Prof. Dr. Peter Thiemann
Winter Term 2024/25

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Functional Programming
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 7

In this exercise we are going to look at applicative functors and parsers.

Applicative Functors

In Haskell, an *applicative functor* is represented by a functor `f` together with two operations

```
pure  :: a -> f a
(<*>) :: f (a -> b) -> f a -> f b
```

which satisfy the applicative laws:

```
pure id <*> v                 =  v
pure (.) <*> u <*> v <*> w    =  u <*> (v <*> w)
pure f <*> pure x             =  pure (f x)
u <*> pure y                  =  pure ($ y) <*> u
```

Each applicative functor is also a regular functor, i.e. we can recover `fmap` in terms of `pure` and `<*>` as followed:

```
fmap' :: (a -> b) -> f a -> f b
fmap' f x = pure f <*> x
```

Alternative Definition

Just as we have seen Kleisli Categories as an alternative definition for monads, we are also going to have a look at an alternative definition of applicative functors as *lax monoidal functors*. This definition is less ergonomic to use, but yields some insight about what applicative functors are.

In this alternative definition, we have two operators

```
unit  :: f ()
(<,>) :: f a -> f b -> f (a, b)
```

which need to satisfy the following laws:

```
unit <,> fx          =  fmap ((),) fx
fx <,> unit          =  fmap (,()) fx
(fx <,> fy) <,> fz   =  fmap (\(x, (y, z)) -> ((x, y), z)) (fx <,> (fy <,> fz))
```

These laws should remind you of the monoid laws: if we ignore the `fmap`s, then we have an associative binary operation `<,>` with a neutral element `unit`:

```
unit <,> fx          =  fx
fx <,> unit          =  fx
(fx <,> fy) <,> fz   =  fx <,> (fy <,> fz)
```

However, without the `fmap`, those values are not equal, but only isomorphic: If `fx` is a container with value `x` as element, then `unit <,> fx` is the same container with value `((), x)`. Those values are not equal, but only isomorphic, i.e. the values have different structure, but carry precisely the same information. Similarly, if `(fx <,> fy) <,> fz` produces a container with element `((x, y), z)`, then `fx <,> (fy <,> fz)` produces a container with element `(x, (y, z))`, where again the elements are not equal, but only isomorphic.

If we think of a value of type `f a` as some kind of computation, which produces a value of type `a`, then the fact that `f` is an applicative functor means that the effects behave like a monoid, and hence computations can be combined by combining the effects like a monoid and collecting the output values in a pair, allowing them to be manually combined via `fmap` afterwards, if required.

We can derive the alternative definition from the original definition:

```
unit = pure ()
fx <,> fy = pure (,) <*> fx <*> fy
```

and vice versa

```
pure x = fmap (\_ -> x) unit
ff <*> fx = fmap (\(f, x) -> f x) (ff <,> fx)
```

Functors vs Applicatives vs Monads

In the previous exercise, we stated that every monad is an applicative functor, but not vice versa, and every applicative functor is a functor, but not vice versa.

But why exactly is it "not vice versa"? What can an applicative functor do, that a functor cannot do necessarily, and what can a monad do, that an applicative functor cannot do necessarily?

The difference between functors and applicatives is easy to see:

```
fmap  ::   (a -> b) -> (f a -> f b)
(<*>) :: f (a -> b) -> (f a -> f b)
```

With `fmap` we can change the elements of a container / the results produced by a computation, but we cannot combine multiple containers / computations. The latter is possible for applicative functors, because the containers / computations behave like a monoid in some sense, and hence can be combined.

The difference between applicative functors and monads is more subtle:

```
(<*>) :: f (a -> b) -> (f a -> f b)
(>>=) :: f a -> (a -> f b) -> f b
```

To make the difference more tangible, let us first define a slight variation of `(>>=)`:

```
(>>>) :: f a -> (a -> f b) -> f (a, b)
mx >>> f = mx >>= \x -> fmap (\y -> (x, y)) (f x)
```

The `(>>>)` operator behaves like `(>>=)` but the resulting computation does not just return the value of the second computation, but the values of both computations. It is easy to see that `(>>=)` and `(>>>)` are equally expressive, as we can also define `(>>=)` in terms of `(>>>)`, by simple throwing the result of the first computation away:

```
(>>=) :: f a -> (a -> f b) -> f b
mx >>= f = fmap (\(_, y) -> y) (mx >>> f)
```

Now, comparing the `(<,>)` and `(>>>)` operators clearly reveals the difference between a monad and an applicative functor:

```
(<,>) :: f a ->       f b  -> f (a, b)
(>>>) :: f a -> (a -> f b) -> f (a, b)
```

When combining multiple computations, then a monad allows to choose the second computation `f b` based on the value of type `a` produced by the first computation `f a`. With the applicative operators this is not possible: the `(<,>)` and `(<*>)` operators are also combining the effects and the produced values, but they are combining the effects independently of the produced values. This means that for applicative functors, the effects are already known *before* the computation is run, whereas for monads, it might be necessary to run part of the computation first, because the intermediary result can influence, which computation will happen afterwards.

As an example, consider the `echo` computation, we saw in a previous exercise sheet:

```
echo :: IO ()
echo = getLine >>= \line -> putStrLn line
```

The `echo` computation first reads a line from the terminal and then prints it back to the user. This means the value `line` produced by running the `getLine` computation influences the effect of the `putStrLn line` computation, i.e. it influences what is printed to the terminal. Writing the `echo` computation by only using operators from the `Applicative` instance is not possible.

Applicatives and `do`-notation

For monads, it is sometimes possible to use applicative operators instead of `do`-notation. This does not always yield more readable code, but in certain cases it does.

It is possible to replace `do`-notation with applicative operators if the results of previous computations do not influence subsequent computations, but they are only returned at the end, e.g.

```
do
  x1 <- e1
  x2 <- e2
  ...
  xn <- en
  return (f x1 x2 ... xn)
```

where the expressions `e1`, `e2`, ..., `en` do not mention any of the variables `x1`, `x2`, ..., `xn`. In this case the `do`-block can be replaced by

```
pure f <*> e1 <*> e2 <*> ... <*> en
```

This notation runs into problems, if some of the computations' results are not needed to produce the result via `f`, e.g.

```
do
  x1 <- e1
  x2 <- e2
  x3 <- e3
  x4 <- e4
  x5 <- e5
  return (f x2 x4)
```

A straightforward translation to applicative syntax would look as followed:

```
pure (\_ x2 _ x4 _ -> f x2 x4) <*> e1 <*> e2 <*> e3 <*> e4 <*> e5
```

As this is rather noisy, Haskell provides the following operators

```
(<*) :: Applicative f => f a -> f b -> f a
(*>) :: Applicative f => f a -> f b -> f b
fx <* fy = pure (\x y -> x) <*> fx <*> fy
fx *> fy = pure (\x y -> y) <*> fx <*> fy
```

The `(<*)` and `(*>)` operators combine two computations by only keeping the first and second result, respectively. As a mnemonic, you can think of the `<` and `>` symbol as pointing in the direction of the computation whose value should be used in the combined computation. With these operators we can write the above `do`-block as

```
pure f <* e1 <*> e2 <* e3 <*> e4 <* e5
```

which is parenthesized as

```
((((pure f <* e1) <*> e2) <* e3) <*> e4) <* e5
```

As we want to ignore the result of `e1`, we use `pure f <* e1` so the combined computation still produces `f` as a result. We then apply `<*> e2` because the result of `e2` should be the first argument to `f`. We then use `<*` again to ignore the result of `e3`, and so on.

To further improve readability, there is also a infix version of `fmap`:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

and a special case of `fmap`, which replaces the elements in the container with a constant value, ignoring the previous elements in the container:

```
(<$)  :: Functor f =>       b  -> f a -> f b
y <$ fx = (\_ -> y) <$> fx
```

Since `pure f <*> x` is equal to `fmap f x`, we can use these operators to further rewrite the above two `do`-blocks as:

```
f <$> e1 <*> e2 <*> ... <*> en
```

and

```
f <$ e1 <*> e2 <* e3 <*> e4 <* e5
```

which eliminates the use of `pure`.

Noise of History

In the previous exercise sheet, we have shown the following type class hierarchy of monads:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

For a monad `return` and `pure` are always the same. The fact that `return` exists at all is because at the beginning of Haskell, people did not know about `Applicative` and the hierarchy simply was

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

At some point, the `Applicative` type class was introduced in a breaking change (the AMP proposal), and `return` was left in the `Monad` typeclass for backwards compatibility. However, `return` has a default definition of `return = pure` so if you write a `Monad`-instance nowadays, you can leave out `return`, and it will automatically use `pure` from your `Applicative` instance instead.

Exercise 1 (Applicative Instances)

In the previous exercise sheet, we implemented various monads. As we did not cover applicative functors at that point in time, we left the `Applicative` instances `undefined`.

Implement `Applicative` instances for the monads from the previous exercise.

Exercise 2 (Parsers)

In the lecture, we saw an implementation of parser combinators. In this exercise we are going to write a parser for a large subset of JSON, and use the fact that parsers are `Applicative`s to end up with a notation that looks almost like a EBNF grammar specification.

You can find the parser combinators from the lecture in the code supplement for this exercise. The definition differs from the lecture by wrapping the parser type in a `newtype` to make it easier to write type class instances for parsers, i.e. instead of

```
type Parser t r = [t] -> [(r, [t])]
```

it uses

```
newtype Parser t r = Parser { runParser :: [t] -> [(r, [t])] }
```

The following steps will guide you through implementing a JSON parser:

1. Running a parser on a token list determines if a *prefix* of the token list is a word in the parser's language.

   This is necessary to be able to combine parsers, because where the first parser stops is where the next parser continues, but once the final parser is obtained, one usually wants to see if the whole token list is a word in the language, and not just a prefix of it.

   Write a function

```
runParserEnd :: Parser t a -> [t] -> [a]
```

Which runs a parser on a token list and returns all results where the token list has been consumed completely.

Examples:

```
>>> runParser (lit 'a') "a"
[('a',"")]
>>> runParser (lit 'a') "aha"
[('a',"ha")]
>>> runParserEnd (lit 'a') "a"
['a']
>>> runParserEnd (lit 'a') "aha"
[]
```

2. Implement `Functor`, `Applicative`, `Monad`, and `Alternative` instances for `Parser t`.

   To use the `Alternative` type class, you need to `import Control.Applicative`. The `Alternative` type class represents `Applicative` functors, which are equipped with an additional monoid structure (a different one than the monoid-like structure we discussed on the lax monoidal functor representation), where `(<|>)` is an associative binary operation with a neutral element `empty`:

   ```
   class Applicative f => Alternative f where
     empty :: f a
     (<|>) :: f a -> f a -> f a
   ```

   Note that all operations needed to create the instances, except for `(>>=)`, are already defined in the API from the lecture – they just have different names!

   For the monad instance you can think of a value of `Parser t a` as a computation which when run on a list of tokens, tries to parse a certain grammar from a prefix of the tokens, and produces values of type `a` for each successful parse plus the unused suffix of the tokens. In some sense, the `Parser t` monad is a combination of a `State [t]` monad with the list monad. A computation described by `px >>= f` first tries to parse with `px` and then `f` can look at the results to choose a parser with which to continue.

   After writing those instances, the following should be a parser for a simple arithmetic language that only has addition and natural numbers. The grammar for this language is given by:

   $$\begin{aligned} \mathsf{nat} &::= [\text{'0'} - \text{'9'}]^+ \\ \mathsf{expr} &::= \mathsf{nat} \mid \text{'('}\ \mathsf{expr}\ \text{'+'}\ \mathsf{expr}\ \text{')'} \end{aligned}$$

   We write terminal symbols (tokens) in single quotes to distinguish them from non-terminal symbols. The meaning of the grammar is:

   - a nat is one or more digits; and
   - a expr is either a nat or an opening parenthesis, followed by an expr, followed by a plus symbol, followed by an expr, followed by a closing parenthesis.

   Note, that to keep the example simple, whitespace is not allowed and parenthesis are mandatory! The syntax tree for this language is:

   ```
   data Expr1 = ENat Int | EAdd Expr1 Expr1 deriving (Eq, Show)
   ```

   and it can be parsed as followed:

```
-- Parsing a single digit
pDigit :: Parser Char Int
pDigit = msatisfy (\c -> readMaybe [c])

-- Converting a list of digits to an integer
digitsToInt :: [Int] -> Int
digitsToInt ds = sum $ zipWith (*) ds $ (10^) <$> reverse [0..length ds-1]

-- Parsing a natural number (we will discuss `some` in the next step)
nat :: Parser Char Int
nat = digitsToInt <$> some pDigit

-- Parsing an Expr1 via the Applicative and Alternative interfaces
expr1 :: Parser Char Expr1
expr1 = ENat <$> nat
    <|> EAdd <$ lit '(' <*> expr1 <* lit '+' <*> expr1 <* lit ')'

-- Alternative: Parsing an Expr1 via the Monad and Alternative interfaces
expr1' :: Parser Char Expr1
expr1' = enat <|> eadd where
  enat = do
    n <- nat
    return $ ENat n
  eadd = do
    lit '('
    e1 <- expr1
    lit '+'
    e2 <- expr1
    lit ')'
    return $ EAdd e1 e2
```

Compare the definition of `expr1` with the grammar above: if you squint a bit, you should almost literally recognize the grammar in the implementation of the parser!

Also compare `expr1` with `expr1'`: while the monad implementation might still be more easy to understand (for someone new to applicatives), it is also significantly longer and it is more difficult to recognize the grammar in the monadic implementation.

3. Before we get to parsing JSON, it is helpful to extend the parser library with a few more parser combinators.

   Implement the following functions:

   ```
   option :: Parser t a -> Parser t (Maybe a)
   many0  :: Parser t a -> Parser t [a]
   many1  :: Parser t a -> Parser t [a]
   sepBy0 :: Parser t a -> Parser t b -> Parser t [a]
   sepBy1 :: Parser t a -> Parser t b -> Parser t [a]
   ```

   `option p` should return a parser, which parses `p` zero or one times. `many0 p` should return a parser, which parses `p` zero or more times. `many1 p` should return a parser, which parses `p` one or more times. `sepBy0 p psep` should return a parser, which parses `p` zero or more times, but between each two parses `p` it parses a separate according to `psep`. `sepBy1 p psep` should return a parser, which parses `p` one or more times, but between each two parses `p` it parses a separate according to `psep`.

   Examples:

```
>>> runParserEnd (option (lit 'a')) ""
[Nothing] -- one successful parse, that returned Nothing
>>> runParserEnd (option (lit 'a')) "a"
[Just 'a']
>>> runParserEnd (option (lit 'a')) "aaa"
[]  -- no successful parses

>>> runParserEnd (many0 (lit 'a')) ""
[[]] -- one successful parse, that returned the empty list
>>> runParserEnd (many0 (lit 'a')) "a"
[['a']]
>>> runParserEnd (many0 (lit 'a')) "aaa"
[['a', 'a', 'a']]

>>> runParserEnd (many1 (lit 'a')) ""
[]  -- no successful parses
>>> runParserEnd (many1 (lit 'a')) "a"
[['a']]
>>> runParserEnd (many1 (lit 'a')) "aaa"
[['a', 'a', 'a']]

>>> runParserEnd (sepBy0 (lit 'a') (lit ',')) ""
[[]]
>>> runParserEnd (sepBy0 (lit 'a') (lit ',')) "a"
[['a']]
>>> runParserEnd (sepBy0 (lit 'a') (lit ',')) "a,a"
[['a', 'a']]
>>> runParserEnd (sepBy0 (lit 'a') (lit ',')) "a,a,a"
[['a', 'a', 'a']]
>>> runParserEnd (sepBy0 (lit 'a') (lit ',')) "a,a,a,"
[] -- no trailing separators allowed
```

Remark: All of the above functions are not specific to parsers, but can be implemented for any `Applicative` with an `Alternative` instance! In particular, `many0`, `many1`, and `option` are already implemented generically in `Control.Applicative` with names `many`, `some`, and `optional`. For more information consult the documentation at: https://hackage.haskell.org/package/base-4.20.0.1/docs/Control-Applicative.html#t:Alternative

4. Implement a parser for a subset of JSON consisting of integers, booleans, null, strings, lists, and objects.

   The datatype for JSON values is given by:

```
data JSON = JInt Int
          | JBool Bool
          | JNull
          | JString String
          | JList [JSON]
          | JObject [(String, JSON)]
          deriving (Show, Eq)
```

The grammar describing the concrete syntax is given by

$$
\begin{aligned}
\text{int} &::= \text{'-'}^{?}\ [\text{'0'} - \text{'9'}]^{+} \\
\text{bool} &::= \text{'true'} \mid \text{'false'} \\
\text{str} &::= \text{'"'}\ [\hat{}\ \text{'"'}]^{*}\ \text{'"'} \\
\text{ws} &::= (\text{' '} \mid \text{'\textbackslash n'} \mid \text{'\textbackslash t'} \mid \text{'\textbackslash r'})^{*} \\
\text{json} &::= \text{int} \mid \text{bool} \mid \text{'null'} \mid \text{str} \mid \text{'['}\ \text{ws}\ \text{jsons}^{?}\ \text{ws}\ \text{']'} \mid \text{'\{'}\ \text{ws}\ \text{items}^{?}\ \text{ws}\ \text{'\}'} \\
\text{jsons} &::= \text{json} \mid \text{json}\ \text{ws}\ \text{','}\ \text{ws}\ \text{jsons} \\
\text{items} &::= \text{item} \mid \text{item}\ \text{ws}\ \text{','}\ \text{ws}\ \text{items} \\
\text{item} &::= \text{str}\ \text{ws}\ \text{':'}\ \text{ws}\ \text{json} \\
\text{json'} &::= \text{'ws'}\ \text{json}\ \text{'ws'}
\end{aligned}
$$

The meaning of the grammar is:

- an int is optionally a '-' symbol followed by one or more digits;
- a bool is either the tokens 'true' or 'false';
- a str starts with a double quote, followed by zero or more characters that are not a double quote, and ends with a double quote;
- ws represents zero or more whitespace symbols (spaces, newlines, tabs, carriage returns). This is necessary, since we are writing a parser directly on `Char` tokens, so we also need to tell the parser where whitespace is allowed;
- a json value is either an int, a bool, the tokens 'null', a str, a list of json values, or an object, i.e. dictionary from strings to json values. Lists and objects require extra care, because JSON requires that list entries and object items are separated by commas, where no trailing comma is allowed, e.g. `[1,2,3]` is ok, but `[1,2,3,]` is not. A list starts with the '[' symbol, followed by zero or more whitspaces, followed by optionally jsons, followed by zero or more whitspaces, followed by the ']' symbol. A jsons is either a single json value or a json value followed by a comma, followed by jsons. An object uses the same pattern as for lists to make sure that each item is separated with a comma, but no trailing comma is allowed; and
- json' is the start symbol, which we use to parse full json documents. It is the same as json, but allows whitespace before and after the actual JSON code.

Hint: You might want to use `sepBy0` instead of literally implementing jsons and items.

Example: If you have a file `example.json` with content

```
{
  "foo": 42,
  "bar": [1, 2, false, null],
  "baz": "boo"
}
```

then you should get the following output when parsing it:

```
>>> runParserEnd json' <$> readFile "example.json"
[
  JObject [
    ("foo", JInt 42),
    ("bar", JList [JInt 1, JInt 2, JBool False, JNull]),
    ("baz", JString "boo")
  ]
]
```