Chapter 1

Getting Started

1.1 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*.

$$A \leftarrow' a' \tag{1.1}$$

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 2

Rule Sequences

2.1 Simple Sequences

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

$$\begin{array}{ccc} A & \leftarrow & 'a' \\ B & \leftarrow & 'b' \\ C & \leftarrow & 'c' \end{array} \tag{2.1}$$

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.

2.2 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 3

Repetitions, Wild Cards and parenthesis

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

3.1 *

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

3.2 ?

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

3.3 .

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
```

Chapter 4

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 <- parser.new()
```

```
## Error:  could not find function "parser.new"
```

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

```
##
## Rule: A
## Def: A<- . .?
## Com:
## Act:
##
## 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 pplying 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 )
```

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", exe = T)
paste1(value(res))

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

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

Chapter 5

Editing Rules and Adding Comments

Chapter 6

Examining What Happened

6.1 Printing the Results

6.2 Graphing the Results