REGULÄRE AUSRÜCKE IN PROGRAMMIERSPRACHEN

Im Gegensatz zu den regulären Ausdrücken die wir aus der theoretischen Informatik kennen, bieten die meisten Programmiersprachen Implementierungen, die über die Leistungsfähigkeit von regulären Sprachen hinausgehen – womit die Bezeichnung „Regulärer Ausdruck“ an einigen Stellen gar nicht mehr gerechtfertigt ist. Die Notation von diesen Ausdrücken unterscheidet sich auch durchaus von denen aus der TI. In Grammatiken finden wir die beliebte EBNF (extended BNF) Notation welche ganz ähnliche Metazeichen verwendet, die jedoch mit unter ganz verschiedene Bedeutungen besitzen – deshalb sollte man diese auf keinen Fall verwechseln!

Selbst in den einzelnen Programmiersprachen gibt es keine einheitliche Definition über den syntaktischen Aufbau eines RegExp. Wir wollen uns deshalb hier auf eine Basis beschränken, die zumindest in den von VCC unterstützten Programmiersprachen verwendet werden kann.

Zeichen	Bedeutung
.	... steht für ein beliebiges Zeichen (außer \n\r – siehe weiter unten).
[ABC]	Eckige Klammern beschreiben eine Auswahl (Zeichenklasse). Dieses Beispiel steht also für genau ein Zeichen A,B oder C.
[a-zA-Z]	Der Bindestrich innerhalb eckiger Klammern bestimmt einen Bereich. Daher steht dieses Beispiel für alle Zeichen des lateinischen Alphabets.
[^A]	Das Dach (^) negiert die Auswahl. Daher wird hier ein beliebiges Zeichen außer "A" beschrieben.
+	... kennzeichnet eine Mengenangabe. Das "+" steht für ein oder mehrere Vorkommen des Zeichens, Zeichenklasse oder Metazeichen.
*	... steht für 0 bis beliebig viele Vorkommen.
{m,n}	... steht für "m" bis "n" Vorkommen des Zeichens oder Metazeichens.
{n}	... steht für "n" Vorkommen des Zeichens oder Metazeichens.
^	... bezeichnet, dass das Zeichen oder Metazeichen am Anfang der Zeichenkette vorkommen muss.
$	... bezeichnet, dass das Zeichen oder Metazeichen am Ende der Zeichenkette vorkommen muss.
(ab)+	... Klammern erlauben Teilausdrücke zu einer Einheit zusammenzufassen.
A|B	... steht für ein Zeichen A ODER B (kann auch auf komplexe geklammerte Teilausdrücke angewendet werden z.B.: (a|b)|[0-9]+ .

Neben diesen RegExp eigenen Metazeichen finden sich noch eine Reihe zusätzlicher Zeichen die teilweise bereits aus dem normalen Programmieralltag bekannt sein dürften. Wichtig: Ein \s ist ungleich \S → casesensitive.

Zeichen	Bedeutung
\n	... steht für einen Zeilenumbruch.
\r	... steht für einen Wagenrücklauf (unter Windows häufig \r\n für einen Zeilenumbruch).
\t	... steht für ein Tabulatorzeichen.
\s	... steht für ein einzelnes Leerzeichen.
\d	... steht für den regulären Teilausdruck [0-9].
\w	... ein Buchstabe, eine Ziffer oder der Unterstrich [a-zA-Z_0-9].

Anhand einiger Beispiele soll die richtige Verwendung der aufgeführten Zeichen verdeutlicht werden:

- ... trifft 3 beliebige Zeichen (außer \r und \n)
- .{1,5} trifft 1,2,3,4 oder 5 beliebige Zeichen (außer \r und \n)
- [0-9]+ trifft eine beliebig lange Ziffernfolge wie etwa: 058237435 oder 0
- a*b trifft beliebig viele a (oder keins) gefolgt von genau einem b
- (a*)|(b*) trifft beliebig viele a ODER beliebig viele b oder auch das leere Wort!
- \d+,\d\d trifft eine Fließkommazahl der Form: 234,32 oder 0,98
  identisch mit [0-9]+,[0-9][0-9]
- [1-9][0-9]* trifft eine beliebig lange Ziffernfolge aber ohne vorangestellte Nullen.
- [\r\t\n\s] trifft ein typisches Whitespace
- [1-5]0 trifft genau die Ziffernfolgen 10, 20, 30, 40 oder 50.

In vielen Fällen werden auch Zeichen benötigt, die im RegExp ein Metazeichen darstellen. Möchte man Beispielsweise den arithmetischen Ausdruck „(234+23)*4“ mit einem regulären Ausdruck erkennen, so müssen wir den Klammern sowie dem Plus und dem Mal ein „\“ voranstellen um die Metazeichenfunktion außer Kraft zu setzen: \([0-9]+\+[0-9]\)*[0-9]

ANWENDUNGSGEBIETE

EINGABGE ÜBERPRÜFUNG

In der Praxis finden sich viele Anwendungsgebiete für reguläre Ausdrücke. Vielleicht das berühmteste Beispiel ist die Eingabevalidierung bei graphischen Benutzerinterfaces sowohl in Webanwendungen als auch in Desktopanwendungen.

Ein Beispiel: Auf einer Portalseite XY kann sich ein neuer Benutzer unter Angabe seiner Emailadresse als neuer User registrieren. Eine ganz einfache Eingabemaske könnte wie folgt aussehen:

Benutzername : |_____|
Emailadresse : |_____|

Passwort : |_____|
Wiederholung : |_____|

Der Anbieter von XY möchte, dass als Benutzernamen nur Zeichenketten aus Buchstaben, Zahlen oder Unterstrichen erlaubt sind. Der Benutzername soll zwischen 3 und 8 Zeichen lang sein. Es soll außerdem überprüft werden ob eine (ansatzweise) gültige Emailadresse eingegeben wurde und das Passwort aus mindestens 5 Zeichen besteht.

Wir definieren einen regulären Ausdruck für den Benutzernamen:
[0-9A-Za-z_]{3,8}

Passt der Ausdruck auf das Eingabewort (in diesem Fall der eingegebene Benutzername) akzeptieren wir die Eingabe des Anwenders, andern Falls weisen wir den Benutzer darauf hin seine Eingabe zu überprüfen und entsprechend zu ändern.

Für die Emailadresse benötigen wir schon einen etwas umfangreicheren RegExp:
[a-zA-Z0-9\._%\+\-]+@[a-zA-Z0-9\.\-]+\.[a-zA-Z]{2,4}

Dabei steht der erste Teil bis zum „@“ für eine Folge von Zeichen die in URL’s allgemein erlaubt sind. Nach dem „@“ folgen mögliche Subdomains und Domainname gefolgt von einer 2-4 stelligen Landeskennung (.de; .com oder auch .info).

Für das Passwort können wir wieder einen einfacheren Ausdruck verwenden:
.....+

Gerade für den letzten Ausdruck würde eine einfache Prüfung der Länge des Eingabewortes mit weniger Mitteln (in Bezug auf Quelltextzeilen) zum Erfolg führen. Betrachtet man das Problem: Validieren von Eingaben etwas allgemeiner lässt sich jedoch jedes Eingabefeld mit einem RegExp verknüpfen und mit Hilfe einer Methode „ValidateForm“ die Eingaben des Benutzers für jedes Feld prüfen ohne später zusätzlichen Code schreiben zu müssen wenn neue Eingabefelder hinzukommen oder entfernt werden:

```
public bool ValidateForm () {
   for (int i = 0; i < Form.Elements.Count; i++) {
      if (Form.Elements[i].RegExp.Match(Form.Elements[i].Text) == false)
          return false; // dieses Feld stimmt nicht
   }
   return true; // Alle Felder sind in Ordnung.
}
```

XML SCHEMAS

Ein weiterer Anwendungsfall sind XML Schemas. Auch hier steht die Validierung von XML Dateien im Mittelpunkt. Mit Hilfe von RegExp können ganz bestimmte Einschränkungen für Attribute definieren. Etwa eine Postleitzahl beginnt mit einem Länderzeichen gefolgt von einem Minus und 5 Ziffern: D-02779.

```
<xsd:simpleType name="PLZ">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-Z]\-[0-9]{5}"/>
    </xsd:restriction>
</xsd:simpleType>
```

Es wird dabei ein Datentyp definiert und von einem Basistyp (hier xsd:string) abgeleitet und zusätzlich durch einen (oder sogar mehrere) Reguläre Ausdrücke „pattern“ eingeschränkt.

Möchte man Beispielsweise ein Passwort mit der Länge von mindestens 5 Zeichen und mindestens einem Buchstaben und mindestens einer Ziffer, so kann man folgenden Typ definieren:

```
<xsd:simpleType name="Passwort">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value=".....+"/>
        <xsd:pattern value=".*[0-9].*"/>
        <xsd:pattern value=".*[A-Za-z].*"/>
    </xsd:restriction>
</xsd:simpleType>
```

Wichtig ist hier, dass jedes „pattern“ passen muss – es findet also ein logisches AND statt.

LITERATUR

Mastering Regular Expressions in Perl, .NET, Java and More [O'Reilly,2Ed,2002]