Parsers

A parser is a program which analyses a text and determines it’s structure. For example,

$$4 + 3 \times 9$$

We usually say that a parser takes a token list to a syntax tree. Almost every (non trivial) program does some form of parsing.

Parsers as functions

Functionally, we can think of a parser as being a function from a token list to a result type:

```haskell
type Parser = String \to Tree
```

Realistically, we should allow the parser to consume part of it’s input and return the unused portion:

```haskell
type Parser = String \to (Tree, String)
```

Parsers as functions

We should parameterise the type of the parser; fixing the output type to Tree is not very general:

```haskell
type Parser result = String \to (result, String)
```

In fact, we can generalise further by saying that the input does not really have to be a string.

```haskell
type Parser token result = [token] \to (result, [token])
```

What about parsers that fail? Or ambiguous grammars?

```haskell
type Parser s a = [s] \to [(a, [s])]
```

We can say that a parser that fails will return an empty list as it’s result.
Some basic parsers

How about a parser that always succeeds in it’s parse attempt (and gives some constant value):

\[
pSucceed :: a \to \text{Parser } s \to s \to [(v, cs)] \]

In use:

*Main> pSucceed 1 "abc"
[(1,"abc")]

Some basic parsers

And a parser that always fails, no matter what the input is:

\[
pFail :: \text{Parser } s \to s \to [] \]

In use:

*Main> pFail "abc"
[]

Some basic parsers

A parser which consumes one token of input and always succeeds in that:

\[
\begin{align*}
\text{item } [] &= [] \\
\text{item } (c:cs) &= [(c,cs)]
\end{align*}
\]

in use:

*Main> item "abc"
[('a','bc')]

Some basic parsers

\[
pSymbol :: \text{Eq } s \Rightarrow s \to \text{Parser } s \to s \to [(v, bs)] \\
pSymbol a (b:bs) | a == b = [(b,bs)] \\
| \text{otherwise } = []
\]

in use:

*Main> pSymbol 'a' "abc"
[('a','bc')] 
*Main> pSymbol 'b' "abc"
[]
Some basic parsers

pToken :: Eq s => [s] -> Parser s [s]
pToken a cs | a == (take n cs) = [(a, drop n cs)]
    | otherwise = []
    where n = length a

in use:
*Main> pToken "abc" "abcde"
["abc","de"]
*Main> pToken "abc" "frobulate"
[]

Our first combinator

The *combinators* are going to allow us to join these trivial parsers together to form some non-trivial parsers.

First up: A combinator that applies one parser, and if that should fail applies a second parser instead:

\texttt{(p \mid> q)} :: \texttt{Parser s a -> Parser s a -> Parser s a}

\texttt{(p \mid> q) input = case p input of}
\texttt{[] \rightarrow q input}
\texttt{anything \rightarrow anything}

This combinator represents the notion of *choice*

The choice combinator in action

*Main> (item \mid> (pSucceed '1')) "abc"
[('a','bc')]
*Main> (item \mid> (pSucceed '1')) ""

*Main> (pFail \mid> (pSucceed 1)) []
[1,[]]

*Main> (pFail \mid> (pSucceed 1)) "abc"
[(1,"abc")]

Sidebar: alternative choice combinator

Sidebar

Just for interest, some parser combinator libraries allow us to generate *all* possible parses, not just the first one that matches.

\texttt{(p \mid> q)} :: \texttt{Parser s a -> Parser s a -> Parser s a}

\texttt{(p \mid> q) input = p input ++ q input}

This definition produces results like this:

*Main> (item \mid> (pSucceed '1')) "abc"
[('a','bc'),('1','abc')]

For simplicity we will use the single-result version of \texttt{\mid>} today, but bear this idea in mind for later (when we discuss error recovering parsers).

End of sidebar
Another (perhaps the other) interesting combinator is one which combines two parsers together to produce a joint result:

\[(\&\&\rightarrow)\text{ :: Parser s a }\rightarrow (a \rightarrow \text{Parser s b}) \rightarrow \text{Parser s b}\]

\[(p \&\& q) \text{ input } = \text{concat } [ q v \text{ input' } | (v,\text{input'}) \leftarrow p \text{ input } ]\]

Some more examples of composition

*Main> ( pToken "ab" \&\& \_ \_ \rightarrow item ) "abc" 
[['c',""]]

*Main> ( pToken "ab" \&\& \_ \_ \rightarrow item ) "xbc"
[]

Composition combinator in action

This combinator is slightly less convenient to use, since we usually want to combine the results of the parsers (this is reflected in the type).

*Main> ( item \&\& \\_ \_ \rightarrow item \&\& \_ \_ \rightarrow pSucceed (c1,c2) ) "abc" 
[['('a','b'),"c"]]

We need to embed functions that gather up the results. This might be easier to read if we lay it out vertically a bit:

\[
(\text{item } \&\& \_ \_ \rightarrow \\
\quad \text{item } \&\& \_ \_ \rightarrow \\
\quad \text{pSucceed } (c1,c2) ) \text{ "abc"}
\]

Predicate-satisfying parser

From now on we can write our parsers in terms of these combinators:

\[\text{pSatisfy :: Eq s }\rightarrow (s \rightarrow \text{Bool}) \rightarrow \text{Parser s s}\]

\[\text{pSatisfy f } = \text{item } \&\& \_ \rightarrow \text{if f c then pSucceed c else pFail}\]

\[\text{pDigit } = \text{pSatisfy isDigit}\]

in use:

*Main> pDigit "123" 
[[''1','23']]

*Main> pDigit "abc" 
[]
Repetition combinators

Combinators that repeat some parser zero or more times (or one or more times):

\[ pMany :: Parser s a -> Parser s [a] \]
\[ pMany p = pMany1 p <|> pSucceed [] \]
\[ pMany1 p = p <&> (\a -> pMany p <&> \b -> pSucceed (a:b)) \]

A digit list parser

We can use this to write a parser for digit lists:

\[ pDigitList = pSymbol '[' <&> \_ -> \]
\[ pDigit <&> \d -> \]
\[ pMany ( pSymbol ',' <&> \_ -> \]
\[ pDigit ) <&> \ds -> \]
\[ pSymbol ']' <&> \_ -> \]
\[ pSucceed (d:ds) \]

*Main> pDigitList "[1,2,3]"
["123"]
*Main> pDigitList "[1,a,3]"
[]

A suggestive correspondence

Look at the type of <&>:

\[ (<&>) :: Parser s a -> (a -> Parser s b) -> Parser s b \]

It reminds us of

\[ (>>=) :: (Monad m) => m a -> (a -> m b) -> m b \]

In fact, we would like to write:

\[
\text{instance Monad Parser where} \\
\quad \text{return} = pSucceed \\
\quad \text{fail} = pFail \\
\quad (>>=) = (<&>)
\]

We will need to modify our parser types for this (but not much).

The new parser type

We need the Parser type to be a data type so that we can create an instance of Monad

\[
data Parser s a = P ([s] -> [(a,[s])])
\]

Therefore we need to modify all the parsers that are aware of the implementation type. For simplicity we will minimise the number of parsers that actually touch the representation:

\[
pSucceed v = P (\cs -> [(v,cs)]) \\
pFail = P (\cs -> []) \\
item = P (\cs -> if (null cs) then [] else [(head cs, tail cs)])
\]
Basic combinators

The basic combinators also need some minor changes.

\[
\text{parse :: Parser } s \text{ a } \rightarrow [s] \rightarrow [(a,[s])] \\
\text{parse } (P \ p) \ \text{inp } = \ p \ \text{inp}
\]

\[
(<|>) :: \text{Parser } s \text{ a } \rightarrow \text{Parser } s \text{ a } \rightarrow \text{Parser } s \text{ a} \\
p <|> q = P (\text{\input } \rightarrow \text{ case } (\text{parse } p \ \text{input}) \ \text{of} \\
[{}] \rightarrow \text{parse } q \ \text{input} \\
\text{anything} \rightarrow \text{anything})
\]

\[
(<&>) :: \text{Parser } s \text{ a } \rightarrow (\text{a } \rightarrow \text{Parser } s \text{ b}) \rightarrow \text{Parser } s \text{ b} \\
p <&> q = P (\text{\input } \rightarrow \\
\text{concat } [\ \text{parse } (q \ v) \ \text{input'} | (v,\text{input'}) \leftarrow \text{parse } p \ \text{input}])
\]

The Class instance

This will do it. Now we can create our class instance:

\[
\text{instance Monad (Parser } s \text{) where} \\
\text{return } = \text{pSucceed} \\
\text{fail } s = \text{pFail} \\
(>>=) = (<&>)
\]

Note how the fail function takes a parameter which we have to ignore (we’ve got nothing that we can do with it).

Some example Monadic parsers

Now that we have an instance of Monad we can create some parsers:

\[
\text{pSymbol } s \\
\quad = \text{do} \\
\quad \quad c \leftarrow \text{item} \\
\quad \quad \text{if } (c==s) \ \text{then return } s \\
\quad \quad \quad \text{else fail } (\text{"expected to see "}++,[c])
\]

\[
\text{pSatisfy :: Eq } s \Rightarrow (s \rightarrow \text{Bool}) \rightarrow \text{Parser } s \text{ s} \\
\text{pSatisfy } f = \text{do} \\
\quad c \leftarrow \text{item} \\
\quad \text{if } f \ c \ \text{then pSucceed } c \\
\quad \quad \text{else fail } ("")
\]

\[
\text{pMany :: Parser } s \text{ a } \rightarrow \text{Parser } s \text{ } [a] \\
\text{pMany } p = \text{pMany1 } p <\mid> \text{ pSucceed } [] \\
\text{pMany1 } p = \text{do} \\
\quad a \leftarrow p \\
\quad b \leftarrow \text{pMany } p \\
\quad \text{return } (a:b)
\]

\[
\text{pDigitList } = \text{do} \\
\quad \text{pSymbol } [''] \\
\quad d \leftarrow \text{pDigit} \\
\quad ds \leftarrow \text{pMany } (\text{do } \text{pSymbol } ',', \gg \text{pDigit }) \\
\quad \text{pSymbol } [',] \\
\quad \text{pSucceed } (d:ds)
\]
A Monad specialization

Something that has come up more than once is the idea of a monad which can return several results. We saw it in `Maybe`, and again in `List`. Now we are seeing in `Parser` as well.

This class captures the idea of combining the values of such a monad (so that we get all the results):

```haskell
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

For a list, for example,

```haskell
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

Other useful(?) instances

```haskell
instance MonadPlus (Parser s) where
    mzero = P (\s -> [])
    mplus = (<|>)
```

This gives us a structure in which to hide the choice combinator. We can then write code such as:

```haskell
pMany p = do pMany1 p ‘mplus‘ pSucceed []
```