Monadic Parsers using Haskell

An introduction

Tony Morris

Abstract

A hands-on introductory tutorial to monadic parsers using the Haskell Programming Language http://haskell.org/. Students will require an installation of the Glasgow Haskell Compiler http://haskell.org/ghc and a text editor. Upon completion of the tutorial students will have created a parser library.

Students will be expected to have a basic understanding of Haskell syntax and familiarity with tools -- in particular, the GHC interpreter (GHCi).

Table of Contents

Get Started

Let us start by creating a file called `MyParser.hs`. Then in that file the contents:

```
module MyParser where

import Data.Char
```

Start the GHC interpreter `ghci` and load the source file.

```
$ ghci
GHCi, version 6.10.2: http://www.haskell.org/ghc/  :? for help
Prelude> :load MyParser.hs
[1 of 1] Compiling MyParser         ( MyParser.hs, interpreted )
Ok, modules loaded: MyParser.
*MyParser>
```

What is a Parser?

A data structure

A parser is a function that accepts an input `String` and either fails or produces a value and the remaining input `String`.

```
data Parser a = P {
  parse :: String -> Maybe (String, a)
}
```

Examine the type of `parse`

```
*MyParser> :type parse
parse :: Parser a -> String -> Maybe (String, a)
```

`parse` accepts a parser and input and either fails or produces a value and the remaining input.

Errors?

Our parser is trivial such that error messages or positions of parse failure are not handled. This is done to simplify this exercise.

Our First Parser

value

Let's create a parser that consumes no input and produces a given value.

```
value :: a -> Parser a
value a = P (\s -> Just (s, a))
```

This parser always succeeds (`Just`).

Experiment

Experiment with this parser

```
*MyParser> :type parse (value "boo")
parse (value "boo") :: String -> Maybe (String, [Char])
*MyParser> parse (value 7) "input"
Just ("input",7)
*MyParser> parse (value "boo") "input"
Just ("input","boo")
```

Failing Parser

Let's write a parser that always fails.

```
failed :: Parser a
failed = P (\_ -> Nothing)
```

This parser always fails since it always produces `Nothing`.

Character Parser

character

Let's create a parser that consumes one character and produces that character, unless the input is empty, in which case fail.

```
character :: Parser Char
character = P (\s -> case s of []  -> Nothing
                               (c:r) -> Just (r, c))
```

Experiment

Experiment with the `character` parser

```
*MyParser> :type parse character
parse character :: String -> Maybe (String, Char)
*MyParser> parse character "abc"
Just ("bc",'a')
*MyParser> parse character ""
```

```
Nothing
```

Choice Parser

|||

Write a function that accepts two parsers and returns a parser that tries the first parser for success. If that parser fails, try the second parser.

```
(|||) :: Parser a -> Parser a -> Parser a
P p1 ||| P p2 = P (\s -> case p1 s of v@(Just _) -> v
                                     Nothing -> p2 s)
```

Experiment

Experiment with the choice parser

```
*MyParser> let p = character ||| value 'x' in parse p "abc"
Just ("bc",'a')
*MyParser> let p = character ||| value 'x' in parse p ""
Just ("",'x')
*MyParser> let p = character ||| failed in parse p "abc"
Just ("bc",'a')
*MyParser> let p = character ||| failed in parse p ""
Nothing
```

Mapping Parser

mapParser

Write a function that accepts a parser and a function from its produced value to another value to return a parser that potentially produces that value. We will see later why this function is useful.

```
mapParser :: Parser a -> (a -> b) -> Parser b
mapParser (P p) f = P (\s -> case p s of Just (r, c) -> Just (r, f c)
                                         Nothing -> Nothing)
```

Experiment

Experiment with the mapping parser

```
*MyParser> let p = mapParser character toUpper in parse p "abc"
Just ("bc",'A')
*MyParser> let p = mapParser character toUpper in parse p ""
Nothing
```

Binding Parser

bindParser

Write a function that accepts a parser and a function from its produced value to another parser producing values of some type and returns a parser producing values of that same type. Again, we will see later why this function is useful.

```
bindParser :: Parser a -> (a -> Parser b) -> Parser b
```

```
bindParser (P p) f = P (\s -> case p s of Just (r, c) -> parse (f c) r
                                          Nothing -> Nothing)
```

Experiment

Experiment with the binding parser

```
*MyParser> let p = bindParser character (\c -> if isUpper c then value c else fa
Nothing
*MyParser> let p = bindParser character (\c -> if isUpper c then value c else fa
Nothing
*MyParser> let p = bindParser character (\c -> if isUpper c then value c else fa
Just ("bc",'A')
```

>>>

Write a function that uses `bindParser` but ignores the value that is produced by the given parser. This will be useful for example, when we wish for a parser to consume white-space but not do anything with that white-space.

```
(>>>) :: Parser a -> Parser b -> Parser b
p >>> q = bindParser p (\_ -> q)
```

Sequencing Parser

sequenceParser

Write a function that accepts a list of parsers and produces a parser of lists by calling `bindParser` and `mapParser` as you go along the list.

```
sequenceParser :: [Parser a] -> Parser [a]
sequenceParser [] = value []
sequenceParser (h:t) = bindParser h (\a -> mapParser (sequenceParser t) (\as ->
```

Experiment

Experiment with the sequencing parser

```
*MyParser> let p = sequenceParser [character, value 'b', character] in parse p
Nothing
*MyParser> let p = sequenceParser [character, value 'b', character] in parse p
Nothing
*MyParser> let p = sequenceParser [character, value 'b', character] in parse p
Just ("","abb")
*MyParser> let p = sequenceParser [character, value 'b', character] in parse p
Just ("c","abb")
```

Sequence N Parser

thisMany

Write a function that accepts an integer (n) and a parser and returns a parser that binds 'n' times. We can do this by using the library function `replicate`.

```
thisMany :: Int -> Parser a -> Parser [a]
thisMany n p = sequenceParser (replicate n p)
```

Zero or Many Parser

list / many1

Let's now write two functions. One takes a parser and applies itself zero or many times (`list`) and the other takes a parser and applies itself one or many times (`many1`). These functions are useful for such things as parsing white-space, where we may take a parser of a single white-space character and obtain a parser of e.g. one or many white-space characters.

```
list :: Parser a -> Parser [a]
list k = many1 k ||| value []

many1 :: Parser a -> Parser [a]
many1 k = bindParser k (\k' -> mapParser (list k) (\kk' -> k' : kk'))
```

Experiment

Experiment with the new parsers

```
*MyParser> let p = many1 character in parse p ""
Nothing
*MyParser> let p = many1 character in parse p "abc"
Just ("","abc")
*MyParser> let p = list character in parse p ""
Just ("","")
*MyParser> let p = list character in parse p "abc"
Just ("","abc")
```

Satisfy

satisfy

Write a parser that consumes a character (or fails if there isn't one) and ensures that character meets a given predicate. We might also use this parser to write another parser (`is`) that parses a specific character.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = bindParser character (\c -> if p c then value c else failed)

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

Experiment

Experiment with these parsers

```
*MyParser> let p = satisfy isUpper in parse p ""
Nothing
*MyParser> let p = satisfy isUpper in parse p "abc"
Nothing
*MyParser> let p = satisfy isUpper in parse p "Abc"
Just ("bc",'A')
*MyParser> let p = is 'a' in parse p ""
Nothing
*MyParser> let p = is 'a' in parse p "abc"
```

```
Just ("bc",'a')
*MyParser> let p = is 'a' in parse p "xbc"
Nothing
```

Satisfiers

satisfy

Let us now write a bunch of parsers that use `satisfy` and apply the predicate.

```
digit :: Parser Char
digit = satisfy isDigit

natural :: Parser Int
natural = mapParser (list digit) read

space :: Parser Char
space = satisfy isSpace

spaces :: Parser String
spaces = many1 space

lower :: Parser Char
lower = satisfy isLower

upper :: Parser Char
upper = satisfy isUpper

alpha :: Parser Char
alpha = satisfy isAlpha

alphanum :: Parser Char
alphanum = satisfy isAlphaNum
```

Experiment

Experiment with some of these parsers

```
*MyParser> parse upper ""
Nothing
*MyParser> parse upper "a"
Nothing
*MyParser> parse upper "A"
Just ("",'A')
*MyParser> let p = list digit in parse p ""
Just ("","")
*MyParser> let p = list digit in parse p "79abc"
Just ("abc","79")
```

Introspection

Bits and Pieces

So far we have constructed a small library of parsers. Some of these parsers are constructed from other parsers. This notion of constructing units of work from other, smaller units of work is the essence of functional programming. Does it scale?

Can we write even more higher-level parsers with our library by *gluing* parsers?

Person

Suppose we have a data structure to represent a person. The person data structure has these attributes:

- Age: positive integer
- First Name: non-empty string that starts with a capital letter
- Surname: string that starts with a capital letter and is followed by 5 or more lower-case letters
- Gender: character that must be 'm' or 'f'
- Phone: string of digits, dots or hyphens but must start with a digit and end with a hash (#)

```
data Person = Person {
  age :: Int,
  firstName :: String,
  surname :: String,
  gender :: Char,
  phone :: String
} deriving Show
```

ageParser

The age parser is easy

```
ageParser :: Parser Int
ageParser = natural
```

Age: positive integer

firstNameParser

The first name parser requires us to piece together the `upper` and `list lower` parser. We do this using `bindParser` and `mapParser`. These two functions are turning out to be quite helpful.

```
firstNameParser :: Parser String
firstNameParser = bindParser upper (\c -> mapParser (list lower) (\cs -> c : cs
```

First Name: non-empty string that starts with a capital letter

surnameParser

The surname parser requires us to piece together the `upper`, `thisMany 5 lower` and `list lower` parsers. Again, we do this using `bindParser` and `mapParser`.

```
surnameParser :: Parser String
surnameParser = bindParser upper (\c -> bindParser (thisMany 5 lower) (\cs -> ma
```

Surname: string that starts with a capital letter and is followed by 5 or more lower-case letters

genderParser

The gender parser uses choice (`|||`) and `is`.

```
genderParser :: Parser Char
genderParser = is 'm' ||| is 'f'
```

Gender: character that must be 'm' or 'f'

phoneBodyParser

We will break the parsing of a phone number into two parts for simplicity. This part will parse the phone number body (without the hash at the end). It uses the `list`, `digit` and `is` parsers along with choice.

```
phoneBodyParser :: Parser String
phoneBodyParser = list (digit ||| is '.' ||| is '-')
```

Phone: string of digits, dots or hyphens but must start with a digit and end with a hash (#)

phoneParser

We will use the `phoneBodyParser` and the `is` parser to create a parser for the phone number. Once again, we glue it all together using `bindParser` and `mapParser`. Notice that we ignore the hash character in the end when we produce the final string.

```
phoneParser :: Parser String
phoneParser = bindParser digit (\d -> bindParser phoneBodyParser (\z -> mapPars
```

Phone: string of digits, dots or hyphens but must start with a digit and end with a hash (#)

Person Parser

Experiment

Experiment with the person parsers

```
*MyParser> parse firstNameParser ""
Nothing
*MyParser> parse firstNameParser "fred"
Nothing
*MyParser> parse firstNameParser "Fred"
Just ("","Fred")
*MyParser> parse surnameParser ""
Nothing
*MyParser> parse surnameParser "fred"
Nothing
*MyParser> parse surnameParser "Fred"
Nothing
*MyParser> parse surnameParser "Frederick"
Just ("","Frederick")
*MyParser> parse genderParser ""
Nothing
*MyParser> parse genderParser "a"
Nothing
*MyParser> parse genderParser "m"
Just ("",'m')
*MyParser> parse genderParser "moo"
Just ("oo",'m')
```

```
*MyParser> parse phoneBodyParser ""
Just ("","")
*MyParser> parse phoneBodyParser "123"
Just ("","123")
*MyParser> parse phoneBodyParser "123-456"
Just ("","123-456")
*MyParser> parse phoneBodyParser "123-456.789"
Just ("","123-456.789")
*MyParser> parse phoneParser ""
Nothing
*MyParser> parse phoneParser "#"
Nothing
*MyParser> parse phoneParser "-#"
Nothing
*MyParser> parse phoneParser "12#"
Just ("","12")
*MyParser> parse phoneParser "1-2#"
Just ("","1-2")
*MyParser> parse phoneParser "123-456.789"
Nothing
*MyParser> parse phoneParser "123-456.789#"
Just ("","123-456.789")
*MyParser> parse phoneParser "123-456.789#"
Just ("","123-456.789")
*MyParser> parse phoneParser "123#"
Just ("","123")
```

So we have parsers for the components of a `Person`. But we need a parser for a person. How can we get one of those?

personParser

By gluing them together with `bindParser` and `mapParser` (of course!).

```
personParser1 :: Parser Person
personParser1 = bindParser ageParser (\age ->
                spaces >>>
                bindParser firstNameParser (\firstName ->
                spaces >>>
                bindParser surnameParser (\surname ->
                spaces >>>
                bindParser genderParser (\gender ->
                spaces >>>
                bindParser phoneParser (\phone ->
                value (Person age firstName surname gender phone))))))
```

Experiment

Experiment with the person parser

```
*MyParser> parse personParser1 ""
Nothing
*MyParser> parse personParser1 "123 Fred Clarkson m 123-456.789#"
Just ("",Person {age = 123, firstName = "Fred", surname = "Clarkson", gender =
*MyParser> parse personParser1 "123 Fred Clarkson m 123-456.789# the rest of th
Just (" the rest of the input",Person {age = 123, firstName = "Fred", surname =
*MyParser> parse personParser1 "123 Fred Clark m 123-456.789# the rest of the i
```

```
Nothing
```

We are successfully parsing a string into a `Person` object with potential failure.

Bind/Map

bindParser/mapParser

We have advanced quite far ahead of what we do with traditional languages. We have glued smaller parts to make larger parts, then glued those larger parts to make even larger parts again.

So what with this `bindParser` and `mapParser` business? They seem rather fundamental. Can we somehow remove the noise to make them easier to use?

You Bet

Haskell has *do notation* to take care of this pattern for us, since it occurs all over the place, not just parsers. First we must implement a type-class:

```
instance Monad Parser where
  (>>=) = bindParser
  return = value
```

Then?

zing!

```
personParser2 :: Parser Person
personParser2 = do age <- ageParser
                   spaces
                   firstName <- firstNameParser
                   spaces
                   surname <- surnameParser
                   spaces
                   gender <- genderParser
                   spaces
                   phone <- phoneParser
                   return (Person age firstName surname gender phone)
```

Conclusion

We have seen how to create a small parser library then use that library to create our own custom parsers. We did this by putting together parts to create more specialised parts. [1]

This process is the essence of functional programming. It scales up to larger programs and demonstrates that functional programming is incredibly practical for solving real-world problems, much more so than traditional methods.

We observed a programming pattern called the monadic model and we exploited Haskell's built-in support for it.

What other types of problems fit this model? The answer(s!) to this question run deep, but that's for another day.

[1] An industrial strength implementation of this type of parser library is Parsec.

A. MyParser.hs

```
module MyParser where

import Data.Char

data Parser a = P {
  parse :: String -> Maybe (String, a)
}

value :: a -> Parser a
value a = P (\s -> Just (s, a))

failed :: Parser a
failed = P (\_ -> Nothing)

character :: Parser Char
character = P (\s -> case s of [] -> Nothing
                               (c:r) -> Just (r, c))

(|||) :: Parser a -> Parser a -> Parser a
P p1 ||| P p2 = P (\s -> case p1 s of v@(Just _) -> v
                                     Nothing -> p2 s)

mapParser :: Parser a -> (a -> b) -> Parser b
mapParser (P p) f = P (\s -> case p s of Just (r, c) -> Just (r, f c)
                                         Nothing -> Nothing)

bindParser :: Parser a -> (a -> Parser b) -> Parser b
bindParser (P p) f = P (\s -> case p s of Just (r, c) -> parse (f c) r
                                          Nothing -> Nothing)

(>>>) :: Parser a -> Parser b -> Parser b
p >>> q = bindParser p (\_ -> q)

sequenceParser :: [Parser a] -> Parser [a]
sequenceParser [] = value []
sequenceParser (h:t) = bindParser h (\a -> mapParser (sequenceParser t) (\as ->

thisMany :: Int -> Parser a -> Parser [a]
thisMany n p = sequenceParser (replicate n p)

list :: Parser a -> Parser [a]
list k = many1 k ||| value []

many1 :: Parser a -> Parser [a]
many1 k = bindParser k (\k' -> mapParser (list k) (\kk' -> k' : kk'))

satisfy :: (Char -> Bool) -> Parser Char
satisfy p = bindParser character (\c -> if p c then value c else failed)

is :: Char -> Parser Char
is c = satisfy (== c)

digit :: Parser Char
digit = satisfy isDigit
```

```
natural :: Parser Int
natural = mapParser (list digit) read

space :: Parser Char
space = satisfy isSpace

spaces :: Parser String
spaces = many1 space

lower :: Parser Char
lower = satisfy isLower

upper :: Parser Char
upper = satisfy isUpper

alpha :: Parser Char
alpha = satisfy isAlpha

alphanum :: Parser Char
alphanum = satisfy isAlphaNum

data Person = Person {
  age :: Int,
  firstName :: String,
  surname :: String,
  gender :: Char,
  phone :: String
} deriving Show

ageParser :: Parser Int
ageParser = natural

firstNameParser :: Parser String
firstNameParser = bindParser upper (\c -> mapParser (list lower) (\cs -> c : cs

surnameParser :: Parser String
surnameParser = bindParser upper (\c -> bindParser (thisMany 5 lower) (\cs -> ma

genderParser :: Parser Char
genderParser = is 'm' ||| is 'f'

phoneBodyParser :: Parser String
phoneBodyParser = list (digit ||| is '.' ||| is '-')

phoneParser :: Parser String
phoneParser = bindParser digit (\d -> bindParser phoneBodyParser (\z -> mapPars

personParser1 :: Parser Person
personParser1 = bindParser ageParser (\age ->
                spaces >>>
                bindParser firstNameParser (\firstName ->
                spaces >>>
                bindParser surnameParser (\surname ->
                spaces >>>
                bindParser genderParser (\gender ->
                spaces >>>
                bindParser phoneParser (\phone ->
                value (Person age firstName surname gender phone))))))
```

```
instance Monad Parser where
  (>>=) = bindParser
  return = value

personParser2 :: Parser Person
personParser2 = do age <- ageParser
                   spaces
                   firstName <- firstNameParser
                   spaces
                   surname <- surnameParser
                   spaces
                   gender <- genderParser
                   spaces
                   phone <- phoneParser
                   return (Person age firstName surname gender phone)
```

B. MyParser.scala

```
package MyParser

import Character._

case class Parser[A](parse: List[Char] => Option[(List[Char], A)]) {
  def |||(p2: => Parser[A]): Parser[A] = Parser(s => this parse s match {
    case v@Some(_) => v
    case None => p2 parse s
  })

  def >>>[B](q: => Parser[B]): Parser[B] = Parser.bindParser[A, B](this, _ => q
}

object Parser {
  def value[A](a: A): Parser[A] = Parser(s => Some(s, a))

  def failed[A]: Parser[A] = Parser(s => None)

  def character: Parser[Char] = Parser[Char] {
    case Nil => None
    case c::r => Some(r, c)
  }

  def mapParser[A, B](p: Parser[A], f: A => B): Parser[B] = Parser(s => p parse
    case Some((r, c)) => Some(r, f(c))
    case None => None
  })

  def bindParser[A, B](p: Parser[A], f: A => Parser[B]): Parser[B] = Parser(s =>
    case Some((r, c)) => f(c) parse r
    case None => None
  })

  def sequenceParser[A](ps: List[Parser[A]]): Parser[List[A]] = ps match {
    case Nil => value(Nil)
    case h::t => bindParser(h, (a: A) => mapParser(sequenceParser(t), (as: List
  }
```

```
  def thisMany[A](n: Int, p: Parser[A]): Parser[List[A]] = sequenceParser(List.r

  def list[A](k: Parser[A]): Parser[List[A]] = many1(k) ||| value(Nil)

  def many1[A](k: Parser[A]): Parser[List[A]] = bindParser(k, (kx: A) => mapPars

  def satisfy(p: Char => Boolean): Parser[Char] = bindParser(character, (c: Char

  def is(c: Char): Parser[Char] = satisfy(_ == c)

  val digit: Parser[Char] = satisfy(isDigit(_))

  val natural: Parser[Int] = mapParser(list(digit), (_: List[Char]).mkString.toI

  val space: Parser[Char] = satisfy(isWhitespace(_))

  val spaces: Parser[List[Char]] = many1(space)

  val lower: Parser[Char] = satisfy(isLowerCase(_))

  val upper: Parser[Char] = satisfy(isUpperCase(_))

  val alpha: Parser[Char] = satisfy(isLetter(_))

  val alphaNum: Parser[Char] = satisfy(isLetterOrDigit(_))
}

case class Person(age: Int, firstName: String, surname: String, gender: Char, ph

object Person {
  import Parser._

  val ageParser: Parser[Int] = natural

  val firstNameParser: Parser[String] = bindParser(upper, (c: Char) => mapParser

  val surnameParser: Parser[String] = bindParser(upper, (c: Char) => bindParser

  val genderParser: Parser[Char] = is('m') ||| is('f')

  val phoneBodyParser: Parser[String] = mapParser(list(digit ||| is('.') ||| is

  val phoneParser: Parser[String] = bindParser(digit, (d: Char) => bindParser(ph

  val personParser1: Parser[Person] = bindParser(ageParser, (age: Int) =>
                                      spaces >>>
                                      bindParser(firstNameParser, (firstName: St
                                      spaces >>>
                                      bindParser(surnameParser, (surname: String
                                      spaces >>>
                                      bindParser(genderParser, (gender: Char) =>
                                      spaces >>>
                                      bindParser(phoneParser, (phone: String) =>
                                      value(Person(age, firstName, surname, gend

  implicit def PersonMonad[A](p: Parser[A]) = new {
    def map[B](f: A => B) = mapParser(p, f)
    def flatMap[B](f: A => Parser[B]) = bindParser(p, f)
```

```
  }

  val personParser2: Parser[Person] = for(age <- ageParser;
                                          _ <- spaces;
                                          firstName <- firstNameParser;
                                          _ <- spaces;
                                          surname <- surnameParser;
                                          _ <- spaces;
                                          gender <- genderParser;
                                          _ <- spaces;
                                          phone <- phoneParser)
                                        yield Person(age, firstName, surname, gen
}
```

C. MyParser.java

```
package MyParser;

import fj.P2;
import fj.F;
import fj.P1;
import static fj.pre.Equal.charEqual;
import static fj.P.p;
import fj.data.Option;
import fj.data.List;
import static fj.data.Validation.parseInt;
import static fj.data.List.asString;
import static fj.data.List.fromString;
import static fj.data.Option.some;
import static fj.data.Option.none;

import static java.lang.Character.*;
import static MyParser.Parser.*;

abstract class Parser<A> {
  abstract Option<P2<List<Character>, A>> parse(List<Character> s);

  Parser<A> or(final P1<Parser<A>> p2) {
    return new Parser<A>() {
      Option<P2<List<Character>, A>> parse(final List<Character> s) {
        final Option<P2<List<Character>, A>> v = Parser.this.parse(s);
        return v.isSome()
            ? v
            : p2._1().parse(s);
      }
    };
  }

  <B> Parser<B> anonymousBind(final P1<Parser<B>> q) {
    return bindParser(this, new F<A, Parser<B>>() {
      public Parser<B> f(final A _) {
        return q._1();
      }
    });
  }

  static <A> Parser<A> value(final A a) {
```

```
  return new Parser<A>() {
    Option<P2<List<Character>, A>> parse(final List<Character> s) {
      return some(p(s, a));
    }
  };
}

static <A> Parser<A> failed() {
  return new Parser<A>() {
    Option<P2<List<Character>, A>> parse(final List<Character> s) {
      return none();
    }
  };
}

static Parser<Character> character() {
  return new Parser<Character>() {
    Option<P2<List<Character>, Character>> parse(final List<Character> s) {
      return s.isEmpty()
          ? Option.<P2<List<Character>, Character>>none()
          : some(p(s.tail(), s.head()));
    }
  };
}

static <A, B> Parser<B> mapParser(final Parser<A> p, final F<A, B> f) {
  return new Parser<B>() {
    Option<P2<List<Character>, B>> parse(final List<Character> s) {
      return p.parse(s).map(new F<P2<List<Character>, A>, P2<List<Character>,
        public P2<List<Character>, B> f(final P2<List<Character>, A> rc) {
          return rc.map2(f);
        }
      });
    }
  };
}

static <A, B> Parser<B> bindParser(final Parser<A> p, final F<A, Parser<B>> f
  return new Parser<B>() {
    Option<P2<List<Character>, B>> parse(final List<Character> s) {
      return p.parse(s).bind(new F<P2<List<Character>, A>, Option<P2<List<Cha
        public Option<P2<List<Character>, B>> f(final P2<List<Character>, A>
          return f.f(rc._2()).parse(rc._1());
        }
      });
    }
  };
}

static <A> Parser<List<A>> sequenceParser(final List<Parser<A>> ps) {
  return ps.isEmpty()
      ? value(List.<A>nil())
      : bindParser(ps.head(), new F<A, Parser<List<A>>>() {
    public Parser<List<A>> f(final A a) {
      return mapParser(sequenceParser(ps.tail()), List.<A>cons().f(a));
    }
  });
}
```

```
static <A> Parser<List<A>> thisMany(final int n, final Parser<A> p) {
  return sequenceParser(List.replicate(n, p));
}

static <A> Parser<List<A>> list(final Parser<A> k) {
  return many1(k).or(new P1<Parser<List<A>>>() {
    public Parser<List<A>> _1() {
      return value(List.<A>nil());
    }
  });
}

static <A> Parser<List<A>> many1(final Parser<A> k) {
  return bindParser(k, new F<A, Parser<List<A>>>() {
    public Parser<List<A>> f(final A kx) {
      return mapParser(list(k), List.<A>cons().f(kx));
    }
  });
}

static Parser<Character> satisfy(final F<Character, Boolean> p) {
  return bindParser(character(), new F<Character, Parser<Character>>() {
    public Parser<Character> f(final Character c) {
      return p.f(c) ? value(c) : Parser.<Character>failed();
    }
  });
}

static Parser<Character> is(final char c) {
  return satisfy(charEqual.eq(c));
}

final static Parser<Character> digit = satisfy(new F<Character, Boolean>() {
  public Boolean f(final Character z) {
    return isDigit(z);
  }
});

final static Parser<Integer> natural = mapParser(list(digit), new F<List<Chara
  public Integer f(final List<Character> z) {
    return parseInt(asString(z)).success();
  }
});

final static Parser<Character> space = satisfy(new F<Character, Boolean>() {
  public Boolean f(final Character z) {
    return isWhitespace(z);
  }
});

final static Parser<List<Character>> spaces = many1(space);

final static Parser<Character> lower = satisfy(new F<Character, Boolean>() {
  public Boolean f(final Character z) {
    return isLowerCase(z);
  }
});
```

```
  final static Parser<Character> upper = satisfy(new F<Character, Boolean>() {
    public Boolean f(final Character z) {
      return isUpperCase(z);
    }
  });

  final static Parser<Character> alpha = satisfy(new F<Character, Boolean>() {
    public Boolean f(final Character z) {
      return isLetter(z);
    }
  });

  final static Parser<Character> alphaNum = satisfy(new F<Character, Boolean>()
    public Boolean f(final Character z) {
      return isLetterOrDigit(z);
    }
  });
}

class Person {
  final int age;
  final String firstName;
  final String surname;
  final char gender;
  final String phone;

  Person(final int age, final String firstName, final String surname, final cha
    this.age = age;
    this.firstName = firstName;
    this.surname = surname;
    this.gender = gender;
    this.phone = phone;
  }

  final static Parser<Integer> ageParser = natural;

  final static Parser<String> firstNameParser = bindParser(upper, new F<Charact
    public Parser<String> f(final Character c) {
      return mapParser(list(lower), new F<List<Character>, String>() {
        public String f(final List<Character> cs) {
          return asString(cs.cons(c));
        }
      });
    }
  });

  final static Parser<String> surnameParser = bindParser(upper, new F<Character
    public Parser<String> f(final Character c) {
      return bindParser(thisMany(5, lower), new F<List<Character>, Parser<Strin
        public Parser<String> f(final List<Character> cs) {
          return mapParser(list(lower), new F<List<Character>, String>() {
            public String f(final List<Character> t) {
              return asString(cs.cons(c).append(t));
            }
          });
        }
      });
```

```
  }
});

final static Parser<Character> genderParser = is('m').or(new P1<Parser<Charac
  public Parser<Character> _1() {
    return is('f');
  }
});

final static Parser<String> phoneBodyParser = mapParser(list(digit.or(new P1<
  public Parser<Character> _1() {
    return is('.');
  }
}).or(new P1<Parser<Character>>() {
  public Parser<Character> _1() {
    return is('-');
  }
})), new F<List<Character>, String>() {
  public String f(final List<Character> z) {
    return asString(z);
  }
});

final static Parser<String> phoneParser = bindParser(digit, new F<Character, 
  public Parser<String> f(final Character d) {
    return bindParser(phoneBodyParser, new F<String, Parser<String>>() {
      public Parser<String> f(final String z) {
        return mapParser(is('#'), new F<Character, String>() {
          public String f(final Character _) {
            return asString(fromString(z).cons(d));
          }
        });
      }
    });
  }
});

final static Parser<Person> personParser1 = bindParser(ageParser, new F<Integ
  public Parser<Person> f(final Integer age) {
    return spaces.anonymousBind(new P1<Parser<Person>>() {
      public Parser<Person> _1() {
        return bindParser(firstNameParser, new F<String, Parser<Person>>() {
          public Parser<Person> f(final String firstName) {
            return spaces.anonymousBind(new P1<Parser<Person>>() {
              public Parser<Person> _1() {
                return bindParser(surnameParser, new F<String, Parser<Person>
                  public Parser<Person> f(final String surname) {
                    return spaces.anonymousBind(new P1<Parser<Person>>() {
                      public Parser<Person> _1() {
                        return bindParser(genderParser, new F<Character, Pars
                          public Parser<Person> f(final Character gender) {
                            return spaces.anonymousBind(new P1<Parser<Person>
                              public Parser<Person> _1() {
                                return bindParser(phoneParser, new F<String, 
                                  public Parser<Person> f(final String phone)
                                    return value(new Person(age, firstName, s
                                  }
                                });
```

```
                                  }
                                });
                              }
                            });
                          }
                        });
                      }
                    });
                  }
                });
              }
            });
          }
        });
      }
    });
  }
}
```

D. Licence

This presentation is released under a Creative Commons - Attribution, Non-Commercial, No Derivatives Works licence.

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE

1. Definitions

   1. "Collective Work" means a work, such as a periodical issue, anthology or
   2. "Derivative Work" means a work that reproduces a substantial part of the
   3. "Licensor" means the individual or entity that offers the Work under the
   4. "Moral rights law" means laws under which an individual who creates a wor
   5. "Original Author" means the individual or entity who created the Work.
   6. "Work" means the work or other subject-matter protected by copyright that
   7. "You" means an individual or entity exercising rights under this Licence
   8. "Licence Elements" means the following high-level licence attributes as s

2. Fair Dealing and Other Rights. Nothing in this Licence excludes or modifies,

3. Licence Grant. Subject to the terms and conditions of this Licence, Licensor

   1. to reproduce the Work, to incorporate the Work into one or more Collectiv
   2. to publish, communicate to the public, distribute copies or records of, e

The above rights may be exercised in all media and formats whether now known or

4. Restrictions. The licence granted in Section 3 above is expressly made subje

   1. You may publish, communicate to the public, distribute, publicly exhibit
   2. You may not exercise any of the rights granted to You in Section 3 above
   3. If you publish, communicate to the public, distribute, publicly exhibit o
   4. For the avoidance of doubt, where the Work is a musical composition:
      1. Performance Royalties Under Blanket Licences. Licensor reserves the
      2. Mechanical Rights and Statutory Royalties. Licensor reserves the ex
   5. Webcasting Rights and Statutory Royalties. For the avoidance of doubt, wh
   6. False attribution prohibited. Except as otherwise agreed in writing by th

   7. Prejudice to honour or reputation prohibited. Except as otherwise agreed i

5. Disclaimer.

EXCEPT AS EXPRESSLY STATED IN THIS LICENCE OR OTHERWISE MUTUALLY AGREED TO BY TH

6. Limitation on Liability.

TO THE FULL EXTENT PERMITTED BY APPLICABLE LAW, AND EXCEPT FOR ANY LIABILITY ARI

If applicable legislation implies warranties or conditions, or imposes obligatio

   1. in the case of goods, any one or more of the following:
         1. the replacement of the goods or the supply of equivalent goods;
         2. the repair of the goods;
         3. the payment of the cost of replacing the goods or of acquiring equiv
         4. the payment of the cost of having the goods repaired; or
   2. in the case of services:
         1. the supplying of the services again; or
         2. the payment of the cost of having the services supplied again.

7. Termination.

   1. This Licence and the rights granted hereunder will terminate automatically
   2. Subject to the above terms and conditions, the licence granted here is pe

8. Miscellaneous.

   1. Each time You publish, communicate to the public, distribute or publicly d
   2. If any provision of this Licence is invalid or unenforceable under applica
   3. No term or provision of this Licence shall be deemed waived and no breach
   4. This Licence constitutes the entire agreement between the parties with res
   5. The construction, validity and performance of this Licence shall be goverr