Unix & Shell-Programmierung SS21

Vorlesungswoche 7

Helga Karafiat

FH Wedel

Erinnerung: Werkzeuge zur Textverarbeitung

Befehl	Beschreibung
`cat`	Text ausgeben, neutrales Element bzgl. |
`cut`	Spalten selektieren und ausschneiden
`sort`	Daten zeilenweise sortieren
`uniq`	doppelte aufeinanderfolgende Zeilen löschen
`wc`	Zeichen, Wörter und Zeilen zählen
`tr`	Zeichen übersetzen bzw Zeichen löschen
`grep`	Zeichenstrom zeilenweise filtern
`sed`	Textmanipulation auf Zeichenströmen
`awk`	Programmiersprache zum Manipulieren von Texten (Pattern Matching Language)
`sh`	und viele andere (z.B. `bash`, `csh`, `perl`, `tcl`, `ruby`): Sprachen zum Erstellen von eigenen komplexen Filtern

Einfache Tools zum Arbeiten mit regulären Ausdrücken

- `grep`
  - `global\regularexpression\print` bzw.
    `global search for a regular expression and print out matched lines`
  - sucht Textzeilen anhand eines angegebenen Musters heraus
  - verschiedene Varianten
    - ★ `grep -E` (oder `egrep`): Verarbeitung von erweiterten regulären Ausdrücken
    - ★ `grep -F` (oder `fgrep`): Auswertung Ausdrücke als feste Zeichenketten (statt regulärer Ausdruck)
- `sed`
  - `stream editor`
  - Erlaubt die Manipulation von Zeichenketten in Zeichenströmen
  - arbeitet zeilenweise
  - hauptsächliche Verwendung: Suchen und Ersetzen
  - `sed -E` um erweiterte reguläre Ausdrücke zu verwenden
- Vorsicht `GNU` (Linux) und `BSD` (MacOS) verhalten sich was reguläre Ausdrücke angeht teilweise sehr unterschiedlich und auch abweichend zu POSIX!

Reguläre Ausdrücke

- Aufgaben: Suchen nach Textmustern (Filtern) oder Suchen und Ersetzen von Textmustern
- Reguläre Ausdrücke (vereinfacht ausgedrückt): Beschreibungssprache für Textmuster
- Motivation:
  - Automatisierung von wiederkehrenden Aufgaben
  - Vermeidung von "Äffchenarbeit", wie
    "Erstelle eine Liste aller Überschriften aus diesen 100 html-Dateien."
    oder "Ersetze alle Vorkommen von 'foo' durch 'bar'."
- Einfachstes Beispiel für einen regulären Ausdruck schon bekannt:
  `grep TODO`
  - sucht alle Zeilen in denen die Zeichenkette "TODO" enthalten ist
    (genaugenommen: Ein `T` gefolgt von einem `O` gefolgt von einem `D` gefolgt von einen `O`).
  - einfachster regulärer Ausdruck ist also eine Zeichenkette
- Unterscheidung laut POSIX in
  Basic Regular Expressions (BRE) und Extended Regular Expressions (ERE)

Aufrufsyntax grep

- `grep [OPTIONS]... PATTERN [FILE]...`
- Wichtige Optionen
  - `-i` ignoriert Groß-/Kleinschreibung
  - `-v` gibt alle Zeilen aus, die nicht matchen
  - `-q` stiller Modus: `grep` gibt keine Matches aus, sondern setzt lediglich Exitcode entsprechend
  - `-s` unterdrückt Fehlermeldungen (häufig zusammen mit `-q`)
  - `-e PATTERN` notwenig zur Angabe mehrerer Patterns
  - `-C ZAHL` Kontext: gibt ZAHL Zeilen um den Match aus (-A für nur dahinter, -B für nur davor)
  - `-n` gibt die Zeilennummern mit aus
- Beispiel: Alle Zeilen mit Vorkommen der Zeichenkette *foo* in der Datei foo.txt finden
  `cat foo.txt | grep 'foo'`
- Tipp: Reguläre Ausdrücke am Besten immer in Quotes schreiben
  (je nach gewünschtem Effekt `weak` oder `strong`)

Aufrufsyntax sed

- `sed [OPTION]... 'COMMAND' [FILE]...`
- `COMMAND`: `s/SUCHWORT/ERSETZUNG/FLAG`
  - `s` am Anfang steht für substitute, also Ersetzen
    (andere Zeichen nur für komplexere Skripte vonnöten)
  - `SUCHWORT`: gesuchter Ausdruck
  - `ERSETZUNG`: das wodurch ersetzt werden soll
  - `FLAG`: steuert Verhalten von `sed`
    - `i`: case insenstive
    - `g`: global, ersetzt also nicht nur das erste Vorkommen in der Zeile, sondern alle Vorkommen
    - `ZAHL`: Ersetzt das ZAHLte Vorkommen in einer Zeile
    - `p`: wenn `-n` aktiv ist, modifizierte Zeile ausgeben
  - `/`: Trennzeichen
    - `/` wird klassisch als Trennzeichen verwendet, kann aber auch jedes andere Zeichen sein
      (das Zeichen wird dadurch definiert, dass es nach dem `s` steht)
    - häufige Trennzeichen: `,`, `;`, `|`, ...

- POSIX-konforme Optionen
  - `-e COMMAND` zur Angabe von mehreren Ausdrücken
  - `-f COMMAND_FILE` zur Angabe von Ausdrücken über eine Skript-Datei
  - `-n` Unterdrücke die Ausgabe der modifizierten Zeilen, stattdessen Ausgabe explizit durch `p`
- `GNU sed` verfügt über Vielzahl an Erweiterungen und Optionen (siehe `man sed`), `--posix` schaltet GNU-Erweiterungen aus (Vorsicht: Option nicht POSIX-konform)
- Beispiele:
  - Alle Vorkommen der Zeichenkette *foo* aus foo.txt durch *bar* ersetzen
    `cat foo.txt | sed 's/foo/bar/g'`
  - Als Ersetzung kann auch nichts angegeben werden, das entspricht dann löschen, z.B. `sed 's/nichts//'`
- Vorsicht: Da `sed` zeilenbasiert arbeitet, können Zeilenumbrüche nicht ohne weiteres gelöscht werden, dafür bietet sich dann `tr` an

Verwendung regulärer Ausdrücke in vi

- `vi` hat gleiche Funktionalität für reguläre Ausdrücke wie `grep` & `sed`
- Suchen im Befehlsmodus mit `/` auch mit regulären Ausdrücken möglich z.B. `/foo`
- Im Befehlsmodus kann mit `:%s` ein Ersetzungsbefehl eingeleitet werden
  z.B. `:%s/foo/bar/g` (% steht für ganze Datei)
- Im `visual-Mode` kann Suchen und Ersetzen auch auf der Auswahl durchgeführt werden
  - Wechseln in den visual-Mode mit v oder V (visual line)
  - Markieren mit den Richtungstasten (h, j, k, l)
  - Eintippen von : führt dazu, dass die Zeilenangabe `'<,'>` eingesetzt wird (ersetzt das %)
  - danach `s` eingeben leitet den Ersetzungsbefehl ein
  - Beispiel: `:'<,'>s/foo/bar/g`
- Nettes vim-Feature: Wenn der Suchteil des Ausdrucks leer gelassen wird, nimmt vim den letzten Ausdruck nach dem mit / gesucht wurde

Aufbau regulärer Ausdrücke

- Reguläre Ausdrücke sind zeichenweise aufgebaut und werden auch Zeichen für Zeichen von links nach rechts ausgewertet
- jedes Zeichen hat eine eindeutig definierte Bedeutung
- Reguläre Ausdrücke bestehen aus
  - Literalen ("normalen" Zeichen) und
  - Metazeichen (Zeichen mit einer besonderen Bedeutung)
- Einfachster regulärer Ausdruck: Zeichenkette aus Literalen (Wort)
- Besonderes Metazeichen: Backslash (\)
  - wird verwendet um
    - manche Metazeichen einzuleiten oder
    - andere Metazeichen zu maskieren, so dass sie als "normales" Zeichen gelesen werden
  - ist selbst immer ein Metazeichen, kann aber mit sich selbst maskiert werden (\\)

Wie “matchen” reguläre Ausdrücke?

- Dass ein regulärer Ausdruckt “matcht” bedeutet, dass er auf einen Teil des Textes (oder den ganzen Text) passt
- Beispiele

```
echo "Hallo Welt" | grep 'Hallo'
      ^^^^^

echo "Hallo Welt" | grep ' Welt'
           ^^^^^

echo "Hallo Welt" | grep 'H'
      ^

echo "Hallo Hallo Welt" | grep 'H'
      ^     ^
```

Metazeichen: Zeilenanker

- Zeilenanker sind:
  - Zeilenanfang, Schreibweise: ^
  - Zeilenende, Schreibweise: $
- Beispiele

```
echo "Hallo Welt" | grep '^Hallo'
      ^^^^^

echo "Hallo Hallo Welt" | grep '^Hallo'
      ^^^^^

echo "Hallo Welt" | grep ' Welt$'
           ^^^^^

echo "Hallo Welt" | grep '^Hallo Welt$'
      ^^^^^^^^^^
```

Metazeichen: Zeichenklassen

- Schreibweise `[...]`
- Bedeutung: beschreiben eine gültige Eingabemenge für ein Zeichen
- Beispiel

```
echo "Hallo Hello Hullo" | grep 'H[ae]llo'
      ^^^^^ ^^^^^
```

  matcht nur auf *Hallo* oder *Hello*, aber sonst nichts!

- Weiteres Beispiel nur Vokale: `[aeiou]`
- Intervallschreibweise möglich z.B. `[a-z]`,`[A-Z]` oder `[0-9]`
- Beliebige kombinierte Schreibweise möglich, z.B. `[a-zA-Z0-9_]`
  (Buchstabe, Zahl oder Unterstrich)

- Negation möglich
  - Syntax: `[^...]`
  - Beispiel `[^xyz]` alle Zeichen außer `x`, `y` und `z`
  - Vorsicht `[^x]` (jedes Zeichen außer `x`) erwartet ein Zeichen und matcht nicht auf leer
    Beispiel: `echo "Hausbau" | grep 'au[^s]'`
- Andere Metazeichen verlieren innerhalb von `[]` ihre Bedeutung
  - kein Quoting notwendig,
    z.B. `[.ab$]` erlaubt das Zeichen: `.`, `a`, `b` oder `$`
  - Sonderfälle: `-`, `]` und `^`
    - wenn nach `^` gesucht wird, darf es nicht nach der öffnenden eckigen Klammer stehen
    - wenn nach `-` gesucht wird, muss es entweder nach der öffnenden oder vor der schließenden eckigen Klammer stehen
    - wenn nach `]` gesucht wird, muss es nach der öffnenden eckigen Klammer stehen
- Beispiel (Matche Großbuchstaben, `[`, `]`, `-`, `^`):

```
echo "Hallo [Name] - wie geht es Dir ^^?" | grep '[][^A-Z-]'
```

Metazeichen: Besondere Zeichenklasse - beliebiges Zeichen

- Schreibweise: `.`
- Bedeutung: beschreibt genau ein beliebiges Zeichen
- Beispiele
  - *Hallo*, *Hello* u.s.w.

    ```
    echo "Hallo Hello Hullo H llo HEllo Hllo" | grep 'H.llo'
          ^^^^^ ^^^^^ ^^^^^ ^^^^^ ^^^^^
    ```

  - Flexible Datumsangabe:

    `[0-3][0-9].[0-3][0-9].[0-9][0-9][0-9][0-9]`

    matcht auf: `14.06.2020`, `06/14/2020`, `14-06-2020`

    aber auch auf: `0123456789`

Metazeichen: Wiederholung

- Schreibweise: `*`
- Bedeutung: Zeichen davor kann vorkommen, also 0 mal bis beliebig oft
- Beispiele

```
echo "Hallo     Welt" | grep 'Hallo *Welt'
      ^^^^^^^^^^^^^^

echo "HalloWelt" | grep 'Hallo *Welt'
      ^^^^^^^^^
```

- kann auf jedes einzelne Zeichen angewendet werden, z.B. auch auf `[...]`
- Beispiel beliebig lange positive Ganzzahl: `[0-9]*`
  - Vorsicht: Erlaubt auch leere Zeichenketten oder solche ohne Zahlen
  - Abhilfe: `[0-9][0-9]*` (mindestens eine Zahl muss vorhanden sein)

- Häufige Kombination `.*` (beliebiges Zeichen, beliebig oft)
  - Vorsicht: Reguläre Ausdrücke sind meistens “greedy” bzw arbeiten nach dem “longest match” Prinzip, d.h. es wird so viel gematcht, wie irgendwie geht
  - Beispiel

```
echo "<tag1>Text</tag1>" | grep '<.*>'
      ^^^^^^^^^^^^^^^^^
```

- Vorsicht: `*` in regulären Ausdrücken ist nicht vergleichbar mit `*` aus den Datei-Wildcards
  - regulärer Ausdruck: Bezug immer auf das davor stehende Zeichen
  - Wildcard: Bezug zu im Dateisystem vorhandenen Dateien

Metazeichen: Intervallausdrücke

- Schreibweise: `\{n,m\}`
- Bedeutung: das Zeichen davor muss mindestens `n` und höchstens `m` mal vorkommen
- Variationen:
  - `\{n\}`: Das Zeichen kommt genau `n` mal vor
  - `\{n,\}`: Das Zeichen kommt mindestens `n` mal vor
- Einfaches Beispiel:

  ```
  echo "Hallo" | grep 'l\{3,\}'
  ```

  matcht nicht, weil nur zwei aufeinanderfolgende `l` enthalten sind und nicht mindestens drei

- Weitere Beispiele

```
echo "aaabbc aabc a aaaab aaaaa" | grep 'a\{2,3\}'
      ^^^    ^^     ^^^   ^^^^^

echo "aaabbc aabc gnargl aaaab" | grep 'a\{2\}'
      ^^     ^^          ^^^^

echo "aaabbc aabc gnargl aaaab" | grep 'a\{3,\}'
      ^^^                ^^^^

echo "aaab ab aabc aaaaac" | sed 's/a\{2,3\}/b/g'
```

Metazeichen: Rückwärtsreferenz / Rückbezug (Backreference)

- Schreibweise: `\(...\)` und `\1` bis `\9`
- Bedeutung:
  "Filtere das was auch immer ein bestimmter früherer Teil des Ausdrucks gefiltert hat"
- Vorgehen in zwei Stufen
  - Bis zu neun Unterausdrücke können in `\(...\)` eingeschlossen werden (Verschachtelung ist möglich)
  - In späterem Teil des gleichen Musters kann `\m` eingesetzt werden, wobei `\m` dann für den gematchten Inhalt des `m`-ten Klammerpaars `\(...\)` steht
- Einfaches Beispiel: Eine Zeile mit zwei aufeinanderfolgenden Vorkommen von *das*

```
echo "Das hier hat nur ein das" | grep '\(das\) \1'

echo "Häufiger Vertipper: das das" | grep '\(das\) \1'
                                    ^^^^^^^
```

- Komplexeres Beispiel:
  Auffinden von allen entweder in einfachen (') oder doppelten (") Hochkommata geklammerten Worten
- Naiver Ansatz:

  ```
  ["'].*["']
  ```

  Schwäche: Findet z.B. auch "Hallo'
- Lösung:

  ```
  \(["']\).*\1
  ```

  Findet nur noch passende Klammerpaare
- Vorsicht mit dem Quoting in der Shell!

- Häufiger Einsatzbereich für Rückersetzung: Suchen und Ersetzen
- Beispiel: Vertauschen von Spalten in einer csv-Datei:

```
Hallo;123;
Test;234;
Welt;1;
```

Vertauschen der Spalten mit

```
cat test.csv | sed 's/\([a-zA-Z]*\);\([0-9]*\);/\2;\1;/'
```

Andere Möglichkeit (genereller bzgl Spalteninhalt)

```
cat test.csv | sed 's/\(.*;\)\(.*;\)/\2\1/'
```

Erweiterte reguläre Ausdrücke

- Alle Ausdrücke bisher Bestandteil von Basic Regular Expressions (BRE)
- Extend Regular Expressions (ERE) bieten einige nützliche Erweiterungen und sind je nach Anwendungfsfall ggfs besser lesbar
- Unterscheidung zwischen BRE und ERE ist historisch gewachsen:
  - Reguläre Ausdrücke sind mathematisches Konzept aus den 1940er Jahren (als einfache Beschreibungssprache für sogennante reguläre Mengen)
  - Erster Zusammenhang mit Computern 1968
    Paper: *Regular Expression Search Algorithm* von Ken Thompsen
  - `ed` der erste UNIX-Editor konnte bereits reguläre Ausrücke
  - Bezeichner `grep` kommt aus dem Befehl für Suchen und Ausgeben von `ed` (reguläre Ausdrücke Engine aus `ed` wurde aufgrund der hohen Beliebtheit als eigenes Programm `grep` umgesetzt)
  - `grep` und `egrep` waren ursprünglich zwei verschiedene Entwicklungen
  - POSIX-BRE orientieren sich am ursprünglichen Umfang von `grep`
  - POSIX-ERE orientieren sich am ursprünglichen Umfang von `egrep`

- Änderungen ERE im Vergleich zu BRE:
  - `{n,m}` statt `\{n,m\}`
    dafür müssen geschweifte Klammern maskiert werden, wenn explizit danach gesucht wird
  - `(...)` statt `\(...\)`
    Sonderfall: Unterstützt gemäß POSIX Gruppierung, aber keine Rückwärtsreferenz!
  - Quantoren (Wiederholung, Intervallausdrücke) können sich durch Gruppierung jetzt auch auf komplette reguläre Ausdrücke oder Wortketten beziehen: z.B. `(ab)*`
- Erweiterungen:
  - Quantor: `+`
    Zeichen bzw. Ausdruck darf 1 mal oder beliebig oft vorkommen - entspricht `{1,}`
  - Quantor: `?`
    Zeichen bzw. Ausdruck darf 0 oder 1 Mal vorkommen - entspricht `{0,1}`
  - Alternation `|`
    matcht wenn einer der Ausdrücke matcht ("Oder-Verknüpfung"), häufig in Kombination mit `()`

- Beispiele:
  - *Hallo* oder *Hello*

```
echo "Hallo Hello Hullo" | grep -E 'Hallo|Hello'
      ^^^^^ ^^^^^

echo "Hallo Hello Hullo" | grep -E 'H(a|e)llo'
      ^^^^^ ^^^^^
```

  - Suchen nach Zeilen, die mit *Subject* oder *To* oder *From* beginnen

```
^(Subject|To|From)
```

  - Suchen nach Zeichenketten aus mindestens 2 Zahlen oder Zeichenketten aus 3 Großbuchstaben

```
echo "123 H4110 FOOBar" | grep -E '[0-9]{2,}|[A-Z]{3}'
      ^^^  ^^^^ ^^^
```

Zeichen	BRE/ERE	Bedeutung in einem Muster
`\`	Beide	Maskierung des folgenden Zeichens oder besondere Bedeutung aktivieren (je nach Kontext)
`.`	Beide	Beliebiges Zeichen
`*`	Beide	Beliebige Anzahl des Zeichens davor (0..). Bei ERE auch auf `()`-Ausdruck anwendbar.
`^`	Beide	Zeilenanfang
`$`	Beide	Zeilenende
`[...]`	Beide	Zeichenklasse: Filterung eines bestimmten Zeichens; Angabe als mögliche Zeichen oder Bereich(e)
`\{n,m\}`	BRE	Intervallausdruck: Zeichenhäufigkeit des Zeichens davor, auch `\{n\}` (genau `n`) oder `\{n,}` (mindestens `n`)
`\( \)`	BRE	Speichert den zum Muster gefundenen Text, kann später mit Escape-Sequenz `\1` bis `\9` wieder eingesetzt werden
`\m`	BRE	Verweist auf das m-te Suchmuster bzw. den dazu gemachten Text (m=1..9)
`{n,m}`	ERE	Genau wie `\{n,m\}` aus BRE
`+`	ERE	Filtert eine mehrere Instanzen des Ausdrucks davor (1-n)
`?`	ERE	Filtert keine oder eine Instanz des Ausdrucks davor (0-1)
`|`	ERE	Filtert den Ausdruck, der davor oder dahinter angegeben ist
`( )`	ERE	Wendet Filter auf die eingeschlossene Gruppe Ausdrücke an

Erweiterungen zum POSIX-Standard

- Hinweis: Rückersetzung im Sinne von `\( \)` und `\1..\9` ist laut POSIX nicht für erweiterte reguläre Ausdrücke definiert
  - Grund: Rückbezüge sind langsam können im schlimmsten Fall exponentielle Laufzeit haben (siehe BUGS-Section in `man grep`)
  - Aber: Alle GNU Tools unterstützen die Rückersetzung auch für `( )` auch bei EREs, BSD tut das nicht (Manpages lesen!)
- Alle GNU Tools unterstützen zudem `\?`,`\+` und `\|` in BREs (analog zu `?`, `+` und `|` aus ERE)
- Nützliche Einschränkung: Wörter definieren
  - Analog zu Zeilenanfang und Zeilenende, gibt es auch Wortanfang und Wortende
  - Wortanfang: `\<`
  - Wortende: `\>`
  - Wortbildung von POSIX nur für den Editor `ed` standardisiert
  - `GNU grep` und `GNU sed` unterstützen diese Option eigentlich immer (Manpage lesen!)

- Beispiele Wortdefinition

```
echo "aaa aaab abcaaa" | grep '\<aaa'
      ^^^ ^^^

echo "aaa aaab abcaaa" | grep 'aaa\>'
      ^^^         ^^^

echo "aaa aaab abcaaa" | grep '\<aaa\>'
      ^^^
```

- `GNU grep` hat spezielle Option `-w` mit der es sich in den Wort-Modus versetzen lässt

```
echo "aaa aaab abcaaa" | grep -w 'aaa'
      ^^^
```

Spezielle POSIX-Klammerausdrücke für Zeichenklassen

- beschreiben Klassen von speziellen Zeichen
- Hintergründe:
  - für nicht-englische Sprache reichen die einfachen Bereiche, wie a-z, manchmal nicht mehr aus, wie z.B. für die Umlaute im Deutschen
  - das Zusammenfassen von bestimmten Zeichen zu "größeren" benannten Klassen (z.B. für Whitespace, alphanumerische Zeichen) wünschenswert
- Nur innerhalb von `[...]` gültig
- Inhalte sind ggfs. abhängig von der eingestellten `locale`

- POSIX Zeichen-Äquivalenzklassen
  - beschreiben Mengen von Zeichen, die als äquivalent angesehen werden
  - z.B. bei französischer `locale`: `[=e=]` entspricht den Zeichen e, è, é, ê

    ```
    echo "Thérèse" | grep '[[=e=]]'
            ^ ^ ^
    ```

    oder z.B. mit deutscher `locale`

    ```
    echo "Hallo Hällö!" | grep 'H[[=a=]]ll[[=o=]]'
          ^^^^^ ^^^^^
    ```

- POSIX Kollationssequenzen
  - bestimmen wie besondere bzw. zusammengesetzte Zeichen z.B. bei der Sortierung und beim Matching behandelt werden (abhängig von der `locale`)
  - z.B. `ll` im Spanischen, welches als einzelnes Zeichen behandelt wird und zwischen `l` und `m` einsortiert

    ```
    echo "tortilla" | grep `torti[[.span-ll.]]a`
          ^^^^^^^^
    ```

- Weitere POSIX Zeichenklassen (`locale`-unabhängig)

Klasse	Filtert Zeichen
`[:alnum:]`	Alphanumerische Zeichen (alphabetisch und numerisch)
`[:alpha:]`	Alphabetische Zeichen
`[:blank:]`	Leer- und Tabulatorzeichen
`[:cntrl:]`	Steuerzeichen
`[:digit:]`	Ziffern
`[:graph:]`	"Schwarze" Zeichen (nicht Whitespace, nicht Steuerzeichen, etc.)
`[:lower:]`	Kleinbuchstaben
`[:print:]`	Druckare Zeichen (wie `[:graph:]` aber plus Leerzeichen)
`[:punct:]`	Interpunktionszeichen
`[:space:]`	Whitespace-Zeichen (wie `[:blank:]` aber plus Zeilenumbrüche etc.)
`[:upper:]`	Großbuchstaben
`[:xdigit:]`	Hexdezimale Ziffern

- Beispiel

```
echo "Hallo123#" | grep '[[:digit:]]'
           ^^^
```

Vorrang der Operatoren

Operator	Bedeutung
`[..] [==] [::]`	Klammersymbole für Zeichenvergleiche und Zeichenketten
`\metazeichen`	Geschützte Metazeichen
`[]`	Klammerausdrücke
`\( \) \ziffer` bzw. `()`	Unterausdrücke und Rückbezüge (BRE) bzw. Gruppierung (ERE)
`* \{ \}` bzw. `* + ? { }`	Wiederholung des vorstehenden Zeichens (BRE) bzw. vorstehenden Ausdrucks (ERE)
kein Symbol	Verkettung, z.B. `ab`
`^ $`	Anker
`|`	Alternation (nur ERE)

Sortierung nach höchster bis niedrigster Vorrang

- Stolperfallen:
  - `def*` entspricht `de(f*)` und nicht `(def)*`
  - `[a-c][d-f]*` entspricht `[abc]([d-f])*` und nicht `([a-c][d-f])*`

Ein paar Echtwelt-Beispiele (viele Wege führen nach Rom)

- Positive Ganzzahlen (führende Nullen erlaubt)
  - `^[0-9][0-9]*$` (BRE/ERE)
  - `^[0-9]\{1,\}$` (nur BRE)
  - `^[0-9]+$` (nur ERE)
  - ...
- Erweiterung: Positive und negative Ganzzahlen
  (führende Nullen erlaubt, Plus und Minus als Präfix erlaubt)
  - `^[+-]\{0,1\}[0-9]\{1,\}$` (nur BRE)
  - `^[-+]?[0-9]+$` (nur ERE)
  - `^(+|-)?[0-9]{1,}$` (nur ERE)
  - ...