Reguläre Ausdrücke

Sebastian Harl
<tokkee@lusc.de>

LUSC Workshop Weekend 2008

03. Oktober 2008

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+
)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x0
1-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9]
)?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-9
]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?
|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Reguläre Ausdrücke

- Engl. „regular expression“ → „regex“ oder „regexp“ (Plural: „regexes“, „regexps“, „regexen“)

Reguläre Ausdrücke

- Engl. „regular expression“ → „regex“ oder „regexp“ (Plural: „regexes“, „regexps“, „regexen“)
- Zweck: Identifizierung von Zeichenketten mit einem bestimmten Muster

Reguläre Ausdrücke

- Engl. „regular expression“ → „regex“ oder „regexp“ (Plural: „regexes“, „regexps“, „regexen“)
- Zweck: Identifizierung von Zeichenketten mit einem bestimmten Muster
- formale Sprache, die von einem Prozessor interpretiert werden kann

Reguläre Ausdrücke

- Engl. „regular expression“ → „regex“ oder „regexp“ (Plural: „regexes“, „regexps“, „regexen“)
- Zweck: Identifizierung von Zeichenketten mit einem bestimmten Muster
- formale Sprache, die von einem Prozessor interpretiert werden kann
- Einsatz: Text-Matching, speziell Parser

Reguläre Ausdrücke - einfache Beispiele

- Teilzeichenketten: `car` in verschiedenen Kontexten: **car**, **car**`toon`, `bi`**car**`bonate`, ...
- Wort-Muster: alle Wörter, die mit `s` beginnen und mit `d` enden: **s**`and`, **s**`ignifie`**d**, ...

Reguläre Ausdrücke - Implementierungen (I)

- `POSIX`
  - definiert in POSIX.2
  - erweiterte und einfache (veraltet) Regexen
  - verfügbar über die libc
  - `find -regex|-iregex`
  - `expr`
  - `grep -E/egrep`, (`grep`)
  - `sed -r`, `awk`

Reguläre Ausdrücke - Implementierungen (II)

- `PCRE` (`Perl Compatible Regular Expression`)
  - `perl` ;-)
  - verfügbar als Bibliothek für viele Programmier- und Skript-Sprachen (C/C++, Python, Ruby, Tcl, PHP, ...)
  - `Postfix` maps
  - `grep -P` (experimentell)

Inhalt

Grundlagen

Grundlegende Konzepte

- Regexen sind Ausdrücke, die eine Menge von Zeichenketten beschreiben

Grundlegende Konzepte

- Regexen sind Ausdrücke, die eine Menge von Zeichenketten beschreiben
- Ausdrücke können durch Operationen zusammengefasst werden:
  - Alternativen: `foo|bar`

Grundlegende Konzepte

- Regexen sind Ausdrücke, die eine Menge von Zeichenketten beschreiben
- Ausdrücke können durch Operationen zusammengefasst werden:
  - Alternativen: `foo|bar`
  - Gruppierung: `foo(|bar)`

Grundlegende Konzepte

- Regexen sind Ausdrücke, die eine Menge von Zeichenketten beschreiben
- Ausdrücke können durch Operationen zusammengefasst werden:
  - Alternativen: `foo|bar`
  - Gruppierung: `foo(|bar)`
  - Quantifizierung: `fo+b?a*r?`

Geschichte (I)

- Ursprung in der Automaten-Theorie und Theorie formaler Sprachen
- 1950er: Mathematiker Stephen Cole Kleene beschreibt das Modell von „Regulären Mengen“
- `SNOBOL` war eine erste Implementierung von Pattern Matching (aber keine Regexen)
- Ken Thompson setzte Kleenes Modell im Editor `QED` um und integrierte es später in den Editor `ed`
- daraus entstand später `grep` mit seiner Unterstützung für Regexen

Geschichte (II)

- seither verschiedene Variationen in `expr`, `awk`, `Emacs`, `vi` und `lex`
- Perl und Tcl Regexen waren ursprünglich von einer Bibliothek von Henry Spencer abgeleitet und entwickelten sich von dort weiter
- PCRE wurde von Philip Hazel entwickelt, um das Verhalten von Perl Regexen nachzubauen

Syntax

POSIX erweiterte Syntax (ERE)

- POSIX einfache Syntax (BRE) existiert hauptsächlich für Rückwärtskompatibilität; Verwendung ist nicht mehr empfehlenswert
- Bedeutung von einigen Escape-Sequenzen hat sich umgedreht (z.B. `\(` und `\)` in BRE statt `(` und `)`)
- `|`, `?` und `+` keine spezielle Bedeutung in BRE
- `\` gefolgt von einer Zahl neu in ERE, aber sollte vermieden werden
- RTFM: `regex(7)`

Meta-Sequenzen (I)

- beliebiges Zeichen: .
  (Vorsicht: Nicht innerhalb von Gruppierungen)

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)
- Zeichenklassen: `[]`
  Bereiche: `-`, falls nicht am Anfang oder Ende oder mit Backslash geschützt

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)
- Zeichenklassen: `[]`
  Bereiche: `-`, falls nicht am Anfang oder Ende oder mit Backslash geschützt
  (Beispiel: `[abcx-z_.]`)

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)
- Zeichenklassen: `[]`
  Bereiche: `-`, falls nicht am Anfang oder Ende oder mit Backslash geschützt
  (Beispiel: `[abcx-z_.]`)
- negierte Zeichenklassen: `[^]`

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)
- Zeichenklassen: `[]`
  Bereiche: `-`, falls nicht am Anfang oder Ende oder mit Backslash geschützt
  (Beispiel: `[abcx-z_.]`)
- negierte Zeichenklassen: `[^]`
  (Beispiel: `[^>]`)

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)
- Zeichenklassen: `[]`
  Bereiche: `-`, falls nicht am Anfang oder Ende oder mit Backslash geschützt
  (Beispiel: `[abcx-z_.]`)
- negierte Zeichenklassen: `[^]`
  (Beispiel: `[^>]`)
- Anfang / Ende: `^`, `$`

Meta-Sequenzen (I)

- beliebiges Zeichen: `.`
  (Vorsicht: Nicht innerhalb von Gruppierungen)
  (Beispiel: `a.c`)
- Zeichenklassen: `[]`
  Bereiche: `-`, falls nicht am Anfang oder Ende oder mit Backslash geschützt
  (Beispiel: `[abcx-z_.]`)
- negierte Zeichenklassen: `[^]`
  (Beispiel: `[^>]`)
- Anfang / Ende: `^`, `$`
  (Beispiel: `^foo$`)

Meta-Sequenzen (II)

- Alternativen: |

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)
- Subausdrücke, Gruppierung: ()
  Referenzierung über $\backslash n$ ($n \in [1, 9]$)
  Perl: $\$n$

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)
- Subausdrücke, Gruppierung: ()
  Referenzierung über $\backslash n$ ($n \in [1, 9]$)
  Perl: $\$n$
  (Beispiel: `(foo)`)

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)
- Subausdrücke, Gruppierung: ()
  Referenzierung über $\backslash n$ ($n \in [1, 9]$)
  Perl: $\$n$
  (Beispiel: `(foo)`)
- Quantifizierung: *, ?, +, $\{n, m\}$

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)
- Subausdrücke, Gruppierung: ()
  Referenzierung über $\backslash n$ ($n \in [1, 9]$)
  Perl: $\$n$
  (Beispiel: `(foo)`)
- Quantifizierung: *, ?, +, $\{n, m\}$
  (Beispiel: `a{3,5}b*`)

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)
- Subausdrücke, Gruppierung: ()
  Referenzierung über $\backslash n$ ($n \in [1, 9]$)
  Perl: $\$n$
  (Beispiel: `(foo)`)
- Quantifizierung: *, ?, +, $\{n, m\}$
  (Beispiel: `a{3,5}b*`)
- Spezielle Sequenzen (nicht POSIX): $\backslash <$, $\backslash >$
  Perl: `\b`

Meta-Sequenzen (II)

- Alternativen: |
  (Beispiel: `foo|bar`)
- Subausdrücke, Gruppierung: ()
  Referenzierung über $\backslash n$ ($n \in [1, 9]$)
  Perl: $\$n$
  (Beispiel: `(foo)`)
- Quantifizierung: *, ?, +, $\{n, m\}$
  (Beispiel: `a{3,5}b*`)
- Spezielle Sequenzen (nicht POSIX): $\backslash <$, $\backslash >$
  Perl: `\b`
  (Beispiel: $\backslash$ <`foo`$\backslash >$)

Zeichenklassen - POSIX und Perl

POSIX	Perl	ASCII
`[:alnum:]`		`[A-Za-z0-9]`
`[:word:]`	`\w`	`[A-Za-z0-9_]`
	`\W`	`[^\w]`
...		

Zeichenklassen - POSIX und Perl

POSIX	Perl	ASCII
`[:alnum:]`		`[A-Za-z0-9]`
`[:word:]`	`\w`	`[A-Za-z0-9_]`
	`\W`	`[^\w]`
...		

(Beispiel: `[xYz[:alnum:]-]`)

Zeichenklassen - POSIX und Perl

POSIX	Perl	ASCII
`[:alnum:]`		`[A-Za-z0-9]`
`[:word:]`	`\w`	`[A-Za-z0-9_]`
	`\W`	`[^\w]`
...		

(Beispiel: `[xYz[:alnum:]-]`)

- RTFM: `regex(7)` bzw. `perlre(1)`

Gierige Quantoren

- üblicherweise sind Quantoren „gierig“, d.h. sie „verschlingen“ so viel wie möglich
- Beispiel: Eingabe „`<foo@bar>, <qux@baz>`“, Muster: „`<.*>`“

Gierige Quantoren

- üblicherweise sind Quantoren „gierig“, d.h. sie „verschlingen“ so viel wie möglich
- Beispiel: Eingabe „`<foo@bar>, <qux@baz>`“, Muster: „`<.*>`“
- Lösungsansatz: Muster: „`<[^>]*>`“

Gierige Quantoren

- üblicherweise sind Quantoren „gierig“, d.h. sie „verschlingen“ so viel wie möglich
- Beispiel: Eingabe „`<foo@bar>, <qux@baz>`“, Muster: „`<.*>`“
- Lösungsansatz: Muster: „`<[^>]*>`“
- faules Verhalten von Quantoren (z.B. Perl): „`<.*?>`“

Zeichenketten-Ersetzung

Zeichenketten-Ersetzung

- z.B. Perl und sed
- s/*zeichenkette*/*ersetzung*/

Zeichenketten-Ersetzung

- z.B. Perl und sed
- `s/`*zeichenkette*`/`*ersetzung*`/`
- Beispiel (sed -r): `s/fu(bar)? baz/foo\1/`

Zeichenketten-Ersetzung

- z.B. Perl und sed
- `s/`*zeichenkette*`/`*ersetzung*`/`
- Beispiel (sed -r): `s/fu(bar)? baz/foo\1/`
- Beispiel (Perl): `s/fu(bar)? baz/foo$1/`

Fragen?

History:

- 2008/10/03: LUSC Workshop Weekend 2008