Reguläre Ausdrücke

Mark Heckmann

July 31, 2012

Contents

REGEX : Mastering regular expressions

Dieser Abscnitt dient als kurze EInführung in regex und stellt die wichtisgten R Funktionen vor, die regex nutzen. Es ist jedoch keine umfassende Einführung oder Erklärung. Hierzu ist e ratsam eine Standardbuch über regex zu lesen. Ein sehr gutes Werk, an das auch folgende Einführung angelehnt ist, ist z.B. Friedl (XXX).

Idee ist fast jeden bereits aus Betriebessystemen Syntax bekannt. In DOS oder Unix ist es möglich, alle Dateien, die z.B. die Endung `.txt` haben folgendrmaßen anzusprechen: `*.txt`. Das Sternechen steht hierbei für eine oder einer Reihe beliebiger Zeichen. Solche Filename patter werden *file globs* oder *wildcards* genannt. In Betriebssystemen gibt es nur wenige Zeichen, die eine derart besondere Bedeutung haben. *regular expressions* sind eine Pattern Sprache, die über weit mehr Möglichkeiten verfügt solche Wortmuster zu defineren.

* und andere Zeichen mit besonderer Bedeutung werde *metacharacters* genannt. Die übrigen Zeichen werden *literals* genannt.

In regular expression können die große Anzahl kleiner metacharacter in sehr vielfältiger eise zu einem regulären Ausruck zusammengesetzt werden. So entstehen Ausrücke die eine komplexe Funktion erfüllen.

Der Bergriff *Reguläre Ausdrücke* heißt auf Englisch *regular expressions*. Es ist übrlich, diesen mit dem *Akronym* regex abzukürzen. So wird ab hier auch in diesem Buch verfahren.

Metacharacters

Eineige Zeichen haben in innerhka von regulären Ausdrücken eine besondere Bedeutung. Diese werden Metacharacters genannt. Darüber hinaus hängt die Bedeutung eininger Metacharacters davon ab, wo genau im regulären Ausdruck sie auftauchen. Dies trifft z.B. auf den Metacharacter^ zu. Mehr dazu später.

Reguläre Ausrücke sind eine Art eigene Sprache zu der mehrere Dialekte existieren. Je nach Programmiersprache bzw. Programm kann es hier leicht Unterschide in der Syntax sowie in den Möglichieten geben. Eineige Funktionen in R, die reguläre Ausrücke verarbeiten verstehen mehrere Dialekte. So z.B. die Art und Weise, wie regex in Perl bennuitzt werden. Im Folgenden werden wir uns auf den in R genutzten Standarddialekt konzentrieren.

```
> text <- c("Erste Zeile dieses Textes.",
           "Dies sei die zweite Zeile.")
> grep("Zeile", text)

[1] 1 2


> grep("Dies", text)

[1] 2
```

Der Ausdruck ist so zu lesen. Suche ein großes `D`, das von einem kleinen `i`, einem `e` und `s` gefolgt wird. Dies trifft nur auf die zweite Zeile zu. In der ersten enthält das Wort `dieses` eine kleines `d` und wird somit nicht erfasst.

Um sowohl das große als auch das kleine *d* zu erfassen können sogenannte *character classes* genutzt werden. Eine characer class wird durch eckige Klammern begrenzt, z.B. `[dD]`. Dies besagt, dass an dieser Stelle im Ausdruck ein großes oder eine kleines d stehen kann. Dies kann folgendemaßen in unseren Ausdruck eingebaut werden, um sowohl die Zeichenkette `dies` als auch `Dies` zu erfassen.

```
> grep("[Dd]ies", text)

[1] 1 2
```

Während außerhalb einer character class die Regel gilt, dass ein bestimmtes Zeichen *UND* dann das nächste bestimmte Zeichen folgt, sind die Zeichen innerhalb der Klasse mit einem `ODER` vernüpft. Es kann also das `D` oder das `d` folgen. Eine chacter class beschriebt jeweils nur wofür ein Zeichen innerhalb des Ausdrucks dtshen kann, nicht mehrere.

Anfang und Ende einer Zeichenkette

Nehmen wir an, wir haben folgende Variablennamen.

```
> varnames <- c("geschl", "verwend", "gbv", "v1", "v2", "v3")
```

Wir möchte aus diesen alle Variablen herausfiltern, die mit einen `v` beginnen. Der Ansatz `grep("v", varnames)` würde uns hier nicht weiterhelfen, da er den Index aller Variabken ausgibt, die überhaupt den Buchstaben `v` enthalten.

Um die Position in einer Zeichenkette mit zu berücksichtigen können die Metacharactere `^` (caret) und `$` Dollarzeichen genutzt werden. Sie markieren den Anfang und das Ende der untersuchten Zeichenkette. Nachfolgender Audsruck gibt nur die Indizes der Variablennamen aus, wo direkt nach dem Beginn der zeichenkette (`^`) ein `v` folgt.

```
> grep("^v", varnames)

[1] 2 4 5 6
```

Wir sind jeodch noch nicht ganz zufrieden. Wir möchten nur die Variaben ausgeben, wo auf `v` eine Zahl folgt. Um dies zu erreichen, können wir angeben, dass nach dem v ein Zeichen zwischen 0 und 9 folgen muss.

```
> grep("^v[0123456789]", varnames)

[1] 4 5 6
```

Für die Zahlen 0 bis 9 ist die Kurzschreibweise `0-9` erlaubt. Somt kann alternativ auch der Ausdruck `grep("^v[0-9]", varnames)` genutzt werden.

Um das gleiche Ergebnis zu erreiche hätten wir auch den alternativen Ansatz wählen können nur jene Variablen auszugeben, die als letztes Zeichen eine Zahl entahlten und direkt danach das Emde der Zeichenkette (`$`) folgt

```
> grep("[0-9]$", varnames)

[1] 4 5 6
```

TODO Ausdrücke zu Vereinfachung `0-9` `A-Z` etc.

Negatierung von character classes

Innerhalb von character clasees hat das Zeichen `^` eine andere Bedeutung als außerhalb. Während es außerhalb den Zeilenanfang markiert, bedeutet es als *erstes* Zeichen innerhalb der eckigen Klammer, dass alle nachfolgenden Zeichen `nicht` erfasst werden. [1] `[^0-3]` bedeutet somit, dass nur Zeichen erfasstwerden sollen, die nicht 0 bis 3 sind.

Alzernativ hätte wir somit alle Variablennamen, bei denen auf das `v` nach dem Zeilenanfang kein Buchstabe (aber z.B. eine Zahl folgt) erfasst.

```
> grep("^v[^a-z]", varnames)

[1] 4 5 6
```

Der Punkt - irgendein Zeichen

Der Punkt ist ein Metacharcter, der für ein belibiges Zeichen steht. Angenommen wir hätten alle Variablen finden wollen, bei denen nach Zeilenbeginnn irgendeinem Zeichen beginnen gefolgt von einer Zahl steht. Dies wäre wie folgt möglich.

```
> grep("^.[0-9]", varnames)

[1] 4 5 6
```

EIne weiteres Beispiel ist eine Datum. Nehmen wir da, ich suche in einem Text das Datum 01.01.2000. Dies könnte auch folgendermaßen geschireben werden: 01/01/2001 oder auch 01-01-2001. Der susdruck könnte demenstprechden Lauten

```
> dates <- c("01.01.2000", "01/01/2000", "01-01-2000", "01 01 2000")
> grep("01.01.2000", dates)

[1] 1 2 3 4
```

`Das Oder`

Der senkrechte Strich (`|`) ist in `R` als logisches `oder` bekannt. Innerhlab von reguären Ausdrücken kann er in ähnloicher Weise genutzt werden. Vorhin haben wir mittels einer character Klasse zwischen den Worten dies und Dies unterschieden. Es ist auch möglich, dies mittels des Metacharacters `|` zu erreichen.

```
> d <- c("dies", "Dies")
> grep("[Dd]ies", d)

[1] 1 2

> grep("Dies|dies", d)

[1] 1 2
```

In Kombination mit einem weiteren Metacharacter, der runden Klammer, kann die Reichweite des Auswahlbereichs auf diese eingeschränkt werden.

```
> grep("(D|d)ies", d)
```

[1] Wenn es nicht direkt nach der öffnenenden eckigen Klammer steht hat es jedoch eine andere Bedeutung als die Negierung.

```
[1] 1 2
```

Innerhalb einer character class ist `|` *kein* Metacharcter sondern ein normales Zeichen.

Schueen wir uns noch eine weiteres Beispiel an. Wir haben folegndes Variablennamen unnd wollen alle ausgeben, die mir `v` oder mit `var` beginnen und auf die ein Zahl folgt.

```
> vars <- c("geschl", "v1", "var4")
> grep("^(v|var)[0-9]", vars)

[1] 2 3
```

Case sensitivity

```
> vars <- c("geschl", "v1", "var4", "V4", "Var5", "VAR6")
> grep("^([Vv]|[Vv][Aa][Rr])[0-9]", vars)

[1] 2 3 4 5 6

> grep("^(v|var)[0-9]", vars, ignore.case=T)

[1] 2 3 4 5 6
```

Begining of word matching

```
> vars <- c("var", "covariable", "variable", "no variable")
> grep("var", vars)

[1] 1 2 3 4

> grep("\\<var\\>", vars)

[1] 1

> grep("\\<var", vars)

[1] 1 3 4
```

Optionale Ausdrücke

`?` zeigt an, dass das vorangehende Zeichen optional ist. Im nachfolgenden Fall wird nach der Zeichenkette *die* gesucht als auch nach *dies*.

```
> h <- c("die", "dies")
> grep("dies?", h)

[1] 1 2


> j <- c("var", "v")
> grep("v(ar)?", j)

[1] 1 2


> j <- c("July fourth", "Jul 4th")
> grep("(July|Jul) (fourth|4th|4)", j)
```

```
[1] 1 2

> grep("(July?) (fourth|4(th)?)", j)

[1] 1 2
```

quantifiers

Die Metacharacters Plus + und Sternchen (*) ähneln dem Fragezeichen. Auch sie beziehen sich auf das vorangehende Zeichen. + versucht das vorgehende Zeichen so oft wie möglich, aber mindestens einmal zu matchen. * ist ähnlich. Es matched das vorangehende Zeichen beliebig oft im gegensatz zu + jedoch auch keinmal. Wenn das Zeichen hingegen keinmal auftaucht, so wird + im Gegensatz zu * keinen match finden.

```
> s <- c("a", "ab", "abb", "abbb")
> grep("ab+", s)

[1] 2 3 4

> grep("ab*", s)

[1] 1 2 3 4
```

Wenden wir die nun auf den Fall der Variablenauswahl an. Da nicht alle Variablen nur eine Zahl hinter der v aufweisen werden sollen nun mindestens eine aber auch mehrere Zahlen erfasst werden.

```
> vars <- c("v_geschl", "v1", "v132")
> grep("^v[0-9]+", vars)

[1] 2 3
```

Komplizieren wir den Fall nun ein wenig, indem manche Variablennamen mit v andere mit var beginnen und Groß-und Kkleinschreibung austauschbar ist.

```
> vars <- c("v_geschl", "v1", "v132", "var13", "VAR2", "Var544")
> grep("^v(ar)?[0-9]+", vars, ignore.case=T)

[1] 2 3 4 5 6
```

Quantifizierung über Intervalle. Bisher haben wir lediglich die Möglichkeit gehabt keinen oder mehrere vorangehende Zeichen zu matchen. Die geschweifte Klammer gibt die Möglichkeit die Anzahl an zu matchenden Zeichen zu begrenzen. Nachfolgenden Beispiel gibt nur jene Variablen aus, bei der dem v mindestens 3 Ziffern folgen.

```
> s <- c("v12", "v10000", "v1", "v123")
> grep("^v[0-9]{3,6}", s)

[1] 2 4
```

Backreferencing

```
> s <- c("Wort Wort", "der der Mann")
> grep("\\<([A-Za-z]+) +\\1\\>", s)

[1] 1 2
```

NEU: Nehmen wir an, wir haben einen string der Anführungszeichen enthält. Diese werden durch den Escape character gekennzeichnet. Außerdem hat der String am Anfang und Ende Anführungszeichen. Es sollen nun ausschließlich die Anführungszeichen am Anfang und Ende entfernt werden, aber nur, wenn beide vorhanden sind. Eine Lösung liefert die Nutzung einer Backreferenz. Der geundene Ausdruck wird durch jene ersetzt, die innerhlab der runden Klammern stehen. Auf diese bezieht sich die Back-Referenz. Dies sind alle Teile des Satzes außer den Anführrugsstrichen.

```
> library(stringr)
> s <- c("\"He said:\"Hello\" - some word\"", "\"Hello!\" he said")
> r <- str_replace(s, "^\"(.*)\"$", "\\1")
```

Escape Sequenzen Um ein Zeichen zu matchen, dass innerhalb von regulären Ausdrücken ein Metacharcter ist und somit eine besondere Bedeutung hat wird der Backslash als Escape Anweisung genutzt. Dieser zeigt an, dass das nacholgende Zeichen nicht als Metacharacter sondern als nmormale Zeichen behandelt werden soll. In R ist es noch ein wenig umständlicher, weil der Backlsah selber in Zeichenketten bereits eine Bdertung hat. Aus diesem Grund benötigen wir einen doppelten Backslash.

```
> s <- c("*.bmp", "*.png")
> grep("\\*", s)

[1] 1 2

> s <- c("wort\\test", "*.png")
> grep("\\\\", s)

[1] 1

> s <- "(Anmerkungen)"
> grep("\\([A-Za-z]+\\)", s)

[1] 1
```

Wortanfang alternative: `\\b`.

```
> l <- c("dies", "das", "und so")
> grep("\\bd", l)

[1] 1 2
```

Subexpression: Eine Subexpression ist eine eiabgeschlossene Einheit eines größerene Ausdrucks. Z.B die linke und rechte Seite von einer Auswahl `1-3|7-9` oder ein Bereich innerhalb einer runden Klammer `A-Z1,1(a-z)*`. Quantifiers beziehen sich immer auf die komplette vorangehende Subexpression. Wenn also eine Quatifier auf ein Klammer folgt, so bezieht diese sich auf den gesamten Ausdruck in der Klammer `(D|d)+`.

Übersicht

Tabelle **??** gibt einen Überblich über die bisher verwendetetn Elemente von regulären Ausdrücken.

TABELLE FRIEDL 1.5.6

0.1 Beispiel

Enthält eine Zeichenkette nur Zahlen?

Table 1: Regex

Metacharacter	Beschreibung	Matches
`.`	Punkt	Ein beliebiges Zeichen
`[ ]`	character class	Irgendein ein Zeichen aus der Klasse
`[^ ]`	negierte character class	Irgendein ein Zeichen außer jene in der Klassen
`^`	caret	Steht für den Zeilenanfang
`$`	Dollarzeichen	Steht für das Zeilenende
`|`	Senkrechter Strich	Auswahl: Einer der Ausdrücke links oder rechts wird gematched
`( )`	Runde Klammern Strich	Bisher: Gültigkeitsberech für `|` definieren
`\<`	Backslash, kleiner	Anfang eines Wortes. Achtung: in R wird ein zweiter Backslash vorangestellt.
`\>`	Backslash, größer	Ende eines Wortes. Achtung: in R wird ein zweiter Backslash vorangestellt.

```
> library(stringr)


> s <- c("123", "A123", "123A")
> str_extract_all(s, "^[0-9]+$")

[[1]]
[1] "123"

[[2]]
character(0)

[[3]]
character(0)


> s <- c("1.15", ".12", "123")
> str_extract_all(s, "^[-+]?[0-9]*(\\.[0-9]*)?")

[[1]]
[1] "1.15"

[[2]]
[1] ".12"

[[3]]
[1] "123"


> s <- c("10.00 Euro", "10Euro", "10 €")
> str_extract_all(s, "^[0-9]*(\\.[0-9]*)?")

[[1]]
[1] "10.00"

[[2]]
[1] "10"

[[3]]
[1] "10"

> str_match(s, "(^[0-9]*(\\.[0-9]*)?).(Euro|€)")

     [,1]         [,2]    [,3]  [,4]
[1,] "10.00 Euro" "10.00" ".00" "Euro"
[2,] "10Euro"     "1"     ""    "Euro"
[3,] "10 €"       "10"    ""    "€"
```

```
> s <- c("A111", "A123", "A1")
> str_extract_all(s, "^[A-Z][0-9]+")

[[1]]
[1] "A111"

[[2]]
[1] "A123"

[[3]]
[1] "A1"

> str_extract_all(s, "^[A-Z][0-9]+?")     # non-greedy

[[1]]
[1] "A1"

[[2]]
[1] "A1"

[[3]]
[1] "A1"

> str_extract_all(s, "^[A-Z][0-9]{2, }?")

[[1]]
[1] "A11"

[[2]]
[1] "A12"

[[3]]
character(0)
```

Text mit Klammern

```
> s <- "ein (Test innerhalb) und (so)"
> str_extract_all(s, "\\([^)]*\\)")

[[1]]
[1] "(Test innerhalb)" "(so)"
```

Regex group capture

Wir haben bereits gesehen, dass auf vorherige Teilausdrücke zugegriffen werde konnte. in Sprachem wie z.B. Perl ist es möglich über eine externe Variable auf die einzelne Gruppen des Ausdrucks zuzugreifen. In R ist dies z.B. mit der Funktion `str_match` aus dem `stringr` Paket möglich. Diese liefert eine Matrix zurück, bei der die erste Spalte das Ergebnis des kompletten Ausdrucks enthält. Die üpbrigen Splaten enthalten sukzessiv die Ergebnisse der Teilausdrücke.

```
> s <- c("(v1 = 0.123)", "(v2 = 0.444)")
> str_match(s, "\\((.*?) = (0\\.[0-9]+)\\)")

     [,1]           [,2] [,3]
[1,] "(v1 = 0.123)" "v1" "0.123"
[2,] "(v2 = 0.444)" "v2" "0.444"
```

Nehmen wir an es liegen Währungsangaben vor. Diese können Nachkommastellen aufweisen und haben eine Währungsangabe hinter der. Folgende Varinate und entsprechden Kombinationen sind denkbar: 10 Euro, 10.00 Euro, 10 Eurozeichen.

```
> s <- c("10.99", "10.00 Euro", "10  Euro", "10 €", "9.99 €")
> str_match(s, "(^[0-9]+(\\.[0-9]*)?) *(Euro|€)?")

     [,1]         [,2]    [,3]  [,4]
[1,] "10.99"      "10.99" ".99" ""
[2,] "10.00 Euro" "10.00" ".00" "Euro"
[3,] "10  Euro"   "10"    ""    "Euro"
[4,] "10 €"       "10"    ""    "€"
[5,] "9.99 €"     "9.99"  ".99" "€"
```

Tabulatoren: Nehmen wir an, wir wollen zwischen der Zahl und der währungsangabe auf einen Tabulatior zulassen. Der Tabulator ist als Metacharater über den Backslash als `\t` definiert. Durch den Zusatz `[ \t]*` können als Trennzeichen nun ein oder mehrere Leerzeichen oder Tabulatoren oder auch in Kombination verwendet werden.

```
> s <- c("10.00\tEuro", "9.99 \t€")
> cat(s)

10.00        Euro 9.99         €

> str_match(s, "(^[0-9]+(\\.[0-9]*)?)[ \t]*(Euro|€)?")

     [,1]          [,2]    [,3]  [,4]
[1,] "10.00\tEuro" "10.00" ".00" "Euro"
[2,] "9.99 \t€"    "9.99"  ".99" "€"
```

Eine Alternative ist der Metacharcter für einen belibigen Whitespace Chacarter `\s`. hierzu zählt der Tabulator, das Leerzeichen, eine neue Zeile und Carriage return. Achtung: Innerhalb einer character class wird dies nicht funktionieren.

```
> str_match(s, "(^[0-9]+(\\.[0-9]*)?)\\s*(Euro|€)?")

     [,1]          [,2]    [,3]  [,4]
[1,] "10.00\tEuro" "10.00" ".00" "Euro"
[2,] "9.99 \t€"    "9.99"  ".99" "€"
```

Wenn eine Klammer nicht gecaptured werden soll wird dazu in Perl die Sequenz `(?:  )` genutzt. In R hat dies zunächst keine Bedeutung da kein Capturing stattfindet. nur die Funktion `str_match` berücksichtigt dies. Nachfolgend eine leicht modifizierte Variante, die nur die wichtisgsten Gruppen auswertet. Verwirrend ist hierbei, dass das Fragezeichen etwas anderes bedutet als sonnst. Hier hat die ganze Sequenz `(?:` *eine* Bedeutung. Dies macht den Code jedoch schwerer lesbar, so dass man stets überlegen muss, ob die Vorteile überwiegen.

```
> s <- c("10.99", "10.00 Euro", "10  Euro", "10 €", "9.99 €")
> str_match(s, "(^[0-9]+(?:\\.[0-9]*)?) *(Euro|€)?")

     [,1]         [,2]    [,3]
[1,] "10.99"      "10.99" ""
[2,] "10.00 Euro" "10.00" "Euro"
[3,] "10  Euro"   "10"    "Euro"
[4,] "10 €"       "10"    "€"
[5,] "9.99 €"     "9.99"  "€"
```

```
> s <- c("123", "12AB")
> str_extract(s, "\\d*")

[1] "123" "12"
```

Das Paket `gsubfn` erlaubt nicht nur einfache Ersetzungen sondern auch Funtkionen zu nutzen. Hierbei werden die

0.2 Ersetzungen

Anonymisieren

```
> s <- c("Mark Heckmann", "Markus Heckmann", "Jan Pries")
> str_replace(s, "(Heckmann|Pries)", "xxx")

[1] "Mark xxx"   "Markus xxx" "Jan xxx"

> str_replace(s, "(Mark|Jan) (Heckmann|Pries)", "\\1 xxx")

[1] "Mark xxx"        "Markus Heckmann" "Jan xxx"

> str_replace(s, "\\<(Mark|Jan)\\> \\<(Heckmann|Pries)\\>", "\\1 xxx")

[1] "Mark xxx"        "Markus Heckmann" "Jan xxx"
```

Auf zwei bzw. drei Nachkommastellen abschneiden, je nachdem, ob die dritte Ziffer eine Null oder keine Null ist.

```
> s <- c(1.25, 1.257, 1.257134)
> str_replace(s, "(\\.\\d\\d[1-9]?)\\d*", "\\1")

[1] "1.25"  "1.257" "1.257"


> library(gsubfn)
> s <- c("12", "(13, 14)")
> gsubfn("[0-9]+", function(x) as.numeric(x) + 1, s)

[1] "13"       "(14, 15)"
```

Nehmen wir an wir wollen an die Namen Mark und Jan jeweils eine 1 dranhängen.

```
> s <- c("der Hermann", "der Jan", "der Mark")
> str_replace(s, "(Mark|Jan)", "\\11")   # oder

[1] "der Hermann" "der Jan1"    "der Mark1"

> gsubfn("(Mark|Jan)", function(x) paste(x, 1, sep=""), s)

[1] "der Hermann" "der Jan1"    "der Mark1"
```

Adding text of the beginning of lineby replacing ^ with some content.

```
> s <- c("some text")
> str_replace(s, "^", "added ")
```

```
[1] "added some text"
```

Nehmen wir an wir wollen alle Zitate, die in Klammermn stehen aus einem Text herausfiltern.

```
> s <- c("Dies war bereits angelegt (Hauser, 1999, S.123). Auch hier (Heckmann, 2012, S.23)
> str_match_all(s, "\\(([^()]*)\\)")[[1]][, 2]

[1] "Hauser, 1999, S.123"  "Heckmann, 2012, S.23"
```

Lookaround

Eine Lookaraound ist ein relativ neues Regex Feature. Nehmen wir an, wir möchten in längere Zahlen die keine Punkte zur Tausendertrennung enthalten Punkte einsetzen.

Lookahead wird durch folgende Sequenz aufgerufen: `(?=  )` Lookbehind Sequenz: `(?<=  )`

Lokkarounds matchen nur Positionen, d.h. sie geben ein Position zurück.

Lookaaround sind nicht in der Standardart implementiert sondern nur im Perl Dialekt.

```
> s <- c("Mark Heckmann")
> str_replace(s, perl("(?=Heckmann)"), "A. ")

[1] "Mark A. Heckmann"
```

Aus Heckmann wird Beckmann.

```
> s <- c("Mark Heckmann")
> str_replace(s, perl("(?=Heckmann)H"), "B")

[1] "Mark Beckmann"
```

Nachfolgendes Lookahead (dritte Zeile) führt dazu, dass die regex nicht füdig wird wenn auf das Wort `Mark` kein `s` folgt.

```
> s <- "Marks Auto"
> str_replace(s, "Marks", "Mark's")

[1] "Mark's Auto"

> str_replace(s, "\\<Marks\\>", "Mark's")       # mit word boundaries

[1] "Mark's Auto"

> str_replace(s, perl("\\bMark(?=s\\b)"), "Mark'")    # mit Lookahead

[1] "Mark's Auto"
```

In einer Kombination aus Lookbehind und Lookahead liefert uns die geneua Position hinter dem wort Mark.

```
> s <- "Marks Auto"
> str_replace(s, perl("(?<=\\bMark)(?=s\\b)"), "'")

[1] "Mark's Auto"
```

Punkte einfügen um Tausenderstellen zu trennen.

```
> s <- c("12344567788")
> # (?<=\\d)        # links sthet eine Zahl
> # (\\d\\d\\d)+$   # mehrmals drei Zahlen und das dann das Ende
> gsub("(?<=\\d)(?=(\\d\\d\\d)+$)", "\\.", s, perl=T)

[1] "12.344.567.788"

> str_replace_all(s, perl("(?<=\\d)(?=(\\d\\d\\d)+$)"), "\\.")

[1] "12.344.567.788"
```

Dieser Ansatz funktioniert nicht innerhalb einer Zeichenkette, da das Dollarzeichen am Schluss das Ende des Strings benötigt. Lösung eine Word-Boundary nutzen.

```
> s <- c("Eine 12344567788 Zahl")
> gsub("(?<=\\d)(?=(\\d\\d\\d)+\\b)", "\\.", s, perl=T)

[1] "Eine 12.344.567.788 Zahl"
```

Lösung ohne Lookbehind:

```
> gsubfn("(\\d)(?=(\\d\\d\\d)+(?!\\d))", function(x, ...)
   paste(x, ".", sep=""), s, perl=T)

[1] "Eine 12.344.567.788 Zahl"
```

Negative Lookaheads `(?<!  )` und Lookbehinds `(?!  )`

Anfang eines Wortes

Text innerhalb von Klammern extrahieren.

```
> s <- c("im Jahr (2001) war was", "in (der Klammer) test")
> str_extract(s, perl("(?<=\\().*(?=\\))"))

[1] "2001"        "der Klammer"

>


> s <- "einige Worte in Folge"
> str_replace_all(s, perl("(?<!\\w)(?=\\w)"), "*")

[1] "*einige *Worte *in *Folge"
```

Ende eines Wortes

```
> s <- "einige Worte in Folge"
> str_replace_all(s, perl("(?<=\\w)(?!\\w)"), "*")

[1] "einige* Worte* in* Folge*"
```

0.3 Beispiele

Einen string zwischen Zahl und Buchstabe splitten mit Lookbehind und Lookahead.

```
> s <- c("V123", "v12")
> strsplit(s, "(?<=[A-Za-z])(?=\\d+)", perl=T)

[[1]]
[1] "V"   "123"

[[2]]
[1] "v"  "12"
```