Exploring Languages with Interpreters and Functional Programming
Chapter 45

H. Conrad Cunningham

4 November 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of November 2018 is a recent version of Firefox from Mozilla.

45 Parsing Combinators

45.1 Chapter Introduction

TODO

45.2 Developint 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)

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 Acknowledgements

I initially developed this prototype set of parsing combinators in Summer 2017 as a part of what is now called the ELI Calculator language case study. I based this case study, in part, on ideas from:

- the Scala-based Expression Tree Calculator case study from my *Notes on Scala for Java Programmers* [Cunningham 2018]: (which was itself adapted from the tutorial Scala for Java Programmers [Schinz 2017] in Spring 2016)
- the Lua-based Expression Language 1 and Imperative Core interpreters I developed for the Fall 2016 CSci 450 course
- Samuel N. Kamin's textbook *Programming Languages: An Interpreter-based Approach*, Addison Wesley, 1990 [Kamin 1990] and my work to implement three (Core, Lisp, and Scheme) of these interpeters in Lua in 2013
- the Wikipedia articles on Regular Grammar, Context-Free Grammar, Backus-Naur Form, Lexical Analysis, Parsing, LL Parser, Recursive Descent Parser, and Abstract Syntax [Wikipedia 2018, 2018b].
- Chapter 21 (Recursive Descent Parser) of the Martin Fowler and Rebecca Parsons's book *Domain-Specific Languages*, Addison Wesley, 2011 [Fowler 2011].
- Section 3.2 (Predictive Parsing) of Andrew W. Appel's textbook *Modern Compiler Implementation in ML*, Cambridge, 1998
  [Appel 1998].
- Chapters 6 (Purely Functional State) and 9 (Parser Combinators) from Paul Chiusano and Runar Bjarnason's *Functional Programming in Scala*, Manning, 2015.

The parsing combinators were built primarily on the approach of [Fowler 2011], with influences by [Chiusano 2015].

In Fall 2018, I divided the 2017 Expression Language Parsing chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Sections 11.1-11.4 became Chapter 44, Calculator Parsing, and sections 11.6-11.7 became Chapter 45, Parsing Combinators (this chapter). I merged Section 11.5 into Chapter 43, a new chapter on the modular structure of the ELI Calculator language interpreter.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed. The HTML version of this document may require use of a browser that supports the display of MathML.

45.6 References

[Appel 1998]: Andrew W. Appel. *Modern Compiler Implementation in ML*, Cambridge, 1998. (Especially section 3.2 "Predictive Parsing")

[Chiusano 2015]: Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015. (Especially chapters 6 "Purely Functional State" and 9 "Parser Combinators")

[Cunningham 2018]: H. Conrad Cunningham. *Notes on Scala for Java Programmers*, 2018 (which is itself adapted from the tutorial [Schinz 2018] Scala for Java Programmers

[Fowler 2011]: Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*, Addison Wesley, 2011. (Especially chapter 21 "Recursive Descent Parser")

[Kamin 1990] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

[Linz 2017] Peter Linz. *An Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett, 2017. (Especially sections 1.2, 3.3, and 5.1)

[Schinz 2017]: Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers, accessed February 2016.

[Sestoft 2012]: Peter Sestoft. *Programming Language Concepts*, Springer, 2012. (Especially sections 1.3, 2.5, and 2.7 and chapter 8)

[Wikipedia 2018a]: Wikipedia. Articles on Formal Grammar, Regular Grammar, Context-Free Grammar, Backus-Naur Form, Extended Backus-Naur Form, and Parsing. Accessed 9 August 2018.

[Wikipedia 2018b]: Wikipedia. Articles on Abstract Syntax and Associative Array, Accessed 9 August 2018.

45.7 Terms and Concepts

TODO