Parsec: Direct Style Monadic Parser Combinators For The Real World

Daan Leijen
University of Utrecht
P.O. Box 80.089, 3508TB
Utrecht, The Netherlands
daan@cs.uu.nl

Erik Meijer
Microsoft Corp.
Redmond, WA
emeijer@microsoft.com

DRAFT

October 4, 2001

Abstract

Despite the long list of publications on parser combinators, there does not yet exist a monadic parser combinator library that is applicable in real world situations. In particular naive implementations of parser combinators are likely to suffer from space leaks and are often unable to report precise error messages in case of parse errors. The Parsec parser combinator library described in this paper, utilizes a novel implementation technique for space and time efficient parser combinators that in case of a parse error, report both the position of the error as well as all *grammar productions that would have been legal at that point in the input.*

1 Introduction

Parser combinators have always been a favorite topic amongst functional programmers. Burge (1975) already described a set of combinators in 1975 and they have been studied extensively over the years by many others (Wadler, 1985; Hutton, 1992; Fokker, 1995; Hutton and Meijer, 1996). In contrast to parser generators that offer a fixed set of combinators to express grammars, these combinators are manipulated as first class values and can be combined to define new combinators that fit the application domain. Another advantage is that the programmer uses only one language, avoiding the integration of different tools and languages (Hughes, 1989).

Despite the theoretical benefits that parser combinators offer, they are hardly used in practice. When we wrote a parser for the language XMλ (Shields and Meijer, 2001) for example, we had a set of real-world requirements on the combinators. They had to be *monadic* in order to make the parse context sensitive, they had to be *efficient* (ie. competitive in speed with happy (Gill and Marlow, 1995) and without space leaks) and they had to return *high quality error messages*. To our surprise, most current monadic parser libraries suffered from shortcomings that made them unsuitable for our purposes; they are not efficient in space or time, and they don't allow for good error messages.

There has been quite a lot of research on the efficiency of parsers combinators (Koopman and Plasmeijer, 1999; Partridge and Wright, 1996; Röjemo, 1995; Meijer, 1992) but those libraries pay almost no attention to error messages. Recently, Swierstra *et al.* (1996; 1999) have developed sophisticated combinators that even perform error correction but unfortunately they use a non-monadic formulation and a separate lexer.

This paper describes the implementation of a set of monadic parser combinators that are efficient and produce good quality error messages. Our main contribution is the overall design of the combinators, more specifically:

- We describe a novel implementation technique for space and time efficient parser combinators. Laziness is essential ingredient in the short and concise implementation. We identify a space leak that contributes largely to the inefficiency of many existing parser combinators described in literature.
- We show how the primitive combinators can be extended naturally with error messages. The user can label grammar production with suitable names. The messages contain not only the position of the error but also

all grammar productions that would have been legal at that point in the input – i.e. the first-set of that production.

The combinators that are described in this paper have been used to implement a 'real world' parser library in Haskell that is called PARSEC. This library is available with documentation and examples from http://www.cs.uu.nl/~daan/parsec.html and is distributed with the GHC compiler.

Throughout the rest of the paper we assume that the reader is familiar with the basics of monadic combinator parsers. The interested reader is referred to Hutton and Meijer (1996) for a tutorial introduction.

2 Grammars and Parsers

The following sections discuss several important restrictions and other characteristics of existing parser combinator libraries that influenced the design of Parsec.

2.1 Monadic vs. Arrow style Parsers

Monadic combinator parsers consist of a monad `Parser a` (typically of the form `String → Result a` for some functor `Result`) with a unit `return` and bind `(>>=)` operation, and a number of parser specific operations, usually a choice combinator `(<|>)` and a function `satisfy` for construction elementary parsers for terminal symbols:

```
type Parser a

return :: a → Parser a
(>>=) :: Parser a → (a → Parser b) → Parser b

satisfy :: (Char → Bool) → Parser Char
(<|>) :: Parser a → Parser a → Parser a
```

An important practical benefit of monadic parser combinators is the fact that Haskell has special syntax (the `do`-notation) that greatly simplifies writing monadic

programs. However, there are also deeper reasons why we prefer using monadic combinators.

Besides bind, there exists another important form of a sequential combinator `(<*>)` which is described by Swierstra and Duponcheel (1996) and later identified as a special case of an *arrow-style* combinator by Hughes (2000). The types of the monadic and arrow-style combinators are closely related, but subtly different:

```
(<*>) :: Parser a → Parser (a → b) → Parser b
(>>=) :: Parser a → (a → Parser b) → Parser b
```

However, their runtime behavior differs as much as their types are similar. Due to parametricity (Wadler, 1989), the second parser of `(<*>)` will never depend on the (runtime) result of the first. In the monadic combinator, the second parser *always* depends on the result of the first parser. An interesting relation between both forms follows directly from their type signatures; arrow-style parser combinators can at most parse languages that can be described by a *context-free* grammar while the monadic combinators can also parse languages described by *context-sensitive* grammars.

Since parsers described with arrow-style combinators never depend on run-time constructed values, it is possible to analyze the parsers before executing them. Swierstra and Duponcheel (1996) use this characteristic when they describe combinators that build lookup tables and perform dynamic error correction.

Monadic combinators are able to parse context sensitive grammars. This is not just a technical nicety. It can be used in many situations that are tradionally handled as a separate pass after parsing. For example, if plain XML documents are parsed with a context-free parser, there is a separate analysis needed to guarantee well-formed ness, i.e. that every start tag is closed by a matching end tag.

A monadic parser can construct a specialized end tag parser when an open tag is encountered. Given an `openTag` parser that returns the tag name of a tag and an `endTag` parser that parses an end tag with the name that is passed as an argument, an XML parser that only accepts well-formed fragments can be structured as follows:

```
xml = do{ name    <- openTag
        ; content <- many xml
        ; endTag name
```

```
        ; return (Node name content)
        }
    <|> xmlText
```

2.2 Left recursion

An important restriction on most existing combinator parsers (and Parsec is no exception) is that they are unable to deal with left-recursion. The first thing a left-recursive parser would do is to call itself, resulting in an infinite loop.

In practice however, grammars are often left-recursive. For example, expression grammars usually use left-recursion to describe left-associative operators.

```
expr   ::= expr "+" factor
factor ::= number | "(" expr ")"
```

As it is, this grammar can not be literally translated into parser combinators. Fortunately, every left-recursive grammar can be rewritten into a right-recursive one (Aho *et al.*, 1986). It is also possible to define a combinator `chainl` (Fokker, 1995) that captures the design pattern of encoding associativety using left-recursion directly, thereby avoiding a manual rewrite of the grammar.

2.2.1 Sharing

One could think that the combinators themselves can observe that `expr` is left-recursive, and thus could prevent going into an infinite loop. In a pure language however, it is impossible to observe sharing from within the program. It follows that parser combinators are unable to analyze their own structure and can never employ standard algorithms on grammars to optimize the parsing process.

All combinator libraries are forced use a *predictive* parsing algorithm, also known as *left-to-right, left-most derivation* or LL parsing (Aho *et al.*, 1986). (LR parsing is still the exclusive domain of separate tools that can analyze the grammar on a meta-level.) However, Claessen and Sands (1999) describe an interesting approach to observable sharing in the context of hardware descriptions which might be used in the context of parser combinators to analyze the structure of a parser at run-time.

2.3 Backtracking

Ambiguous grammars have more than one parse tree for a sentence in the language. Only parser combinators that can return more than one value can handle ambiguous grammars. Such combinators use a list as their reply type.

In practice however, you hardly ever need to deal with ambiguous grammars. In fact it is often more a nuisance than a help. For instance, for parser combinators that return a list of successes, it doesn't matter whether that list contains zero, one or many elements. They are all valid answers. This makes it hard to give good error messages (see below). Furthermore it is non-trivial to tame the space and (worst case exponential) time complexity of full backtracking parsers.

However, even when we restrict ourselves to non-ambiguous grammars, we still need to backtrack because the parser might need to look arbitrary far ahead in the input.

Naive implementations of backtracking parser combinators suffer from a space leak. The problem originates in the definition of the choice combinator. It either always tries its second alternative (because it tries to find all possible parses), or whenever the first alternative fails (because it requires arbitrary lookahead). As a result, the parser `(p <|> q)` holds on to its input until `p` returns, since it needs the original input to run parser `q` when `p` has failed. The space leak leads quickly to either a stack/heap overflow or reduction in speed on larger inputs.

2.4 Errors

Parsers should report errors when the input does not conform to the grammar. A good parser error message contains the position of the error in the input as well as the cause of the error. Besides the cause of an error, the message can contain *all* possible productions that would have been legal at that point in the input. These correspond to the FIRST set of a non-terminal.

Beside error reporting the parser might try to *correct* the error. After detecting an error, the input is modified by deleting or inserting tokens which might lead to valid input again. Swierstra and Duponcheel (1996) describe how automatic error correction can be implemented with arrow-style parser combinators.

As explained above, current (nondeterministic) parser combinators are not very good at reporting errors. The combinators report neither the position nor the

possible causes of an error. It is hard to report an error since the the parsers can always look arbitrarily far ahead in the input (they are LL(∞) and it becomes hard to decide what the error message should be.

It is for the two reasons above that in Parsec *we restrict ourselves to predictive parsers with limited lookahead.* The `<|>` combinator is left-biased and will return the first succeeding parse tree (i.e. even if the grammar is ambiguous only one parse tree is returned). The Parsec combinators will report all possible causes of an error. The messages can be customized by the user – instead of giving the error message on the character level it contains a grammar production description.

2.5 LL Grammars

The following sections derive a space efficient and error reporting combinator parsers. The space leak can be fixed by restricting the lookahead. As a side effect this also improve the quality of the error messages that are implemented later in this paper.

LL grammars have the distinctive properties that they are non-ambiguous and not left-recursive. A grammar is LL(k) if the associated predictive parser needs at most k tokens lookahead to disambiguate a parse tree. For example, the following grammar is LL(2):

```
S ::= PQ | Q
P ::= "p"
Q ::= "pq"
```

When a the first token is `"p"`,F we still don't know if we are in the `PQ` or `Q` production, only upon seeing the second token (`"p"` or `"q"`) we know what to choose.

The usual list of successes combinators have the interesting property that they have a dynamic lookahead to an arbitrary large k; We will call this an LL(∞) parser. The combinators will look arbitrarily far ahead due to the definition of the `(<|>)` combinator. Whenever the first parser fails, the second will be tried instead, no matter how many tokens the first parser has consumed! The previous grammar can be translated literally into combinators:

```
s  = do{ p; q } <|> q
```

```
p  = char 'p'
q  = do{ char 'p'; char 'q' }
```

Unfortunately, this doesn't hold in general. There is a specific case where we can't literally translate the grammar. Here is the previous grammar again written in a slightly different way:

```
S ::= PQ
P ::= "p" | ϵ
Q ::= "pq"
```

When we literally translate this grammar we get:

```
s = do{ p; q }
p = char 'p' <|> return 'p'
q  = do{ char 'p'; char 'q' }
```

The `<|>` combinator is now local to the `p` parser. It returns a result right after the first character is consumed. If the input was `"pq"` it will recognize the `"p"` character as part of the `p` production and fail when trying `q`! The `<|>` combinator should be used at the point where lookahead is actually needed and can not be used locally in the production.

In general, every `PQ` where `P` $\Rightarrow^* \epsilon$ (i.e. `P` has an empty derivation) and where FIRST(`P`) $\cup$ FIRST(`Q`) $\neq \emptyset$ (i.e. their first-sets overlap (Aho *et al.*, 1986)), should be rewritten to `P'Q | Q` where `P'` equals production `P` but no longer includes an ϵ derivation. If a grammar is *left-factored* (Aho *et al.*, 1986) this transformation happens automatically.

LL(∞) is a powerful grammar class. Any non-ambiguous context-free grammar can be transformed into an LL(∞) grammar. In practice, there are many languages that require arbitrary lookahead; for example, type signatures in Haskell or declarations in C.

3 Restricting lookahead

The following sections will focus on implementing a set of monadic combinators that circumvent the space leak of naive combinators and add good error messages.

To solve the space leak of the naive parser combinators, we turn to deterministic predictive parsing with limited lookahead. An LL(1) parser has a lookahead of a single token – it can always decide which alternative to take based on the current input character. In practice this means that the parser `(p <|> q)` never tries parser `q` whenever parser `p` has *consumed any input*.

To use an LL(1) strategy, each parser is keeps track of its input consumption. We call this the *consumer*-based approach. A parser has either `Consumed` input or returned a value without consuming input, `Empty`. The return value is either a single result and the remaining input, `Ok a String`, or a parse error, `Error`:

```
type Parser a   = String → Consumed a

data Consumed a = Consumed (Reply a)
                | Empty (Reply a)

data Reply a    = Ok a String | Error
```

Note that the real Parsec library is parameterized with the type of the input and a user definable state.

3.1 Basic combinators

Given the concrete definition of our `Parser` type, we can now turn to the implementation of the basic parser combinators.

The `return` combinator succeeds immediately without consuming any input, hence it returns the `Empty` alternative:

```
return x
  = \input -> Empty (Ok x input)
```

The `satisfy` combinator consumes a single character when the test succeeds but returns `Empty` when the test fails, or when it encounters the end of the input:

```
satisfy :: (Char → Bool) → Parser Char
satisfy test
```

p	q	(p >>= q)
Empty	Empty	Empty
Empty	Consumed	Consumed
Consumed	Empty	Consumed
Consumed	Consumed	Consumed

Figure 1: Input consumption of `(>>=)`

```
  = \input -> case (input) of
      []     -> Empty Error
      (c:cs) | test c    -> Consumed (Ok c cs)
             | otherwise -> Empty Error
```

With the `satisfy` combinator we can already define some useful parsers:

```
char c  = satisfy (==c)
letter  = satisfy isAlpha
digit   = satisfy isDigit
```

The implementation of the `(>>=)` combinator is the first one where we take consumer information into account. Figure 1 summarizes the input consumption of a parser `(p >>= f)`. If `p` succeeds without consuming input, the result is determined by the second parser. However, if `p` succeeds while consuming input, the sequence starting with `p` surely consumes input *Thanks to lazy evaluation, it is therefore possible to immediately build a reply with a* `Consumed` *constructor even though the final reply value is unknown.*

```
(>>=) :: Parser a → (a → Parser b) → Parser b
p >>= f
  = \input -> case (p input) of
      Empty reply1
        -> case (reply1) of
             Ok x rest -> ((f x) rest)
             Error     -> Empty Error

      Consumed reply1
        -> Consumed
           (case (reply1) of
              Ok x rest
                    -> case ((f x) rest) of
                         Consumed reply2 -> reply2
```

```
                    Empty reply2    -> reply2
          error -> error
        )
```

Due to laziness, a parser `(p >>= f)` directly returns with a `Consumed` constructor if `p` consumes input. The computation of the final reply value is *delayed*. This 'early' returning is essential for the efficient behavior of the choice combinator.

An LL(1) choice combinator only looks at its second alternative if the first hasn't consumed any input – regardless of the final reply value! Now that the `(>>=)` combinator immediately returns a `Consumed` constructor as soon as some input has been consumed, the choice combinator can choose an alternative as soon as some input has been consumed. It no longer holds on to the original input, fixing the space leak of the previous combinators.

```
(<|>) :: Parser a → Parser a → Parser a
p <|> q
  = \input -> case (p input) of
      Empty Error -> (q input)
      Empty ok    -> case (q input) of
                       Empty _   -> Empty ok
                       consumed  -> consumed
      consumed    -> consumed
```

Note that if `p` succeeds without consuming input the second alternative is favored if it consumes input. This implements the "longest match" rule.

With the bind and choice combinator we can define almost any parser. Here are a few useful examples:

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

many1 :: Parser a -> Parser [a]
many1 p
  = do{ x  <- p;
      ; xs <- (many1 p <|> return [])
      ; return (x:xs)
      }
```

```
identifier
  = many1 (letter <|> digit <|> char '_')
```

Note that the formulation of the `many1` parser works because the choice combinator doesn't backtrack anymore.

3.2 Related work

It is interesting to compare this approach with previous work on efficient parser combinators. Röjemo (1995) uses a continuation based approach in combination with a `cut` combinator. The `cut` combinator is used to implement an LL(1) variant of the choice combinator. A variant of Röjemo's solution is given by Koopman and Plasmeijer (1999). In his thesis (1992), Meijer describes several alternative implementations of the `cut` combinator using continuation based parsers.

The main contribution of this paper is the simplicity of the consumer based approach when compared to an implementation based on continuations. Due to laziness, the algorithm can be specified declaratively, while getting the same operational 'interleaved' behavior as with continuations. It is also easier to constructively add error messages to the combinators, which is done later in this paper.

The consumer based design is perhaps most closely related to the work of Partridge and Wright (1996). They implement a predictive LL(1) parser using four return values in their parser reply:

```
data Reply a  = Ok a String
              | Epsn a String
              | Err
              | Fail
```

The `Epsn` (epsilon) and `Fail` alternatives are used when the parser hasn't consumed any input. The correspondence with a consumer based design is clear:

Partridge & Wright		Consumer based design
`Ok x input`	≡	`Consumed (Ok x input)`
`Epsn x input`	≡	`Empty (Ok x input)`
`Err`	≡	`Consumed (Error)`
`Fail`	≡	`Empty (Error)`

Library	chars/second	allocated/char	resident/char
parsec + scanner	115,000	409	13
parsec	88,000	896	6
parselib	78,000	730	23
uuparsing + scanner	61,000	928	58

Figure 2: Comparison of libraries

Unfortunately, the approach of Partridge and Wright still suffers from the space leak. The information about input consumption is tupled with the information about the success of the parser. The choice operator now holds on the input since information about both the success and the consumption of a parser is returned, which can only be done after a reply is completely evaluated.

3.3 Measurements

We have done some prelimary measurements on the effectiveness of the consumer based design. We took four different libraries and let them parse the standard libraries of the Zurich Oberon system (Wirth, 1988). To make the test as honest as possible, we wrote the Oberon parser using standard arrow-style combinators and mapped the basic combinators of each library to these combinators. This enables each library to use the exactly the same parser sources.

The libraries tested are:

- *parsec*. The full Parsec library, including the error message mechanism that is developed later in this paper. The library can parse context-sensitive grammars with infinite lookahead. There are two variants tested, "parsec" is a version where the entire grammar, including the lexical part, is described using parser combinators and "parsec+scanner" is a version where a seperate hand-written scanner is used.

- *uuparsing*. A sophisticated arrow-style library developed at the University of Utrecht (Swierstra and Azero Alcocer, 1999). A prominent feature is that the library automatically corrects the input on errors and (thus) always succeeds. The library parses context-free with infinite lookahead. The parser in our test uses a seperate hand-written scanner for Oberon.

- *parselib*. The 'standard' monadic parser library that is distributed with the Hugs interpreter. This is a monadic parser library developed by Graham Hutton and Erik Meijer (Hutton and Meijer, 1996). The library parses

context sensitive with infinite lookahead and can even deal with ambigious grammars but gives no error messages at all. The entire grammar is described using parser combinators.

Each library was compiled with GHC 5.02 with the `-O2` flag and tested against all 102 standard library files of the Zurich Oberon system. The largest of these files consists of 115,000 characters and 3302 lines, and the total line count is 87,000. The libraries were run with a 64 Mb heap on a 550 MHz Pentium running FreeBSD. Detailed results can be found at `http://www.cs.uu.nl/~daan/pbench.html`.

Figure 2 summarizes the results. It shows the average number of characters parsed per second, the number of bytes allocated per character and the number of bytes resident per character. The residency gives the maximal portion of the heap that was live during the execution of the program.

The measurements should be interpreted with care since each library uses different parsing strategies and has different features. For example, in contrast to the other libraries, the ParseLib library can deal with ambigious grammars. The bottom line however is that each library uses exactly the same parser source to parse the same Oberon sources and it seems that the consumer based design pays off in practice.

3.4 Infinite lookahead, again

With all these optimization efforts, the parser combinators are now restricted to LL(1) grammars. Unfortunately, most (programming language) grammars are not LL(1) and even require arbitrary lookahead.

Dually to the approach sketched in (Röjemo, 1995; Hutton and Meijer, 1996; Koopman and Plasmeijer, 1999; Meijer, 1992) where a special combinator is introduced to mark explicitly when *no* lookahead is needed, we add a special combinator to mark explicitly where arbitrary lookahead *is* allowed.

The `(try p)` parser behaves exactly like parser `p` but pretends it hasn't consumed any input when `p` fails:

```
try :: Parser a → Parser a
try p
```

```
  = \input -> case (p input) of
                Consumed Error  -> Empty Error
                other           -> other
```

Consider the parser `(try p <|> q)`. Even when parser `p` fails while consuming input (`Consumed Error`), the choice operator will try the alternative `q` since the `try` combinator has changed the `Consumed` constructor into `Empty`. Indeed, if you put `try` around all parsers you will have an LL(∞) parser again!

Although not discussed in their paper, the `try` combinator could just as easily be applied with the four reply value approach of Partridge and Wright (1996), changing `Err` replies into `Fail` replies. The approach sketched here is dual to the three reply values of Hutton (1992). Hutton introduces a `noFail` combinator that turns empty errors into consumed errors! It effectively prevents backtracking by manual intervention.

3.5 Lexing

The `try` combinator can for example be used to specify both a lexer and parser together. Take for example the following parser:

```
expr =   do{ string "let"; whiteSpace; letExpr }
     <|> identifier
```

As it stands, this parser doesn't work as expected. On the input `letter` for example, it fails with an error message.

```
>run expr "letter"
parse error at (line 1,column 4):
unexpected "t"
expecting white space
```

The `try` combinator should be used to backtrack on the `let` keyword. The following parser correctly recognises the input `letter` as an identifier.

```
expr =   do{ try (string "let"); whiteSpace; letExpr }
     <|> identifier
```

In contrast with other libraries, the `try` combinator is not built into a special choice combinator. This improves modularity and allows the construction of lexer libraries that use `try` on each lexical token. The Parsec library is distributed with such a library and in practice, `try` is only needed for grammar constructions that require lookahead.

4 Error Messages

The restriction to LL(1) makes it much easier for us to generate good error messages. First of all, the error message should include the position of an error. The parser input is tupled with the current position – the parser state.

```
type Parser a   = State -> Consumed a

data State      = State String Pos
```

Beside the position, it is very helpful for the user to return the grammar productions that would have led to correct input at that position. This corresponds to the FIRST set of that production. During the parsing process, we will dynamically compute first sets for use in error messages. This may seem expensive but laziness ensures that this only happens when an actual error occurs.

An error message contains a position, the unexpected input and a list of expected productions – the first set.

```
data Message    = Message Pos String [String]
```

To dynamically compute the first set, not only `Error` replies but also `Ok` replies should carry an error message. Within the `Ok` reply, the message represents the error that would have occurred if this successful alternative wasn't taken.

```
data Reply a    = Ok a State Message
                | Error Message
```

4.1 Basic parsers

The `return` parser attaches an empty message to the parser reply.

```
return :: a -> Parser a
return x
  = \state ->
    Empty (Ok x state (Message pos [] []))
```

The implementation of the `satisfy` parser changes more. It updates the parse position if it succeeds and returns an error message with the current position and input if it fails.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy test
  = \(State input pos) ->
       case (input) of
         (c:cs) | test c
                -> let newPos   = nextPos pos c
                       newState = State cs newPos
                   in seq newPos
                        (Consumed
                          (Ok c newState
                              (Msg pos [] [])))

         (c:cs) -> Empty (Error
                     (Msg pos [c] []))
         []     -> Empty (Error
                     (Msg pos "end of input" []))
```

Note the use of `seq` to strictly evaluate the new position. If this is done lazily, we would introduce a new space leak – the original input is retained since it is needed to compute the new position at some later time.

The `(<|>)` combinator computes the dynamic first set by merging the error messages of two `Empty` alternatives – regardless of their reply value. Whenever both alternatives do not consume input, both of them contribute to the possible causes of failure. Even when the second succeeds, the first alternative should propagate its error messages into the `Ok` reply.

```
(<|>) :: Parser a → Parser a → Parser a
p <|> q
  = \state ->
      case (p state) of
        Empty (Error msg1)
                 -> case (q state) of
```

```
                      Empty (Error msg2)
                        -> mergeError msg1 msg2
                      Empty (Ok x inp msg2)
                        -> mergeOk x inp msg1 msg2
                      consumed
                        -> consumed
        Empty (Ok x inp msg1)
                 -> case (q state) of
                      Empty (Error msg2)
                        -> mergeOk x inp msg1 msg2
                      Empty (Ok _ _ msg2)
                        -> mergeOk x inp msg1 msg2
                      consumed
                        -> consumed
        consumed -> consumed

mergeOk x inp msg1 msg2
  = Empty (Ok x inp (merge inp1 inp2))

mergeError msg1 msg2
  = Empty (Error (merge msg1 msg2))

merge (Msg pos inp exp1) (Msg _ _ exp2)
  = Msg pos inp (exp1 ++ exp2)
```

Notice that the positions of the error message passed to `merge` should always be the same. Since the choice combinator only calls `merge` when both alternatives have not consumed input, both positions are guaranteed to be equal.

The sequence combinator computes the first set by merging error messages whenever one of the parsers doesn't consume input. In those cases, both of the parsers contribute to the error messages.

4.2 Labels

Although error messages are nicely merged, there is still no way of adding names to productions. The new combinator `(<?>)` labels a parser with a name.

The parser `(p <?> msg)` behaves like parser `p` but when it fails *without* consuming input, it sets the expected productions to `msg`. It is important that it only does so when no input is consumed since otherwise it wouldn't be something that is expected after all:

```
(<?>) :: Parser a -> String -> Parser a
p <?> exp
  = \state ->
      case (p state) of
        Empty (Error msg)
              -> Empty (Error (expect msg exp))
        Empty (Ok x st msg)
              -> Empy (Ok x st (expect msg exp))
        other -> other

expect (Msg pos inp _) exp
  = Msg pos inp [exp]
```

The label combinator is used to return error messages in terms of high-level grammar productions rather than at the character level. For example, the elementary parsers are redefined with labels:

```
digit      = satisfy isDigit <?> "digit"
letter     = satisfy isAlpha <?> "letter"
char c     = satisfy (==c)   <?> (show c)

identifier = many1 (letter <|> digit <|> char '_')
```

4.3 Labels in practice

Error messages are quite improved with these labels, even when compared to widely used parser generators like YACC. Here is an example of applying the `identifier` parser to the empty input.

```
>run identifier ""
parse error at (line 1,column 1):
unexpected end of input
expecting letter, digit or '_'
```

Normally all important grammar productions get a label attached. The previous error message is even better when the `identifier` parser is also labeled. Note that this label overrides the others.

```
>run identifier "@"
```

```
parse error at (line 1,column 1):
unexpected "@"
expecting identifier
```

The following example illustrates why `Ok` replies need to carry error messages with them.

```
test = do{ (digit <|> return '0')
         ; letter
         }
```

The first set of this parser consists of both a `digit` and a `letter`. On illegal input both these production should be included in the error message. Operationally, the `digit` parser will fail and the `return '0'` alternative is taken. The `Ok` reply however still holds the `expecting digit` message. When the `letter` parser fails, both productions are shown:

```
>run test "*"
parse error at (line 1,column 1):
unexpected "*"
expecting digit or letter
```

4.4 Related work

Error reporting is first described by Hutton (1992). However, the solution proposed is quite subtle to apply in practice, involving deep knowledge about the underlying implementation. The position of the error is not reported as the combinators backtrack by default – this makes it hard to generate good quality error messages. Röjemo (1995) adds error messages with source positions using a predictive parsing strategy.

Error correcting parsers are parsers that always continue parsing. Swierstra *et al.* (1996; 1999) describe sophisticated implementations of error correction. These parser probably lend themselves well to customizable error messages as described in this paper.

5 Conclusions

We hope to have showed to parser combinators can be an acceptable alternative to parser generators in practice. Moreover, the efficient implementation of the combinators is surprisingly concise – laziness is essential for this implementation technique.

At the same time, we have identified weaknesses of the parser combinators approach, most notably the left-recursion limitation and the inability to analyse the grammar at run-time.

6 Acknowledgements

Doaitse Swierstra has been a source of inspiration with his intimate knowledge of error-correcting parser strategies. We would also like to thank Mark Shields for his help on the operational semantics of these combinators. Johan Jeuring has provided many suggestions that improved the initial draft of this paper.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986. ISBN 0-201-10194-7.

W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975. ISBN 0-201-14450-6.

Koen Claessen and David Sands. *Observable sharing for functional circuit description*. In ASIAN'99, 5th Asian Computing Science Conference, LNCS, 1742:62–73, Phuket, Thailand, 1999. Springer-Verlag. http://www.cs.chalmers.se/~dave/papers/observable-sharing.ps.

Jeroen Fokker. *Functional parsers*. In Advanced Functional Programming, First International Spring School, LNCS, 925:1–23, Båstad, Sweden, May 1995. Springer-Verlag. http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps.

Andy Gill and Simon Marlow. *Happy – The Parser Generator for Haskell*, 1995. University of Glasgow. http://www.haskell.org/happy.

Steve Hill. *Combinators for parsing expressions*. Journal of Functional Programming, 6(3):445–463, May 1996.

John Hughes. *Why Functional Programming Matters.* Computer Journal, 32(2):98–107, 1989.

John Hughes. *Generalising monads to arrows.* Science of Computer Programming, 37:67–111, 2000.
http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps.

Graham Hutton and Erik Meijer. *Monadic parser combinators.* Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps.

Graham Hutton. *Higher-order functions for parsing.* Journal of Functional Programming, 2(3):232–343, July 1992.
http://www.cs.nott.ac.uk/Department/Staff/gmh/parsing.ps.

Pieter Koopman and Rinus Plasmeijer. *Efficient combinator parsers.* In Implementation of Functional Languages, LNCS, 1595:122–138. Springer-Verlag, 1999.

Erik Meijer. *Calculating Compilers.* PhD thesis, Nijmegen University, 1992.

Andrew Partridge and David Wright. *Predictive parser combinators need four values to report errors.* Journal of Functional Programming, 6(2):355–364, March 1996.

Niklas Röjemo. *Garbage collection and memory efficiency in lazy functional languages.* PhD thesis, Chalmers University of Technology, 1995.

Mark Shields and Erik Meijer. *Type-indexed rows.* In Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), London, England, pages 261–275. ACM press, January 2001.
http://www.cse.ogi.edu/~mbs/pub/type_indexed_rows.

Doaitse Swierstra and Pablo Azero Alcocer. *Fast, error correcting parser combinators: A short tutorial.* In SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, LNCS, 1725:111–129. Springer-Verlag, November 1999.
http://www.cs.uu.nl/groups/ST/Software/UU_Parsing.

Doaitse Swierstra and Luc Duponcheel. *Deterministic, error correcting combinator parsers.* In Advanced Functional Programming, Second International Spring School, LNCS, 1129:184–207. Springer-Verlag, 1996.
http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/AFP2.ps.

Philip Wadler. *How to replace failure with a list of successes.* In Functional Programming Languages and Computer Architecture, LNCS, 201:113–128. Springer-Verlag, 1985.

Philip Wadler. *Theorems for free.* In Mac Queen, editor, 4'th International Conference on Functional Programming and Computer Architecture, pages 347–359, London, September 1989. Addison-Wesley.

Philip Wadler. *The essence of functional programming.* In 19'th Symposium on Principles of Programming Languages, pages 1–14, Albuquerque, New Mexico, January 1992. ACM press.
http://cm.bell-labs.com/cm/cs/who/wadler/topics/monads.html.

Niklaus Wirth. *The programming language Oberon.* Software Practice and Experience, 19(9), 1988. The Oberon language report.
http://www.oberon.ethz.ch.