Monadic Parsing: A Case Study

Bernard Pope, Simon Taylor and Mark Wielaard
Department of Computer Science
The University of Melbourne
Parkville, Vic. 3052, Australia

Abstract

One of the selling points for functional languages is the ease with which parsers for simple languages can be expressed. In this report we give an introduction to monadic and operator precedence parsing in functional languages, using the parsing of propositional logic formulas as an example.

1 Introduction

A classical showcase for higher-order functional programming is *functional parsing*. The idea, which goes back at least to Burge [2], is that powerful parsers can be built simply by combining simple parsers (essentially functions from input strings to output tokens) by higher-order functions, so-called *parser combinators*, whose job it is to build complex structure trees from simpler trees. This approach has been followed up by Hutton [5], Fokker [4] and others.

Recently the so-called *monadic* programming style has become popular in functional programming, and monadic parsing has appeared as a useful special case [6]. The Hugs distribution [7] includes a "parser library" which includes support for monadic style parsing.

In this essay we describe methods for parsing in functional languages. Section 2 provides a gentle introduction to monadic parsing. Section 3 discusses the case of operator precedence parsing. In Section 4 we go through the details of a non-trivial example, namely the translation between modern notation for propositional logic and that used in *Principia Mathematica* [12].

2 Functional Parsing and Monads

2.1 Functional parsing: an overview

Functional programming languages provide an elegant method for building complex parsers out of simpler parsers. Elementary parsers for the primitives of a language (for example, the terminal symbols of a BNF grammar) can be combined together by higher order functions (called *parser combinators*) to form more powerful composite parsers which can in turn be combined to form a complete parser for a given language. This method of creating functional parsers is well known, and Fokker [4] provides a thorough exposition. The essence of parser combinators is their ability to provide mechanisms for combining a number of parsers in sequence or alternation. This is particularly useful if we view parsing as the recursive application of BNF grammar rules in a top-down, left-to-right manner (left-to-right, recursive descent parsing).

What does a functional parser look like?[1] In our definition, a functional parser is a function which takes a list of tokens as input and returns either a parse-result (if the parse partially succeeds) or a failure-flag (if the parse fails). A parse-result is a pair (*value*, *tokens-left*), such that *value* represents

[1] We will restrict ourselves to deterministic parsers in this paper, although non-deterministic parsers (those that parse an input in a number of possible ways) are possible, see [6].

$$
\begin{array}{lcl}
exp & ::= & term\ op\ term \parallel term \\
term & ::= & signed\ term \parallel signed \\
signed & ::= & basic \parallel \text{`-'}\ basic \\
basic & ::= & var \parallel \text{`('}\ exp\ \text{`)'} \\
op & ::= & \text{`|'} \parallel \text{`->'} \parallel \text{`<->'}
\end{array}
$$

Figure 1: BNF grammar for propositional logic

the structure of the tokens seen so far (typically this will be a parse tree), and *tokens-left* is the list of unparsed tokens from the input. A failure-flag is a unique return value that indicates that the parser could not parse any of the input. The relevance of the failure-flag will be discussed below.

2.2 A high-level notation for functional parsing

Using the notation that '$\oplus$' represents a sequencing operator, and '$\cup$' represents an alternation operator, a high-level design of a functional parser can be given. Consider two parsers, α and β, and a list of input tokens κ. We can specify the sequential application of α and β on κ by the statement '$(\alpha \oplus \beta)\ \kappa$'. This is the application of α on κ, followed by the application of β on $\overline{\kappa}$, such that $\overline{\kappa}$ is the list of tokens remaining after α has consumed a (possibly zero) number of tokens from κ. In other words, the $\oplus$ operator allows multiple parsers to be applied in sequence, such that each successive parser is applied to the tokens which remain after the previous parser has finished. The statement '$(\alpha \cup \beta)\ \kappa$' represents the application of α on κ, or, if α fails to parse any tokens, the application of β on κ. Another way of understanding the '$\cup$' operator is that it allows for an alternative parser to be applied to the input, if the preceding parser fails.

There is one important detail that has been overlooked thus far: parsers return a parse-result. The notation does not appear to allow for such results to be manipulated. This is true, however, we are only looking for a high-level description of functional parsers. Thus, we will assume that the results of parsers will be implicitly available for manipulation. When two parsers are sequenced, we have said that the second parser has access to the *tokens-left* from the first parser. We will also assume that the second parser has access to the *value* constructed by the first parser, and that the *value* s of all parsers in a sequence may be combined together to form a new composite *value* , if so required.

Let us now look at a simple example. A BNF grammar is defined in figure 1 for sentences in propositional logic. The terminal symbols of this grammar are the binary operators defined by the rule *op*, parentheses, and variables (*var*, a non-empty string of alphabetic characters). Conjunction is represented by the sequencing of the terms, and conjunction binds tighter than any of the binary connectives, so for example '`p q | r`' means '$(p \wedge q) \vee r$'.

Using our high-level notation, the specification for an expression (*exp*) parser is simple. If *termP* is a parser for terms and *opP* is a parser for operators then *expP* (a parser for expressions) is defined as: $expP = (termP \oplus opP \oplus termP) \cup termP$. This can be read as: an expression parser is a term parser followed by an operator parser followed by a term parser, or it is just a term parser. A high-level description of the rest of the parser follows: [2]

$$
\begin{array}{lcl}
termP & = & (signedP \oplus termP) \cup signedP \\
signedP & = & (basicP) \cup (notP \oplus signedP) \\
basicP & = & (varP) \cup (bracketP \oplus expP \oplus bracketP)
\end{array}
$$

[2]Note that *notP*, *bracketP*, and *varP* are elementary parsers for the negation symbol, parentheses, and variables respectively.

```
> data Parser a = P (String -> Maybe (a, String))
>
> instance Monad Parser where
>   -- return            :: a -> Parser a
>   return v             = P (\inp -> Just (v, inp))
>
>   -- >>=               :: Parser a -> (a -> Parser b) -> Parser b
>   (P p) >>= f          = P (\inp -> case p inp of
>                                       Just(v, out) -> (papply (f v) out)
>                                       Nothing -> Nothing)
>   -- end of Monad instance definition
>
>   papply               :: Parser a -> String -> Maybe (a, String)
>   papply (P p) inp     = p inp
```

Figure 2: A determinsitic parser as a state monad

2.3 A *brief* introduction to monads, in reference to parsing

What does monadic-style programming offer the field of functional parsing? Firstly, the monadic operators *bind* and *plus* provide useful mechanisms for achieving the sequencing and alternation of parsers respectively. Secondly, the *do notation* provided by Haskell provides a very readable method for expressing the sequential application of parsers, and hides the plumbing of state information between each parser. Functional parsers written in the monadic-style have the *look and feel* of an imperative recursive descent parser, without any comprimise to the purity of the implementation language. In summary, monads allow us to impose an order of evaluation and handle state manipulation, in a pure functional language. Furthermore, with the requisite syntatic sugar added, all of this can be written in an intuitive syntax.

Readers unfamiliar with the technical details of monads (in functional languages) are referred to the following articles [6, 9, 11, 8, 3, 10]

Determinsitic parsers (those that produce exactly one parse-result) are in effect instances of the state monad, as described in Jones [8]. A Haskell implementation of a determinsitic parser monad is given in figure 2. It is similar to Jones' state monad, except that it is renamed to Parser to indicate a narrower meaning in the context of parsing.

Using the nomenclature introduced earlier in this section, an object of type 'Parser a', is a mapping from a string to either a parse-result, or to a failure-flag. Note the use of the *Maybe* type constructor within the *Parser* type constructor. This allows a Parser to return either '`Just(val, toks_left)`' (a parse-result) or '`Nothing`' (a failure-flag indicating that the parser failed to parse any input). The function `papply` simply allows us to get inside the parser datatype and apply it to some input.

Note that Haskell uses the symbol '`>>=`' to reresent the monadic bind operator. There is a strong (and intentional) connection between the bind operator and the $\oplus$ operator given above in the high-level description of functional parsers. This connection will be explored in the next sub-section of the paper where we introduce the do-notation of Haskell.

2.4 The connection between *do notation* and the *monadic bind*

The overall intention of this sub-section is to present to the reader the connection between the low level implementation of monadic bind, the medium level syntax of the do notation, and the high-level notation of the $\oplus$ operator. Recall, that in our high-level defintion of functinal parsers, to sequence two parsers, α and β, we used the notation '$(\alpha \ \oplus \ \beta)$'. In Haskell, using the do-notation, we write the sequencing

of two parsers, `alphaP` and `betaP`, as '`do {alphaP; betaP}`'. Recall also, that we have defined parsers to be functions that return either a failure-flag, or a parse-result. Our high-level notation required that the results of a parser are implicitly available to all other parsers following in sequence. The do-notation of Haskell accomodates the manipulation of parse-results in two ways. The primary concern of the sequencing operator is that the *tokens-left* from the first parser are given as input to the second parser. This is where the do-notation makes use of the bind operator. Syntactic and semantic defintions for the do-notation are given in [3]. If we look inside the implementation of the do-notation, it is easy to see how the *tokens-left* are threaded between parsers in sequence:[3]

```
do{e}       =  e

do{e; es}   =  e >>= \_ -> do{es}
```

In the above recursive definition of (part of) the do-notation, `e` represents an expression, and `es` represents a list of expressions separated by semicolons. It is useful to think of the application of a parser to some input as an expression. Thus we can view '`do {alphaP; betaP}`' as '`alphaP >>= \_ -> betaP`', which can be read as: 'bind `alphaP` to a function which ignores its input and returns `betaP`'. Looking back to our definition of `>>=` (the monadic bind operator), we can see that the above do-sentence will apply `alphaP` to its input tokens to produce a parse-result, the *value* of the parse-result produced by `alphaP` will be ignored, and `betaP` will be applied to the *tokens-left* produced by `alphaP`.

The secondary concern of the sequencing operator is what should happen if we do not want to ignore the *value* produced by a parser? There is a further rule that we must add to the above definition of the do-notation to solve this problem:

```
do{pat <- e; es}  =  e >>= f where f pat = do{es}; f _ = zero
```

This extra rule allows us to manipulate the *value* produced by a parser applied in a sequence of parsers. We can now write the following Haskell code: '`do {val <- alphaP; betaP val}`'. In this case the *value* produced by `alphaP` is made available to `betaP` through the variable `val` (a Haskell pattern), which allows pattern matching on the *value* produced. If the pattern fails to match, then a default defintion of the function `f` is given which returns `zero` (an empty parse, or failed parse) and no further sequencing is performed. This has been a brief description of the do-notation of Haskell, however, we can see that we now have a concrete mechanism for implementing the high-level operator $\oplus$.

2.5 Achieving alternation in our monadic parser

We will now turn our attention to achieving alternation for our example monadic parser, and thus implement the high-level operator $\cup$. For determinsitic parsers, applying two parsers in alternation essentially requires that the first parser be applied to the input, if it fails to parse any of the input then the second parser should be applied to the input. Therefore, we need some manner for representing the failure of a parser. To indicate that a parser has failed to parse any input, we allow it to return the value `Nothing`. In monadic jargon, we are giving our parser monad a zero, and thus can make it an instance of the MonadZero type class:

```
> instance MonadZero Parser where
>    -- zero              :: Parser a
>    zero                 = P (\inp -> Nothing)
```

Thus, zero for our parser monad, is a parser that takes some input, and returns a failure-flag '`Nothing`'. This is the same zero mentioned in the definition of the do-notation of Haskell.[4] The monadic equivalent

[3] This description of the do-notation is derived from that given in [3].

[4] In order for do-notation to be possible for a monad it must have a zero, and so a zero is important for both alternation and sequencing.

$$
\begin{array}{lll}
< expr > ::= & < expr > \wedge < expr > & \| < expr > \vee < expr > \| \\
& < expr > \rightarrow < expr > & \| < expr > \leftrightarrow < expr > \| \\
& \neg < expr > \| (< expr >) \| \textbf{var} &
\end{array}
$$

Figure 3: Operator grammar for propositional logic.

of the high-level $\cup$ operator is the 'plus' operator, which in Haskell is represented by the symbol `++`. To implement alternation for our parser monad in Haskell, we must make the monad an instance of the MonadPlus type class, and define the `++` operator:

```
> instance MonadPlus Parser where
>   -- ++                 :: Parser a -> Parser a -> Parser a
>   (P p) ++ (P q)        = P (\inp -> case p inp of
>                                        Just (v, inp') -> Just (v, inp')
>                                        Nothing -> q inp)
```

The definition for the `++` operator applies the first parser to the input, if it returns a parser result, then the whole expression should return that parse result, however, if it fails, then the second parser should be applied to the input. It can be seen that we also have a concrete mechanism for representing the high-level operator $\cup$.

2.6 Some efficiency considerations

Now that we have achieved sequencing and alternation for our parser monad in Haskell, it should be straightforward to translate the high-level parser description (given earlier) into Haskell code. Examination of the grammar in figure 1, will reveal that the grammar is not LL1. The direct translation of the high-level parser into Haskell would result in a correct parser. However, in many circumstances a degree of backtracking (through alternation) would occur, making the parser inefficient. Where possible, this inefficiency can be avoided by converting the grammar of the langauge into an equivalent LL1 grammar.

3 Operator precedence parsing

3.1 Operator precedence and relations

Given a language where different operators have different precedences one could describe a context free grammar for such a language by creating nonterminals for the different levels of precedence. This is what we did in figure 1. <*exp*> uses <*term*>, which uses <*signed*>, which uses <*basic*>. We can easily see that brackets have higher precedence than minus, which has higher precedence than and, which has higher precedence than the other operators. The other operators have the same precedence since they are mentioned in the same rule.

As described in [1] an operator precedence grammar is a context-free grammar where no production right side is empty or has two adjacent non-terminals.[5] Just as it is more convenient to write a regular expression for a regular language. It is sometimes easier to describe a language with just operator precedences, especially when we have a lot of different precedences. (We will see an example of this in the next section.)

To make the grammar in figure 1 into an operator precedence grammar we could introduce a symbol $\wedge$ and write it as in figure 3. Then we would have to give the precedences of the operators.

[5] Notice that we use a simplification of the concept described in [1].

	$\neg$	$\wedge$	$\vee$	$\rightarrow$	$\leftrightarrow$	(	)
$\neg$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$
$\wedge$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\lessdot$
$\vee$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
$\rightarrow$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
$\leftrightarrow$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$

Figure 4: Operator precedence relations for the propositional grammar.
(Operators at the side are to the left, operators on top are to the right.)

We can use three precedence relations, $\lessdot$, $\doteq$ and $\gtrdot$ to define the precedence of the different operators. Where $\boldsymbol{a} \lessdot \boldsymbol{b}$ means "$\boldsymbol{a}$ yields precedence to $\boldsymbol{b}$", $\boldsymbol{a} \doteq \boldsymbol{b}$ means "$\boldsymbol{a}$ has the same precedence as $\boldsymbol{b}$" and $\boldsymbol{a} \gtrdot \boldsymbol{b}$ means "$\boldsymbol{a}$ takes precedence over $\boldsymbol{b}$". We will assume that only one of these relations holds between two operators, but it may be that both $\boldsymbol{a} \lessdot \boldsymbol{b}$ and $\boldsymbol{b} \lessdot \boldsymbol{a}$ holds, which means that the operator to the left of the other has always higher precedence.

The intuitive way of determining what precedence relations should hold between a pair of terminals is by looking at the traditional notions of associativity and precedence of the operators. One can think of the precedence relation holding between two operators in an expression as which operator must evaluate first. We will give an example of this way of thinking about the relations after we have defined the precedence relations for our example grammar.

In our example the operator $\wedge$ should have higher precedence than the operator $\vee$. So we make $\vee \lessdot \wedge$ and $\wedge \gtrdot \vee$.

To show that an operator is associative to the left we make $\boldsymbol{a} \gtrdot \boldsymbol{a}$, and if an operator is associative to the right we make $\boldsymbol{a} \lessdot \boldsymbol{a}$. So in our example all binary operators have $\boldsymbol{op} \gtrdot \boldsymbol{op}$, but also $\vee \gtrdot \rightarrow$ and $\rightarrow \gtrdot \vee$, this expresses that $\vee$ and $\rightarrow$ have the 'same' precedence, but the one on the left binds stronger than the one on the right.

The brackets are a bit more subtle. We want to say that operators between brackets have higher precedence than operators outside the brackets. We can do this by defining the following relations for the operators with respect to '(' and ')' :

$$\begin{array}{ll} \boldsymbol{a} \lessdot (& (\lessdot \boldsymbol{a} \\ ) \gtrdot \boldsymbol{a} & \boldsymbol{a} \gtrdot) \\ (\lessdot (&) \gtrdot) \end{array}$$
$$(\doteq)$$

Symbols (non operators) can be viewed as operators which immediately evaluate. So our parser algorithm will treat them in a special way. (One could also define for every operator -including brackets- $\mathbf{var} \gtrdot \boldsymbol{a}$ and $\boldsymbol{a} \lessdot \mathbf{var}$ to express this.) Figure 3.1 shows all the relations for the propositional grammar.

Lets see what all this means for a simple example. If we have the following expression:

$$\neg\ \mathbf{q}\ \vee\ \mathbf{r}\ \wedge\ (\ \mathbf{p}\ \leftrightarrow\ \mathbf{q}\ )$$

We will assume that the **var** symbols immediately evaluate. If we now show only the relations between the normal operators we get:

$$\neg\ \gtrdot\ \vee\ \lessdot\ \wedge\ \lessdot\ (\ \lessdot\ \leftrightarrow\ \gtrdot\ )$$

If we read this from left to right it says: First evaluate the $\neg$, before you evaluate the $\vee$, then wait with evaluating the $\vee$ until we have evaluated the $\wedge$ and the $\wedge$ should wait for the evaluation of the

(and the (should wait for the evaluation of the $\leftrightarrow$, this $\leftrightarrow$ must evaluate before the). After the $\leftrightarrow$ has evaluated we have the following situation (again replacing evaluated expressions with the relations holding between the operators left):

$$\vee \lessdot \wedge \lessdot (\doteq)$$

Which means that the $\vee$ should evaluate after the $\wedge$ evaluates, which should wait until the (evaluates. The $\doteq$ between the brackets means that they should now evaluate. (Which is just what the $\leftrightarrow$ between the brackets evaluated to.) Now we can evaluate the $\wedge$ and finally we evaluate the $\vee$.

Or if we would put an $\lessdot$ before the whole expression and a $\gtrdot$ after the whole expression (again without the normal symbols), we would get:

$$\lessdot \neg \gtrdot \vee \lessdot \wedge \lessdot (\lessdot \leftrightarrow \gtrdot) \gtrdot$$

Which could be read as a 'normal' bracketed expression. Expressing the intended evaluation order.

3.2 Operator-precedence parsing algorithm

We will now construct an algorithm for parsing operator precedence grammars. As seen in the explanation above the way we can read an operator precedence grammar is from left to right, when we see a symbol we just accept it as is, and when it is an operator we will examine the operator to the right of it. If the operator has a greater or equal precedence ($\lessdot$ or $\doteq$) than the operator to the right of it we have to evaluate the operator, if the operator has a lower precedence ($\lessdot$) than the operator to the right of it. We will have to wait until the operator right from of it evaluates. We can implement this using two stacks, one stack for the symbols and one stack for the 'suspended' operators. Figure 5 shows an imperative way of expressing this algorithm.[6]

4 Example: propositional formulas a la Principia Mathematica

Principia Mathematica [12] introduces a notation for propositional formulas designed to eliminate most parentheses from formulas. The method to accomplish this is to add dots to break an expression into parts, more dots meaning a larger break. A conjunction consists of at least one dot. Dots associated with a conjunction result in less of a break than dots associated with a disjunctive connective such as $\vee$, $\rightarrow$ or $\leftrightarrow$. Negation of anything other than a simple proposition still requires parentheses.

The precedence rules for dotted connectives are as follows:

- Dots for conjunction have precedence left and right over any smaller group of dots until either end of the proposition.
- Dots beside a disjunctive operator have precedence away from the operator over any smaller number of dots next to a disjunctive connective or a smaller or equal number of dots for conjunction.
- Both conjunction and the disjunctive connectives are left associative.

This language was designed for parsing by humans. Formal language description mechanisms had not been discovered in 1927 when *Principia Mathematica* was written. As it happens, the notation can be described using an operator precedence grammar with an infinite number of operators as shown in figure 7.

The interesting part of the precedence table is the handling of the conjunction and disjunction operators. Most operator precedence grammars have a finite number of operators, but here there is no limit to the number of dots that can accompany a conjunction or disjunction.

All that is required to parse this grammar using the operator precedence parser defined in section 3 is to create a function encapsulating figure 7 and define parsers for the operator symbols.

[6] This is a simplification of the algorithm given in [1].

```
symbol_stack = empty
operator_stack = empty

get_symbol;
while ( not symbol = END ) do
begin
        if sym is a symbol then push sym onto symbol_stack
        else if sym is an operator then
        begin
                if ( empty(operator_stack) ) then
                        push(sym, operator_stack)
                else
                begin
                        while ( order(top(operator_stack), symbol) = LT ||
                                order(top(operator_stack), symbol) = EQ ) do
                                        apply pop(operator)
                        push(sym, operator_stack)
                end
        end

        get_symbol
end

while operator_stack <> empty do
begin
        apply pop(operator_stack)
end
```

Figure 5: Imperative operator-precedence parsing algorithm

Modern notation	*Principia Mathematica* notation
p ∨ q	p ∨ q
p q	p.q
p (q → r)	p.q → r
p → (q ∨ (r ↔ s))	p ↔: q ∨. r ↔ s
p (q r ∨ s) → (t u ↔ v)	p:q.r.∨ s:→:t.u.↔ v

Figure 6: Examples of *Principia Mathematica* notation

	$conj(n1)$	$disj(n2,\ n3)$	$\neg$	var
$conj(n4)$	**if** $n4 > n1$ **then** $\lessdot$ **else** $\gtrdot$	**if** $n4 > n2$ **then** $\lessdot$ **else** $\gtrdot$	$\lessdot$	$\lessdot$
$disj(n5,\ n6)$	**if** $n6 \geq n1$ **then** $\lessdot$ **else** $\gtrdot$	**if** $n6 > n2$ **then** $\lessdot$ **else** $\gtrdot$	$\lessdot$	$\lessdot$
$\neg$	$\gtrdot$	$\gtrdot$	—	$\lessdot$
var	$\gtrdot$	$\gtrdot$	—	—
(	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$
)	$\gtrdot$	$\gtrdot$	$\gtrdot$	—
$begin$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$

	(	)	end
$conj(n4)$	$\lessdot$	$\gtrdot$	$\gtrdot$
$disj(n5,\ n6)$	$\lessdot$	$\gtrdot$	$\gtrdot$
$\neg$	$\lessdot$	$\gtrdot$	$\gtrdot$
var	—	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\doteq$	—
)	—	$\gtrdot$	$\gtrdot$
$begin$	$\lessdot$	—	—

Figure 7: Operator precedences for the notation described in Principia Mathematica

***conj*(n)** Denotes a conjunction with n dots.

***disj*$(n_1,\ n_2)$** Denotes a disjunctive operator ($\vee$, $\rightarrow$ or $\leftrightarrow$) with n_1 dots to the left and n_2 dots to the right.

begin Denotes the start of the string to be parsed.

end Denotes the end of the string to be parsed.

Action	Stack	Remaining input
		$\$p : q.r. \vee \ s\$$
shift	$\$$	$p : q.r. \vee \ s\$$
shift	$\$ \ll p$	$: q.r. \vee \ s\$$
shift	$\$ \ll p \gg :$	$q.r. \vee \ s\$$
reduce	$\$ \ll :$	$q.r. \vee \ s\$$
shift	$\$ \ll : \ll q$	$.r. \vee \ s\$$
shift	$\$ \ll : \ll q \gg .$	$r. \vee \ s\$$
reduce	$\$ \ll : \ll .$	$r. \vee \ s\$$
shift	$\$ \ll : \ll . \ll r$	$. \vee \ s\$$
shift	$\$ \ll : \ll . \ll r \gg . \vee$	$s\$$
reduce	$\$ \ll : \ll . \gg . \vee$	$s\$$
reduce	$\$ \ll : \ll . \vee$	$s\$$
shift	$\$ \ll : \ll . \vee \ll s$	$\$$
shift	$\$ \ll : \ll . \vee \ll s \gg \$$	
reduce	$\$ \ll : \ll . \vee \gg \$$	
reduce	$\$ \ll : \gg \$$	
reduce	$\$\$$	

Figure 8: Parsing steps for the formula $p : q.r. \vee \ s$

5 Conclusion and further work

Monads provide an elegant way to define parsers for LL(1) grammars. The main contribution of the monadic style is to allow the hiding of the threading of the parser state through the computation. Modelling of failure and non-determinism can easily be provided. The main drawback of monadic style parsers is that it is very easy to produce parsers which are very inefficient due to backtracking.

Operator precedence grammars are a useful class of grammars for parsing expression-type grammars. In some cases, operator precedence parsing can be used where no BNF grammar exists for the language, as for the *Principia Mathematica* notation introduced in section 4. A generic operator precedence parser can be written in any higher-order language, making parsing operator precedence grammars as easy as defining the precedence table and providing parsers for the operators.

Logic programming languages also provide good support for writing simple parsers. Non-determinism and failure are built into the language. DCG notation is provided to hide the parser state arguments to avoid unnecessary plumbing. It would be interesting to compare DCG parsers with monadic parsers with regard to style and expressiveness.

In summary, the monadic and operator precedence parsers discussed here are very useful for rapid prototyping and for applications where error-diagnosis, error-recovery and maximal efficiency are not important considerations, and for some grammars which cannot be implemented using standard parser generators.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] W. Burge. *Recursive Programming Techniques.* Addison-Wesley, 1975.

[3] J. Jeuring E. Meijer. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 228–266. Springer, 1995.

[4] J. Fokker. Functional parsers. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 1995.

[5] G. Hutton. Higher order functions for parsing. *Journal of Functional Programming*, 2:323–343, 1992.

[6] G. Hutton and E. Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.

[7] M. Jones. Hugs 1.3 home page, `http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html`.

[8] M. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1995.

[9] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[10] P. Wadler. The essence of functional programming. In *19'th Annual Symposium on Principles of Programming Languages*, 1992. Have to check this reference.

[11] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[12] A. N. Whitehead and B. Russell. *Principia Mathematica*, volume 1. Cambridge University Press, second edition, 1927.