The pegR Users Guide

M. S. Legrand

Contents

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 Example

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*. In this case we used the letter A. It may also be called the *rule name* or the *rule label*. [1] The portion on the righthand side of the $\leftarrow$ is called the *rule definition*. Together, the *rule id* , the arrow, and the *rule definition* form the *rule source.*

In this case the rule definiton consists of a single quoted string consisting of the letter 'a'. Quoted strings are called a literals. This rule says when presented some text to process: accept and consume that quoted string.

Adding this rule to our peg can be done in one of two ways: via the + operator, or via the add_rule function:

Using the + operator to add a rule

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

or **Using the add_rule command**

[1] A rule identifier may be composed of any combindation of letters, numbers and an underscore, but must begin with a letter.

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

Once the rule is added, we may parse by applying [2] it to some input text.

Again there are two equivalent ways to do this: the operator **[[]]** and the **apply_rule** function.

Using operator form [[]]

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

or

Using apply_rule

```
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 )
```

To access these components directly we may use

```
status(result)

## [1] TRUE
```

```
consumed(result)

## [1] "a"
```

```
value(result)

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

[2] We use the term **apply a rule** rather than parse, because from our perspective, any rule is a potential root. This point of view is useful, when debugging, since we can break a large parse into subcomponents and test them individually.

2.2 DataFrames

The set of rules contained in a parser can easily be extracted to a dataframe using the **as.data.frame** function. For example:

```
peg <- new.parser()
peg <- peg + c("Pet1<-'cat'", "{list('meow')}") + c("Pet2<-'dog' ",
    "{list('bow wow')}")
pets <- as.data.frame(peg)
pets

##   rule.id rule.definition rule.description action.specification
## 1    Pet1     Pet1<-'cat'                NA         list('meow')
## 2    Pet2     Pet2<-'dog'                NA      list('bow wow')
```

Once in the rules are in data.frame form, they can be manipulated, saved for later use or used to initialize a new parser, as shown below.

```
pegPets <- new.parser(pets)
pegPets

##
## Rule: Pet1
## Def: Pet1<-'cat'
## Des:
## Act: list('meow')
##
## Rule: Pet2
## Def: Pet2<-'dog'
## Des:
## Act: list('bow wow')
```

2.3 Building Upon other Rule Sets

Since rules sets can be save as data.frames and later retrieved, this provides a way of building up libraries of useful rules.

One such library, is the **commonRules** data.frame. As an illustration consider the problem of picking out all words that follow a number.

```
data(commonRules)  # loads the commonRules data.frame
peg <- new.parser(commonRules, action.exe = TRUE)  #initializes the parser with those
peg <- peg + c("NW<-NUM WORD", "{list(v[[2]])}", "#Just the word part") +
    c("HIDEWORD<-WORD", "{}", "#hide word") + c("R<-(NW/HIDEWORD)+")
txt <- "The story of the 3 pigs and 1 wolf."
res <- peg[["R"]](txt)
```

```
value(res)
```

```
## $NW
## [1] "pigs"
##
## $NW
## [1] "wolf"
```

Chapter 3

The Grammar

In this chapter, we give a brief introduction to the core components of the peg grammar together with some simple very simple examples. You might also want to consider checking out the slides at `http://bford.info/pub/lang/peg-slides.pdf`, which provides a short but beautiful presentation of the grammar.

3.1 Rule Sequences

By the term *sequence*, we mean a sequential applicaton of rules, where the success of the sequence requires the success of each rule of the sequence applied in turn.

3.1.1 Simple Example

As an example, we begin by considering 3 rules:

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

To start we create a new peg parser and add these three rules using the operator "+" method. This entails simply stringing the rules together as follows:

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

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

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

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")
```

accepts and consumes 'abc'. However note,

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

is also accepted (status =TRUE), but only 3 out 4 characters are consumed.
Remark: Of course, if we had formed the rule 'ABC' as

$$ABC \leftarrow \quad 'a' \quad 'b' \quad 'c' \tag{3.2}$$

The exact same logic would have taken place, with 'a', 'b', 'c' playing the role anonymous rules. (Anonymous, since they were not named.)

3.1.2 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 <- peg + "R1<-'a' 'b' !'c'" + "R2<- 'a' 'b' &'c'"
```

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

3.2 Repetitions, Wild Cards and parenthesis

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

3.2.1 Zero Or More: the Star Operator (*)

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

```
peg <- new.parser()
peg <- peg + "A<- 'a' 'a'*"
status(peg[["A"]]("aaaaa"))

## [1] TRUE
```

accepts any string beginning with consecutive a's

3.2.2 Zero or One: the Optional Operator (?)

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 <- peg + "A<- 'a' 'a'?"
status(peg[["A"]]("aaaaa"))

## [1] TRUE
```

This will reject if there 3 or more a's

3.2.3 One of Anything, the Dot Operator (.)

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 <- peg + "A<- . .?"
status(peg[["A"]]("xy"))

## [1] TRUE
```

3.2.4 Ranges

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

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

This will accept all integers.

3.2.5 Exercises

1. What does the rule
$$R < -'a' 'b' 'c' !. \tag{3.3}$$
do? How does this differ from
$$R < -'a' 'b' 'c' \tag{3.4}$$
2. Write a rule that accepts a string with exactly one letter x
3. Write a rule that accepts decimal numbers.

3.3 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 <- peg + "X <- 'ab' / 'ac' / 'a'" + "Y <-'a' / 'ab' / 'ac'"
```

Applying X to 'aaa' we get

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

Applying X to 'abc' we get

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

But applying Y to 'abc' we get

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

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

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

3.3.1 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

Chapter 4

Actions

4.1 Return Values

In the absense of actions the behaviour for return values can be summarized as follows:

- Atoms return the characters consumed as a list.
- Expressions are built from atoms and other expressions.
- Returns of an expression is built from the returns its components, most often just combining the lists together as a single list.

Actions provide a way to modify the return those return values.
Some actions we might want to perform are:

- Paste the values together to form a single string [1]
- Drop the values altogether
- Convert the values to numeric
- Replace the values with different values, for example changing the case

To illustrate an action consider the problem of capitalizing every occurance of the letter 'a'. Since this action is only to affect a, so we first make a rule for A for the letter 'a' together an action for that rule. [2]

4.2 Captializing all occurance of the letter 'a'

[1] actually a list with a single string as its entry

[2] Here we create the rule from it's definition and add the action simultaneously. In the next chapter, we discuss on how to add/edit actions of existing rules.

```
peg <- new.parser()
peg <- add_rule(peg, "A<-'a'", act = "list('A')")
```

If we run this on a we see it capitalizes

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

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

Several observations can be made:

1. We put A before . so that is A attempted first
2. We added an action as text to A by including the named parameter act="list('A')"
3. The act="list('A')" is internally turned into a function *f(v){list('A')* } a list containing 'A'.
4. The list returned by the function replaces the input list*v=list(atom='a')* which has the effect of capitalizing a.
5. The final list is gotton by applying the value function to the result, value(res.
6. The value(res) is list containing 'b','A', 'c', ' ', 'w', 'o', 'l', 'f', ' ', 'b', 'a', 'y'
7. *"paste1"*[3] is used to paste the elements of the list together.

4.2.1 Including a Description

It should be painfully obvious, that we need a mechanism to include some description with our rule and action. This is simply to, just include a *des*= component when adding a rule. For example:

```
peg <- new.parser()
peg <- add_rule(peg, "A<-'a'", act = "list('A')", des = "replace a by A")
```

4.2.2 Other ways to capitalize every letter 'a'

There are several otherapproaches with varing syntax that can be used to do exactly the same thing:

[3] Paste1 is analagous to *paste0* but operates on lists instead

Capitalize alternative II

Here make a couple of changes.

- Instead of specifying exe=TRUE in our call to *apply_rule* we set the default *action.exe=TRUE* in the constructor of the parser.
- We paste the list together inside the action for rule G

```
peg <- new.parser(action.exe = TRUE)  # Set action.exe default to TRUE
peg <- add_rule(peg, "A<- 'a'", act = "list('A')",
    des = "replace a by A")
peg <- add_rule(peg, "G<- (A / .)+", act = "list(paste1(v))")
res <- apply_rule(peg, "G", "bad wolf bay")
value(res)[[1]]  # Extact the only element of the list

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

Capitalize alternative III

This is alternative II, rewritten to using the operator form

```
peg <- new.parser(action.exe = TRUE)
peg <- peg + c("A<- 'a'", act = "list('A')", des = "replace a by A") +
    c("CAP<- (A / .)+", act = "list(paste1(v))")
res <- peg[["CAP"]]("bad wolf bay")
value(res)[[1]]

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

Note: Each rule and action is supplied as a single character vector, with the action named by *act*.

Capitalize alternative IV

This is same as alternative III, but with a twist, we no longer name the action, but the string representating the action must be enclosed with **"{}"**. Also, for the description, instead of using a named field, *"des="*, we indicate that it is a description by placing a # symbol at the beginning. Thus descriptions can be thought of as comments for rules.

```
peg <- new.parser(action.exe = TRUE)
peg <- peg + c("A<- 'a'", "{list('A')}", "# Replace a by A") +
    c("CAP<- (A / .)+", "{list(paste1(v))}")
res <- peg[["CAP"]]("bad wolf bay", exe = T)
value(res)[[1]]
```

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

Capitalize alternative V

This is same as alternative IV, but with another twist, instead of "{list(paste1(v))}", we use a shortcut "{-}".

```
peg <- new.parser(action.exe = TRUE)
peg <- peg + c("A<- 'a'", "{list('A')}", "# Replace a by A") +
    c("CAP<- A / .", "{-}")
res <- peg[["CAP"]]("bad wolf bay")
value(res)[[1]]

## [1] "b"
```

There are two shortcuts:

1. {-} : the paste1 shortcut, it pastes the values (same as {paste1(v)})
2. {} : the eat shortcut, it eats the values (same as {list()})

4.3 List names

Often, it is useful to pick out a particular element of a value list. By default

- The return list for an atom has the element named "atom"
- A list without any explicit name inherits the name of the rule which created it.

```
peg <- new.parser(action.exe = TRUE)
peg <- peg + c("DIGIT<-[0-9]", "{list(as.numeric(v[[1]])) }") +
    c("OP<- '+'/'-'", "{-}") + c("S<-' '", "{}") +
    c("R<-(DIGIT/OP/S/.)*")
res <- peg[["R"]]("1 + 2 = X")
value(res)

## $DIGIT
## [1] 1
##
## $OP
## [1] "+"
##
## $DIGIT
## [1] 2
```

```
##
## $atom
## [1] "="
##
## $atom
## [1] "X"
```

Of course, if one choose, one can always set their own names to the values.

```
peg <- new.parser(action.exe = TRUE)
peg <- peg + c("DIGIT<-[0-9]", "{ list( steve=as.numeric(v[[1]])) }")
res <- peg[["DIGIT"]]("3")
value(res)

## $steve
## [1] 3
```

4.4 Salient Points

Some salient point about actions are:

1. Actions are attached to rules
2. Actions are used to modify the return value of a rule
3. Actions accept an list arg v and return a list
4. Actions are added as a string which is to represent the body of a function f(v)
5. To invoke an action, either the *exe flag* should be set to TRUE, or *action.exe* should be set to TRUE.
6. Actions are declared either by naming (act=...) or by enclosing with braces "{... }"
7. The action values can have names assigned to the members of the list
8. Descriptions are a way of adding comments to rules.

4.4.1 Exercises

1. Write a parser that convert binary numbers to base 10
2. Write a parser that returns a list of all words in a paragraph.
3. Write a parser to capitalize the last letters of all words in a paragraph.
4. Write a parser to cound the number of sentences in a paragraph.
5. Write a parser to capitalize the first word of every sentence in a paragraph

Chapter 5

Editing Rules, Actions and Descriptions

In the preceding sections, we discussed adding rules with actions and descriptions to the parser, but not how to modify them once they have been added.

5.1 Editing the Rule Definitions of an Existing Rule

Rule definitions can be ammended, although not renamed. Changing the rule definition can be done in a couple of ways:

5.1.1 Set Definition

```
peg <- new.parser()
peg <- add_rule(peg, "ARTICLE<-'a'")
inspect_rule(peg, "ARTICLE")

## Rule: ARTICLE
## Def: ARTICLE<-'a'
## Des:
## Act:

peg <- set_definition(peg, "ARTICLE", "ARTICLE<-'a'/'an'/'the'")
```

Note: We keep the same rule name.

5.1.2 Set Definition

```
peg <- new.parser()
peg <- peg + "ARTICLE<-'a'"
peg[["ARTICLE"]] <- "ARTICLE<-'a'/'an'/'the'"
```

Again Note: We keep the same rule name.

5.2 Actions

Setting actions to a rule after the rule has been added to the parser is relatively straight forward, and can be done in a variety of ways:

5.2.1 Set Action

```
peg <- new.parser()
peg <- add_rule(peg, "A<-'a'")
peg <- set_action(peg, "A", "list('A')")
inspect_rule(peg, "A")

## Rule: A
## Def: A<-'a'
## Des:
## Act: list('A')
```

Here we inspected the rule *"A"* by using the **inspect_rule** command. Of course, we could have gotton all the rules by issuing *print(peg)* or simply *peg* by itself on a single line.

Now we can reset the action to produce 2 captials A's by

```
peg <- set_action(peg, "A", "list('AA')")
inspect_rule(peg, "A")

## Rule: A
## Def: A<-'a'
## Des:
## Act: list('AA')
```

And we can remove the action by assigning NULL to the action

```
peg <- set_action(peg, "A", NULL)
peg

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

To remove the rule altogether

```
peg <- delete_rule(peg, "A")
peg
```

5.2.2 The Operator Approach: peg[[*Rule.Id*]]

The operator approach is as follows:

```
peg <- new.parser()
peg <- peg + "A<-'a'"
peg[["A"]] <- "{list('A')}"  # Same as set_action(peg, 'A', 'list('A')')
peg[["A"]]  # Same as inspect_rule(peg, 'A')

## Rule: A
## Def: A<-'a'
## Des:
## Act: list('A')
```

To reset the action to produce 2 capital A's

```
peg[["A"]] <- "{list('AA')}"  # Same as set_action(peg, 'A', 'list('AA')')
peg[["A"]]  # Same as inspect_rule(peg, 'A')

## Rule: A
## Def: A<-'a'
## Des:
## Act: list('AA')
```

To remove the action

```
peg[["A"]] <- "{NULL}"  # Same as set_action(peg, 'A', 'list('AA')')
peg[["A"]]  # Same as inspect_rule(peg, 'A')

## Rule: A
## Def: A<-'a'
## Des:
## Act: list('AA')
```

To remove the rule altogether

```
peg[["A"]] <- NULL
peg

##
## Rule: A
```

```
## Def: A<-'a'
## Des:
## Act: list('AA')
```

5.3 Descriptions

Descriptions are comments for rules.

5.3.1 Set Description

We can attach a description to a rule as follows:

```
peg <- new.parser()
peg <- add_rule(peg, "A<-'a'")
peg <- set_description(peg, "A", "consumes 'a' ")
inspect_rule(peg, "A")

## Rule: A
## Def: A<-'a'
## Des: consumes 'a'
## Act:
```

Now we can reset the description to another phrase by

```
peg <- set_description(peg, "A", "Consumes a single character 'a'")
inspect_rule(peg, "A")

## Rule: A
## Def: A<-'a'
## Des: Consumes a single character 'a'
## Act:
```

And we can remove the description by assigning NULL to the description

```
peg <- set_description(peg, "A", NULL)
peg

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

5.3.2 The Operator Approach: peg[[*Rule.Id*]]

We can attach a description to a rule as follows:

```
peg <- new.parser()
peg <- peg + "A<-'a'"
peg[["A"]] <- "# consumes 'a'"  # Same as set_description(peg, 'A', 'consumes 'a'')
peg[["A"]]  # Same as inspect_rule(peg, 'A')

## Rule: A
## Def: A<-'a'
## Des: consumes 'a'
## Act:
```

To reset the description to another phrase by

```
peg[["A"]] <- "#Consumes a single character 'a'"  # Same as set_description(peg, 'A',
peg[["A"]]  # Same as inspect_rule(peg, 'A')

## Rule: A
## Def: A<-'a'
## Des: Consumes a single character 'a'
## Act:
```

To remove the description

```
peg[["A"]] <- "#"  # Same as set_description(peg, 'A', NULL)
peg[["A"]]  # Same as inspect_rule(peg, 'A')

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

5.4 Simultaneously Doing More than One Update of a Given Rule

With the operator notation, we can also simultaneous update the definition, action, and comment of a rule.

```
peg <- new.parser()
peg <- peg + "ARTICLE<-'a'"
peg[["ARTICLE"]] <- c("ARTICLE<-'a'/'an'/'the'", "{}",
    "#Eat the articles")
peg[["ARTICLE"]]
```

```
## Rule: ARTICLE
## Def: ARTICLE<-'a'/'an'/'the'
## Des: Eat the articles
## Act: list()
```

Chapter 6

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

6.1 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 <- peg + "A<-'a'" + "B<-'b'" + "C<-'c'" + "X<- A / B A/ C B A"
```

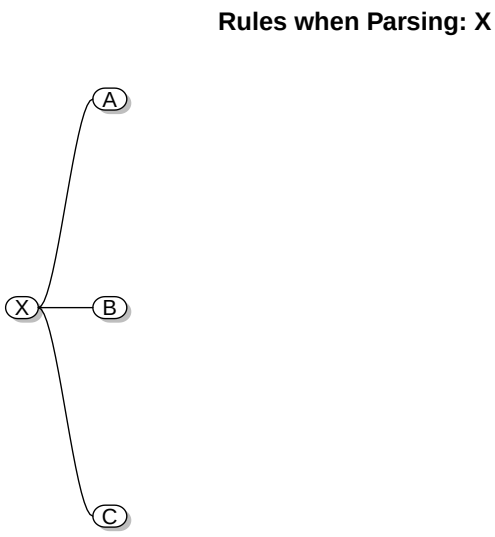
Then applying X to "cba" we see

```
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 )
```

6.2 Plotting the Results

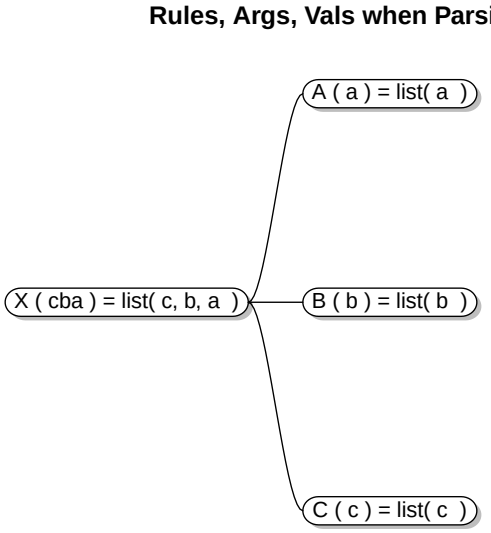
Plotting the same result we have

```
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 <- 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"
# 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 )
```

6.3 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 can be useful to explore the syntax, without having to create a new parser.

```
res <- qp("'a' 'b'/'c'")("a")
value(res)

## list()
```

```
res <- qp("'a' 'b'/'c'")("ab")
value(res)

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

```
res <- qp("'a' 'b'/'c'")("ac")
value(res)

## list()
```

```
res <- qp("'a' 'b'/'c'")("c")
value(res)

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

6.4 Tips

Ok, you're ready to create your first great 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, You can always add actions later.
- 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.
- Use the debugger: Don't be afraid to use the pegR debugger. It won't bite.
- Be patient: Breaking the monitor doesn't really help!

Chapter 7

Debugging

Of course the first rule in good programming is to never make any mistakes. However sometimes reality sets in and we have to deal with it.

7.1 When Rule Hangs: Examine the call sequence

Sometimes when the rule is applied it hangs.

7.1.1 An example

Consider the following rules

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

Applying 'A' to "ab" produces an bright red 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.

7.2 Using the Rule Debugger: Stepping through the Rule Set

To use the rule depugger, issue the textbfdebug.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:

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

This will recognize "dbca" Running the debugger we see

```
debug.pegR(peg)
peg[["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(peg)
peg[["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