CSCI 365 Problem Set 8: Applicative

Due Friday, 5 April 2024

Specification

To receive credit for this problem set:

- Complete at least 9 out of 12 exercises.
- Submit `AParser.hs` and `SExpr.hs`. You should take the versions that I have provided and add your solutions to them.
- Any code you write must adhere to the Haskell style guide linked from the course web page.

Introduction

A *parser* is an algorithm which takes unstructured data as input (often a `String`) and produces structured data as output. For example, when you load a Haskell file into `ghci`, the first thing it does is *parse* your file in order to turn it from a long `String` into an *abstract syntax tree* representing your code in a more structured form.

A Parser for Things is a function from Strings to Maybe a Pair of a Thing and a String.

Concretely, we will represent a parser for a value of type `a` as a function which takes a `String` represnting the input to be parsed, and succeeds or fails; if it succeeds, it returns the parsed value along with whatever part of the input it did not use.

```
newtype Parser a where
  P :: (String -> Maybe (a, String)) -> Parser a

runParser :: Parser a -> String -> Maybe (a, String)
runParser (P f) = f
```

For example, `satisfy` takes a `Char` predicate and constructs a parser which succeeds only if it sees a `Char` that satisfies the predicate (which it then returns). If it encounters a `Char` that does not satisfy the predicate (or an empty input), it fails.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = P f
  where
    f [] = Nothing    -- fail on an empty input
    f (x:xs)          -- check if x satisfies the predicate
                        -- if so, return x along with the remainder
                        -- of the input (that is, xs)
```

```
        | p x       = Just (x, xs)
        | otherwise = Nothing  -- otherwise, fail
```

Using satisfy, we can also define the parser `char`, which expects to see exactly a given character and fails otherwise.

```
char :: Char -> Parser Char
char c = satisfy (== c)
```

For example:

```
*Parser> runParser (satisfy isUpper) "ABC"
Just ('A',"BC")
*Parser> runParser (satisfy isUpper) "abc"
Nothing
*Parser> runParser (char 'x') "xyz"
Just ('x',"yz")
```

For convenience, I've also provided you with a parser for positive integers:

```
posInt :: Parser Integer
posInt = P f
  where
    f xs
      | null ns   = Nothing
      | otherwise = Just (read ns, rest)
      where (ns, rest) = span isDigit xs
```

Tools for building parsers

However, implementing parsers explicitly like this is tedious and error-prone for anything beyond the most basic primitive parsers. The real power of this approach comes from the ability to create complex parsers by *combining* simpler ones. And this power of combining will be given to us by. . . you guessed it, `Applicative`.

Exercise 1

First, you'll need to implement a `Functor` instance for `Parser`.

Hint: You may find it useful to implement a function

```
first :: (a -> b) -> (a,c) -> (b,c)
```

or to note that such a function is already available (with an even more general type) in the `Data.Bifunctor` module.

Exercise 2

Now implement an `Applicative` instance for `Parser`:

- `pure a` represents the parser which consumes no input (that is, it returns the `String` unchanged) and successfully returns a result of `a`.

- `p1 <*> p2` represents the parser which first runs `p1` (which might consume some input and produce a function), then passes the *remaining* input to `p2` (which might consume more input and produces some value), then returns the result of applying the function to the value. However, if either `p1` or `p2` fails then the whole thing should also fail (put another way, `p1 <*> p2` only succeeds if both `p1` and `p2` succeed).

Hint: You should be able to make good use of the `Applicative` instance for `Maybe`.

So what is this good for? Suppose we have a type `Employee` defined as follows:

```
type Name = String
data Employee = Emp { name :: Name, phone :: String }
```

If we have a `Name` and a `String`, we can apply the `Emp` constructor to them to create an `Employee` value. But what if we have not a `Name` and a `String`, but a way to *parse* a `Name` and a `String`? Well, we can use the `Applicative` instance for `Parser` to make an employee parser from name and phone parsers. That is, if

```
parseName  :: Parser Name
parsePhone :: Parser String
```

then

```
Emp <$> parseName <*> parsePhone :: Parser Employee
```

is a parser which first reads a name from the input, then a phone number, and returns them combined into an `Employee` record. Of course, this assumes that the name and phone number are right next to each other in the input, with no intervening separators. We'll see later how to make parsers that can throw away extra stuff that doesn't directly correspond to information we want to parse.

You can also test your `Applicative` instance using other simple applications of functions to multiple parsers. You should implement each of the following exercises using the `Applicative` interface to put together simpler parsers into more complex ones. **Do *not* implement them using the low-level definition of a `Parser`!** In other words, for these exercises you should think of the `Parser` type as a black box—you should *not* pattern-match on `Parser`. Pretend that you do not have access to the `Parser` constructor or even know how the `Parser` type is defined.

Note that you should add these functions to the module's export list (the list of things in parentheses right after `module AParser` at the top of the file).

Exercise 3 Create a parser

```
abParser :: Parser (Char, Char)
```

which expects to see the characters `'a'` and `'b'` and returns them as a pair. That is,

```
*AParser> runParser abParser "abcdef"
Just (('a','b'),"cdef")
*AParser> runParser abParser "aebcdf"
Nothing
```

Exercise 4 Now create a parser

```
abParser_ :: Parser ()
```

which acts in the same way as `abParser` but returns `()` instead of the characters `'a'` and `'b'`.

```
*AParser> runParser abParser_ "abcdef"
Just ((),"cdef")
*AParser> runParser abParser_ "aebcdf"
Nothing
```

Exercise 5 Create a parser `intPair` which reads two integer values separated by a single space and returns the integer values in a list. You should use the provided `posInt` to parse the integer values.

```
*AParser> runParser intPair "12 34"
Just ([12,34],"")
```

Exercise 6

`Applicative` by itself can be used to make parsers for simple, fixed formats. But for any format involving *choice* (*e.g.* "…after the colon there can be a number **or** a word **or** parentheses…") `Applicative` is not quite enough. To handle choice we turn to the `Alternative` class, defined (essentially) as follows:

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

`(<|>)` is intended to represent *choice*: that is, `f1 <|> f2` represents a choice between `f1` and `f2`. `empty` should be the identity element for `(<|>)`, and often represents *failure*.

Write an `Alternative` instance for `Parser`:

Hint: there is already an `Alternative` instance for `Maybe` which you may find useful.

- `empty` represents the parser which always fails.
- `p1 <|> p2` represents the parser which first tries running `p1`. If `p1` succeeds then `p2` is ignored and the result of `p1` is returned. Otherwise, if `p1` fails, then `p2` is tried instead.

Unlike the previous exercise, to implement the `Alternative Parser` instance you will have to actually dig into the definition of `Parser`.

Exercise 7

Now implement a parser

```
intOrUppercase :: Parser ()
```

which parses either an integer value or an uppercase character, and fails otherwise. Again, here you should just rely on the `Applicative` and `Alternative` interfaces; your implementation should not rely on the definition of `Parser`.

```
*Parser> runParser intOrUppercase "342abcd"
Just ((), "abcd")
*Parser> runParser intOrUppercase "XYZ"
Just ((), "YZ")
*Parser> runParser intOrUppercase "foo"
Nothing
```

Parsing S-expressions

Your solutions to this section should go in `SExpr.hs`.

All told, we now have the following:

- the definition of a basic `Parser` type
- a few primitive parsers such as `satisfy`, `char`, and `posInt`
- `Functor`, `Applicative`, and `Alternative` instances for `Parser`

So, what can we do with this? It may not seem like we have much to go on, but it turns out we can actually do quite a lot.

Again, from this point on you should only need to write code that uses interfaces provided by the `Functor`, `Applicative`, and `Alternative` instances, and does not depend on the details of the `Parser` implementation. In fact, `AParser.hs` does not export the `Parser` constructor, so when using it in another module it is literally impossible to depend on the details of its implementation.

Exercise 8

First, let's see how to take a parser for (say) widgets and turn it into a parser for *lists* of widgets. In particular, there are two functions you should implement: `zeroOrMore` takes a parser as input and runs it consecutively as many times as possible (which could be none, if it fails right away), returning a list of the results. `zeroOrMore` always succeeds. `oneOrMore` is similar, except that it requires the input parser to succeed at least once. If the input parser fails right away then `oneOrMore` also fails.

For example, below we use `zeroOrMore` and `oneOrMore` to parse a sequence of uppercase characters. The longest possible sequence of uppercase characters is returned as a list. In this case, `zeroOrMore` and `oneOrMore` behave identically:

```
*AParser> runParser (zeroOrMore (satisfy isUpper)) "ABCdEfgH"
Just ("ABC","dEfgH")
*AParser> runParser (oneOrMore (satisfy isUpper)) "ABCdEfgH"
Just ("ABC","dEfgH")
```

The difference between them can be seen when there is not an uppercase character at the beginning of the input. `zeroOrMore` succeeds and returns the empty list without consuming any input; `oneOrMore` fails.

```
*AParser> runParser (zeroOrMore (satisfy isUpper)) "abcdeFGh"
Just ("","abcdeFGh")
*AParser> runParser (oneOrMore (satisfy isUpper)) "abcdeFGh"
Nothing
```

Implement `zeroOrMore` and `oneOrMore` with the following type signatures:

```
zeroOrMore :: Parser a -> Parser [a]
oneOrMore  :: Parser a -> Parser [a]
```

Hint: To parse one or more occurrences of `p`, run `p` once and then parse zero or more occurrences of `p`. To parse zero or more occurrences of `p`, first try parsing one or more; if that fails, return the empty list.

Exercise 9 There are a few more utility parsers needed before we can accomplish the final parsing task. First, `spaces` should parse a consecutive list of zero or more whitespace characters (use the `isSpace` function from the standard `Data.Char` module).

```
spaces :: Parser String
```

Exercise 10 Next, `ident` should parse an *identifier*, which for our purposes will be an alphabetic character (use `isAlpha`) followed by zero or more alphanumeric characters (use `isAlphaNum`). In other words, an identifier can be any nonempty sequence of letters and digits, except that it may not start with a digit.

```
ident :: Parser String
```

For example:

```
*AParser> runParser ident "foobar baz"
Just ("foobar"," baz")
*AParser> runParser ident "foo33fA"
Just ("foo33fA","")
*AParser> runParser ident "2bad"
Nothing
*AParser> runParser ident ""
Nothing
```

Exercise 11

S-expressions are a simple syntactic format for tree-structured data, originally developed as a syntax for Lisp programs. We'll close out our demonstration of parser combinators by writing a simple S-expression parser.

An *identifier* is represented as just a `String`; the format for valid identifiers is represented by the `ident` parser you wrote in the previous exercise.

```
type Ident = String
```

An "atom" is either an integer value (which can be parsed with `posInt`) or an identifier.

```
data Atom where
  N :: Integer -> Atom
  I :: Ident   -> Atom
  deriving Show
```

Finally, an S-expression is either an atom, or a list of one or more S-expressions.[1]

[1] Actually, this is slightly different than the usual definition of S-expressions in Lisp, which also includes binary "cons" cells; but it's good enough for our purposes.

```
data SExpr where
  A    :: Atom    -> SExpr
  Comb :: [SExpr] -> SExpr
  deriving Show
```

Textually, S-expressions consist of either an atom, *or* an open parenthesis followed by one or more S-expressions followed by a close parenthesis. Spaces are allowed anywhere.

$$\begin{aligned} atom &::= int \\ &\mid ident \end{aligned}$$

$$\begin{aligned} S &::= atom \\ &\mid (S^*) \end{aligned}$$

For example, the following are all valid S-expressions:

```
5
foo3
(bar (foo) 3 5 874)
(((lambda x (lambda y (plus x y))) 3) 5)
(   lots  of   (  spaces   in  )  this ( one ) )
```

I have provided Haskell data types representing S-expressions in `SExpr.hs`. Write a parser for S-expressions, that is, something of type

```
parseSExpr :: Parser SExpr
```

Hints: To parse something but ignore its output, you can use the `(*>)` and `(<*)` operators, which have the types

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

`p1 *> p2` runs `p1` and `p2` in sequence, but ignores the result of `p1` and just returns the result of `p2`. `p1 <* p2` also runs `p1` and `p2` in sequence, but returns the result of `p1` (ignoring `p2`'s result) instead.

For example:

```
*AParser> runParser (posInt *> spaces) "345  "
Just (345,"")
```

- Start by defining `token :: Parser a -> Parser a` which runs the given parser but also eats up any whitespace afterwards.
- You might also like to define `parens :: Parser a -> Parser a` which parses the same thing as the given parser but inside parenthesis. For example, if `posInt` expects to see something like `35`, `parens posInt` would expect to see something like `(35)`. Be sure to use `token` for parsing both parenthesis, so that spaces are allowed after the parentheses.
- Now write `parseAtom :: Parser Atom` (using `token` appropriately around `posInt` and `ident`).
- Finally, write `parseSExpr`.

Further Exploration

Exercise 12 Take a look at the paper *Applicative Programming with Effects*, available from `https://openaccess.city.ac.uk/id/eprint/13222/1/`.[2] This is the paper which originally introduced the `Applicative` abstraction, and is still surprisingly readable. Read at least sections 1 and 2; you may find sections 3 and 4 interesting as well. Explain one thing you learned from reading the paper, or how something in the paper connects to something we did in class.

[2] McBride, Conor, and Ross Paterson. "Applicative programming with effects." Journal of functional programming 18, no. 1 (2008): 1-13.