

# FUNCTIONAL PEARL

## *Monadic parsing in Haskell*

GRAHAM HUTTON

*University of Nottingham, Nottingham, UK*

ERIK MEIJER

*University of Utrecht, Utrecht, The Netherlands*

### 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, 1992a, 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 and 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:

$$\begin{aligned} \text{return } a \gg= f &= f \ a \\ p \gg= \text{return} &= p \\ p \gg= (\lambda a \rightarrow (f \ a \gg= g)) &= (p \gg= (\lambda a \rightarrow f \ a)) \gg= g \end{aligned}$$

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 `chain11` 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
```