Predictive parser combinators need four values to report errors

ANDREW PARTRIDGE AND DAVID WRIGHT

Department of Computer Science, University of Tasmania
GPO Box 252C, Hobart, Tasmania 7001, Australia
(e-mail: {A.S.Partridge, D.A.Wright}@cs.utas.edu.au*)*

Abstract

A combinator-based parser is a parser constructed directly from a BNF grammar, using higher-order functions (combinators) to model the alternative and sequencing operations of BNF. This paper describes a method for constructing parser combinators that can be used to build efficient predictive parsers which accurately report the cause of parsing errors. The method uses parsers that return values (parse trees or error indications) decorated with one of four tags.

Capsule Review

Combinator parsers have become popular in functional programming circles in recent years, although some questions have been raised regarding their practicality. This paper describes how to construct efficient predictive combinator parsers, and in addition addresses an important pragmatic issue which is not usually handled well, even by conventional parsers: the accurate identification of parsing errors.

1 Introduction

The idea behind combinator-based parsers is to construct a parser directly from a BNF grammar by modelling the alternative and sequencing operations of BNF using higher-order functions called *parser combinators*. Benefits of this technique include: the parser can be written straightforwardly in the language in which the rest of the program is written; semantic actions are easily incorporated; and special purpose combinators can be defined in terms of the basic combinators, using the full power of a lazy functional language.

Although combinator-based parsers in general can deal naturally with ambiguous grammars, this paper is only concerned with the use of combinators to construct *predictive parsers* for LL(1) grammars (Aho *et al.*, 1986). A predictive parser is a recursive-descent parser that needs no backtracking. LL(1) grammars have the property that at most one of a set of alternatives will match at least one token (Lewis and Stearns, 1968). Programming languages, for example, are usually context-free languages whose grammars can be translated into LL(1) form by eliminating left-recursion and common left factors.

The main contribution of this paper is an improvement in the error reporting provided by parser combinators when used to write predictive parsers. Using the method described in this paper, the first token following the longest parse of the input is automatically reported as the cause of a parse error. However, parsers constructed using the method described in this paper do not recover from parse errors.

The programming examples in the paper are given in Gofer (Jones, 1994). Any other lazy language with higher-order functions and a polymorphic type system could be used.

2 A parser combinator monad

Monads are a well-established means of structuring functional programs (Wadler, 1990), and it is convenient, but by no means essential, to present this work using them. We will use the monad syntax provided by the Gofer system (Jones, 1994). In this syntax, one can use indentation preceded by the keyword `do` instead of writing the `bind` operator explicitly. The `result` combinator is still written explicitly.

A *functor* is a type constructor over which a `map` function is defined. A *monad* is a functor with `result` and `bind` functions defined. Here are type class definitions for functors, monads and monads with a zero and a plus, similar to those provided in the Gofer constructor class prelude:

```
class Functor f where
    map :: (a -> b) -> (f a -> f b)

class Functor m => Monad m where
    result :: a -> m a
    bind :: m a -> (a -> m b) -> m b

class Monad m => Monad0 m where
    zero :: m a

class Monad0 c => MonadPlus c where
    (++) :: c a -> c a -> c a
```

To model a BNF grammar the `bind` operator will be used to construct parsers for sequences; `result` will have the dual purpose of enabling the construction of the result of a successful parse and modelling the ϵ parser from BNF; `zero` will be defined as a primitive parser that fails after consuming no input tokens; and plus (`++`) will be used to build parsers for alternatives. We also define a primitive parser `lit` for parsing literals. The `map` function may be used to apply a function to the successful result of a parser.

This is by no means a complete set of primitive parsers and parser combinators, but it is adequate for describing our method.

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

3 Two-value combinators

Hutton (1992) presents a set of parser combinators based on the list of successes method of Wadler (1985). Using this technique, parsers return a list of successful parses. While this permits the writing of parsers for ambiguous grammars, the resulting parsers cannot indicate where parse errors occur, since failure of a parser to match is represented by an empty list.

For unambiguous grammars, parsers need only return a single parse. This leads Hutton to the idea of parsers that return either the `Ok` tagged result of a single successful parse paired with the remaining input, or a `Fail` tagged indication of the location and nature of parse failure. Because there are two tag values, we shall call the combinator sets used to build such parsers *two-value combinators*. The following types and access function are suitable for two-value combinators:

```
data Parser a b = Parser ([a] -> Reply (b, [a]) (Error a))
data Reply x y = Ok x | Fail y
type Error a = a

getParser (Parser pf) = pf
```

In the type `Parser a b`, `a` is the type of tokens to be parsed, and `b` is the type of the result of a successful parse. In general, `Error` could be an algebraic type with separate tags for indicating different classes of errors, such as lexical errors, expected end of file, and unexpected end of file.

The `map` function applies a function to the successful result of a parser:

```
instance Functor (Parser a) where
   map f p = Parser (\inp ->
                case getParser p inp of
                   Fail e          -> Fail e
                   Ok (x, inp1)    -> Ok (f x, inp1))
```

The primitive parser `zero` fails without consuming any tokens:

```
instance MonadO (Parser a) where
   zero = Parser (\(tok:toks) -> Fail tok)
```

The function `lit` takes a token as an argument and returns a parser that checks whether the next input token matches the argument. If it does, the token is returned as the result of the parse, tagged with `Ok`; otherwise the failing token is returned, tagged with `Fail`. If it is necessary to indicate the absolute position of a failing token, tokens may be tagged with their position in the input at the lexical analysis stage. This information can then be extracted to indicate the location of the error if the top-level parse fails.

```
lit :: Eq a => a -> Parser a a
lit x = Parser (\(y:ys) -> if (x==y) then Ok (y, ys) else Fail y)
```

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

Table 1. *Behaviour of the* ++ *and* bind *combinators with two values*

a	b	a 'bind' (_ -> b)	a ++ b
Fail	Fail	Fail	Fail
Fail	Ok	Fail	Ok
Ok	Fail	Fail	Ok
Ok	Ok	Ok	Ok [a]

[a] Technically, if the grammar is LL(1), both parses cannot succeed. The result is defined to be Ok here so that it does not depend on the result from the second parse. Hence, the second parse need not be evaluated if the first succeeded.

Table 1 gives the abstract behaviour required of bind and ++ for the two-value method. The abstraction is that only the tags on the values returned by the parsers a and b are necessary to compute the tag returned by the result parser.

Here is the code for bind and result using the two-value method:

```
instance Monad (Parser a) where
   p1 'bind' p2 = Parser (\inp ->
                     case getParser p1 inp of
                        Fail e1        -> Fail e1
                        Ok (x, inp1)   ->
                           case getParser (p2 x) inp1 of
                              Fail e2      -> Fail e2
                              Ok (y, inp2) -> Ok (y, inp2))

   result x = Parser (\inp -> Ok (x, inp))
```

The bind combinator constructs parsers for sequences. Its first argument is a parser, and its second argument is a function that returns a parser after being given the result of the first parse as argument. Table 1 shows that both argument parsers must succeed for the sequence parser to succeed. The result function takes one argument and returns a parser that consumes no input tokens and succeeds, returning the argument as the result of the parse.

Table 1 shows that in an alternative either parse may succeed for the alternative to succeed:

```
instance MonadPlus (Parser a) where
   p1 ++ p2 = Parser (\inp ->
                  case getParser p1 inp of
                     Fail e -> getParser p2 inp
                     Ok x   -> Ok x)
```

If a tree of ++'s contains a result, the result must appear as the second argument of the rightmost ++. This is necessary to ensure that the longest successful parse is always returned, because the ++ combinator only attempts to match its second parser argument to the input if its first argument failed to match.

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

There are two problems with the two-value approach: the parsers tend to show the location of errors as appearing earlier in the source than they really are, and ++ sometimes unnecessarily tries to match its second argument onto the input. These problems are linked, as can be seen by considering the following grammar:

```
ex1 = do {a; b} ++ c
```

which is Gofer monad syntax for:

```
ex1 = (a `bind` (\_ -> b)) ++ c
```

If `a` matches but `b` does not, the `bind` subexpression will return `Fail`, so the ++ operator will attempt to match `c`. Since we assume that the grammar is unambiguous, there is no possibility that `c` will match here, so this is wasted work. Furthermore, when `c` fails to match, the ++ operator has insufficient information to decide whether to report one failed parse or the other. In this case it happens to return the error from the `c` parser, effectively indicating that the error lies in the piece of the input that was correctly matched by parser `a`!

4 Two tag values plus position information

For accurate error reporting it is usual to tag tokens with their row and column positions. This information can be used to compare the relative positions of two tokens in the input, enabling ++ to choose the longer of the two parses when one of them fails. This guarantees that error messages indicate the position of an error as occurring at the end of the longest parse, and so solves the problems with the two-value combinators from the previous section.

Note that even a failed parse is a candidate for being the longer of two parses being considered by ++. For example, a parse that consumes a number of tokens and then fails is longer than one that consumes no tokens at all and succeeds. (Recall that with an LL(1) grammar, at most one of the alternatives may match a token.)

Because the production rules are left factored and any `result` parsers appear at the far right of any trees of ++'s, the ++ combinator can safely assume that if the first parse succeeds it is longer than the second parse. Only if the first parse fails does the second parse have to be attempted to find its length. In this case, ++ should return the longer of the two parses, regardless of which succeeded.

Unfortunately, using this method the type of ++ becomes cluttered because of the need to examine the positions of tokens. This makes the combinators less abstract, so we seek a better method.

5 Three-value combinators

An alternative solution to the problem with two-value combinators is given by Hutton (1992). Hutton's solution is to make parsers return values with one of three tag values: `Ok`, `Fail`, and `Err`.

The `Err` value is generated using a new combinator called `noFail`. The `noFail` combinator translates `Fail` tags into `Err` tags, but leaves `Ok` tags alone. For example,

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

the use of `noFail` in `(p ++ q ++ nofail r)` captures the notion that failure of any alternative to succeed gives rise to an error. The use of `noFail` in `(p 'bind' (\x -> nofail q))` captures the notion that failure of one parser after success of another gives an error rather than just failure.

Explicit use of the `noFail` combinator requires some care to obtain informative error reporting. Fortunately, if we assume an LL(1) grammar it is possible to automate the generation of the `Err` value. The next section shows how this is done.

6 Four-value combinators

The following observation is the key to automating the generation of the `Err` value: with an LL(1) grammar, if an alternative successfully consumes at least one token (possibly followed by failure), we can assume that the other alternative will not be able to successfully consume any tokens. Conversely, if an alternative consumes no tokens (successfully or not), then the other alternative may be able to consume some tokens.

Note the essential difference between `Err` and `Fail`: `Err` should be returned by a parser that failed after successfully consuming at least one token, in which case no alternative parser should be tried; `Fail` is returned by a parser that failed without consuming any tokens at all, in which case an alternative parser may be tried.

It is also necessary to distinguish between a successful parse that consumed at least one token and one that consumed no tokens. A parser that successfully consumes at least one token but then fails must return the `Err` value. However, a parser may succeed without consuming any tokens (for example, the `result` parser). It is therefore possible for a parser to successfully consume no tokens, followed by failure. Here the parser has consumed no tokens, and must return the `Fail` value. This is necessary in case the parser is part of an alternative, as the other alternative should then be tried.

To distinguish between a successful parse that consumed at least one token and a successful parse that consumed no tokens, we introduce the new tag value `Epsn` and redefine the meaning of `Ok`. The meanings of the tag values are now satisfyingly symmetrical:

- `Epsn`: the parser succeeded without consuming any tokens.
- `Ok`: the parser succeeded after consuming at least one token.
- `Fail`: the parser failed without consuming any tokens.
- `Err`: the parser failed after consuming at least one token.

Table 2 shows the abstract behaviour of `++` and `bind` using the four tag values. The requirement to respect the definitions of the tag values completely defines the behaviour of the `bind` combinator. Note particularly two points about the behaviour of `bind`. First, even if both parses in a sequence are successful, `bind` only returns `Ok` if at least one of the two parsers consumed some input, otherwise returns `Epsn`. Second, if the first parse returns `Epsn` and the second parse returns `Fail`, the result is `Fail` rather than `Err`.

The `++` combinator is not so straightforward, since the definitions of the tag values do not completely constrain its behaviour. We therefore introduce the following

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

Table 2. *Behaviour of the* ++ *and* bind *combinators with four values*

a	b	a 'bind' (_ -> b)	a ++ b
Err	Err	Err	Err
Err	Fail	Err	Err
Err	Epsn	Err	Err
Err	Ok	Err	Err
Fail	Err	Fail	Err
Fail	Fail	Fail	Fail
Fail	Epsn	Fail	Epsn
Fail	Ok	Fail	Ok
Epsn	Err	Err	Err
Epsn	Fail	Fail	Epsn
Epsn	Epsn	Epsn	Epsn
Epsn	Ok	Ok	Ok
Ok	Err	Err	Ok
Ok	Fail	Err	Ok
Ok	Epsn	Ok	Ok
Ok	Ok	Ok	Ok

desirable properties for ++. These properties, in conjunction with the definitions of the tag values, completely constrain the behaviour of ++:

- *associativity*, so that `(a ++ b) ++ c` is equivalent to `a ++ (b ++ c)`.
- `Ok` *is a left zero*, to save ++ from looking at the right parse when the left parse has consumed some input.
- `Fail` *is a left and right identity*, so that if one alternative fails to match anything at all, ++ will return the other alternative.
- `Err` *is a left zero*, to guarantee that the longest possible parse is returned.
- `(Epsn ++ Ok) = Ok` *and* `(Epsn ++ Err) = Err`, so that `Epsn` no longer needs to appear at the rightmost position in a tree of ++'s.

The behaviour of ++ in Table 2 satisfies all of these properties, as well as satisfying the definitions of the four tag values.

Note that if an LL(1) grammar is assumed, the cases `(Ok ++ Ok)`, `(Ok ++ Err)`, `(Err ++ Ok)`, and `(Err ++ Err)` cannot arise. The definitions of these cases in the table are consistent with lazy evaluation of the right argument of ++. For example, `(Ok ++ x)` is defined to be `Ok` for all `x`, including `x` equal to `Ok` and to `Err`. Similarly, `(Err ++ x)` is defined to be `Err` for all `x`. This enables the implementation to avoid evaluation that would only be necessary to check that the grammar is indeed LL(1).

We now give Gofer code to implement the combinators and primitive parsers using the four-value scheme:

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

instance Functor (Parser a) where
   map f p = Parser (\inp ->
                case getParser p inp of
                   Err e          -> Err e
```

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

```
                    Fail e          -> Fail e
                    Epsn (x, inp1) -> Epsn (f x, inp1)
                    Ok (x, inp1)   -> Ok (f x, inp1))

instance Monad (Parser a) where
   p1 `bind` p2
      = Parser (\inp ->
           case getParser p1 inp of
              Err e1          -> Err e1
              Fail e1         -> Fail e1
              Epsn (x, inp1) ->
                 case getParser (p2 x) inp1 of
                    Err e2          -> Err e2
                    Fail e2         -> Fail e2
                    Epsn (y, inp2) -> Epsn (y, inp2)
                    Ok (y, inp2)   -> Ok (y, inp2)
              Ok (x, inp1)   ->
                 case getParser (p2 x) inp1 of
                    Err e2          -> Err e2
                    Fail e2         -> Err e2
                    Epsn (y, inp2) -> Ok (y, inp2)
                    Ok (y, inp2)   -> Ok (y, inp2))

   result x = Parser (\inp -> Epsn (x, inp))

instance MonadPlus (Parser a) where
   p1 ++ p2
     = Parser (\inp ->
          case getParser p1 inp of
             Err e  -> Err e
             Fail e -> getParser p2 inp
             Epsn x ->
                case getParser p2 inp of
                   Err e2  -> Err e2
                   Fail e2 -> Epsn x
                   Epsn y  -> Epsn y
                   Ok y    -> Ok y
             Ok x   -> Ok x)
```

The functions `lit` and `zero` are the same as their two-value versions.

We now prove that parsers constructed from the four-value combinators defined above return the longest parse. If there is an error, they will therefore pinpoint it as occurring at the end of the longest parse.

Lemma 6.1

The functions `++`, `bind`, `result` and `lit` respect the definitions of the tag values.

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

Proof
By case analysis of the code. Details omitted. □

Theorem 6.1
Parsers for LL(1) grammars constructed from the four-value combinators defined above return the longest parse.

Proof
By induction on the structure of parsers, as follows:

Basis cases: If the grammar is either `lit t` or `result v`, the result holds by Lemma 6.1.

Inductive assumption: The theorem holds for parsers `a` and `b`.

Inductive cases: If the grammar is `(a 'bind' (\x -> b))`, then the result follows from the inductive assumption and Lemma 6.1.

If the grammar is `(a ++ b)`, then by cases on the result of `a`:

- If `a` partially matches, then by Lemma 6.1 it returns either `Ok` or `Err`. Examination of the code for `++` shows that `(a ++ b)` returns the result from `a` in this case. Because the grammar is LL(1) and `a` partially matches, `b` will not match at all, so `a` is the longer parse. Hence `(a ++ b)` returns the result from the longer parse in this case.
- If `a` does not match at all, then by Lemma 6.1 it returns either `Epsn` or `Fail`. Here, `b` may match, partially match, or not match at all. Hence, the parse returned by `b` is at least as long as the parse returned by `a`. Examination of the code for `++` shows that `(a ++ b)` mostly returns the result from `b` in this case. The only exception is that the result from `a` is returned when `a` returns `Epsn` and `b` returns `Fail`, in which case both parses are equally long.

□

Finally, Table 3 shows the results for various inputs when the parsers `ex1` and `ex2` (below) are implemented using the two-value and four-value combinators defined in this paper.

```
ex2 = do {a; b} ++ epsilon
      where epsilon = result 'e'

a = lit 'a'; b = lit 'b'; c = lit 'c'
```

7 Conclusion

In traditional parser construction, bottom-up approaches are used. This has the benefits of improved efficiency and the ability to recover from errors. The grammar transformations necessary are often complicated, but they can be done automatically using tools such as `yacc` (Aho *et al.*, 1986).

https://doi.org/10.1017/S0956796800001714 Published online by Cambridge University Press

Table 3. *Comparing* `ex1` *and* `ex2` *with two-value and four-value combinators*

Input	ex1 (2 values)	ex1 (4 values)	ex2 (2 values)	ex2 (4 values)
"ab"	Ok ('b',[])	Ok ('b',[])	Ok ('b',[])	Ok ('b',[])
"ad"	Fail 'a'	Err 'd'	Ok ('e',"ad")	Err 'd'
"c"	Ok ('c',[])	Ok ('c',[])	Ok ('e',"c")	Epsn ('e',"c")
"f"	Fail 'f'	Fail 'f'	Ok ('e',"f")	Epsn ('e',"f")

One advantage of parser combinators over the traditional techniques is that they can be used with very little infrastructure: there is no need to buy, build, maintain or understand any complex compiler-compiler tools. The necessary grammar transformations can be performed by hand, or by using *parser transformers* (Fokker, 1995).

Previous sets of parser combinators are not very good at pinpointing the location of parse errors, making them unsuitable for building programming language parsers. This paper has shown a method for producing combinator-based parsers that automatically report the first token following the longest parse of the input as the cause of a parse error. The method uses parsers that return values (parse trees or error indications) decorated with one of four tags. The resulting parsers do not try alternatives unnecessarily, making them more efficient than parsers built using two-value combinators. Another small benefit of the four-value method is that, because the tag values returned by the parser combinators are determined solely by the tag values returned by the argument parsers, it is possible to code the combinators using continuation-passing style (Appel, 1992).

Acknowledgements

Thanks to the anonymous reviewers for carefully parsing the paper and reporting errors and helpful comments.

References

Aho, A. V., Sethi, R. and Ullman, J. D. (1986) *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

Appel, A. W. (1992) *Compiling with continuations*.. Cambridge University Press.

Fokker, J. (1995) Functional Parsers. In *Advanced Functional Programming: Lecture Notes in Computer Science, 925*, J. Jeuring and E. Meijer (editors). Springer-Verlag.

Hutton, G. (1992) Higher-order functions for parsing. *Journal of Functional Programming*, **2**(3): 323–343.

Jones, M. (1994) *Gofer 2.30 release notes*, `http://www.cs.nott.ac.uk:80/Department/Staff/mpj/`.

Lewis, P. M. II and Stearns, R. E. (1968) Syntax-directed transduction. *J. ACM*, **15**(3): 465–488.

Wadler, P. (1985) How to Replace Failure by a List of Successes. *Functional Programming Languages and Computer Architecture: Lecture Notes in Computer Science, 201*, pp. 113–128. Springer-Verlag.

Wadler, P. (1990) Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pp. 61–78. ACM.