Chapter 1

Introduction to pegR

PegR is an R based tool for writing and executing parsing expression grammers. The steps are

1. Create a peg object to hold the peg grammer
2. Add/edit a some rules
3. Add/edit some actions
4. Apply the rules to some text (with or without using actions)
5. Keep repeating steps 2 and 3 until you get it right

Chapter 2

Getting Started

2.1 A Simple Rule

After installing, the first thing to do is to load the pegR library:

```
library(pegr)
```

Next we need to create a new peg parser object

```
peg <- new.parser()
```

Now we define a rule, which we label A to accept a single charcter, the letter a.

$$A \leftarrow' a' \tag{2.1}$$

The portion on the left hand side of the $\leftarrow$, is the called the *rule identifier*. It may also be called the *rule name* or the *rule label*. The portion on the righthand side of the $\leftarrow$ is called the *rule definition*. In this case there is a quoted string consisting of the letter 'a'. The quoted string is called a literal. The rule says when presented some text to process: accept and consume the quoted string.

Adding this rule to our peg can be done in one of two ways:

```
peg + "A<-'a'"

##
## Rule: A
## Def: A<-'a'
## Com:
## Act:
```

or

```
add_rule(peg, "A<-'a'")
```

Once the rule is added, we may parse by applying it to some input text. Again there are two equivalent ways to do this:

```
result <- peg[["A"]]("a")
```

or

```
result <- apply_rule(peg, "A", "a")
```

Printing the result we see 3 components:

```
result

## Call: Rule=A; Input Arg="a"
## Status: TRUE
## Consumed: ( 1 / 1 )
## Evaluates to: list( a )
```

1. *Status:* True means the rule accepted the input
2. *Consumed:* Tells us how many characters were consumed
3. *Value:* Give us a list containing values computed during the parser (We will talk more about this when we introduce *actions*.)

To access these components directly we use

```
status(result)

## [1] TRUE
```

```
consumed(result)

## [1] "a"
```

```
value(result)

## $atom
## [1] "a"
```

Chapter 3

Rule Sequences

3.1 Simple Sequences

By the term *sequence*, we mean a sequential applicaton of rules. As an example, consider 3 rules:

$$\begin{array}{lcl} A & \leftarrow & 'a' \\ B & \leftarrow & 'b' \\ C & \leftarrow & 'c' \end{array} \tag{3.1}$$

The first form of rule adding rules to the parser allows us to string the rules together as follows:

```
peg <- new.parser()
peg + "A<-'a'" + "B<-'b'" + "C<-'c'"

##
## Rule: A
## Def: A<-'a'
## Com:
## Act:
##
## Rule: B
## Def: B<-'b'
## Com:
## Act:
##
## Rule: C
## Def: C<-'c'
## Com:
## Act:
```

Now to form a rule which looks for the leter *a* followed immediately by *b* and then *c*, we add

```
peg + "ABC<- A B C"

##
## Rule: A
## Def: A<-'a'
## Com:
## Act:
##
## Rule: ABC
## Def: ABC<- A B C
## Com:
## Act:
##
## Rule: B
## Def: B<-'b'
## Com:
## Act:
##
## Rule: C
## Def: C<-'c'
## Com:
## Act:
```

Here the space between the symbols A, B, C on the right hand side indicate that we first must satisfy A, then B, the C. Now evaluating on 'abc' we see

```
peg[["ABC"]]("abc")

## Call: Rule=ABC; Input Arg="abc"
## Status: TRUE
## Consumed: ( 3 / 3 )
## Evaluates to: list( a,b,c )
```

accepts and consumes 'abc'. However note,

```
peg[["ABC"]]("abcx")

## Call: Rule=ABC; Input Arg="abcx"
## Status: TRUE
## Consumed: ( 3 / 4 )
## Evaluates to: list( a,b,c )
```

is also accepted (status =TRUE), but only 3 out 4 characters are consumed.

3.2 Look-Ahead Operators

The reason for allowing to accept without having the entire imput consumed for the look-ahead operators *!* and *&*.

The operator *!* means "not followed by", *&* means followed by. For example consider

```
peg <- new.parser()
peg + "R1<-'a' 'b' !'c'" + "R2<- 'a' 'b' &'c'"

##
## Rule: R1
## Def: R1<-'a' 'b' !'c'
## Com:
## Act:
##
## Rule: R2
## Def: R2<- 'a' 'b' &'c'
## Com:
## Act:
```

Then

```
status(peg[["R1"]]("abc"))

## [1] FALSE
```

returns FALSE (is rejected) but

```
status(peg[["R1"]]("ab31"))

## [1] TRUE
```

returns TRUE and consumes

```
consumed(peg[["R1"]]("ab31"))

## [1] "ab"
```

2 charscters (the 'ab' portion) Likewise

```
status(peg[["R2"]]("abc"))

## [1] TRUE
```

is rejected but

```
status(peg[["R2"]]("ab"))

## [1] FALSE
```

is accepted

Chapter 4

Repetitions, Wild Cards and parenthesis

Repetitions, wild cards and grouping are included to make the rule specificaton easier.

4.1 *

The * allows of 0 or more repetitions of the preceding. For example

```
peg <- new.parser()
peg + "A<- 'a' 'a'*"

##
## Rule: A
## Def: A<- 'a' 'a'*
## Com:
## Act:

status(peg[["A"]]("aaaaa"))

## [1] TRUE
```

accepts any string beginning with consecutive a's

4.2 ?

However, if we only want to accept if the string begins with 1 or 2 a' we may use the optional operator:

```
peg <- new.parser()
peg + "A<- 'a' 'a'?"

##
## Rule: A
## Def: A<- 'a' 'a'?
## Com:
## Act:

status(peg[["A"]]("aaaaa"))

## [1] TRUE
```

This will reject if there 3 or more a's

4.3 .

A single dot will match any single character, so to accept a string with length 1 or 2 we might use

```
peg <- new.parser()
peg + "A<- . .?"

##
## Rule: A
## Def: A<- . .?
## Com:
## Act:

status(peg[["A"]]("xy"))

## [1] TRUE
```

4.4 Ranges

Ranges are specified by the usual bracket dash notation, for example consider

```
peg <- new.parser()
peg + "INT <- [0-9]+"

##
## Rule: INT
## Def: INT <- [0-9]+
## Com:
## Act:
```

This will accept all integers.

4.5 Exercises

1. 1 What does the rule

$$R < -'a' \ 'b' \ 'c' \ !. \tag{4.1}$$

   do? How does this differ from

$$R < -'a' \ 'b' \ 'c' \tag{4.2}$$

2. 2 Write a rule that accepts a string with exactly one letter x
3. 3 Write a rule that accepts decimal numbers.

Chapter 5

Prioritzed Choice

Priortized Choice examines each rule in turn and selects the first match. While we use blank spaces for sequencece, we ”/” for prioritzed choice.

Consider the following example:

```
peg <- new.parser()
peg + "X <- 'ab' / 'ac' / 'a'" + "Y <-'a' / 'ab' / 'ac'"

##
## Rule: X
## Def: X <- 'ab' / 'ac' / 'a'
## Com:
## Act:
##
## Rule: Y
## Def: Y <-'a' / 'ab' / 'ac'
## Com:
## Act:
```

Applying X to ’aaa’ we get

```
peg[["X"]]("aaa")  # status true; consumes 'a'

## Call: Rule=X; Input Arg="aaa"
## Status: TRUE
## Consumed: ( 1 / 3 )
## Evaluates to: list( a )
```

Applying X to ’abc’ we get

```
peg[["X"]]("abc")  # status true; consumes 'ab'

## Call: Rule=X; Input Arg="abc"
```

```
## Status: TRUE
## Consumed: ( 2 / 3 )
## Evaluates to: list( ab )
```

But applying Y to 'abc' we get

```
peg[["Y"]]("abc")  # status true; consumes 'a'

## Call: Rule=Y; Input Arg="abc"
## Status: TRUE
## Consumed: ( 1 / 3 )
## Evaluates to: list( a )
```

What happened? For Y, the rule 'a' precedes the rules for 'ab' and 'ba' so when an 'a' is encountered, the rule is satisfied and the single 'a' is consumed (and returned in the values of the list). Thus the order is critcal.

Again applying X to 'acb' we get

```
peg[["X"]]("acb")  #staus true; consumes 'ac'

## Call: Rule=X; Input Arg="acb"
## Status: TRUE
## Consumed: ( 2 / 3 )
## Evaluates to: list( ac )
```

But applying Y to to 'acb' again consumes only the 'a'

```
peg[["Y"]]("acb")  #staus true; consumes 'ac'

## Call: Rule=Y; Input Arg="acb"
## Status: TRUE
## Consumed: ( 1 / 3 )
## Evaluates to: list( a )
```

5.1 Exercises

1. What does the rule A ← 'a' 'b' / 'c' do?
2. What does the rule A ← 'a' ('b' / 'c') do?
3. What does the rule A ← 'a' / 'b' 'c' do?
4. What does the rule A ← ('a' / 'b') 'c' do?
5. What does the rule A ← A / 'a' / '' do?
6. What does the rule A ← '' / 'a' / A do?

7. Write a parser that rejects unbalance parenthesis

ChapterActions

Actions allow us to do tranformations on the values of rules. Some salient point about actions are:

1. Actions are attached to rules
2. Actions are functions that accept an list arg v and return a list
3. Actions can be added either as a function f(v) or as a string which is to represent the body of f(v)
4. To invoke an action, the exe flag should be set to TRUE

To illustrate an action consider the example to capitalize the letter 'a'. The action will only be performed on a, so we first make a rule for a and an action for that rule

```
fn <- function(v) {
    return(list("A"))
}
peg <- new.parser()
add_rule(peg, "A<-'a'", act = fn)
```

If we run this on a we see it capitalizes

```
add_rule(peg, "G<- (A / .)+")
res <- apply_rule(peg, "G", "bad wolf bay", exe = T)
paste1(value(res))

## [[1]]
## [1] "bAd wolf bAy"
```

Here we used the *paste1"* functions, which is analagous to *paste0* but operates on lists instead.

A simple example would be to capitalize all occurances of the letter a.

```
peg <- new.parser()
peg + c("A<- 'a'", act = "list('A')") + c("CAP<- A / .",
    act = "list(paste1(v))")

##
## Rule: A
## Def: A<- 'a'
## Com:
## Act: Inline: list('A')
##
## Rule: CAP
```

```
## Def: CAP<- A / .
## Com:
## Act: Inline: list(paste1(v))
```

Several things should be noted:

1. We put 'a' before . so that is A attempted first
2. We added an action as text to A (and CAP)
3. each rule is now a vector

5.2 Exercises

1. Write a parser that convert binary numbers to base 10
2. Write a parser to captilize the last letters of all words in a sentence

Chapter 6

Editing Rules and Adding Comments

Chapter 7

Helpful Tools and Hits

In this section we provide some tools and few hints we hope will prove useful.

- Printing Results:
- Graphing the Results
- Using qp

7.1 Printing the Results

Sometimes it is instructive to see how your peg grammar actually parsed a given text input, if nothing but to make sure it's doing what you intended it to do. This is easily done by setting the record flag to TRUE and printing. As our first example consider

```
peg <- new.parser()
peg + "A<-'a'" + "B<-'b'" + "C<-'c'" + "X<- A / B A/ C B A"

##
## Rule: A
## Def: A<-'a'
## Com:
## Act:
##
## Rule: B
## Def: B<-'b'
## Com:
## Act:
##
## Rule: C
## Def: C<-'c'
## Com:
```

```
## Act:
##
## Rule: X
## Def: X<- A / B A/ C B A
## Com:
## Act:
```

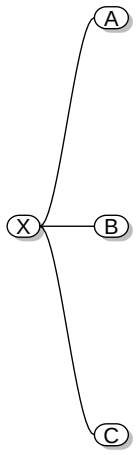
Then applying X to ”cba” we see

```
res <- peg[["X"]]("cba", record = TRUE)
res

## Call: Rule=X; Input Arg="cba" ; Options:
## Status: TRUE
## Consumed: ( 3 / 3 )
## Evaluates to: list( c,b,a )

tree(res)

## ____X(cba) = list(c, b, a )
##     |____C(c) = list(c )
##     |____B(b) = list(b )
##     |____A(a) = list(a )
```

7.2 Plotting the Results

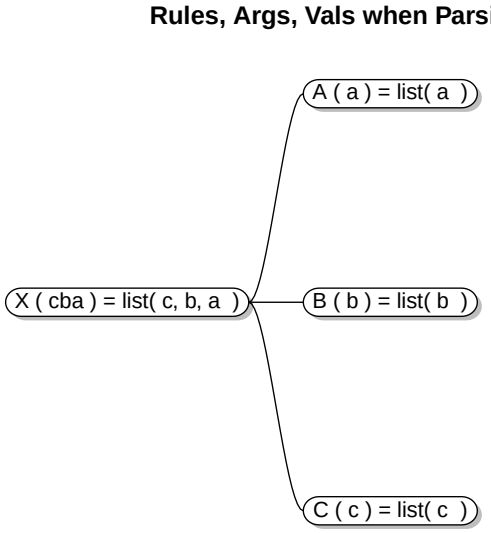
Plotting the same result we have

```
plot(res)
```



By default, only the rule id are shown, but we can get all availabe information by setting show flag to ”all”

```
plot(res, show = "all")
```



As a second example, consider the following calculator:

```
peg <- new.parser()
peg + "NUM<-('-')? [0-9]+" + "ATOM <- NUM / ( '(' & SUM & ')' )" +
    "DIV <- ATOM '/' PROD" + "MULT <- ATOM '*' PROD" +
    "PROD <- MULT / DIV / ATOM" + "SUB <- PROD '-' SUM" +
    "ADD <- PROD '+' SUM" + "SUM <- ADD / SUB /PROD"

##
## Rule: ADD
## Def: ADD <- PROD '+' SUM
## Com:
## Act:
##
## Rule: ATOM
## Def: ATOM <- NUM / ( '(' & SUM & ')' )
## Com:
```

```
## Act:
##
## Rule: DIV
## Def: DIV <- ATOM '/' PROD
## Com:
## Act:
##
## Rule: MULT
## Def: MULT <- ATOM '*' PROD
## Com:
## Act:
##
## Rule: NUM
## Def: NUM<-('-')? [0-9]+
## Com:
## Act:
##
## Rule: PROD
## Def: PROD <- MULT / DIV / ATOM
## Com:
## Act:
##
## Rule: SUB
## Def: SUB <- PROD '-' SUM
## Com:
## Act:
##
## Rule: SUM
## Def: SUM <- ADD / SUB /PROD
## Com:
## Act:

# next we set actions to the nodes
peg[["NUM"]] <- c(act = "list(as.numeric(paste(v,collapse='')))")
peg[["ADD"]] <- c(act = "list( v[[1]]+v[[3]] )")
peg[["SUB"]] <- c(act = "list( v[[1]] - v[[3]] )")
peg[["MULT"]] <- c(act = "list( v[[1]] * v[[3]] )")
peg[["DIV"]] <- c(act = "list( v[[1]] / v[[3]] )")
```

Here the root node for all calculations will be ”SUM”. Applying to 1+2*3 we see

```
res <- peg[["SUM"]]("1+2*3", record = TRUE)
value(res)[[1]]

## [1] "1"

tree(res)
```

```
## _____SUM(1+2*3) = list(1, +, 2, *, 3 )
##     |_____ADD(1+2*3) = list(1, +, 2, *, 3 )
##           |_____PROD(1) = list(1 )
##           |     |_____ATOM(1) = list(1 )
##           |           |_____NUM(1) = list(1 )
##           |_____SUM(2*3) = list(2, *, 3 )
##                 |_____PROD(2*3) = list(2, *, 3 )
##                       |_____MULT(2*3) = list(2, *, 3 )
##                             |_____ATOM(2) = list(2 )
##                             |     |_____NUM(2) = list(2 )
##                             |_____PROD(3) = list(3 )
##                                   |_____ATOM(3) = list(3 )
##                                         |_____NUM(3) = list(3 )
```

7.3 Qp

To help with tiny simple, expressions, there is a tool called qp, which allows one to parse a some simple expressions that contain only literals and connectives (i.e no labeled rules). This is most useful when first beginning to learn the PEG grammar.

7.4 Tips

Ok, your ready to create your grand grammar. But how to get started? Here are some tips:

- Rules first, get the rules right first, the actions can come later. If actions have already been attached, disable them by setting exe=FALSE
- Think small, break the problem into smaller pieces. When a rule doesn't behave as you hoped, break up into smaller rules which can be more easily tested and understood. which you can examine the output more easily.
- Test Test Test, write small tests for each problem /subproblem . You may even consider using the wonderful testthat package.
- Be patient

Chapter 8

Debugging

The first rule is to never make any mistakes. However ...

8.1 When Rule Hangs: Examine the call sequence

Sometimes mistakes are made and when the rule is applied it hangs.

8.1.1 An example

Consider the following rules

```
peg <- new.parser()
peg + "A<- 'a' B / C / 'ab' " + "B<- C / 'b'" + "C<-A"

##
## Rule: A
## Def: A<- 'a' B / C / 'ab'
## Com:
## Act:
##
## Rule: B
## Def: B<- C / 'b'
## Com:
## Act:
##
## Rule: C
## Def: C<-A
## Com:
## Act:
```

Applying 'A' to "ab" produces an infinite recursion error.

```
peg[["A"]]("ab")
```

```
## Error:  evaluation nested too deeply:  infinite recursion
/ options(expressions=)?
```

The easist way to analyze what happened, is to set a limit on the calling depth of the rules and apply again.

```
set_rule_stack_limit(peg, 20)
peg[["A"]]("ab")
```

```
## Error:  Max Call Depth of Rule Stack Exceeded!  To see calling
sequence use get_rule_stack
```

We still get an error, namely the rule stack limit was exceeded, but now we can analyse calling sequence.

```
stack <- get_rule_stack(peg)
stack
```

```
##    node.id pos
## 1        A   1
## 2        B   2
## 3        C   2
## 4        A   2
## 5        C   2
## 6        A   2
## 7        C   2
## 8        A   2
## 9        C   2
## 10       A   2
## 11       C   2
## 12       A   2
## 13       C   2
## 14       A   2
## 15       C   2
## 16       A   2
## 17       C   2
## 18       A   2
## 19       C   2
## 20       A   2
```

A quick inspection shows us that after level 3, we appear to be in an infinite loop cycling between C and A without consuming any more characters. Thus we need to reconsider our rules and the relationship between C and A.

8.2 Using the Rule Debugger: Stepping through the Rule Set

To use the rule depugger, issue the debug.pegR command and then apply to the rule to debug and step through using n.

The list of commands for the debugger is:

- ”h, help: shows this help”
- ”n: step to the next rule”
- ”c: continue to the next breakpoint”
- ”clr: clear all breakpoints”
- ”+brk@: add break point at both enter and exit points of a rule”
- ” : example +brk@ RULE.ID”
- ”+brk@ >: add break point at the enter point of a rule”
- ” : example +brk@¿ RULE.ID”
- ”+brk@ <: add break point at the exit point of a rule”
- ” : example +brk@¡ RULE.ID”
- ”−brk@: delete break points of a rule”
- ”−brk@ >: delete break point at the enter point of a rule”
- ” : example -brk@¿ RULE.ID”
- ”−brk@ <: delete break point at the exit point of a rule”
- ” : example -brk@¡ RULE.ID”
- ”value: display the return value (value is a list created upon exiting with status of TRUE)”
- ”q: quit the debugger”
- ”r: restart debugger”
- ”l: list all rule breakpoints”

Note For each rule, one can set a break point when entering the rule, and another breakpoint when exiting. When exiting a rule, one can obtain the value of produced by the rule by issuing the *value* command. Otherwise, the rule debugger is pretty much what one would expect.

Consider the following example:

```
pegR <- new.parser()
pegR + "A<-B 'a'" + "B<-C 'b'" + "C<-D 'c'" + "D<-'d'"

##
## Rule: A
## Def: A<-B 'a'
## Com:
## Act:
##
## Rule: B
## Def: B<-C 'b'
## Com:
## Act:
##
## Rule: C
## Def: C<-D 'c'
## Com:
## Act:
##
## Rule: D
## Def: D<-'d'
## Com:
## Act:
```

This will recognize ”dbca” Running the debugger we see

```
debug.pegR(pegR)
pegR[["A"]]("dcba")

## Rdb> Commands: h, n, c, clr, +brk@, -brk@, Q, r, l
## ==>Entering Rule: A
##    Rule Definiton: A<-B 'a'
##    Input text: 'dcba'
## Rdb> n
## ==>Entering Rule: B
##    Rule Definiton: B<-C 'b'
##    Input text: 'dcba'
## Rdb> +brk@>D
## Rdb> c
## ==>Entering Rule: D
##    Rule Definiton: D<-'d'
##    Input text: 'dcba'
## Rdb> v
## Return Value Not Availabe
## Rdb> n
## <==Exiting Rule: D
```

```
##    Rule Definiton: D<-'d'
##    Status: TRUE ; Rule D accepted the input 'dcba'
##    Consumed: 'd'
## Rdb> v
## Returned value:
## $atom
## [1] "d"
##
## Rdb> n
## <==Exiting Rule: C
##    Rule Definiton: C<-D 'c'
##    Status: TRUE ; Rule C accepted the input 'dcba'
##    Consumed: 'dc'
## Rdb> v
## Returned value:
## $atom
## [1] "d"
##
## $atom
## [1] "c"
##
## Rdb> n
## <==Exiting Rule: B
##    Rule Definiton: B<-C 'b'
##    Status: TRUE ; Rule B accepted the input 'dcba'
##    Consumed: 'dcb'
## Rdb> v
## Returned value:
## $atom
## [1] "d"
##
## $atom
## [1] "c"
##
## $atom
## [1] "b"
##
## Rdb> n
## <==Exiting Rule: A
##    Rule Definiton: A<-B 'a'
##    Status: TRUE ; Rule A accepted the input 'dcba'
##    Consumed: 'dcba'
## Rdb> v
## Returned value:
## $atom
```

```
## [1] "d"
##
## $atom
## [1] "c"
##
## $atom
## [1] "b"
##
## $atom
## [1] "a"
##
## Rdb> n
##  Bye
```

The sequence of steps above output was:

1. 'n', sending us to the next
2. $+brk@ > D$, setting a break point for entering the rule "D"
3. 'c', skipping to the next break point which is entering rule "D"
4. 'v', attempting to see the return value, but we have just entered "D", so there is no return value
5. 'n', stepping to next, now we have returned from "D", and we see "D" has consumed 'd'
6. 'v', inspecting the return value of D, we see a list with the single element, i.e. list('d')
7. 'n', stepipng to the next, now are we returning from "C" and we see that "C" has consumed 'dc'
8. 'v', inspecting the return value again we see that C returns a list having two elements, i.e. list('d','c')
9. 'n' stepping to the next, now are now returning from "B" and we see that "B" has consumed 'dcb'
10. 'v' inspectin the return value again, we see that B returns is a list with three elements list('d','c','b')
11. 'n' stepping to the next, now we returning from "A", see that "A" has consumed 'dcba'
12. 'v' The return value of D is a list with two elements list('d','c','b', 'a')
13. 'n' stepping again , we are out of the evaluation

Doing it again, but with the input 'xcba' we see

```
debug.pegR(pegR)
pegR[["A"]]("xcba")

## Rdb> Commands: h, n, c, clr, +brk@, -brk@, Q, r, l
## ==>Entering Rule: A
##    Rule Definiton: A<-B 'a'
##    Input text: 'xcba'
## Rdb> +brk@<D
## Rdb> c
## <==Exiting Rule: D
##    Rule Definiton: D<-'d'
##    Status: FALSE ; Rule D rejected the input: 'xcba'
##    Consumed: ''
## Rdb> n
## <==Exiting Rule: C
##    Rule Definiton: C<-D 'c'
##    Status: FALSE ; Rule C rejected the input: 'xcba'
##    Consumed: ''
## Rdb> n
## <==Exiting Rule: B
##    Rule Definiton: B<-C 'b'
##    Status: FALSE ; Rule B rejected the input: 'xcba'
##    Consumed: ''
## Rdb> n
## <==Exiting Rule: A
##    Rule Definiton: A<-B 'a'
##    Status: FALSE ; Rule A rejected the input: 'xcba'
##    Consumed: ''
## Rdb> n
##  Bye
```

This time we:

1. '+$brk@ < D$', immediately setting a break point for exiting the rule "D"
2. 'c', which skips to the next break point which is exiting rule "D", note D is rejecting the input
3. 'n', which moves the next rule adb gives us a rejection by C
4. 'n', which moves the next rule adb gives us a rejection by B
5. 'n' which moves the next rule adb gives us a rejection by A
6. 'n' which brings us out of the out of the evaluation