J. Functional Programming 1 (1): 1–000, January 1993

FUNCTIONAL PEARLS

Monadic Parsing in Haskell

Graham Hutton
University of Nottingham

Erik Meijer
University of Utrecht

1 Introduction

This paper is a tutorial on defining recursive descent parsers in Haskell. In the spirit of *one-stop shopping*, the paper combines material from three areas into a single source. The three areas are functional parsers (Burge, 1975; Wadler, 1985; Hutton, 1992; Fokker, 1995), the use of monads to structure functional programs (Wadler, 1990; Wadler, 1992a; Wadler, 1992b), and the use of special syntax for monadic programs in Haskell (Jones, 1995; Peterson *et al.* , 1996). More specifically, the paper shows how to define monadic parsers using `do` notation in Haskell.

Of course, recursive descent parsers defined by hand lack the efficiency of bottom-up parsers generated by machine (Aho *et al.*, 1986; Mogensen, 1993; Gill & Marlow, 1995). However, for many research applications, a simple recursive descent parser is perfectly sufficient. Moreover, while parser generators typically offer a fixed set of combinators for describing grammars, the method described here is completely extensible: parsers are first-class values, and we have the full power of Haskell available to define new combinators for special applications. The method is also an excellent illustration of the elegance of functional programming.

The paper is targeted at the level of a good undergraduate student who is familiar with Haskell, and has completed a grammars and parsing course. Some knowledge of functional parsers would be useful, but no experience with monads is assumed. A Haskell library derived from the paper is available on the web from:

```
http://www.cs.nott.ac.uk/Department/Staff/gmh/bib.html#pearl
```

2 A type for parsers

We begin by defining a type for parsers:

```
newtype Parser a = Parser (String -> [(a,String)])
```

That is, a parser is a function that takes a string of characters as its argument, and returns a list of results. The convention is that the empty list of results denotes failure of a parser, and that non-empty lists denote success. In the case of success, each result is a pair whose first component is a value of type `a` produced by parsing and processing a prefix of the argument string, and whose second component is the unparsed suffix of the argument string. Returning a list of results allows us to build parsers for ambiguous grammars, with many results being returned if the argument string can be parsed in many different ways.

3 A monad of parsers

The first parser we define is `item`, which successfully consumes the first character if the argument string is non-empty, and fails otherwise:

```
item :: Parser Char
item  = Parser (\cs -> case cs of
                         ""     -> []
                         (c:cs) -> [(c,cs)])
```

Next we define two combinators that reflect the monadic nature of parsers. In Haskell, the notion of a *monad* is captured by a built-in class definition:

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

That is, a type constructor `m` is a member of the class `Monad` if it is equipped with `return` and `(>>=)` functions of the specified types. The type constructor `Parser` can be made into an instance of the `Monad` class as follows:

```
instance Monad Parser where
   return a = Parser (\cs -> [(a,cs)])
   p >>= f  = Parser (\cs -> concat [parse (f a) cs' |
                                (a,cs') <- parse p cs])
```

The parser `return a` succeeds without consuming any of the argument string, and returns the single value `a`. The `(>>=)` operator is a *sequencing* operator for parsers. Using a deconstructor function for parsers defined by `parse (Parser p) = p`, the parser `p >>= f` first applies the parser `p` to the argument string `cs` to give a list of results of the form `(a,cs')`, where `a` is a value and `cs'` is a string. For each such pair, `f a` is a parser which is applied to the string `cs'`. The result is a list of lists, which is then concatenated to give the final list of results.

The `return` and `(>>=)` functions for parsers satisfy some simple laws:

```
           return a >>= f   =   f a
             p >>= return   =   p
 p >>= (\a -> (f a >>= g))  =   (p >>= (\a -> f a)) >>= g
```

In fact, these laws must hold for any monad, not just the special case of parsers. The laws assert that — modulo the fact that the right argument to `(>>=)` involves a binding operation — `return` is a left and right unit for `(>>=)`, and that `(>>=)` is associative. The unit laws allow some parsers to be simplified, and the associativity law allows parentheses to be eliminated in repeated sequencings.

4 The do notation

A typical parser built using `(>>=)` has the following structure:

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

Such a parser has a natural operational reading: apply parser `p1` and call its result value `a1`; then apply parser `p2` and call its result value `a2`; ...; then apply parser `pn` and call its result value `an`; and finally, combine all the results by applying a semantic action `f`. For most parsers, the semantic action will be of the form `return (g a1 a2 ... an)` for some function `g`, but this is not true in general. For example, it may be necessary to parse more of the argument string before a result can be returned, as is the case for the `chainl1` combinator defined later on.

Haskell provides a special syntax for defining parsers of the above shape, allowing them to be expressed in the following, more appealing, form:

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

This notation can also be used on a single line if preferred, by making use of parentheses and semi-colons, in the following manner:

```
do {a1 <- p1; a2 <- p2; ...; an <- pn; f a1 a2 ... an}
```

In fact, the *do notation* in Haskell can be used with any monad, not just parsers. The subexpressions `ai <- pi` are called generators, since they generate values for the variables `ai`. In the special case when we are not interested in the values produced by a generator `ai <- pi`, the generator can be abbreviated simply by `pi`.

Example: a parser that consumes three characters, throws away the second character, and returns the other two as a pair, can be defined as follows:

```
p :: Parser (Char,Char)
p  = do {c <- item; item; d <- item; return (c,d)}
```

5 Choice combinators

We now define two combinators that extend the monadic nature of parsers. In Haskell, the notion of a *monad with a zero*, and a *monad with a zero and a plus* are captured by two built-in class definitions:

```
class Monad m => MonadZero m where
   zero :: m a
```

```
class MonadZero m => MonadPlus m where
   (++) :: m a -> m a -> m a
```

That is, a type constructor `m` is a member of the class `MonadZero` if it is a member of the class `Monad`, and if it is also equipped with a value `zero` of the specified type. In a similar way, the class `MonadPlus` builds upon the class `MonadZero` by adding a `(++)` operation of the specified type. The type constructor `Parser` can be made into instances of these two classes as follows:

```
instance MonadZero Parser where
   zero   = Parser (\cs -> [])

instance MonadPlus Parser where
   p ++ q = Parser (\cs -> parse p cs ++ parse q cs)
```

The parser `zero` fails for all argument strings, returning no results. The `(++)` operator is a (non-deterministic) *choice* operator for parsers. The parser `p ++ q` applies both parsers `p` and `q` to the argument string, and appends their list of results.

The `zero` and `(++)` operations for parsers satisfy some simple laws:

```
      zero ++ p   =  p
      p ++ zero   =  p
  p ++ (q ++ r)   =  (p ++ q) ++ r
```

These laws must in fact hold for any monad with a zero and a plus. The laws assert that `zero` is a left and right unit for `(++)`, and that `(++)` is associative. For the special case of parsers, it can also be shown that — modulo the binding involved with `(>>=)` — `zero` is the left and right zero for `(>>=)`, that `(>>=)` distributes through `(++)` on the right, and (provided we ignore the order of results returned by parsers) that `(>>=)` also distributes through `(++)` on the left:

```
             zero >>= f   =  zero
        p >>= const zero  =  zero
          (p ++ q) >>= f  =  (p >>= f) ++ (q >>= f)
p >>= (\a -> f a ++ g a)  =  (p >>= f) ++ (p >>= g)
```

The zero laws allow some parsers to be simplified, and the distribution laws allow the efficiency of some parsers to be improved.

Parsers built using `(++)` return many results if the argument string can be parsed in many different ways. In practice, we are normally only interested in the first result. For this reason, we define a (deterministic) choice operator `(+++)` that has the same behaviour as `(++)`, except that at most one result is returned:

```
(+++)  :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p ++ q) cs of
                             []     -> []
                             (x:xs) -> [x])
```

All the laws given above for `(++)` also hold for `(+++)`. Moreover, for the case of `(+++)`, the precondition of the left distribution law is automatically satisfied.

The `item` parser consumes single characters unconditionally. To allow conditional parsing, we define a combinator `sat` that takes a predicate, and yields a parser that consumes a single character if it satisfies the predicate, and fails otherwise:

```
sat  :: (Char -> Bool) -> Parser Char
sat p = do {c <- item; if p c then return c else zero}
```

Example: a parser for specific characters can be defined as follows:

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

In a similar way, by supplying suitable predicates to `sat`, we can define parsers for digits, lower-case letters, upper-case letters, and so on.

6 Recursion combinators

A number of useful parser combinators can be defined recursively. Most of these combinators can in fact be defined for arbitrary monads with a zero and a plus, but for clarity they are defined below for the special case of parsers.

- Parse a specific string:

```
string       :: String -> Parser String
string ""     = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

- Parse repeated applications of a parser `p`; the `many` combinator permits zero or more applications of `p`, while `many1` permits one or more:

```
many   :: Parser a -> Parser [a]
many p  = many1 p +++ return []

many1  :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```

- Parse repeated applications of a parser `p`, separated by applications of a parser `sep` whose result values are thrown away:

```
sepby          :: Parser a -> Parser b -> Parser [a]
p `sepby` sep   = (p `sepby1` sep) +++ return []

sepby1         :: Parser a -> Parser b -> Parser [a]
p `sepby1` sep = do a  <- p
                    as <- many (do {sep; p})
                    return (a:as)
```

- Parse repeated applications of a parser `p`, separated by applications of a parser `op` whose result value is an operator that is assumed to associate to the left, and which is used to combine the results from the `p` parsers:

```
chainl  :: Parser a -> Parser (a -> a -> a) -> a -> Parser a
chainl p op a  = (p `chainl1` op) +++ return a

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = do {a <- p; rest a}
                 where
                    rest a = (do f <- op
                                 b <- p
                                 rest (f a b))
                             +++ return a
```

Combinators `chainr` and `chainr1` that assume the parsed operators associate to the right can be defined in a similar manner.

7 Lexical combinators

Traditionally, parsing is usually preceded by a lexical phase that transforms the argument string into a sequence of tokens. However, the lexical phase can be avoided by defining suitable combinators. In this section we define combinators to handle the use of space between tokens in the argument string. Combinators to handle other lexical issues such as comments and keywords can easily be defined too.

- Parse a string of spaces, tabs, and newlines:

```
space :: Parser String
space  = many (sat isSpace)
```

- Parse a token using a parser `p`, throwing away any *trailing* space:

```
token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}
```

- Parse a symbolic token:

```
symb   :: String -> Parser String
symb cs = token (string cs)
```

- Apply a parser `p`, throwing away any *leading* space:

```
apply  :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p})
```

8 Example

We illustrate the combinators defined in this article with a simple example. Consider the standard grammar for arithmetic expressions built up from single digits using the operators `+`, `-`, `*` and `/`, together with parentheses (Aho *et al.*, 1986):

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{expr}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\
\mathit{term} & ::= & \mathit{term}\ \mathit{mulop}\ \mathit{factor} \mid \mathit{factor} \\
\mathit{factor} & ::= & \mathit{digit} \mid (\,\mathit{expr}\,) \\
\mathit{digit} & ::= & \mathtt{0} \mid \mathtt{1} \mid \ldots \mid \mathtt{9} \\
\\
\mathit{addop} & ::= & \mathtt{+} \mid \mathtt{-} \\
\mathit{mulop} & ::= & \mathtt{*} \mid \mathtt{/}
\end{array}
$$

Using the `chainl1` combinator to implement the left-recursive production rules for *expr* and *term*, this grammar can be directly translated into a Haskell program that parses expressions and evaluates them to their integer value:

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr   = term   `chainl1` addop
term   = factor `chainl1` mulop
factor = digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit  = do {x <- token (sat isDigit); return (ord x - ord '0')}

addop = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}
```

For example, evaluating `apply expr " 1 - 2 * 3 + 4 "` gives the singleton list of results `[(-1,"")]`, which is the desired behaviour.

9 Acknowledgements

Thanks for due to Luc Duponcheel, Benedict Gaster, Mark P. Jones, Colin Taylor, and Philip Wadler for their useful comments on the many drafts of this article.

References

Aho, A., Sethi, R., & Ullman, J. (1986). *Compilers — principles, techniques and tools.* Addison-Wesley.

Burge, W.H. (1975). *Recursive programming techniques.* Addison-Wesley.

Fokker, Jeroen. 1995 (May). Functional parsers. *Lecture notes of the Baastad Spring school on functional programming.*

Gill, Andy, & Marlow, Simon. 1995 (Jan.). *Happy: the parser generator for Haskell.* University of Glasgow.

Hutton, Graham. (1992). Higher-order functions for parsing. *Journal of functional programming*, **2**(3), 323–343.

Jones, Mark P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of functional programming*, **5**(1), 1–35.

Mogensen, Torben. (1993). *Ratatosk: a parser generator and scanner generator for Gofer.* University of Copenhagen (DIKU).

Peterson, John, *et al.* . 1996 (May). *The Haskell language report, version 1.3.* Research Report YALEU/DCS/RR-1106. Yale University.

Wadler, Philip. (1985). How to replace failure by a list of successes. *Proc. conference on functional programming and computer architecture.* Springer–Verlag.

Wadler, Philip. (1990). Comprehending monads. *Proc. ACM conference on Lisp and functional programming.*

Wadler, Philip. (1992a). The essence of functional programming. *Proc. principles of programming languages.*

Wadler, Philip. (1992b). Monads for functional programming. Broy, Manfred (ed), *Proc. Marktoberdorf Summer school on program design calculi.* Springer–Verlag.