Exploring Languages with Interpreters
and Functional Programming
Chapter 45

H. Conrad Cunningham

4 November 2018

Contents

45 Parsing Combinators .................................................. 2
  45.1 Chapter Introduction .................................... 2
  45.2 Developint Parsing Combinators ....................... 2
    45.2.1 State actions and combinators .................... 2
    45.2.2 Completing a combinator library ................ 3
    45.2.3 Adding parse tree generations (TODO) ........... 5
  45.3 Standard libraries for parsing (TODO) ................. 5
  45.4 Exercises ..................................................... 5
  45.5 Acknowledgements ......................................... 6
  45.6 References .................................................... 7
  45.7 Terms and Concepts ....................................... 7

Copyright (C) 2017, 2018, H. Conrad Cunningham
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 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 \( S ::= A \mid B \), which has two alternatives.

\[
\text{parseS :: String} \to (\text{Bool}, \text{String})
\]

\[
\text{parseS xs =}
\]

\[
\begin{align*}
\text{case parseA xs of} & \quad \text{-- try A} \\
(\text{True}, \ ys) & \to (\text{True}, \ ys) \quad \text{-- A succeeds} \\
(\text{False}, \ _ ) & \to
\end{align*}
\]

\[
\begin{align*}
\text{case parseB xs of} & \quad \text{-- else try B} \\
(\text{True}, \ ys) & \to (\text{True}, \ ys) \quad \text{-- B succeeds} \\
(\text{False}, \ _ ) & \to (\text{False}, \ xs) \quad \text{-- both A,B fail}
\end{align*}
\]

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

\[
\text{String} \to (\text{Bool}, \text{String})
\]

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:

\[
\text{type Parser a b} = \text{a} \to (\text{b}, \text{a})
\]

In the case of parseS, we specialize this to:

\[
\text{Parser String Bool}
\]
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
```
As with \texttt{parseS}, we can generalize \texttt{parseA} as a combinator \texttt{parseSeq}.

\begin{verbatim}
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)
\end{verbatim}

Thus we can redefine \texttt{parseA} in terms of the \texttt{parseSeq} combinator and simpler parsers \texttt{parseC} and \texttt{parseD}.

\begin{verbatim}
parseA = parseSeq parseC parseD
\end{verbatim}

Similarly, we consider the \texttt{parseB} prototype, which implements a repetition rule \( B ::= \{ E \} \).

\begin{verbatim}
parseB xs =
    case parseE xs of
        -- try E
        (True, ys) -> parseB ys -- try again
        (False, ys) -> (True, xs) -- stop
\end{verbatim}

As above, we generalize this as combinator \texttt{parseStar}.

\begin{verbatim}
parseStar :: Parser a Bool -> Parser a Bool
parseStar p1 xs ->
    case p1 xs of
        (True, ys) -> parseStar p1 ys
        (False, _ ) -> (True, xs)
\end{verbatim}

We can redefine \texttt{parseB} in terms of combinator \texttt{parseStar} and simpler parser \texttt{parseE}.

\begin{verbatim}
parseB = parseStar parseB
\end{verbatim}

Finally, consider parsing prototype \texttt{parseC}, which implements an optional rule \( C ::= \[ F \] \).

\begin{verbatim}
parseC xs =
    case parseF xs of
        -- try F
        (True, ys) -> (True,ys)
        (False, _ ) -> (True,xs)
\end{verbatim}
We generalize this pattern as \texttt{parseOpt}, as follows.

\[
\texttt{parseOpt} :: \texttt{Parser a Bool} \rightarrow \texttt{Parser a Bool}
\]

\[
\texttt{parseOpt p1} =
\texttt{\( \lambda x s \rightarrow \case \) p1 xs of}
\]

\[
\texttt{(True, y s) \rightarrow (True, y s)}
\]

\[
\texttt{(False, _) \rightarrow (True, x s)}
\]

We can thus redefine \texttt{parseC} in terms of simpler parser \texttt{parseF} and combinator \texttt{parseOpt}.

\[
\texttt{parseC = parseOpt parseF}
\]

In this simple example grammar, function \texttt{parseD} is a simple instance of a sequence and \texttt{parseE} and \texttt{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.

\[
\texttt{parseAltList} :: [\texttt{Parser a Bool}] \rightarrow \texttt{Parser a Bool}
\]

\[
\texttt{parseSeqList} :: [\texttt{Parser a Bool}] \rightarrow \texttt{Parser a Bool}
\]

\[
\texttt{parseFail, parseSucceed :: Parser a Bool}
\]

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

TODO: Update and document the Parser Combinator library code.

\textbf{45.2.3 Adding parse tree generations (TODO)}

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

\textbf{45.3 Standard libraries for parsing (TODO)}

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

\textbf{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 interpreters in Lua in 2013


- Chapter 21 (Recursive Descent Parser) of the Martin Fowler and Rebecca Parsons's book Domain-Specific Languages, Addison Wesley, 2011 [Fowler 2011].


- 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


[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)


45.7 Terms and Concepts

TODO