.

```
parser :: ((Char, String) → Either String (a, String)) → Parser a
parser f = Parser (safeSplit ∘ dropWhile isSpace >>= f)
  where
    safeSplit []     = Left "Unexpected end of input"
    safeSplit (h : t) = Right (h, t)
```

Functions matching the input type can then be lifted into the *Parser* type. Here's a function that consumes an identifier and returns what it consumed:

```
identifier :: ((Char, String) → Either String (String, String))
identifier (h, r)
  | isAlpha h = Right (first (h:) (span isAlphaNum r))
  | otherwise = Left ("Expected identifier, but found " ++ show h)
```

When we use <|>, if both of the options fail then the resulting error will be the one raised by the right parser, but we want an error that reports their combination. Here's a helper that tries a list of parsers and reports a custom error if none of them succeed.

```
parsers :: String → [Parser a] → Parser a
parsers name ps = foldl' (<|>) empty ps <|> parser err
  where err (h, _) = Left ("Expected " ++ name ++ " but found " ++ show h)
```

So, to parse expressions, one can combine a series of parsers with a call to the *parsers* function like so:

```
expression = parsers "an expression"
  [application, variable, function, parens expression]
```

Now we can define some simple parsing operations to construct parsers. The first is a function that takes a character, and produces a parser that consumes that character, producing an error if the next character to parse is not the one it was given.

```
symbol :: Char → Parser ()
symbol c = parser symbol'
  where
    symbol' (h, r)
      | c ≡ h      = Right ((), r)
      | otherwise = Left ("Expected " ++ show c ++ ", but found " ++ show h)
```

The second operation is a parser that will consume a variable and return the name it consumed, making it exactly the parser form of the *identifier* function.

```
variable :: Parser String
variable = parser identifier
```

We can produce new operations by combining these existing ones. Here is a function which, given a parser, produces a parser that consumes parentheses around the given parse.

```
parens :: Parser a → Parser a
parens p = symbol '(' *> p <* symbol ')'
```

These definitions are all we need for now. As we extend our language beyond the pure lambda calculus, we will add extra functions.

1.2 Implementation

With the parser given above, for the given syntax definition for expression, define:

1) A data structure that represents an expression (10 marks).

   data *Expr*

2) A parser for an expression into your new type (20 marks).

   expression :: *Parser Expr*

   You are encouraged to include a syntax definition alongside your parser. Your concrete syntax should be simple: use the Unicode symbol for lambdas, and don't permit functions with more than one parameter.

3) A function for reducing an expression to normal form (20 marks).

   evaluate :: *Expr* → *Expr*

   You are encouraged to include reduction rules alongside your definitions. Don't forget to discuss which order your reduction uses, and how you implement substitution when applying functions.

Add a *main* function which takes a file name to parse and run on the command line, printing out the result of parsing and evaluating the program. Command line arguments are accessible at *System.Environment.getArgs*.

2 Types, Statements, and Variables (40 marks)

Consider the following changes to the previous syntax:

$$e \ ::= \ x \mid \lambda x : \tau . e \mid e \ e \mid (e)$$

$$\tau \ ::= \ B \mid \tau \to \tau$$

This question requires you to extend your existing implementation with types, primitive values, and other extensions, using the appropriate monads. Use the Unicode rightwards arrow symbol for function types when updating the parser.

2.1 The Parser

In order to handle the extensions above, we will need some more primitive parsing operations. The first is for parsing keywords like **true** and **false**, which is similar to parsing a variable but confirms that the consumed identifier matches the expected name.

```
keyword :: String → Parser ()
keyword key = parser (identifier >>= keyword')
  where
```

```
  keyword' (name, r)
    | name ≡ key = Right ((), r)
    | otherwise  = Left ("Expected " ++ key ++ ", but found " ++ name)
```

Next we need to parse natural numbers, so we define a parser that consumes all of the leading digits and turns them into a *Int*.

```
number :: Parser Int
number = parser number'
  where
    number' (h, r)
      | isDigit h  = Right (first (read ∘ (h:)) (span isDigit r))
      | otherwise = Left ("Expected a number, but found " ++ show h)
```

Finally, we need to parse string literals. We'll only consider simple strings, with no escape characters.

```
string :: Parser String
string = parser string'
  where
    string' (h, r)
      | isQuote h = Right (second tail (break isQuote r))
      | otherwise = Left ("Expected a number, but found " ++ show h)
    isQuote = (≡ '"')
```

This provides us with all we need to begin extending the language.

2.2 Implementation

Implement *Reader*, *Writer*, and *State* monad transformers, and use them appropriately to define:

1) Boolean and natural numbers, along with a simple type system that has the base types *Bool* and *Nat* (10 marks).

   You will need to update your parser to support primitive literals and type annotations, and introduce some basic operations for the new primitives.

$$e \ ::= \ \ldots \mid \textbf{true} \mid \textbf{false} \mid \mathbb{N}$$

$$B \ ::= \ \mathit{Bool} \mid \mathit{Nat}$$

   You are encouraged to include your typing rule definitions.

```
type Check
data Type
check :: Expr → Check Type
```

2) Strings with the base type *String*, and the ability to handle program arguments using the value $argc : Nat$ and the function $argv : Nat \to String$ (10 marks).

   Your parser must also support simple string literals.

$$e \ ::= \ \ldots \mid \mathbb{S}$$

$$B \ ::= \ \ldots \mid String$$

3) The unit value with the base type *Unit*; statements separated by semicolons, which reduce in order and evaluate to the result of the final expression; and a function $print : String \to Unit$ which prints to the output (10 marks).

$$e \ ::= \ \ldots \mid unit \mid e;\ e$$

$$B \ ::= \ \ldots \mid Unit$$

4) Mutable variables, with assignment reducing to the assigned expression (10 marks).

$$e \ ::= \ \ldots \mid \textbf{var}\ x : \tau = e\ \textbf{in}\ e \mid x = e$$

Note that the evaluation must still be pure (that is, not contain any *IO*). The program arguments should be collected before evaluation, and printing to the output should only occur after the evaluation has completed. The type of *evaluate* should be adjusted to compensate for this.

3 Exceptions (10 marks)

Consider this final extension to the previous syntax:

$$e \ ::= \ \ldots \mid \textbf{throw}\ e \mid \textbf{try}\ e\ \textbf{catch}\ x.e$$

This question requires you to extend your existing implementation with exceptions and a simple try-catch mechanism.

3.1 Implementation

Implement a *Result* monad transformer, and use it in combination with the previous monads to define:

1) A throw expression which uses a string as an exception, along with a try-catch expression to catch any exception thrown in the try body, binding the thrown string to an identifier in the catch body (10 marks).

The type of *evaluate* must take into account a potential uncaught exception.