Reguläre Ausdrücke

Proseminar UNIX Tools

Ferdinand Beyer

Technische Universität München

08.11.2005

Gliederung

Gliederung

Beschreibung

Reguläre Ausdrücke

- dienen der Beschreibung einer **formalen Sprache**

Beschreibung

Reguläre Ausdrücke

- dienen der Beschreibung einer **formalen Sprache**
- d.h. sie beschreiben eine **Menge von Zeichenketten**

Beschreibung

Reguläre Ausdrücke

- dienen der Beschreibung einer **formalen Sprache**
- d.h. sie beschreiben eine **Menge von Zeichenketten**
- werden üblicherweise als **Stringmuster** (engl.: *pattern*) notiert

Beschreibung

Reguläre Ausdrücke

- dienen der Beschreibung einer **formalen Sprache**
- d.h. sie beschreiben eine **Menge von Zeichenketten**
- werden üblicherweise als **Stringmuster** (engl.: *pattern*) notiert
- lassen sich relativ einfach mit **endlichen Automaten** implementieren

Gliederung

Bezeichnungen

Gebräuchliche Namen und Abkürzungen

- Englisch: *regular expressions*

Bezeichnungen

Gebräuchliche Namen und Abkürzungen

- Englisch: *regular expressions*
- Abkürzungen: RegExp, Regex, RE

Bezeichnungen

Gebräuchliche Namen und Abkürzungen

- Englisch: *regular expressions*
- Abkürzungen: RegExp, Regex, RE
- In `man`-Pages oft `PATTERN`

Gliederung

Einsatzgebiete in UNIX-Tools

Einsatzgebiete für reguläre Ausdrücke

- Suchen (und eventuell Ersetzen) von Zeichenketten eines bestimmten Musters in Texten
  (z.B. *vi*, *emacs*, *sed*)

Einsatzgebiete in UNIX-Tools

Einsatzgebiete für reguläre Ausdrücke

- Suchen (und eventuell Ersetzen) von Zeichenketten eines bestimmten Musters in Texten
  (z.B. *vi*, *emacs*, *sed*)
- Filtern von Informationen
  (z.B. *grep*)

Einsatzgebiete in UNIX-Tools

Einsatzgebiete für reguläre Ausdrücke

- Suchen (und eventuell Ersetzen) von Zeichenketten eines bestimmten Musters in Texten
  (z.B. *vi*, *emacs*, *sed*)
- Filtern von Informationen
  (z.B. *grep*)
- Erkennen und Klassifizieren von Teilstrings (*tokens*) für die lexikalische Analyse von Texten ($\rightarrow$ Compilerbau)
  (z.B. *lex*)

Gliederung

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)
- *Perl / Perl-compatible regular expressions* (PCRE)

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)
- *Perl / Perl-compatible regular expressions* (PCRE)

Hinweise

- BREs existieren hauptsächlich noch für Abwärtskompatibilität mit älteren Programmen

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)
- *Perl / Perl-compatible regular expressions* (PCRE)

Hinweise

- BREs existieren hauptsächlich noch für Abwärtskompatibilität mit älteren Programmen
- Im Nachfolgenden wird die Notation für EREs besprochen

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)
- *Perl / Perl-compatible regular expressions* (PCRE)

Hinweise

- BREs existieren hauptsächlich noch für Abwärtskompatibilität mit älteren Programmen
- Im Nachfolgenden wird die Notation für EREs besprochen
- Anschließend werden die Unterschiede zu BREs aufgezeigt

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)
- *Perl / Perl-compatible regular expressions* (PCRE)

Hinweise

- BREs existieren hauptsächlich noch für Abwärtskompatibilität mit älteren Programmen
- Im Nachfolgenden wird die Notation für EREs besprochen
- Anschließend werden die Unterschiede zu BREs aufgezeigt
- PCREs entsprechen weitgehend EREs

Notationsarten

Die wichtigsten Formen von regulären Ausdrücken

- *Basic ("simple"/"obsolete") regular expressions* (BRE)
- *Extended ("modern") regular expressions* (ERE)
- *Perl / Perl-compatible regular expressions* (PCRE)

Hinweise

- BREs existieren hauptsächlich noch für Abwärtskompatibilität mit älteren Programmen
- Im Nachfolgenden wird die Notation für EREs besprochen
- Anschließend werden die Unterschiede zu BREs aufgezeigt
- PCREs entsprechen weitgehend EREs
- Relevant für UNIX-Tools sind nur BREs und EREs

Gliederung

Gliederung

Zeichen und Metazeichen

- Ein regulärer Ausdruck besteht aus **Zeichenliteralen** und **Metazeichen**

Zeichen und Metazeichen

- Ein regulärer Ausdruck besteht aus **Zeichenliteralen** und **Metazeichen**
- Die Metazeichen `[ ] ( ) { } ? + * ^ $ \ .` haben eine besondere Bedeutung für den Ausdruck

Zeichen und Metazeichen

- Ein regulärer Ausdruck besteht aus **Zeichenliteralen** und **Metazeichen**
- Die Metazeichen `[ ] ( ) { } ? + * ^ $ \ .` haben eine besondere Bedeutung für den Ausdruck
- Sonstige Zeichen sind Zeichenliterale und stehen für sich selbst

Zeichen und Metazeichen

- Ein regulärer Ausdruck besteht aus **Zeichenliteralen** und **Metazeichen**
- Die Metazeichen `[ ] ( ) { } ? + * ^ $ \ .` haben eine besondere Bedeutung für den Ausdruck
- Sonstige Zeichen sind Zeichenliterale und stehen für sich selbst
- Um Metazeichen als Zeichenliterale zu behandeln, müssen sie mit vorangestelltem **Backslash** \ notiert werden ("maskieren"; engl.: "to escape")

Zeichen und Metazeichen

- Ein regulärer Ausdruck besteht aus **Zeichenliteralen** und **Metazeichen**
- Die Metazeichen `[ ] ( ) { } ? + * ^ $ \ .` haben eine besondere Bedeutung für den Ausdruck
- Sonstige Zeichen sind Zeichenliterale und stehen für sich selbst
- Um Metazeichen als Zeichenliterale zu behandeln, müssen sie mit vorangestelltem **Backslash** \ notiert werden ("maskieren"; engl.: "to escape")

Beispiele

- `au` findet "au" in "Haus" und "Maus"
- `\(TODO\)` findet den String "(TODO)"

Gliederung

Beliebige Zeichen

- Ein **Punkt** . deckt ein beliebiges Zeichen ab

Beliebige Zeichen

- Ein **Punkt** . deckt ein beliebiges Zeichen ab
- Einzige Ausnahme ist das **Newline**-Zeichen, dass in den meisten Implementierungen standardmäßig nicht vom Punkt getroffen wird.

Beliebige Zeichen

- Ein **Punkt** . deckt ein beliebiges Zeichen ab
- Einzige Ausnahme ist das **Newline**-Zeichen, dass in den meisten Implementierungen standardmäßig nicht vom Punkt getroffen wird.

Beispiel

`.and` findet “Hand”, “Land”, “Band”, “Rand”, “Wand” etc.

Gliederung

Zeichenauswahlen (1)

- Eine **Zeichenauswahl** ist eine Liste von Zeichen, die in **eckige Klammern** [ ] eingeschlossen notiert wird

Zeichenauswahlen (1)

- Eine **Zeichenauswahl** ist eine Liste von Zeichen, die in **eckige Klammern** [ ] eingeschlossen notiert wird
- Die Auswahl deckt **genau ein Zeichen** der Liste ab

Zeichenauswahlen (1)

- Eine **Zeichenauswahl** ist eine Liste von Zeichen, die in **eckige Klammern** [ ] eingeschlossen notiert wird
- Die Auswahl deckt **genau ein Zeichen** der Liste ab
- Mit dem **Bindestrich** - können **Zeichenbereiche** in die Auswahl aufgenommen werden

Zeichenauswahlen (1)

- Eine **Zeichenauswahl** ist eine Liste von Zeichen, die in **eckige Klammern** `[ ]` eingeschlossen notiert wird
- Die Auswahl deckt **genau ein Zeichen** der Liste ab
- Mit dem **Bindestrich** - können **Zeichenbereiche** in die Auswahl aufgenommen werden

Beispiele

- `[LHM][au]nd` findet "Land", "Hand", "Hund", "Mund", etc.
- `[0-9a-fA-F]` findet eine Ziffer in Hexadezimal-Schreibweise

Zeichenauswahlen (2)

- Beginnt die Auswahl mit einem **Zirkumflex** ^, so handelt es sich um eine **negierte Auswahl**, d.h. sie repäsentiert alle Zeichen, die **nicht** in der Liste stehen

Zeichenauswahlen (2)

- Beginnt die Auswahl mit einem **Zirkumflex** ^, so handelt es sich um eine **negierte Auswahl**, d.h. sie repäsentiert alle Zeichen, die **nicht** in der Liste stehen
- Soll die Auswahl das Zeichen] selbst enthalten, muss dieses als **erstes Zeichen der Liste** notiert werden

Zeichenauswahlen (2)

- Beginnt die Auswahl mit einem **Zirkumflex** ^, so handelt es sich um eine **negierte Auswahl**, d.h. sie repäsentiert alle Zeichen, die **nicht** in der Liste stehen
- Soll die Auswahl das Zeichen] selbst enthalten, muss dieses als **erstes Zeichen der Liste** notiert werden
- Andere Metazeichen (inklusive \) verlieren innerhalb von Auswahlen ihre Sonderbedeutung und müssen nicht maskiert werden

Zeichenauswahlen (2)

- Beginnt die Auswahl mit einem **Zirkumflex** `^`, so handelt es sich um eine **negierte Auswahl**, d.h. sie repäsentiert alle Zeichen, die **nicht** in der Liste stehen
- Soll die Auswahl das Zeichen `]` selbst enthalten, muss dieses als **erstes Zeichen der Liste** notiert werden
- Andere Metazeichen (inklusive \) verlieren innerhalb von Auswahlen ihre Sonderbedeutung und müssen nicht maskiert werden

Beispiel

`[^dwDW]..` findet drei-buchstabige Wörter, die nicht mit “d” oder “w” beginnen

Zeichenauswahlen (3)

- Für einige Zeichenauswahlen existieren vordefinierte **Zeichenklassen**

Zeichenauswahlen (3)

- Für einige Zeichenauswahlen existieren vordefinierte **Zeichenklassen**
- Diese werden **innerhalb** von Zeichenauswahlen zwischen `[:` und `:]` notiert

Zeichenauswahlen (3)

- Für einige Zeichenauswahlen existieren vordefinierte **Zeichenklassen**
- Diese werden **innerhalb** von Zeichenauswahlen zwischen `[:` und `:]` notiert

Verfügbare Zeichenklassen

`[:alnum:]`	Buchstaben & Ziffern	`[:blank:]`	Leerzeichen & Tabulator
`[:alpha:]`	Buchstaben	`[:space:]`	Jede Art von Leerzeichen
`[:lower:]`	Kleinbuchstaben	`[:punct:]`	Interpunktionszeichen
`[:upper:]`	Großbuchstaben	`[:print:]`	Alle druckbaren Zeichen
`[:digit:]`	Ziffern	`[:graph:]`	Druckbare, nicht-leere Zeichen
`[:xdigit:]`	Hexadezimal-Ziffern	`[:cntrl:]`	Kontrollzeichen

Zeichenauswahlen (3)

- Für einige Zeichenauswahlen existieren vordefinierte **Zeichenklassen**
- Diese werden **innerhalb** von Zeichenauswahlen zwischen `[:` und `:]` notiert

Verfügbare Zeichenklassen

`[:alnum:]`	Buchstaben & Ziffern	`[:blank:]`	Leerzeichen & Tabulator
`[:alpha:]`	Buchstaben	`[:space:]`	Jede Art von Leerzeichen
`[:lower:]`	Kleinbuchstaben	`[:punct:]`	Interpunktionszeichen
`[:upper:]`	Großbuchstaben	`[:print:]`	Alle druckbaren Zeichen
`[:digit:]`	Ziffern	`[:graph:]`	Druckbare, nicht-leere Zeichen
`[:xdigit:]`	Hexadezimal-Ziffern	`[:cntrl:]`	Kontrollzeichen

- Der Vorteil von `[[:alnum:]]` gegenüber `[a-zA-Z0-9]` ist die Unterstützung von **internationalen Zeichen** je nach Systemeinstellung (*Locale*)

Gliederung

Quantoren

- Mit **Quantoren** kann angegeben werden, **wie oft** ein Element vorkommen darf. bzw. muss

Quantoren

- Mit **Quantoren** kann angegeben werden, **wie oft** ein Element vorkommen darf. bzw. muss
- Quantoren werden unmittelbar nach dem Element notiert:
  - `{n}` **genau** `n` Vorkommen
  - `{n,}` **mindestens** `n` Vorkommen
  - `{n,m}` **mindestens** `n` und **höchstens** `m` Vorkommen

Quantoren

- Mit **Quantoren** kann angegeben werden, **wie oft** ein Element vorkommen darf. bzw. muss
- Quantoren werden unmittelbar nach dem Element notiert:
  - `{n}` **genau** `n` Vorkommen
  - `{n,}` **mindestens** `n` Vorkommen
  - `{n,m}` **mindestens** `n` und **höchstens** `m` Vorkommen
- Kurzformen:
  - `?` Vorkommen **optional** ($\hat{=}$ `{0,1}`)
  - `*` **beliebig viele** Vorkommen ($\hat{=}$ `{0,}`)
  - `+` **mindestens ein** Vorkommen ($\hat{=}$ `{1,}`)

Quantoren

- Mit **Quantoren** kann angegeben werden, **wie oft** ein Element vorkommen darf. bzw. muss
- Quantoren werden unmittelbar nach dem Element notiert:
  - `{n}` **genau** `n` Vorkommen
  - `{n,}` **mindestens** `n` Vorkommen
  - `{n,m}` **mindestens** `n` und **höchstens** `m` Vorkommen
- Kurzformen:
  - `?` Vorkommen **optional** ($\hat{=}$ `{0,1}`)
  - `*` **beliebig viele** Vorkommen ($\hat{=}$ `{0,}`)
  - `+` **mindestens ein** Vorkommen ($\hat{=}$ `{1,}`)

Beispiele

- `Go{2,}gle` findet "Google" mit 2 oder mehr "o"

Quantoren

- Mit **Quantoren** kann angegeben werden, **wie oft** ein Element vorkommen darf. bzw. muss
- Quantoren werden unmittelbar nach dem Element notiert:
  - `{n}` **genau** `n` Vorkommen
  - `{n,}` **mindestens** `n` Vorkommen
  - `{n,m}` **mindestens** `n` und **höchstens** `m` Vorkommen
- Kurzformen:
  - `?` Vorkommen **optional** ($\hat{=}$ `{0,1}`)
  - `*` **beliebig viele** Vorkommen ($\hat{=}$ `{0,}`)
  - `+` **mindestens ein** Vorkommen ($\hat{=}$ `{1,}`)

Beispiele

- `Go{2,}gle` findet "Google" mit 2 oder mehr "o"
- `[:;]-?[][)(}{|]+` findet verschiedene Smileys ;-)

Gliederung

Unterausdrücke

- Mit **runden Klammern** ( ) können beliebige Elemente zu **Unterausdrücken** (engl.: *sub patterns*) gruppiert werden

Unterausdrücke

- Mit **runden Klammern** `(` `)` können beliebige Elemente zu **Unterausdrücken** (engl.: *sub patterns*) gruppiert werden
- In vielen Anwendungen kann man mit der Notation `\1`, `\2`, ..., `\n` einen **Rückbezug** (engl.: *back reference* auf den vom `i`-ten Unterausdruck getroffenen Teilstring herstellen (*BRE-spezifisch, wird jedoch auch von vielen ERE-Anwendungen unterstützt*)

Unterausdrücke

- Mit **runden Klammern** `(` `)` können beliebige Elemente zu **Unterausdrücken** (engl.: *sub patterns*) gruppiert werden
- In vielen Anwendungen kann man mit der Notation `\1`, `\2`, ..., `\n` einen **Rückbezug** (engl.: *back reference* auf den vom `i`-ten Unterausdruck getroffenen Teilstring herstellen (*BRE-spezifisch, wird jedoch auch von vielen ERE-Anwendungen unterstützt*)

Beispiel

`([[:alpha:]]+)([[:blank:]]+\1)+` findet zwei oder mehr hintereinander folgende Vorkommen eines Wortes

Gliederung

Verschiedenes

- Mit dem **Pipe-Zeichen** | lassen sich **Alternativen** (“oder”) notieren

Verschiedenes

- Mit dem **Pipe-Zeichen** | lassen sich **Alternativen** (“oder”) notieren
- Der **Zirkumflex** ^ steht (außerhalb von Zeichenauswahlen) für den **Zeilenanfang**, das **Dollar-Zeichen $** für das **Zeilenende**

Verschiedenes

- Mit dem **Pipe-Zeichen** | lassen sich **Alternativen** (“oder”) notieren
- Der **Zirkumflex** ^ steht (außerhalb von Zeichenauswahlen) für den **Zeilenanfang**, das **Dollar-Zeichen $** für das **Zeilenende**
- Viele Implementierungen (z.B. PCRE) weitere Funktionen und Kurzschreibweisen, auf die hier nicht näher eingegangen wird

Verschiedenes

- Mit dem **Pipe-Zeichen** | lassen sich **Alternativen** ("oder") notieren
- Der **Zirkumflex** ^ steht (außerhalb von Zeichenauswahlen) für den **Zeilenanfang**, das **Dollar-Zeichen $** für das **Zeilenende**
- Viele Implementierungen (z.B. PCRE) weitere Funktionen und Kurzschreibweisen, auf die hier nicht näher eingegangen wird

Beispiele

- `true|false` findet "true" oder "false"

Verschiedenes

- Mit dem **Pipe-Zeichen** | lassen sich **Alternativen** ("oder") notieren
- Der **Zirkumflex** ^ steht (außerhalb von Zeichenauswahlen) für den **Zeilenanfang**, das **Dollar-Zeichen $** für das **Zeilenende**
- Viele Implementierungen (z.B. PCRE) weitere Funktionen und Kurzschreibweisen, auf die hier nicht näher eingegangen wird

Beispiele

- `true|false` findet "true" oder "false"
- `^[[:digit:]]+` findet Zahlen am Anfang einer Zeile

Verschiedenes

- Mit dem **Pipe-Zeichen** | lassen sich **Alternativen** ("oder") notieren
- Der **Zirkumflex** ^ steht (außerhalb von Zeichenauswahlen) für den **Zeilenanfang**, das **Dollar-Zeichen $** für das **Zeilenende**
- Viele Implementierungen (z.B. PCRE) weitere Funktionen und Kurzschreibweisen, auf die hier nicht näher eingegangen wird

Beispiele

- `true|false` findet "true" oder "false"
- `^[[:digit:]]+` findet Zahlen am Anfang einer Zeile
- `^$` findet eine leere Zeile

Gliederung

Unterschiede zwischen BREs und EREs

- Die Zeichen | + ? { } ( ) sind Literale, **keine Metazeichen**

Unterschiede zwischen BREs und EREs

- Die Zeichen | + ? { } ( ) sind Literale, **keine Metazeichen**
- Die Klammern für Quantoren sind \{ \}, für Unterausdrücke \(\)

Unterschiede zwischen BREs und EREs

- Die Zeichen `|` `+` `?` `{` `}` `(` `)` sind Literale, **keine Metazeichen**
- Die Klammern für Quantoren sind `\{` `\}`, für Unterausdrücke `\(` `\)`
- `^` und `$` sind nur am Anfang, bzw. am Ende des (Unter-)Ausdrucks möglich, sonst werden sie als Literale interpretiert

Unterschiede zwischen BREs und EREs

- Die Zeichen `|` `+` `?` `{` `}` `(` `)` sind Literale, **keine Metazeichen**
- Die Klammern für Quantoren sind `\{` `\}`, für Unterausdrücke `\(` `\)`
- `^` und `$` sind nur am Anfang, bzw. am Ende des (Unter-)Ausdrucks möglich, sonst werden sie als Literale interpretiert
- Es gibt **keine** Notation für **Alternativen**

Unterschiede zwischen BREs und EREs

- Die Zeichen `|` `+` `?` `{` `}` `(` `)` sind Literale, **keine Metazeichen**
- Die Klammern für Quantoren sind `\{` `\}`, für Unterausdrücke `\(` `\)`
- `^` und `$` sind nur am Anfang, bzw. am Ende des (Unter-)Ausdrucks möglich, sonst werden sie als Literale interpretiert
- Es gibt **keine** Notation für **Alternativen**
- Die Quantoren `+` und `?` müssen in der allgemeinen Quantoren-Schreibweise als `{1,}` und `{0,1}` notiert werden

Gliederung

Gliederung

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken
- Es durchsucht **Eingabedateien** oder die **Standardeingabe** und schreibt jede **Zeile** mit Suchtreffern in die **Standardausgabe**

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken
- Es durchsucht **Eingabedateien** oder die **Standardeingabe** und schreibt jede **Zeile** mit Suchtreffern in die **Standardausgabe**
- `grep` erlaubt **BRE- und ERE-Notation** (`grep -E` oder `egrep`)

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken
- Es durchsucht **Eingabedateien** oder die **Standardeingabe** und schreibt jede **Zeile** mit Suchtreffern in die **Standardausgabe**
- `grep` erlaubt **BRE- und ERE-Notation** (`grep -E` oder `egrep`)
- In beiden Notierungsarten werden **Back References** unterstützt

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken
- Es durchsucht **Eingabedateien** oder die **Standardeingabe** und schreibt jede **Zeile** mit Suchtreffern in die **Standardausgabe**
- `grep` erlaubt **BRE- und ERE-Notation** (`grep -E` oder `egrep`)
- In beiden Notierungsarten werden **Back References** unterstützt

Einsatzgebiete

- Sequentielles durchsuchen **vieler** Dateien

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken
- Es durchsucht **Eingabedateien** oder die **Standardeingabe** und schreibt jede **Zeile** mit Suchtreffern in die **Standardausgabe**
- `grep` erlaubt **BRE- und ERE-Notation** (`grep -E` oder `egrep`)
- In beiden Notierungsarten werden **Back References** unterstützt

Einsatzgebiete

- Sequentielles durchsuchen **vieler** Dateien
- **Filtern** von Informationen, z.B. aus Logdateien

grep (1)

- `grep` (von `g/re/p`) ist ein Programm zur **Suche und Filterung** von Text unter Verwendung von Regulären Ausdrücken
- Es durchsucht **Eingabedateien** oder die **Standardeingabe** und schreibt jede **Zeile** mit Suchtreffern in die **Standardausgabe**
- `grep` erlaubt **BRE- und ERE-Notation** (`grep -E` oder `egrep`)
- In beiden Notierungsarten werden **Back References** unterstützt

Einsatzgebiete

- Sequentielles durchsuchen **vieler** Dateien
- **Filtern** von Informationen, z.B. aus Logdateien
- Im Zusammenspiel mit anderen UNIX-Tools, z.B. in **Shellprogrammen**

grep (2)

Aufrufsyntax und Optionen (Ausschnitt)

```
[e]grep [OPTION] PATTERN [FILE]
```

`-E`	Verwende EREs statt BREs (vgl. `egrep`)
`-o`	Nur tatsächliche Treffer statt ganzer Zeilen ausgeben
`-v`	Nur Zeilen *ohne* Treffer ausgeben
`-i`	Groß-/Kleinschreibung ignorieren
`-c`	Nur Anzahl der gefundenen Zeilen ausgeben
`-l`	Nur die Dateinamen mit Treffern zeigen
`-n`	Zusätzlich Zeilennummern ausgeben
`-s`	Unterdrückt Fehlermeldungen in der Ausgabe

grep (3)

Beispiele

- `grep -c '\\begin{' vortrag.tex`
  Zählt die Vorkommen von `\begin{` in der Datei `vortrag.tex`

grep (3)

Beispiele

- `grep -c '\\begin{' vortrag.tex`
  Zählt die Vorkommen von `\begin{` in der Datei `vortrag.tex`
- `grep '#.*$' *`
  liest alle Kommentare aus den Dateien im aktuellen Verzeichnis

Gliederung

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:
  - Lese die nächste Zeile der Eingabe in den **Eingabepuffer**

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:
  - Lese die nächste Zeile der Eingabe in den **Eingabepuffer**
  - Entscheide, welche **Befehle** für diese Zeile ausgeführt werden sollen

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:
  - Lese die nächste Zeile der Eingabe in den **Eingabepuffer**
  - Entscheide, welche **Befehle** für diese Zeile ausgeführt werden sollen
  - Führe **sukzessive** alle Befehle aus

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:
  - Lese die nächste Zeile der Eingabe in den **Eingabepuffer**
  - Entscheide, welche **Befehle** für diese Zeile ausgeführt werden sollen
  - Führe **sukzessive** alle Befehle aus
  - Kopiere die Zeile in die **Standardausgabe** (vgl. `sed -n`)

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:
  - Lese die nächste Zeile der Eingabe in den **Eingabepuffer**
  - Entscheide, welche **Befehle** für diese Zeile ausgeführt werden sollen
  - Führe **sukzessive** alle Befehle aus
  - Kopiere die Zeile in die **Standardausgabe** (vgl. `sed -n`)
- `sed` ändert die Eingabedateien nicht. Um eine Rückspeicherung zu erhalten, kann die Ausgabe mit > in die Datei zurückgeschrieben werden

Einsatzgebiete

- Automatische, wiederkehrende **Textmanipulationen**

sed (1)

- `sed` (*stream editor*) ist ein **nicht-interaktiver, zeilenorientierter Texteditor**
- Arbeitsweise:
  - Lese die nächste Zeile der Eingabe in den **Eingabepuffer**
  - Entscheide, welche **Befehle** für diese Zeile ausgeführt werden sollen
  - Führe **sukzessive** alle Befehle aus
  - Kopiere die Zeile in die **Standardausgabe** (vgl. `sed -n`)
- `sed` ändert die Eingabedateien nicht. Um eine Rückspeicherung zu erhalten, kann die Ausgabe mit > in die Datei zurückgeschrieben werden

Einsatzgebiete

- Automatische, wiederkehrende **Textmanipulationen**
- Oft in Zusammenspiel mit anderen Tools, z.B. in **Shellprogrammen**

sed (2)

Aufrufsyntax und Optionen

```
sed [-E] [-n] [-e COMMAND] [-f SCRIPT] [FILE]
```

`-E`	Verwende EREs statt BREs
`-n`	Zeilenpuffer nicht implizit in die Ausgabe schreiben
`-e COMMAND`	Füge den Befehl COMMAND der Befehlsliste hinzu
`-f SCRIPT`	Lese Befehle aus der Skript-Datei SCRIPT

sed (2)

Aufrufsyntax und Optionen

```
sed [-E] [-n] [-e COMMAND] [-f SCRIPT] [FILE]
```

`-E`	Verwende EREs statt BREs
`-n`	Zeilenpuffer nicht implizit in die Ausgabe schreiben
`-e COMMAND`	Füge den Befehl COMMAND der Befehlsliste hinzu
`-f SCRIPT`	Lese Befehle aus der Skript-Datei SCRIPT

- Falls nur ein Befehlstext verwendet werden soll, kann `-e` weggelassen werden

sed (2)

Aufrufsyntax und Optionen

```
sed [-E] [-n] [-e COMMAND] [-f SCRIPT] [FILE]
```

`-E`	Verwende EREs statt BREs
`-n`	Zeilenpuffer nicht implizit in die Ausgabe schreiben
`-e COMMAND`	Füge den Befehl COMMAND der Befehlsliste hinzu
`-f SCRIPT`	Lese Befehle aus der Skript-Datei SCRIPT

- Falls nur ein Befehlstext verwendet werden soll, kann -e weggelassen werden
- Jede Zeile eines Befehlsskripts kann einen Befehl enthalten

sed (3)

Aufbau eines sed-Befehls

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

sed (3)

Aufbau eines sed-Befehls

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

- Eine **Adresse** ist entweder

sed (3)

Aufbau eines sed-Befehls

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

- Eine **Adresse** ist entweder
  - Eine **Zeilennummer**

sed (3)

Aufbau eines sed-Befehls

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

- Eine **Adresse** ist entweder
  - Eine **Zeilennummer**
  - **$** für die letzte Zeile

sed (3)

Aufbau eines sed-Befehls

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

- Eine **Adresse** ist entweder
  - Eine **Zeilennummer**
  - **$** für die letzte Zeile
  - Ein **regulärer Ausdruck** der Form `/RegExp/`

sed (3)

Aufbau eines sed-Befehls

```
[Startadresse [, Endadresse]] Funktion [Argumente]
```

- Eine **Adresse** ist entweder
  - Eine **Zeilennummer**
  - **$** für die letzte Zeile
  - Ein **regulärer Ausdruck** der Form `/RegExp/`
- **Ohne** Adresse wird der Befehl auf **jede** Zeile angewendet

sed (4)

Ausgewählte Funktionen

`p`	Zeile explizit ausgeben
`d`	Zeile löschen (bzw. nicht ausgeben)
`q`	Scriptausführung nach dieser Zeile beenden
`s/re/str/[flags]`	Regulären Ausdruck *re* mit *str* ersetzen (**Back-References** in *str* möglich) Ein gebräuchliches Flag ist **g** (*global*), das mehrere Ersetzungen pro Zeile ermöglicht

sed (5)

Beispiele

- `ls /u/halle | sed -E 's/(.*)/\1@in.tum.de/' > email.txt`
  Speichert eine Liste der E-Mail-Adressen der Recherhallen-Benutzer in der Datei `email.txt`

sed (5)

Beispiele

- `ls /u/halle | sed -E 's/(.*)/\1@in.tum.de/' > email.txt`
  Speichert eine Liste der E-Mail-Adressen der Recherhallen-Benutzer in der Datei `email.txt`
- `sed -n '1,3p' email.txt`
  Gibt die ersten drei Zeilen von `email.txt` aus

sed (5)

Beispiele

- `ls /u/halle | sed -E 's/(.*)/\1@in.tum.de/' > email.txt`
  Speichert eine Liste der E-Mail-Adressen der Recherhallen-Benutzer in der Datei `email.txt`
- `sed -n '1,3p' email.txt`
  Gibt die ersten drei Zeilen von `email.txt` aus
- `sed '/^$/d' text`
  Löscht leere Zeilen aus dem Text

Vielen Dank!

Vielen Dank für die Aufmerksamkeit!