Exploring Languages with Interpreters and Functional Programming Chapter 45

H. Conrad Cunningham

04 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

45 Parsing Combinators

45.1 Chapter Introduction

TODO

45.2 Developing Parsing Combinators

In Chapter 44, we examined a set of prototype parsing functions and then used them as patterns for hand-coding of recursive descent parsing functions. We can benefit by generalizing these functions and collecting them into a library.

45.2.1 State actions and combinators

Consider `parseS`, one of the prototype parsing functions from a previous section. It parses the grammar rule `S ::= A | B`, which has two alternatives.

```
parseS :: String -> (Bool,String)
parseS xs =
    case parseA xs of                 -- try A
        (True,  ys) -> (True, ys)   -- A succeeds
        (False, _ ) ->
            case parseB xs of         -- else try B
                (True, ys) -> (True,  ys) -- B succeeds
                (False, _) -> (False, xs) -- both A,B fail
```

Note that `parseS` and the other prototype parsing functions have the type:

```
String -> (Bool,String)
```

The occurrence of type `String` in the argument of the function represents the *state* of the input before evaluation of the function; the second occurrence of `String` represents the state after evaluation. The type `Bool` represents the *result* of the evaluation.

In an imperative program, the state is often left implicit and only the result type is returned. However, in a purely functional program, we must also make both the state change explicit.

Functions that have a type similar to `parseS` are called *state actions* or *state transitions*. We can generalize this parsing state transition as a function type:

```
type Parser a b = a -> (b,a)
```

In the case of `parseS`, we specialize this to:

```
Parser String Bool
```

In the case of richer parsing case studies for the prefix and infix parsers, we specialize this type as:

```
Parser [Token] (Either ErrMsg Expr)
```

Given the `Parser` type, we can define a set of *combinators* that allow us to combine simpler parsers to construct more complex parsers. These combinators can pass along the state implicitly, avoiding some tedious and repetitive work.

We can define a combinator `parseAlt` that generalizes the `parseS` prototype function above. It implements a recognizer, so we fix type `b` to `Bool`, but leave type argument `a` general.

```
parseAlt :: Parser a Bool -> Parser a Bool -> Parser a Bool
parseAlt p1 p2 =
    \xs ->
        case p1 xs of
            (True,  ys) -> (True, ys)
            (False, _ ) ->
                case p2 xs of
                    (True,  ys) -> (True,  ys)
                    (False, _ ) -> (False, xs)
```

Note the use of the anonymous function in the body. Function `parseAlt` takes two `Parser` values and then returns a `Parser` value. The `Parser` function returned binds in the two component function values. When this function is applied to the parser input (which is the argument of the anonymous function), it applies the two component parsers as needed.

We can easily redefine `parseS` in terms of the `parseAlt` combinator and simpler parsers `parseA` and `parseB`.

```
parseS = parseAlt parseA parseB
```

Given parsing input `inp`, we can invoke the parser with the expression:

```
parseS inp
```

Note that this formulation enables us to handle the passing of state among the component parsers implicitly, much as we can in an imperative computation. But it still preserves the nature of purely functional computation.

45.2.2 Completing a combinator library

Now consider the `parseA` prototype, which implements a two-component sequencing rule `A ::= C D`.

```
parseA xs =
    case parseC xs of    -- try C
        (True,  ys) ->   -- then try D
            case parseD ys of
                (True, zs) -> (True,  zs)   -- C D succeeds
                (False, _) -> (False, xs)   -- both C, D fail
        (False, _ ) -> (False,xs)   -- C fails
```

As with `parseS`, we can generalize `parseA` as a combinator `parseSeq`.

```
parseSeq :: Parser a Bool -> Parser a Bool -> Parser a Bool
parseSeq p1 p2 =
    \xs ->
        case p1 xs of
            (True,  ys) ->
                case p2 ys of
                    t@(True, zs) -> t
                    (False,  _ ) -> (False, xs)
            (False, _ ) -> (False,  xs)
```

Thus we can redefine `parseA` in terms of the `parseSeq` combinator and simpler parsers `parseC` and `parseD`.

```
parseA = parseSeq parseC parseD
```

Similarly, we consider the `parseB` prototype, which implements a repetition rule `B ::= { E }`.

```
parseB xs =
    case parseE xs of              -- try E
        (True,  ys) -> parseB ys   -- try again
        (False, ys) -> (True,xs)   -- stop
```

As above, we generalize this as combinator `parseStar`.

```
parseStar :: Parser a Bool -> Parser a Bool
parseStar p1 =
    \xs ->
        case p1 xs of
            (True,  ys) -> parseStar p1 ys
            (False, _ ) -> (True, xs)
```

We can redefine `parseB` in terms of combinator `parseStar` and simpler parser `parseE`.

```
parseB = parseStar parseB
```

Finally, consider parsing prototype `parseC`, which implements an optional rule `C ::= [ F ]`.

```
parseC xs =
   case parseF xs of               -- try F
        (True,  ys) -> (True,ys)
        (False, _ ) -> (True,xs)
```

We generalize this pattern as `parseOpt`, as follows.

```
parseOpt :: Parser a Bool -> Parser a Bool
parseOpt p1 =
    \xs ->
        case p1 xs of
```

```
            (True,  ys) -> (True, ys)
            (False, _ ) -> (True, xs)
```

We can thus redefine `parseC` in terms of simpler parser `parseF` and combinator `parseOpt`.

```
parseC = parseOpt parseF
```

In this simple example grammar, function `parseD` is a simple instance of a sequence and `parseE` and `parseF` are simple parsers for symbols. These can be directly implemented as basic parsers, as before. However, the technique work if these are more complex parsers built up from combinators.

For convenience and completeness, we include extended alternative and sequencing combinators and parsers that always fail or always succeed.

```
parseAltList :: [Parser a Bool] -> Parser a Bool
parseSeqList :: [Parser a Bool] -> Parser a Bool
parseFail, parseSucceed :: Parser a Bool
```

The combinators in this library are in the Haskell module ParserComb.hs. A module that does some testing is TestParserComb.hs.

TODO: Update and document the Parser Combinator library code.

45.2.3 Adding parse tree generations

TODO: Expand this library to allow returns of "parse trees" and error messages.

45.3 Standard libraries for parsing

TODO

There are a number of relatively standard parsing combinator libraries—e.g., the library Parsec. Readers who wish to develop other parsers may want to study that library.

45.4 Exercises

TODO

45.5 Chapter Source Code

TODO

45.6 Acknowledgements

For the general acknowledgements for the ELI Calculator case study and Chapters 41-46 through Spring 2019, see the Acknowledgements section of Chapter 41.

I developed the parsing combinators in this chapter primarily using the approach of Fowler and Parsons [2], with some influence by Chiusano and Bjarnason [1]. I generalized the concrete parsing functions from Chapter 44 to construct the combinators.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

45.7 Terms and Concepts

TODO

45.8 References

[1] Paul Chiusano and Runar Bjarnason. 2015. *Functional programming in Scala* (First ed.). Manning, Shelter Island, New York, USA.

[2] Martin Fowler and Rebecca Parsons. 2010. *Domain specific languages*. Addison-Wesley, Boston, Massachusetts, USA.