Ramble: A Parser Combinator in R

Chapman Siu

9 May 2015

Abstract

This paper presents Ramble, a basic parser combinator written in the R language. Combinator parsing is a parser which resembles BNF notation. Here we present a simple way which such a parser can be built and used within the R language. The R language has first class functions and this is heavily used within Ramble to build up and combine simple parsers into more complex parsers.

1 Introduction

Combinator parsing differs from tools like Lex and Yacc due to its lazy nature. Large parsers are instead build piecewise from smaller parsers using high order functions. For example, we will define higher order functions for sequencing, alternation and repetition. Parsers of this nature are quick to build, understand and modify.

Semantic actions can be added to parsers, allowing actions to be manipulated at will. Many of the concepts explored within R and Ramble are mirrored closely by examples provided by Hutton [1992].

2 Parsing Using Combinators

A parser is simply a function from a string of symboles to a result value. Since a parser might not consume the whole string, part of this result will be the suffix for the input string. Other times a parser may not produce a result at all. This may happen when the parser expects a digit, but receives a letter instead. When this happens, rather than having the parser fail immediately, we instead use the empty list `list()` to indicate failure, and a list with values "result" and "leftover" `list(result, leftover)` ; where "result" is parsed value and the unconsumed input is "leftover".

2.1 Primitive Parsers

When using combinatory parsers, we build from the simple building blocks first, before creating more complex parsers. The first of these parsers are the primitive parsers.

```
succeed <- function(string) {
  return(function(nextString) {
```

```
    return(list(result = string, leftover=nextString))
  })
}
```

The succeed parser always succeeds, and does not actually consume any output. We can say that the value of this parser is infact predetermined. For example `succeed("1") ("abc")` will always return the value 1.

The next parser is the item parser. This will consume a single character of input if possible and return the rest of the string.

```
item <- function(...){
  return(function(string){
    if(length(string)==0){return(NULL)}
    return (if(string=="") list() else list(result=substr(string, 1, 1),
                                            leftover=substring(string, 2)))
  })
}
```

For example `item("123")` will return 1.

The next parser allows us to recognise single symbols. Rather than providing a separate methods to recognize individual characters, it is more convenient to write a simpler parser via a predicate to determine if arbitrary symbols are a member of the set.

```
satisfy <- function(p) {
  return(function(string) {
    if (length(string)==0) {
      return(list())
    }
    else if (string==""){
      return(list())
    }
    else {
      result_ = list(result=substr(string, 1, 1),
                     leftover=substring(string, 2))
      if (p(result_$result)) {
        return(succeed(result_$result)(result_$leftover))
      }
      else{
        return(list())
      }
    }
  })
}
```

This parser can be trivially extended to allow for literal characters to be parsed. And hence we would have our `literal` parser.

```
literal <- function(char) {
  satisfy(function(x){return(x==char)})
}
```

Here we can see how both the `literal` and `satisfy` parsers work and interact. If we were to apply the parser, `literal("a") ("abc")` this would succeed with the result `list(result="a", "abc")`. This is a simple example how we extend the parser in a variety of ways.

2.2 Combinators

Now that the primitive parsers are complete, we can now approach how they may be combined in a manner similiar to the BNF notation. The `alt` combinator corresponds to alterntive in BNF. The approach taken here is based on Fairbairn [1986]; recognising as either sequential manner, returning the result of the first parser to succeed and failure if neither does.

```
alt <- function(p1, p2) {
  return(function(string){
    result <- p1 (string)
    if(!is.null(result$leftover)) {return(result)}
    else{
      return(p2 (string))
    }
  })
}
```

We can then define the combinator which recognises sequential grammar, analogous to sequencing in BNF.

```
then <- function(p1, p2) {
  return(function(string) {
    result <- p1 (string)
    if (length(result) == 0) {
      return (list())
    }
    else {
      result_ <- p2 (result$leftover)
      if (length(result_$leftover) == 0 || is.null(result_$leftover)) {
        return(list())
      }
      return(list(result=Unlist(append(list(result$result),
             list(result_$result))),
             leftover=result_$leftover))
    }
  })
}
```

Where the Unlist function is provided as follows:

```
Unlist <- function(a.list) {
  hasLowerLevel = TRUE
  while(hasLowerLevel) {
    a.list1 <- unlist(a.list, recursive=FALSE, use.names=FALSE)
    if (is(a.list1, 'list')) {
```

```
      a.list <- a.list1
    }
    else {
      hasLowerLevel = FALSE
      return(a.list)
    }
  }
  return(a.list)
}
```

The difference between this function and the `unlist` function in R is that it does not recurse all the way to the bottom and still preserves type of the variables, which is important in this context.

For both of the `alt` and `then` parsers, can be altered to be in the binary operand form in R by using the following

```
`%alt%` <- alt
`%then%` <- then
```

This will then allow the follwoing code to work as intended, `(item() %then% item()) ("abc")`, to produce `list(result=c("a", "b"), leftover=c("c"))`.

2.3 Manipulating Values

Actions of a parser can be added in parsing, allowing arbitrary semantic actions, such as type conversion on the fly. This makes it easy to define new parsers or terms through using our primitives.

```
using <- function(p, f) {
  return(function(string) {
    result <- p (string)
    if(length(result) == 0) {return(list())}
    return(list(result=f(result$result),
                leftover=result$leftover))
  })
}
```

In BNF notation, repetition occurs often enought to merit its own notation. Here we will have the equivalent notation for many and some, which represents zero or more occurences and one or more occurences of the same predicate.

```
many <- function(p) {
  return(function(string) {
    ((p %then% many(p)) %alt% succeed(NULL)) (string)
  })
}

some <- function(p) {
  return(function(string){
    (p %then% many(p)) (string)
  })
}
```

And from here we can construct many more complex parsers such as parsing a string.

```
String <- function(string) {
  if (string=="") {
    return (succeed(NULL))
  }
  else {
    result_=substr(string, 1, 1)
    leftover_=substring(string, 2)
    return((literal(result_) %then%
              String(leftover_)) %using%
               function(x) {paste(unlist(c(x)), collapse="")})
  }
}
```

Where running `String ("123") ("123 abc")` would return `123` as expected.

2.4 An Example

To conclude this introduction to combinatory parsing using R, we will work through a short example on a program which work with airthmetic expressions. To understand how this would function, consider first parsing integers:

```
Digit <- function(...) {satisfy(function(x) {return(!!length(grep("[0-9]", x)))})}

nat <- function() {
  some(Digit()) %using%
    function(x) {paste(unlist(c(x)), collapse="")}
}

token <- function(p) {
  space() %then%
    p %then%
    space() %using%
    function(x) {return(unlist(c(x))[2])}
}

natural <- function(...) {token(nat())}
```

Firstly, we can understand the concept of a digit, which is then parsed as one or more and coerced together, and finally checked when padded with zeros. This is what the combination of these parsers actually perform. Next to understand how to parse arithmetic expressions, we know that the order of operations is brackets, then multiplication and division and finally addition and subtraction. This can be expressed in BNF notation as follows:

```
expr ::= Num num | expr $add expr | expr $sub expr
                 | expr $mul expr | expr $div expr
```

Then this expression can be expanded a little more and form the order of precedence to assist with combinatory parsing:

```
expr ::= term + term | term - term | term
term ::= factor * factor | factor / factor | factor
factor ::= digit+ | (expr)
```

which allows for bracket expressions. So then these statements can be written in our parser combinator language to parse arithmetic expressions.

```
expr <- ((term %then%
            symbol("+") %then%
            expr %using% function(x) {
              print(unlist(c(x)))
              return(sum(as.numeric(unlist(c(x))[c(1,3)])))
            }) %alt%
           (term %then%
              symbol("-") %then%
              expr %using% function(x) {
                print(unlist(c(x)))
                return(Reduce("-", as.numeric(unlist(c(x))[c(1,3)])))
              }) %alt% term)
term <- ((factor %then%
             symbol("*") %then%
             term %using% function(x) {
               print(unlist(c(x)))
               return(prod(as.numeric(unlist(c(x))[c(1,3)])))
             }) %alt%
           (factor %then%
              symbol("/") %then%
              term %using% function(x) {
                print(unlist(c(x)))
                return(Reduce("/", as.numeric(unlist(c(x))[c(1,3)])))
              }) %alt% factor)
factor <- ((
    symbol("(") %then%
      expr %then%
      symbol(")") %using%
      function(x){
        print(unlist(c(x)))
        return(as.numeric(unlist(c(x))[2]))
        })
    %alt% natural())
```

Which will yield the appropriate results. For example

```
> expr("(1+1)*2")
[1] "1" "+" "1"
[1] "(" "2" ")"
[1] "2" "*" "2"
[1] "1" "+" "1"
[1] "(" "2" ")"
[1] "2" "*" "2"
[1] "1" "+" "1"
```

```
[1] "(" "2" ")"
[1] "2" "*" "2"
$result
[1] 4
$leftover
[1] ""
```

References

J. Fairbairn. Making form follow function. *University of Cambridge Computer Laboratory*, Technical Report 89, 1986.

G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2, 1992.