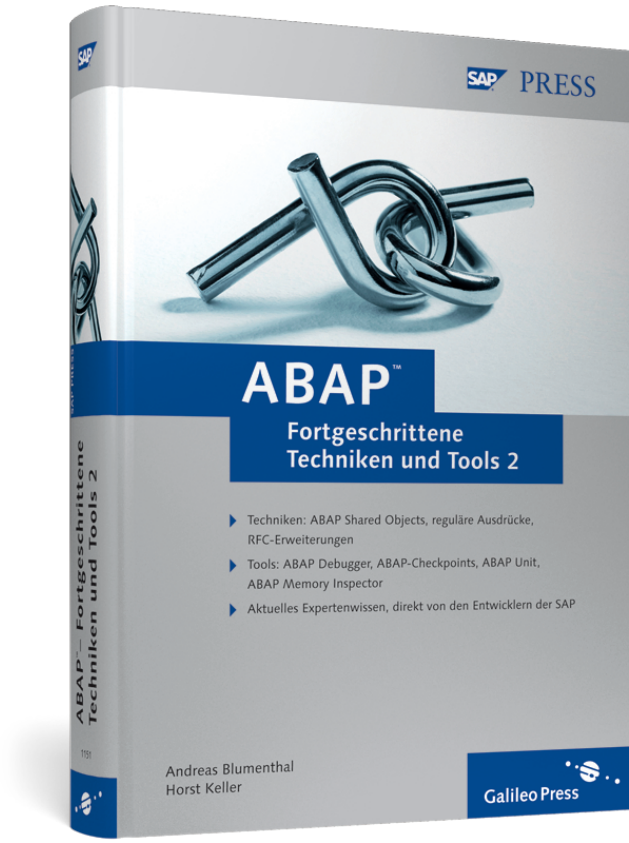
Andreas Blumenthal, Horst Keller

ABAP™ – Fortgeschrittene Techniken und Tools, Band 2



Galileo Press

Bonn • Boston

Auf einen Blick

Inhalt

Vorwort

Dieses Buch ist der zweite Band des Buchs *ABAP – Fortgeschrittene Techniken und Tools*, das vor vier Jahren bei SAP PRESS erschienen ist. Wie sein Vorgänger ist es eine Zusammenfassung einer Reihe von Artikeln über ABAP und das ABAP-Umfeld, die in englischer Sprache im SAP Professional Journal (*http://www.sappro.com*) erschienen sind. Der Ansatz des ersten Bandes, ABAP-spezifische Artikel des im deutschsprachigen Raum nicht so stark verbreiteten SAP Professional Journals einer erweiterten und zugleich fachlich spezieller interessierten Leserschaft in einem Buch zugänglich zu machen, stieß auf recht positive Resonanz. Deshalb wollen wir dieses Konzept mit dem vorliegenden Band fortführen und präsentieren Ihnen hier die in den Jahren 2004 bis 2007 erschienenen ABAP-spezifischen Artikel des SAP Professional Journals in deutscher Sprache und in einer aktualisierten Fassung.

Wieder handelt es sich um Artikel, die von den Mitarbeitern der SAP, die die Sprache ABAP und die dazugehörigen Werkzeuge entwickeln, selbst geschrieben wurden, und damit um Wissen aus erster Hand. Somit bleiben auch die im Vorwort des ersten Bandes gemachten Aussagen im Wesentlichen gültig. Auch das vorliegende Buch dient der Abrundung anderer SAP PRESS-Bücher, die im Bereich »NetWeaver Core AS&DM ABAP« (das ist der derzeitige Name der ABAP-Gruppe) der SAP entstanden sind. Dies gilt insbesondere für den Einführungsband *ABAP Objects – ABAP-Programmierung mit SAP NetWeaver* (SAP PRESS 2006), in dem die Themengebiete des vorliegenden Buchs entweder nur kurz, mit einem anderen Tenor oder gar nicht erwähnt werden. Im vorliegenden Buch sind diese speziellen Themengebiete als abgeschlossene Einheiten in Kapiteln zusammengefasst.

Alle Artikel dieses Buchs wurden einer eingehenden fachlichen und terminologischen Überarbeitung durch ihre Autoren und die Herausgeber unterzogen, teilweise zusammengefasst, neu geordnet oder sogar neu geschrieben. Während einige Artikel im Original den Stand der Releases 6.20/6.40 beschreiben, spiegelt die vorliegende Überarbeitung das aktuelle Release 7.0 des SAP NetWeaver Application Servers ABAP wider. Auf wesentliche Neuerungen, die für die Releases nach 7.0 entwickelt wurden und die derzeit im Rahmen eines »Feature Set Backports« nach Release 7.0, EhP2 übernommen werden, weisen wir jeweils hin.

Inhaltlich bieten wir wieder einen bunten Strauß von Themen, von dem wir hoffen, dass für jeden ABAP-Entwickler etwas dabei ist:

- Wir beginnen mit zwei Kapiteln über die Zeichenkettenverarbeitung, Stand Release 7.0. Obwohl es hierbei ab Release 7.0, EhP2 erhebliche Verbesserungen durch Zeichenkettenausdrücke und neu eingebaute Zeichenkettenfunktionen geben wird, bleiben viele der in **Kapitel 1** gemachten Aussagen gültig. Dies gilt insbesondere auch für das Verständnis vorhandener Programme, in denen die neuen Sprachkonstrukte noch nicht verwendet werden. Die grundlegende Beschreibung regulärer Ausdrücke in **Kapitel 2** ist ohnehin weitestgehend Release-unabhängig.
- Nachdem wir im ersten Band bereits XSLT und das Format asXML für die Serialisierung von ABAP-Daten nach XML eingeführt haben, beschäftigt sich **Kapitel 3** mit den Simple Transformations (ST), einer SAP-eigenen XML-basierten Programmiersprache, die rein auf Transformationen zwischen ABAP-Daten und XML zugeschnitten ist und entsprechende Performance-Vorteile liefert.
- **Kapitel 4** gibt Ihnen eine Einführung in die explizite Verwendung des Shared Memorys eines Applikationsservers mit den Mitteln von ABAP Shared Objects. Nach der Lektüre dieses Kapitels sollten Sie in der Lage sein, wenig veränderliche Daten, die von vielen Programmen gleichzeitig verwendet werden, platzsparend ein einziges Mal im Shared Memory abzulegen und anderen Programmen darauf Zugriff zu gewähren.
- Die nächsten beiden Kapitel sind der ABAP-Remote-Kommunikation und hierbei insbesondere dem Remote Function Call (RFC) gewidmet – ein äußerst wichtiges Thema, das leider häufig zu kurz kommt. Während **Kapitel 5** einen umfassenden Überblick über die RFC-Programmierung liefert, führt **Kapitel 6** in den neuen Background RFC (bgRFC) ein, der die vorhergehenden Techniken transaktionaler RFC (tRFC) und queued RFC (qRFC) ablöst.
- Mit den folgenden zwei Kapiteln halten Sie die schriftlich ausgearbeitete Form der erfolgreichsten und meistbesuchten ABAP-Workshops der SAP TechEds der vergangenen Jahre in den Händen. Das »ABAP Trouble Shooting« mit verschiedenen Test- und Analysewerkzeugen (**Kapitel 7**) und der Umgang mit dem ABAP Debugger (**Kapitel 8**) sind Themen, mit denen sich jeder ABAP-Entwickler wohl oder übel auseinandersetzen muss. Hier erhalten Sie viele wertvolle Tipps.
- **Kapitel 9** führt in die sogenannten Checkpoints ein. Checkpoints instrumentieren ABAP-Programme zu Testzwecken. Sie umfassen neben den vom Debugging bekannten Breakpoints auch Assertions und Logpoints, die allesamt von außerhalb eines Programms gesteuert werden können. Kein modernes ABAP-Programm sollte auf den Einsatz von Checkpoints verzichten.

- Mit **Kapitel 10** erhalten Sie eine prägnante Einführung in das ABAP-Testwerkzeug schlechthin, nämlich in ABAP Unit für Modultests. Nach der Lektüre dieses Kapitels sollte Sie nichts mehr davon abhalten, die gesamte Funktionalität Ihrer Programme mit solchen Tests zu überprüfen und damit ihre Stabilität zu garantieren.
- **Kapitel 11** schließt den Kreis der Testwerkzeuge mit einer Einführung in den ABAP Memory Inspector. In Zeiten von mehr und mehr dynamisch erzeugten Objekten und Datenobjekten wird es immer wichtiger, den dadurch zur Laufzeit belegten Speicher zu kontrollieren, um Speicherlecks vorzubeugen. Das Werkzeug ABAP Memory Inspector ist hierfür unerlässlich.
- Abschließend gibt **Kapitel 12** einen einführenden Überblick über das Enhancement Framework und das Switch Framework, die seit Release 7.0 zur Verfügung stehen. Mithilfe dieser Technologie können verschiedene Branchenlösungen in einem Core-System entwickelt und ausgeliefert werden. Weiterhin läutet das Enhancement Framework eine neue Ära der modifikationsfreien Änderung von SAP-Programmen ein.

Wir danken allen Autoren, die nochmals die Mühe einer Überarbeitung ihrer Artikel auf sich genommen haben, Wellesley Information Services, dass sie uns die Genehmigung für die deutschsprachige Veröffentlichung der Artikel erteilt haben, und Galileo Press für die Übersetzung der Artikel ins Deutsche. Bei Galileo Press sei insbesondere Frau Mirja Werner und Herrn Stefan Proksch vom Lektorat SAP PRESS für die gute Betreuung bei der Erstellung dieses Buchs gedankt.

Wie immer weisen wir abschließend darauf hin, dass wir, wie im Deutschen leider üblich, bei Personenbezeichnungen durchgehend die männliche Form verwenden, diese aber stellvertretend für die männliche und weibliche Form verstehen. Wenn wir also von Entwicklern und Benutzern sprechen, schließt dies selbstverständlich auch Entwicklerinnen und Benutzerinnen ein. Die Verwendung der männlichen Form folgt letztlich auch der Terminologie des SAP NetWeaver Application Servers, die ebenfalls die männliche Form verwendet, beispielsweise Benutzer, Benutzername und Benutzerkonto.

Andreas Blumenthal
Vice President, NetWeaver Core AS&DM ABAP

Horst Keller
Knowledge Architect, NetWeaver Core AS&DM ABAP

Reguläre Ausdrücke sind ein mächtiges und weit verbreitetes Standardwerkzeug zur effizienten Verarbeitung von zeichenartigen Daten. Dieses Kapitel bietet eine allgemeine Einführung in reguläre Ausdrücke und beschreibt deren Integration in die ABAP-Welt. Anhand zahlreicher Beispiele wird gezeigt, wie reguläre Ausdrücke die Performance von ABAP-Programmen verbessern können.

Ralph Benziger und Björn Mielenhausen

2 Reguläre Ausdrücke für die Zeichenkettenverarbeitung in ABAP

2.1 Reguläre Ausdrücke in der Textverarbeitung

Die Textverarbeitung macht einen großen Teil der Informationsverarbeitung aus, insbesondere im Umfeld typischer SAP-Geschäftsanwendungen. Hier bezieht sich Textverarbeitung nicht nur auf die Arbeit mit Text in natürlicher Sprache, sondern auch auf andere Datentypen, die in einer Textdarstellung gespeichert werden, zum Beispiel Datumsangaben, Währungen oder XML-Daten. Auf solche Informationen zuzugreifen, sie zu analysieren und zu ändern sind häufige Aufgaben in einer ABAP-Entwicklung.

Im Allgemeinen lassen sich Textverarbeitungsaufgaben in drei Kategorien gliedern:

- **Validierung**
  Stimmen Informationen im Hinblick auf syntaktische Eigenschaften mit einer Spezifikation überein?
- **Extraktion**
  Lokalisierung von Informationen in einer größeren Datenmenge, basierend auf dem Kontext, nicht auf der Position
- **Transformation**
  Konvertierung von Informationen in unterschiedliche strukturelle Darstellungen

Reguläre Ausdrücke sind ein standardisiertes und weit verbreitetes Werkzeug für die effiziente und effektive Verarbeitung von textbasierten Informationen.

Vor SAP NetWeaver 7.0 gab es in ABAP aber keine native Unterstützung für reguläre Ausdrücke, und sie konnten in ABAP-Programmen nur mithilfe von allerlei Tricks und damit nur sehr eingeschränkt verwendet werden.[1]

Vor der eingehenden Beschäftigung mit den Details zur Verwendung von regulären Ausdrücken werden die genannten Kategorien von Textverarbeitungsaufgaben, für die sich reguläre Ausdrücke ideal eignen, etwas näher betrachtet.

2.1.1 Drei Typen von Textverarbeitungsaufgaben

Anhand einiger kleiner Beispiele werden die drei Typen von Textverarbeitungsaufgaben illustriert: Informationen validieren, extrahieren und transformieren. Anschließend wird der erforderliche Aufwand untersucht, um eines dieser Beispiele mit konventionellem ABAP, das heißt ohne die Hilfe von regulären Ausdrücken zu lösen.

Informationen validieren

Angenommen, Sie schreiben einen ABAP-basierten Webservice, um Kreditkarteninformationen als Teil eines E-Commerce-Systems zu verarbeiten. Das Design der Methode sieht vor, dass die Kreditkartennummer und das Gültigkeitsdatum als Zeichenfolgen übergeben werden. Um ein sicheres Design zu gewährleisten, ist es sinnvoll, die Gültigkeit der empfangenen Daten oder zumindest deren Plausibilität zu überprüfen. In diesem Fall soll sichergestellt werden, dass die Kreditkartennummer eine 15- oder 16-stellige Zahl und das Gültigkeitsdatum eine zweistellige Zahl, gefolgt von einem Schrägstrich und einer weiteren zweistelligen Zahl, ist:

```
1234567812345678  02/06  ->  akzeptieren
123456781234XXXX  03/07  ->  ablehnen
1234567812345678  04.08  ->  ablehnen
```

Während sich eine solche Validierung der Kreditkartennummer in ABAP problemlos mithilfe des `CO`-Operators (contains only) und der `strlen( )`-Funktion (string length) implementieren lässt, erfordert die Überprüfung des Gültigkeitsdatums etwas mehr Aufwand.

Im Allgemeinen gilt: Je flexibler die Spezifikation ist, desto schwieriger wird es, sie in eine knappe und präzise ABAP-Implementierung umzusetzen. Wenn

1 Ein Beispiel ist der Aufruf von JavaScript aus ABAP heraus, nur um dort einen regulären Ausdruck auszuwerten.

die Spezifikation für das Datum beispielsweise einstellige Monatsangaben sowie vier- und zweistellige Jahresangaben zulassen würde, würden sich die Anforderungen an das ABAP-Coding drastisch erhöhen.

Informationen extrahieren

Eine weitere häufige Aufgabe ist das Extrahieren von Informationen aus einer größeren Datenmenge. Angenommen, Sie erstellen einen Webservice zur Bereitstellung von Dokumenten für Benutzer. Das Backend empfängt eine URL vom Frontend, die neben weiteren Informationen die ID des angeforderten Dokumentes enthält:

http://docserve.sap.com/serve?user=ralph&docid=NW2004&lang=EN

In diesem Beispiel wird Code zum Extrahieren des Dokumentnamens `NW2004` aus der Zeichenfolge benötigt. Weitere typische Beispiele sind die Extraktion von Pfaden und Dateierweiterungen aus Dateinamen oder von E-Mail-Adressen aus Headern von E-Mail-Nachrichten.

Informationen transformieren

Bei dieser Aufgabengruppe geht es um die Umwandlung einer strukturellen Darstellung von Informationen in eine andere. Angenommen, in Ihrer Anwendung werden Telefonnummern in einem lesbaren Format gespeichert, das Leerzeichen, Klammern und Bindestriche umfasst:

```
(800) 123-4567
+49 6227 747474
```

Um die Anwendung mit einer externen Anwendung zu verbinden, die Telefonnummern als Ziffernfolgen erfordert, müssen Sie Code schreiben, der alle Zeichen, bei denen es sich nicht um Ziffern handelt, aus den Informationen entfernt. Wenn die Anzahl der Zeichen, die keine Ziffern sind, bekannt und gering ist, kann diese Aufgabe in ABAP problemlos mithilfe einiger `REPLACE`-Anweisungen ausgeführt werden. Dieser Ansatz wird jedoch schnell mühsam und auch fehleranfällig, wenn die Anzahl der zu testenden Nicht-Ziffern sehr hoch ist.

Zu weiteren Beispielen für die Informationstransformation zählen das Ersetzen von mehreren identischen Zeichen durch ein einziges Zeichen oder das Entfernen von HTML-Tags (`<...>`) aus Dokumenten im HTML-Format.

2.1.2 Reguläre Ausdrücke als Retter in der Not

Obwohl die aufgeführten Beispiele oft nur sehr kleine Teilaufgaben innerhalb einer größeren Entwicklung darstellen, erfordern ihre Lösungen unverhältnismäßig viel ABAP-Code. Genau für solche Aufgaben können reguläre Ausdrücke optimal eingesetzt werden.

Um einen ersten Vorgeschmack auf die Vorteile von regulären Ausdrücken zu erhalten, wird eine der beschriebenen einfachen Aufgaben mit ihrer Hilfe gelöst. Listing 2.1 zeigt zunächst den traditionellen ABAP-Code, mit dem sämtliche Tags aus einem HTML-Dokument entfernt werden, das in der Variablen `mytext` gespeichert ist:

```
WHILE mytext CS '<'.
  begin = sy-fdpos.
  FIND '>' IN SECTION OFFSET begin OF mytext
                                   MATCH OFFSET end.
  len = end - begin + 1.
  REPLACE SECTION OFFSET begin LENGTH len OF mytext WITH ``.
ENDWHILE.
```

Listing 2.1 Traditioneller ABAP-Code

Wie Sie sehen, beinhaltet dieser Code eine Schleife mit zwei Suchvorgängen und eine Anweisung zum Ersetzen, um nur die einfache Aufgabe durchzuführen, alle Elemente zwischen spitzen Klammern sowie diese Klammern selbst auszuwechseln. Dass diese einfache Aufgabe einen derart umfangreichen Code erfordert, ist sicher nicht zufriedenstellend.

An dieser Stelle soll schon einmal gezeigt werden, wie dieses ABAP-Codefragment durch die Verwendung eines regulären Ausdrucks in einen einzeiligen Code umgewandelt werden kann, der, abhängig von der zu verarbeitenden Textmenge, zudem mindestens eine Größenordnung schneller ist:

```
REPLACE ALL OCCURRENCES OF REGEX '<[^>]*>' IN mytext WITH ``.
```

2.2 Einführung in die regulären Ausdrücke in ABAP

Nach einer kurzen Einführung in die Syntax von regulären Ausdrücken und deren Integration in die Sprache ABAP erfahren Sie in diesem Abschnitt, wie Sie reguläre Ausdrücke in Ihren eigenen ABAP-Programmen verwenden können, um die genannten Kategorien der Informationsverarbeitung effizient und effektiv auszuführen.

2.2.1 Grundlagen

Reguläre Ausdrücke, oder kurz *Regexe* (für *Regular Expressions*), sind ein Nebenprodukt der theoretischen Informatik, das in den 1950er-Jahren vom kanadischen Mathematiker Stephen Kleene eingeführt wurde (damals wurde die Informatik natürlich noch Mathematik genannt). Regexe leisteten nicht nur einen Beitrag in der Mathematik, sondern erfreuten sich auch als Teil des UNIX-Betriebssystems und dessen textorientierten Werkzeugen wie *grep*, *awk* und *sed* größter Beliebtheit.

Heute unterstützen praktisch alle modernen Programmiersprachen, einschließlich ABAP, reguläre Ausdrücke entweder nativ oder über zusätzliche Bibliotheken. Frühere Versionen von ABAP boten noch keine Unterstützung für reguläre Ausdrücke und erforderten verschiedene Umwege, um externe Regex-Funktionalität zu nutzen. Ab SAP NetWeaver 7.0 ist die native Regex-Unterstützung nun aber endlich verfügbar.

Grundlegende Ausdrücke

Ein regulärer Ausdruck ist ein *Textmuster*, das stellvertretend für eine oder mehrere *Zeichenfolgen* steht. Die Muster regulärer Ausdrücke bestehen aus normalen Zeichen (Literalzeichen) sowie einigen speziellen Zeichen, die *Operatoren*, *Sonder-* oder *Metazeichen* genannt werden:

```
. * + ? ^ $ ( ) {  }  [ ] \  |
```

Der *Punktoperator* ist beispielsweise ein Platzhalter, der ein beliebiges Einzelzeichen darstellt. Folglich steht das Muster `cat` für die Zeichenfolge `cat`, wohingegen das Muster `c.t` Zeichenfolgen darstellt, die aus drei Zeichen bestehen, mit `c` beginnen und auf `t` enden. Beachten Sie daher, dass ein Regex ohne Operatoren genau eine Zeichenfolge darstellt, nämlich sich selbst.

Ein regulärer Ausdruck repräsentiert eine Menge von Zeichenfolgen. Wenn eine Zeichenfolge eine der von einem Regex repräsentierten Zeichenfolgen ist, wird auch gesagt, dass der Regex mit der Zeichenfolge übereinstimmt oder dass der Regex auf die Zeichenfolge passt (Englisch: *matches*). Das Ermitteln von Übereinstimmungen für Zeichenfolgen ist eine der häufigsten Verwendungen von Regexen: Um zu überprüfen, ob eine Zeichenfolge zu einer Gruppe von Zeichenfolgen mit einer bestimmten Eigenschaft gehört, wird ermittelt, ob ein Regex, der diese Eigenschaft codiert, zur Zeichenfolge passt. Im Beispiel codiert der Regex `c.t` die Eigenschaft »eine Zeichenfolge aus drei Zeichen, die mit `c` beginnt und auf `t` endet«, sodass der Regex `c.t` zum Beispiel mit `cut`, `cat` und `cot` übereinstimmt (mit `car` jedoch *nicht*).

Eine weitere Möglichkeit, um mehr als eine Zeichenfolge über einen regulären Ausdruck zu repräsentieren, besteht darin, sämtliche *Alternativen* aufzulisten und diese durch einen *Pipe-* bzw. *Oder-Operator* (`|`) zu trennen. Der Regex `ten|(twen|thir)ty` passt ausschließlich zu den Zeichenfolgen `ten`, `twenty` und `thirty`. Dieses Beispiel zeigt zudem, dass ähnlich wie in arithmetischen Ausdrücken runde *Klammern* zur Strukturierung des Ausdrucks verwendet werden können.

Soll eine Übereinstimmung für ein Operatorzeichen ermittelt werden, kann diesem Zeichen der *Rückstrich-* bzw. *Backslash-Operator* vorangestellt werden, um es in ein Literal zu verwandeln. Dieser Vorgang wird als *Escaping* bezeichnet, und der Rückstrich ist ein Fluchtsymbol. Beispielsweise stimmt der Regex `10\.0` mit `10.0` überein. Der Regex `10.0` stimmt mit `10.0` überein, darüber hinaus jedoch auch mit `1000`, `10A0` etc. Das Weglassen des Backslash-Operators vor Dezimalpunkten ist ein typischer Anfängerfehler, der häufig unbemerkt bleibt, da der fehlerhafte Regex trotzdem mit dem gewünschten Text übereinstimmt – allerdings nicht nur mit diesem!

Mengen und Klassen

Das ausdrückliche Auflisten zahlreicher Alternativen kann mühsam werden und sich vor allem auch negativ auf die Performance auswirken (siehe Abschnitt 2.4.2, »Performancekontrolle«). Glücklicherweise gibt es als *Mengen* bezeichnete reguläre Ausdrücke, um alternative Einzelzeichen kurz und klar zu definieren.

Der Mengenausdruck `[abc123.!?]` stimmt mit allen innerhalb der eckigen Klammern aufgelisteten Einzelzeichen überein (und nur mit diesen). Beachten Sie, dass alle Metazeichen (Backslash ausgenommen) ihre spezielle Bedeutung innerhalb von Mengen verlieren, sodass der Punkt in der gezeigten Menge nur mit einem literalen Punkt übereinstimmt, nicht jedoch mit einem beliebigen Zeichen.

Darüber hinaus ist die *Negation* einer Menge möglich, indem der Negationsoperator (`^`) als erstes Zeichen in einer Menge hinzugefügt wird. Eine negierte Menge stimmt mit jedem einzelnen Zeichen *außer* den in der Menge aufgelisteten Zeichen überein. Auf diese Weise stimmt der Ausdruck `[^0-9a-f]` zum Beispiel mit allen Einzelzeichen außer einer hexadezimalen Ziffer überein. Beachten Sie, dass das Winkelzeichen für eine Negation an erster Position stehen muss, da es anderenfalls als literales Winkelzeichen interpretiert wird.

Um einen der Mengenoperatoren oder die schließende Klammer selbst zur Menge hinzuzufügen, können Sie einfach, wie beschrieben, den Backslash-Operator als Fluchtsymbol benutzen. Der Ausdruck `[\^\-\]]` stimmt beispiels-

weise mit einem Winkelzeichen, einem Bindestrich oder einer schließenden eckigen Klammer überein.

Ein *Bereich* bietet eine noch präzisere Möglichkeit, um Mengen zu definieren. Hier müssen die Zeichen nicht einzeln aufgelistet werden, sondern der Bereichsoperator (`-`) wird verwendet, um den Start- und den Endpunkt eines Zeichenintervalls festzulegen. Anstelle von `[0123456789abcdef]` kann so die einfachere Schreibform `[0-9a-f]` verwendet werden, um eine einzelne hexadezimale Ziffer zu charakterisieren.

Bereiche sind zwar insbesondere in technischen Domänen sehr nützlich, aber Sie sollten dabei immer beachten, dass Bereiche von der Textumgebung abhängig sind! Beispielsweise hängen vom Bereich `[ä-ß]` abgedeckte Zeichen stark von der aktuellen Codepage ab. Daher ist es schwierig, ABAP-Code zu pflegen, der solche Bereiche verwendet. Glücklicherweise lassen sich die meisten Probleme im Zusammenhang mit Bereichen ganz einfach vermeiden, indem Sie vordefinierte Zeichenuntermengen benutzen, die als *Zeichenklassen* bezeichnet werden.

In Zeichenklassen enthaltene Zeichen sind zwar ebenfalls von der Textumgebung abhängig, jedoch bleibt die zugedachte Bedeutung jeder Klasse für alle Codepages einheitlich. Um eine Klasse zu verwenden, fügen Sie den Klassennamen oder die zugehörige Kurzform (siehe Tabelle 2.1) in eine Mengendefinition ein. Zum Beispiel:

- Die Mengendefinition `[[:alpha:][:digit:].,?!\-"']` passt zu allen Einzelzeichen, die typischerweise in Schriftstücken verwendet werden.
- Die Mengendefinition `[[:word:]@\-+.]` passt zu allen Einzelzeichen, die typischerweise in E-Mail-Adressen verwendet werden.

Klassenname	Kurzform	Zeichen	Negierte Kurzform
`[:alpha:]`		alle alphanumerischen Zeichen	
`[:digit:]`	`\d`	alle Ziffern	`\D`
`[:upper:]`	`\u`	alle Großbuchstaben	`\U`
`[:lower:]`	`\l`	alle Kleinbuchstaben	`\L`
`[:word:]`	`\w`	alle alphanumerischen Zeichen zuzüglich Unterstrich (_)	`\W`
`[:space:]`	`\s`	alle Leerzeichen, Tabulatoren, Zeilenvorschübe, Zeilenumbrüche und Seitenvorschübe	`\S`
`[:punct:]`		alle Interpunktionszeichen	

Tabelle 2.1 Einige vordefinierte Zeichenklassen

Die Kurzformen der Zeichenklassen können auch außerhalb von Mengen verwendet werden, um die Lesbarkeit der Ausdrücke zu verbessern. Die negierten Kurzformen wie `\D` entsprechen negierten Zeichenmengen wie `[^\d]` und können daher nicht innerhalb von Mengendefinitionen verwendet werden.

Wiederholungen

Bisher wurden lediglich Platzhalter eingeführt, die für Einzelzeichen stehen. Für die Definition von Platzhaltern für mehrere Zeichen steht eine Vielzahl von *Wiederholungs-* bzw. *Verkettungsoperatoren* (englisch *Quantifiers*) zur Verfügung (siehe Tabelle 2.2).[2] Ein Verkettungsoperator wiederholt den unmittelbar vorangestellten Regex und erzeugt damit Verkettungen dieses Regexes.

Verkettungsoperatoren	Bedeutung
`a+`	mindestens eine Wiederholung des Zeichens `a`
`a*`	keine oder mehrere Wiederholungen des Zeichens `a`
`a{n}`	exakt *n* Wiederholungen des Zeichens `a`
`a{n,m}`	mindestens *n* und maximal *m* Wiederholungen des Zeichens `a`
`a?`	ein optionales Zeichen `a` (das heißt keine oder eine Wiederholung des Zeichens `a`)

Tabelle 2.2 Regex-Verkettungsoperatoren für Wiederholungen

Wie bei Alternativen lässt sich mithilfe von Klammern der Bezug von Verkettungsoperatoren auf größere Unterausdrücke erweitern:

- `abc*` wiederholt `c` und stimmt mit `ab`, `abc`, `abcc`, `abccc` etc. überein.
- `a(bc)*` wiederholt `bc` und stimmt mit `a`, `abc`, `abcbc`, `abcbcbc` etc. überein.
- `[abc]*` wiederholt `[abc]` und stimmt mit `a`, `b`, `c`, `aa`, `ab`, `ac`, `ba`, `bb`, `bc`, `ca`, `cb`, `cc`, `aaa`, `aab` etc. sowie der leeren Zeichenfolge überein.

Wie im letzten Beispiel gezeigt, wird der quantifizierte Unterausdruck für jede Wiederholung erneut ausgewertet, das heißt, der Unterausdruck stimmt möglicherweise bei jeder Iteration mit einer anderen Zeichenfolge überein. Folglich stimmt `[abc]*` nicht nur mit Zeichenfolgen überein, die aus identischen

2 Das Arsenal an Verkettungsoperatoren ist umfangreicher als eigentlich notwendig. Beispielsweise kann `(a+)?` anstelle von `a*` oder `aa*` anstelle von `a+` verwendet werden. Es ist sogar zusätzlich `a{n,}` vorhanden, was `a{n}a*` entspricht.

Zeichen bestehen, sondern auch mit beliebigen Kombinationen aus den Zeichen `a`, `b` und `c`.

Die richtige Anzahl an Wiederholungen wird für jeden Unterausdruck implizit so bestimmt, dass der gesamte Regex möglichst übereinstimmt, sofern dies überhaupt möglich ist. Das bedeutet, dass der Verkettungsoperator beim Anwenden des Regexes `a*aa` auf die Zeichenfolge `aaaa` eine Übereinstimmung von `a*` für die ersten beiden Zeichen `a` ermittelt und die verbleibenden Zeichen `a` den literalen Zeichen `a` im Regex vorbehalten bleiben.

2.2.2 Reguläre Ausdrücke für die Validierung

Nach der Einführung der grundlegenden Operatoren für reguläre Ausdrücke wenden sich die Ausführungen nun einigen Programmieraufgaben zu. Sie werden überrascht sein, wie nützlich bereits die beschriebenen einfachen Regexe eingesetzt werden können.

In diesem Abschnitt sollen Eingaben auf bestimmte Merkmale hin überprüft werden, indem diese Merkmale in einem regulären Ausdruck codiert werden. Idealerweise stimmt der Regex nur dann mit der Eingabe überein, wenn diese das hier interessante Merkmal aufweist. Die Probleme beim Schreiben eines solchen regulären Ausdrucks sind entweder *falsch positive* Ergebnisse (das heißt Eingaben, für die selbst dann eine Übereinstimmung ermittelt wird, wenn sie nicht das gewünschte Merkmal aufweisen) oder *falsch negative* Ergebnisse (das heißt Eingaben, für die selbst dann keine Übereinstimmung ermittelt wird, auch wenn sie das gewünschte Merkmal aufweisen).

Eine bestimmte Anzahl an falsch positiven oder falsch negativen Ergebnissen (jedoch selten beide) ist für zahlreiche Anwendungen sogar akzeptabel. In diesem Fall ist ein Kompromiss zwischen der Komplexität des nötigen Regexes und seiner Performance einerseits und der Anzahl an akzeptablen falschen Übereinstimmungen andererseits erforderlich. Damit die in diesem Artikel verwendeten regulären Ausdrücke lesbar bleiben, wird eine bestimmte Anzahl an falsch positiven, jedoch keine falsch negativen Ergebnisse akzeptiert.

Plausibilitätsprüfung von Kreditkartennummern

Bereits in Abschnitt 2.1.1, »Drei Typen von Textverarbeitungsaufgaben«, wurde die Überprüfung von Kreditkartennummern auf Plausibilität hin angesprochen. Doch was genau *ist* eine plausible Kreditkartennummer? Angenommen, eine Anwendung erfordert, dass keine ungültigen Zeichen in der Nummer enthalten, aber bestimmte Formatierungen der Nummer zugelassen sind.

In diesem Fall kann davon ausgegangen werden, dass eine gültige Kreditkartennummer ausschließlich Ziffern, Leerzeichen und Bindestriche enthält. Ein regulärer Ausdruck, der ausschließlich mit solchen Kreditkartennummern übereinstimmt, lässt sich problemlos als `(\d|\s|-)+` schreiben.

Eine gültige Kreditkartennummer besteht demnach aus einer beliebigen Kombination von Ziffern, Leerstellen und Bindestrichen. Um die Gültigkeit eines in einer Variablen `cc` gespeicherten Eingabewertes zu überprüfen, muss getestet werden, ob der reguläre Ausdruck mit dem Inhalt der Variablen `cc` übereinstimmt. In ABAP lässt sich diese Aufgabe leicht mit der statischen Methode `matches( )` der Klasse `cl_abap_matcher` durchführen (siehe Listing 2.2).[3]

```
IF cl_abap_matcher=>matches(
      pattern = '(\d|\s|-)+'
      text    = cc           ) = abap_false.
  "invalid credit card number
  RETURN.
ENDIF.
```

Listing 2.2 Statische Methode matches( ) der Klasse cl_abap_matcher

Abhängig von speziellen Anforderungen, soll jetzt möglicherweise die Anzahl an falsch positiven Ergebnissen reduziert werden, indem zusätzlich überprüft wird, ob die Eingabe exakt 15 oder 16 Stellen umfasst. Werden Leerstellen und Bindestriche zunächst verboten, genügt der einfache Regex `\d{15,16}` für die Prüfung, ob die Kartennummer die richtige Anzahl von Ziffern enthält.

Die Kombination beider Ausdrücke, um die richtige Anzahl an Ziffern zu prüfen und dabei gleichzeitig Leerstellen und Bindestriche zuzulassen, ist nicht so einfach, wie zunächst vermutet werden könnte. Ein erster trivialer Versuch `(\d|\s|-){15,16}` ist falsch, da Leerzeichen und Bindestriche hier ebenfalls als Stellen gezählt werden. Dadurch wird zum Beispiel fälschlicherweise eine Übereinstimmung für eine Zeichenfolge ermittelt, die zehn Ziffern und fünf Leerzeichen umfasst. Dies entspricht jedoch eindeutig nicht dem gewünschten Ergebnis. Um dieses Problem zu lösen, muss die Wirkung des Verkettungsoperators auf das Teilmuster `\d` beschränkt werden, während zusätzliche Leerzeichen und Bindestriche unbegrenzt zulässig sind:

```
(\s|-)*(\d(\s|-)*){15,16}
```

3 Mit den Releases 7.0, EhP2 und 7.1/7.2 wurde hierfür sogar eine eingebaute Funktion `matches` eingeführt, die direkt einen Wahrheitswert zurückgibt (eine sogenannte Prädikatfunktion). Der logische Ausdruck vereinfacht sich bei ihrer Verwendung zu `IF matches(...)`.

Zur Erklärung wird mit dem ursprünglichen Ausdruck `\d{15,16}` zum Zählen der richtigen Anzahl von Ziffern begonnen, in den die hervorgehobenen Abschnitte eingefügt werden, um zusätzliche Leerzeichen und Bindestriche zuzulassen. Jede Position der Kreditkartennummer kann nun eine beliebige Anzahl an Leerzeichen und Bindestrichen enthalten.

Validierung von Dezimalzahlen

In einem weiteren Beispiel soll gezeigt werden, wie ein Regex erstellt wird, der mit Dezimalzahlen[4] übereinstimmt. Generell besteht eine Dezimalzahl aus mehreren Ziffern, denen ein Dezimaltrennzeichen und weitere Ziffern nachgestellt sind. Es soll jedoch die Möglichkeit berücksichtigt werden, dass einige dieser Abschnitte möglicherweise nicht vorhanden sind (zum Beispiel könnten im englischen Sprachraum die Ziffern vor dem Dezimaltrennzeichen fehlen). Wird dies direkt auf einen Regex übertragen, lautet dieser:

```
\d*\.?\d*.
```

Beachten Sie, dass der Backslash-Operator als Fluchtsymbol für den Dezimalpunkt verwendet wird. Ohne diesen Operator behält der Punkt seine Bedeutung als Metazeichen bei. Wenngleich der genannte Ausdruck im Allgemeinen funktioniert, werden auch alleinstehende Dezimalpunkte (`.`) ermittelt, was zu einem falsch positiven Ergebnis führt. Wenn dieses fehlerhafte Ergebnis ausgeschlossen wird und auch nachgestellte Dezimalpunkte ohne Nachkommastellen (wie in `10.`) nicht zugelassen werden sollen, kann der Regex folgendermaßen neu gruppiert werden: `\d*(\.\d+)?`

Durch das Ändern des zweiten Sternchenoperators in einen Plusoperator wird sichergestellt, dass dem Dezimalpunkt mindestens eine Ziffer folgt. Indem der Fragezeichenoperator an das Ende verschoben wird, werden für den gesamten Ausdruck optional auch Ganzzahlen zugelassen.

Um in ABAP mit großen Zahlen vom Typ `p` zu arbeiten, sollte möglicherweise auch eine Unterstützung für Tausendertrennzeichen (`,`) hinzugefügt werden. Eine nur vermeintliche Lösung wäre eine Lockerung der Zeicheneinschränkung für den ganzzahligen Abschnitt des Wertes, indem `\d*` durch `(\d|,)*` ersetzt wird. Dadurch stimmt der Ausdruck jedoch mit unsinnigen Zeichenfolgen, wie zum Beispiel `1,,,.0`, überein.

4 Hier wird sich auf die englischsprachige Notation mit Dezimalpunkt und Komma als Tausendertrennzeichen bezogen, die in technischen Systemen häufig Anwendung findet.

Dies lässt sich vermeiden, wenn eine Eigenschaft für Trennzeichen gefunden wird, die für alle gültigen Ganzzahlen gilt. Gemäß Definition werden Trennzeichen durch exakt drei Ziffern getrennt. Diese Beschreibung kann weiter vereinfacht werden durch die Aussage, dass auf jedes Trennzeichen genau drei Ziffern folgen. Wird diese Eigenschaft in einem Regex codiert, lautet dieser:

```
\d{1,3}(,\d{3})*(\.\d+)?
```

Der Anfangsteil `\d{1,3}` stimmt mit bis zu drei Ziffern vor dem ersten Trennzeichen (falls vorhanden) überein, wohingegen der letzte Unterausdruck `(\.\d+)?` erneut die Nachkommastellen der Zahl abdeckt (falls vorhanden).

2.2.3 Suchen mit regulären Ausdrücken

Bisher bezogen sich die Erläuterungen auf das Übereinstimmen von regulären Ausdrücken mit Zeichenfolgen, etwa um zu prüfen, ob eine Zeichenfolge eine bestimmte Struktur aufweist. In diesem Abschnitt liegt der Schwerpunkt auf einer weiteren wichtigen Anwendungsmöglichkeit für Regexe: das Finden von Informationen durch die *Suche* nach übereinstimmenden Elementen.

Regexe werden im Allgemeinen zum Suchen verwendet, um die Position von mindestens einer Übereinstimmung für den entsprechenden Regex in einer Zeichenfolge zu ermitteln. Trotzdem wird häufig informell die Wendung »nach einem Regex suchen« verwendet, wenn tatsächlich »nach Übereinstimmungen für diesen Regex suchen« gemeint ist. Es ist daher wichtig, dass Sie den Unterschied zwischen Übereinstimmung und *Suchen* im Kopf behalten, wenn Sie den Rest dieses Kapitels lesen. Bedenken Sie zudem, dass in einigen Programmiersprachen, insbesondere Perl, der Begriff Übereinstimmung (Matching) auch für Suchvorgänge verwendet wird, was zusätzlich für Verwirrung sorgt.

Erste längste Übereinstimmungen

Was geschieht bei der Suche nach einem Regex (bzw. nach den *Übereinstimmungen* für einen Regex) in einem bestimmten Text? Zuerst werden alle möglichen Übereinstimmungen des Regexes innerhalb des Textes ermittelt. Anschließend wird in jeder Gruppe aus überlappenden Übereinstimmungen die am weitesten links beginnende Übereinstimmung als Ergebnis zurückgegeben. Ist die Übereinstimmung nicht eindeutig, so wird die längste zurückgeliefert. Als Beispiel soll mit dem Regex `a[ao]*a` in der im Folgenden gezeigten Zeichenfolge gesucht werden:

```
a[ao]*a:   ooaooaoaoXoooaaooooooaoo
             aooa                  (1)
             aooaoa                (2)
               aoa                 (3)
                        aa         (4)
                        aaooooooa  (5)
                         aooooooa  (6)
```

Die ersten beiden Übereinstimmungen (1 und 2) sind die am weitesten links beginnenden Übereinstimmungen in der ersten Gruppe überlappender Übereinstimmungen. Da die zweite Übereinstimmung die längere ist, wird diese als Ergebnis der Suche zurückgegeben. In der zweiten Gruppe handelt es sich bei den Übereinstimmungen 4 und 5 um die am weitesten links ermittelten Übereinstimmungen. Da die zweite Übereinstimmung die längere ist, wird diese als weiteres Ergebnis der Suche zurückgegeben. Nachdem die Ergebnisse auf diese Weise eindeutig bestimmt wurden, muss der Benutzer, der die Suche aufgerufen hat, ein Ergebnis auswählen (in diesem Fall 2 oder 5). Die meisten Suchschnittstellen geben entweder die erste längste Übereinstimmung von links an oder alternativ alle längsten Übereinstimmungen.[5]

Die Regel, dass die am weitesten links beginnende (das heißt erste) längste Übereinstimmung (Leftmost-longest-Regel) zurückgegeben wird, ist mit das natürlichste Verhalten für Suchvorgänge und wurde als POSIX-Standard definiert. Es gibt jedoch auch andere Ansätze für die Suche, die kurz in Abschnitt 2.4, »Technische Aspekte regulärer Ausdrücke«, erwähnt werden.

Beachten Sie, dass ABAP in Wirklichkeit eine intelligentere Suche durchführt und nicht erst intern alle Übereinstimmungen berechnet, bevor die erste längste Übereinstimmung zurückgegeben wird. Es ist jedoch hilfreich, sich dieses Grundprinzip vorzustellen, wenn Sie den Suchprozess gedanklich nachvollziehen wollen.

Da bei der Suche die längste Übereinstimmung als Ergebnis zurückgegeben wird, wird der Vorgang als *gierig* bezeichnet. Dies gilt nicht nur für den gesamten Regex, sondern auch für jeden einzelnen Unterausdruck und macht sich am meisten bei Wiederholungen bemerkbar. An dieser Stelle spielt diese Tatsache noch keine besondere Rolle, es muss jedoch auf das Problem zurückgekommen werden, wenn auf Untergruppen eingegangen wird (siehe Abschnitt 2.3.1, »Arbeiten mit Untergruppen«).

5 Die ABAP-Anweisung `FIND` macht da keine Ausnahme.

Um lediglich zu prüfen, ob eine Übereinstimmung für einen Regex in einem bestimmten Text vorhanden ist, kann die statische Methode `contains( )` der Klasse `cl_abap_matcher` verwendet werden (siehe Listing 2.3).[6]

```
IF cl_abap_matcher=>contains(
      pattern = '\d+'
      text    = mytext ) = abap_true.
  "match found ...
ENDIF.
```

Listing 2.3 Statische Methode contains( ) der Klasse cl_abap_matcher

In Abschnitt 2.2.4 erfahren Sie, wie Sie detaillierte Informationen zum übereinstimmenden Text erhalten, wie beispielsweise die Position oder die Länge der gefundenen Übereinstimmung.

Anker und Grenzen

Mit einer Handvoll Operatoren, die mit *Positionen* anstelle von Zeichen übereinstimmen, können Sie zusätzlich steuern, wo Übereinstimmungen in einem Text gefunden werden sollen.

Die *Anker* `^` und `$` entsprechen dem Anfang bzw. dem Ende einer Zeile. Folglich geben beide Methodenaufrufe in Listing 2.4 den Wert `abap_false` zurück, da `cat` sich weder am Anfang noch am Ende der Zeichenfolge `mytext` befindet. Aufgrund ihres UNIX-Ursprungs reagieren Anker ebenfalls auf in die Zeichenfolge eingebettete Zeilenumbrüche. Falls dieses Verhalten nicht erwünscht ist, kann stattdessen auf die Ankervarianten `\A` und `\z` zurückgegriffen werden, die mit dem tatsächlichen Anfang bzw. Ende der gesamten Zeichenfolge übereinstimmen.[7]

```
mytext = 'the cat sat on the mat'.
rc1 = cl_abap_matcher=>contains(
         pattern = '^cat'
         text    = mytext ).
rc2 = cl_abap_matcher=>contains(
         pattern = 'cat$'
         text    = mytext ).
```

Listing 2.4 Methodenaufrufe für Anker

6 Auch hierfür gibt es seit den Releases 7.0, EhP2 und 7.1/7.2 eine eingebaute Prädikatfunktion `contains`.

7 Die Unterschiede bei der Groß- und Kleinschreibung lassen sich damit erklären, dass es noch einen weiteren Operator `\Z` gibt, der fast die gleiche Funktion wie `\z` erfüllt, jedoch alle Zeilenumbrüche am Ende des durchsuchten Textes ignoriert.

Die *Wortgrenzen* \< und \> entsprechen dem Anfang bzw. dem Ende eines Wortes:

```
\<.at:    Cathy's cat spat at Matt
.at\>:    Cathy's cat spat  at Matt
\<.at\>:  Cathy's cat spat at Matt
```

Technisch gesehen stimmen Wortgrenzen mit Positionen zwischen durch \w und \W darstellbaren Zeichenfolgen überein. Demzufolge ist das Hinzufügen von Wortgrenzen bei dem oberflächlich sehr sinnvoll erscheinenden Regex \<\w+\> aufgrund der Regel der ersten längsten Übereinstimmung eigentlich redundant.

Es gibt noch einige weitere Begrenzungsoperatoren. Der Operator \b entspricht \<|\>, passt also auf Anfang und Ende eines Wortes. Der interessantere Operator \B stimmt mit den Zwischenräumen innerhalb eines Wortes, das heißt Positionen zwischen zwei aufeinanderfolgenden Zeichen \w, überein.

Kontextbasierte Suche nach Informationen

Ein typischer Anwendungsfall für Regexe ist das kontextbasierte Suchen nach Informationen. Erinnern Sie sich an das Beispiel zur Extraktion von Informationen, in dem die Dokument-ID aus einer bestimmten URL extrahiert werden sollte.

Die Position der Informationen wird durch einige *Begrenzungsmerkmale* gekennzeichnet, das heißt eindeutige Zeichen oder Schlüsselwörter, die das Vorhandensein der gewünschten Informationen eindeutig anzeigen. In diesem Beispiel wird der Wert der Dokument-ID durch das begrenzende Schlüsselwort &docid= auf der linken Seite und das nachfolgende Zeichen & auf der rechten Seite präzise angezeigt:

http://docserve.sap.com/serve?user=ralph&docid=NW2004&lang=EN

Um diese Informationen schnell zu finden, könnte eine Suche mit dem Regex &docid=\w+& durchgeführt werden, wenn angenommen wird, dass die Dokument-ID nur aus Buchstaben, Ziffern und Unterstrichen besteht. Da dies aber keine Fälle abdeckt, in denen die docid das erste oder letzte Argument in der URL ist, wäre (\?|&)docid=\w+(&|$) eine bessere Lösung.

2.2.4 Reguläre Ausdrücke in ABAP-Programmen

Nachdem Sie nun die ersten Regexe für das Abgleichen und die Suche schreiben können, wird der Schwerpunkt im Folgenden auf ABAP gelegt und beschrieben, wie Sie die Vorteile von regulären Ausdrücken in Ihrem ABAP-Code nutzen können.[8] Um diesen Erklärungen folgen zu können, ist es hilfreich, mit den allgemeinen Textverarbeitungstechniken in ABAP vertraut zu sein. Um Ihre Kenntnisse diesbezüglich aufzufrischen, können Sie in Kapitel 1, »Effektive Zeichenkettenverarbeitung in ABAP«, nachschlagen.

Suchen und Ersetzen mit FIND und REPLACE

Die wichtigsten Anweisungen für die Textverarbeitung in ABAP sind die Anweisungen `FIND` und `REPLACE`. Ab SAP NetWeaver 7.0 bieten beide Anweisungen eine native Unterstützung für reguläre Ausdrücke. Regexe lassen sich dort einfach dadurch verwenden, indem das Schlüsselwort `REGEX` vor dem Suchmuster eingefügt wird.[9]

Die gefundenen Teilfolgen werden durch die Zusätze `MATCH LINE`, `MATCH OFFSET` und `MATCH LENGTH` zurückgegeben. Um die Übereinstimmungen eines Regexes `mypattern` in der Zeichenfolge `mytext` zu finden, wird daher folgender Code verwendet:

```
DATA: length TYPE i, offset TYPE i.
FIND REGEX mypattern IN mytext
     MATCH OFFSET off MATCH LENGTH len.
IF sy-subrc = 0.
  WRITE: / 'found match of length', len,
           'at position', off.
ENDIF.
```

Beachten Sie, dass die Länge des übereinstimmenden Textes (anders als bei einer `SUBSTRING`-Suche, siehe Kapitel 1, »Effektive Zeichenkettenverarbeitung

8 Eine vollständige Referenz zur Verwendung regulärer Ausdrücke in ABAP sowie die vollständige Erklärung der dort erlaubten `REGEX`-Syntax finden Sie wie gewohnt in der ABAP-Schlüsselwortdokumentation. Sie brauchen nur nach dem Stichwort »regulär« zu suchen oder aus der F1-Hilfe der entsprechenden ABAP-Anweisungen zu den regulären Ausdrücken zu navigieren.

9 Die Anweisungen `FIND` und `REPLACE` wurden zu Release 7.0 zudem im Hinblick auf die Verarbeitung von internen Tabellen und die Harmonisierung von Zusätzen überarbeitet: In internen Tabellen mit zeichenartigem Zeilentyp kann über den Zusatz `IN TABLE` zeilenweise gesucht und ersetzt werden. Zur `FIND`-Anweisung wurde ein neuer `ALL OCCURRENCES OF`-Zusatz hinzugefügt, der im Wesentlichen dem gleichlautenden Zusatz der `REPLACE`-Anweisung entspricht.

in ABAP«) von der Länge des Suchmusters abweichen kann. Um die tatsächliche übereinstimmende Zeichenfolge abzurufen, kann der für ABAP typische Offset-/Längenzugriff verwendet werden:[10]

```
DATA text_matched TYPE string.
text_matched = mytext+off(len).
```

Um nach *allen* Übereinstimmungen innerhalb des Textes zu suchen, kann der Zusatz `ALL OCCURRENCES OF` der `FIND`-Anweisung verwendet werden. Dann können Sie mit dem Zusatz `MATCH COUNT` die Anzahl an ermittelten Übereinstimmungen abrufen. Mit den anderen Zusätzen `MATCH ...` werden jedoch nur Informationen zur *letzten* gefundenen Übereinstimmung zurückgegeben. Die vollständige Liste der Übereinstimmungen kann mit dem Zusatz `RESULTS` abgerufen werden, der eine interne Tabelle vom Dictionary-Typ `match_result_tab` mit dem Zeilentyp `match_result` füllt (siehe Tabelle 2.3), wobei »Ersetzung« sich auf die Verwendung mit `REPLACE` bezieht.

Komponente	Typ	Bedeutung
`line`	`i`	Zeile der gefundenen Übereinstimmung/ durchgeführten Ersetzung in internen Tabellen (anderenfalls 0)
`offset`	`i`	Position der gefundenen Übereinstimmung/ durchgeführten Ersetzung
`length`	`i`	Länge der gefundenen Übereinstimmung/ durchgeführten Ersetzung
`submatches`	`TABLE OF submatch_result`	interne Tabelle für Untergruppen (siehe Abschnitt 2.3.1, »Arbeiten mit Untergruppen«)

Tabelle 2.3 Struktur match_result

Für jede gefundene Übereinstimmung fügt die `FIND`-Anweisung eine neue Zeile vom Dictionary-Typ `match_result` in die Tabelle ein. Um auf diese Informationen zuzugreifen, wird in der Regel eine `LOOP`-Anweisung verwendet (siehe Listing 2.5).

```
DATA res           TYPE match_result_tab.
FIELD-SYMBOLS <m> TYPE match_result.
FIND ALL OCCURRENCES OF REGEX mypattern IN mytext RESULTS res.
```

10 Oder Sie arbeiten gleich mit Untergruppen (siehe Abschnitt 2.3.1 »Arbeiten mit Untergruppen«).

```
LOOP AT res ASSIGNING <m>.
  WRITE / mytext+<m>-offset(<m>-length).
ENDLOOP.
```

Listing 2.5 LOOP-Anweisung für den Zugriff auf Informationen

Der Zusatz `RESULTS` akzeptiert auch flache Strukturen vom Typ `match_result`, jedoch bleiben wie auch bei den `MATCH ...`-Zusätzen lediglich die Informationen zur letzten Übereinstimmung erhalten. Die Verwendung von `RESULTS` mit Strukturen ist daher nur dafür vorgesehen, um bei der Verwendung von `FIRST OCCURRENCE` auf die Informationen zu Untergruppen zuzugreifen (weitere Informationen hierzu finden Sie in Abschnitt 2.3.1, »Arbeiten mit Untergruppen«).

Der aus Kapitel 1, »Effektive Zeichenkettenverarbeitung in ABAP«, bekannte Zusatz `ACCEPTING|IGNORING CASE` kann natürlich auch mit Regexen verwendet werden, um bei der Suche in einem Text die Groß- und Kleinschreibung zu berücksichtigen. Der Zusatz `IN BYTE MODE` ist bei der Angabe `REGEX` dagegen ausgeschlossen, da binäre Muster von regulären Ausdrücken nicht unterstützt werden.

Alle bisherigen Erklärungen zu `FIND` gelten gleichermaßen für die `REPLACE`-Anweisung. In diesem Fall heißen die entsprechenden Zusätze jedoch `REPLACEMENT LINE`, `REPLACEMENT OFFSET`, `REPLACEMENT LENGTH` und `REPLACEMENT COUNT`.

Weitere Informationen zu den Anweisungen `FIND` und `REPLACE` sowie einige Empfehlungen zur Verwendung mit Zeichenfolgen finden Sie in Kapitel 1.

Reguläre Ausdrücke und ABAP Objects

Für moderne ABAP-Programme wird eine objektorientierte Methodologie als bevorzugtes Programmiermodell empfohlen. Um Regexe in objektorientierten Programmen natürlich verwenden zu können, bietet ABAP die beiden Klassen `cl_abap_regex` und `cl_abap_matcher` an.

Die Regex-Klasse `cl_abap_regex` speichert den eigentlichen regulären Ausdruck in einem internen, vorkompilierten Format (weitere Informationen zur Regex-Erstellung finden Sie in Abschnitt 2.4.2, »Performancekontrolle«). Die Matcher-Klasse `cl_abap_matcher` verknüpft ein `cl_abap_regex`-Objekt mit einem Text und dient als zentrale Schnittstelle für die weitere Verarbeitung. Insbesondere bietet diese Klasse Methoden zur sequenziellen Suche, Abfrage und Ersetzung einzelner Übereinstimmungen (*Match-by-Match*).

Um einen Text zu verarbeiten, erstellen Sie zunächst ein Regex-Objekt mit der Anweisung `CREATE OBJECT` (siehe Listing 2.6).

```
DATA myregex TYPE REF TO cl_abap_regex.
CREATE OBJECT myregex
       EXPORTING
         pattern     = 'a*b'
         ignore_case = abap_true.
```

Listing 2.6 Regex-Objekt mit CREATE OBJECT erstellen

Der optionale Parameter `ignore_case` steuert, ob die Groß- und Kleinschreibung bei der Ermittlung der Übereinstimmungen berücksichtig werden soll oder nicht. Bei fehlender Angabe wird die Groß- und Kleinschreibung berücksichtigt. Es sind weitere optionale Parameter verfügbar, von denen einige im Folgenden noch beschrieben werden.

Anschließend erstellen Sie ein Matcher-Objekt, um den neuen Regex mit dem zu verarbeitenden Text zu verknüpfen (siehe Listing 2.7).

```
DATA: mytext    TYPE string,
      mymatcher TYPE REF TO cl_abap_matcher.
CREATE OBJECT mymatcher
       EXPORTING regex = myregex
                 text  = mytext.
```

Listing 2.7 Matcher-Objekt erstellen

Ein Regex-Objekt kann in beliebig vielen Matcher-Objekten wiederverwendet werden. Aus Gründen der Performance sollte ein mehrfach verwendeter Regex deshalb auch eher als Regex-Objekt erzeugt werden, als immer wieder in Anweisungen oder eingebauten Funktionen (ab den Releases 7.0, EhP2 und 7.1/7.2) neu kompiliert zu werden.

Alternativ bietet die Matcher-Klasse auch eine Factory-Methode für die handliche Erzeugung von Regex und Matcher in einem einzigen Schritt, die dann jedoch keine gemeinsame Nutzung von Regex-Objekten zulässt:

```
mymatcher = cl_abap_matcher=>create(
              pattern = 'a*b'
              text    = mytext ).
```

Das Matcher-Objekt speichert eine interne Kopie des zu verarbeitenden Textes, sodass der Inhalt einer Variablen `mytext`, die an den Konstruktor übergeben wird, nicht von nachfolgenden Ersetzungsvorgängen geändert wird. Auf das Ergebnis von Ersetzungen muss in diesem Fall über `mymatcher->text` anstelle von `mytext` zugegriffen werden. Wenn `mytext` vom Typ `string` ist, verursacht das Kopieren von `mytext` in das Objekt dabei zunächst kaum Kosten, da hier

das aus Kapitel 1, »Effektive Zeichenkettenverarbeitung in ABAP«, bekannte Sharing greift. Erst ein Schreibzugriff auf diese Zeichenfolge durch einen Ersetzungsvorgang führt dazu, dass die gemeinsame Verwendung aufgehoben wird, was dann ein potenziell kostenintensiver Vorgang sein kann.

Die Matcher-Klasse bietet eine Vielfalt von Methoden zum Suchen, Abfragen und Ersetzen von Übereinstimmungen (siehe Tabelle 2.4).

Suchmethoden	Abfragemethoden	Ersetzungsmethoden
`find_next( )`	`get_match( )`	`replace_found( )`
`match( )`	`get_line( )`	`replace_next( )`
	`get_offset( )`	
`find_all( )`	`get_length( )`	`replace_all( )`
	`get_submatch( )`	

Tabelle 2.4 Methoden von cl_abap_matcher

Die Interaktion mit einem Matcher-Objekt basiert auf dem Konzept der *aktuellen Übereinstimmung*, das heißt der letzten während des Suchvorgangs gefundenen Übereinstimmung. Eine aktuelle Übereinstimmung wird durch die Methode `find_next( )` bestimmt und von der Methode `replace_found( )` zerstört. Die Abfragemethoden erfordern eine aktuelle Übereinstimmung, haben jedoch keine Auswirkung auf diese.

Anfänglich ist keine aktuelle Übereinstimmung vorhanden, daher muss zunächst `find_next( )` aufgerufen werden. Sobald eine aktuelle Übereinstimmung vorhanden ist, können Sie diese mit den Abfragemethoden untersuchen, um zum Beispiel Informationen wie Offset und Länge zu erhalten. Durch das erneute Aufrufen von `find_next( )` wird mit der nächsten Übereinstimmung fortgefahren (falls vorhanden). Das Ergebnis des Aufrufs `find_next( )` wird als Boole'scher Wert[11] zurückgegeben. Nachdem alle Übereinstimmungen ermittelt wurden, wird die aktuelle Übereinstimmung invalidiert, und weitere Aufrufe von `find_next( )` schlagen mit dem Wert `abap_false` fehl. Listing 2.8 zeigt, wie dieses Muster verwendet werden kann, um alle Ganzzahlen zu addieren, die in einem Text enthalten sind.

```
DATA: sum       TYPE i,
      mymatch   TYPE match_result,
      mymatcher TYPE REF TO cl_abap_matcher.
```

11 Natürlich in der ABAP-typischen Ausprägung `abap_bool`.

```
mymatcher = cl_abap_matcher=>create(
  pattern = '\d+' text = mytext ).
WHILE mymatcher->find_next( ) = abap_true.
  mymatch = mymatcher->get_match( ).
  sum = sum + mymatcher->text+mymatch-offset(mymatch-length).
ENDWHILE.
```

Listing 2.8 Summierung aller in einem Text enthaltenen Ganzzahlen

Jedes Mal, wenn der Matcher eine aktuelle Übereinstimmung gespeichert hat, kann die Methode `replace_found( )` aufgerufen werden, um diese aktuelle Übereinstimmung durch einen neuen Text zu ersetzen. Dadurch wird die aktuelle Übereinstimmung zerstört, sodass alle darauffolgenden Aufrufe einer Abfragemethode eine Ausnahme auslösen. Durch das erneute Aufrufen von `find_next( )` wird jedoch die nächste nicht verarbeitete Übereinstimmung ermittelt und als aktuelle Übereinstimmung gesetzt.

Indem Sie abwechselnd `find_next( )` und `replace_found( )` aufrufen, können Sie die Übereinstimmungen im Text problemlos durchlaufen und für die einzelnen Übereinstimmungen Aktionen ausführen. Dadurch haben Sie die Möglichkeit, zum Beispiel sämtliche Ganzzahlen in einem Textdokument um den Wert 1 zu erhöhen, da der Ersetzungstext berechnet werden kann, *nachdem* die Übereinstimmung ermittelt wurde (siehe Listing 2.9).

```
DATA: value      TYPE i,
      value_text TYPE string,
      mymatch    TYPE match_result,
      mymatcher  TYPE REF TO cl_abap_matcher.
mymatcher = cl_abap_matcher=>create(
  pattern = '\ d+' text = mytext ).
WHILE mymatcher->find_next( ) = abap_true.
  mymatch = mymatcher->get_match( ).
  value = mymatcher->text+mymatch-offset(mymatch-length) + 1.
  value_text = value.
  mymatcher->replace_found( value_text ).
ENDWHILE.
```

Listing 2.9 Erhöhen aller Ganzzahlen in einem Text um den Wert 1

Abbildung 2.1 zeigt die Statusänderungen eines Matcher-Objektes während eines typischen Nutzungszyklus.

Die Matcher-Klasse bietet zum Suchen und Ersetzen noch einige weitere Methoden an. Die Methode `replace_next( )` ist eine Kurzform von `find_next( )`, unmittelbar gefolgt von `replace_found( )`. Die Methode erfordert daher keine aktuelle Übereinstimmung.

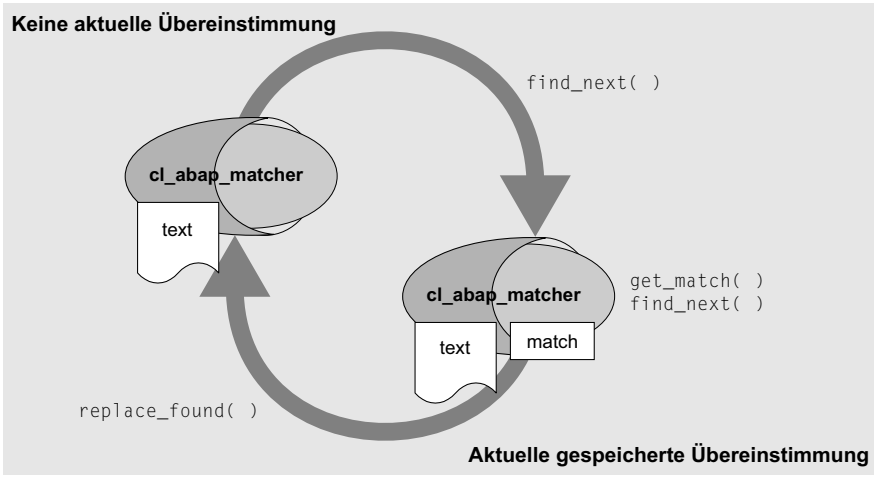


Abbildung 2.1 Typische Verwendung eines cl_abap_matcher-Objektes

Die Methoden `find_all( )` und `replace_all( )` werden für alle noch übrigen Übereinstimmungen im Text ausgeführt. Die Aufrufe beider Methoden invalidieren zudem die aktuelle Übereinstimmung. Die Methode `match( )` vergleicht den Regex mit dem übrigen Text, der noch nicht verarbeitet wurde. Zwar wird die Methode `match( )` in der Regel unmittelbar nach der Erzeugung des Matchers aufgerufen, um eine Übereinstimmung für den gesamten Text zu ermitteln, jedoch ist dies nicht zwingend erforderlich.

Statische Methoden

Die statischen Methoden `matches( )` und `contains( )` wurden bereits in der Einführung in reguläre Ausdrücke erwähnt. Beide statischen Methoden sind für die Verwendung in bedingten Anweisungen als Ersatz für Zeichenfolgenoperatoren wie `CS` und `CP` bestimmt.

Sehr häufig wird lediglich die Information benötigt, ob für einen Regex überhaupt Übereinstimmungen in einem Text vorhanden sind. Es ist jedoch auch möglich, weitere Informationen zu dieser Übereinstimmung abzurufen, wie es auch bei der älteren Vorgehensweise mit Operatoren wie `CS` durch Verwendung von `sy-fdpos` möglich war. Hierzu rufen Sie die statische Methode `get_object( )` auf, um ein Matcher-Objekt zu erhalten, das den Status der vorhergehenden Operation `matches( )` oder `contains( )` darstellt. Anschließend können Sie ganz normal die eingeführten Abfragemethoden einsetzen, um weitere Informationen zu diesem Objekt zu erhalten (siehe Listing 2.10).

```
IF cl_abap_matcher=>contains(
      pattern = mypattern
      text    = mytext    ) = abap_true.
  mymatcher =
      cl_abap_matcher=>get_object( ).
  off = mymatcher->get_offset( ).
  WRITE: / 'found match at offset', off.
ENDIF.
```

Listing 2.10 Informationen zum Objekt abfragen

Theoretisch ist es sogar möglich, die Methoden zum Suchen und Ersetzen für dieses so erhaltene Objekt aufzurufen. Diese Vorgehensweise wird allerdings als schlechter Programmierstil angesehen und deshalb nicht empfohlen.

Hinweis

Ab den Releases 7.0, EhP2 und 7.1/7.2 enthält ABAP einen großen Satz eingebauter Zeichenkettenfunktionen. Von diesen Funktionen nehmen wiederum viele Regexe als Parameter entgegen. Beispiele sind `matches`, `contains`, `find`, `replace` und `match`. Diese Funktionen können statt der Methoden der genannten Klassen verwendet werden, wenn es um einzelne kontextfreie Abgleiche geht, in denen die sonstige hier aufgeführte Funktionalität nicht benötigt wird.

2.2.5 Reguläre Ausdrücke außerhalb von ABAP-Programmen

Reguläre Ausdrücke können auch außerhalb von ABAP-Programmen getestet und verwendet werden. Ab SAP NetWeaver 7.0 bietet der neue ABAP Debugger in zahlreichen Suchdialogfenstern Unterstützung für Regexe. Die Unterstützung in weiteren Werkzeugen ist für zukünftige Releases geplant.

Im Folgenden werden das Regex Toy und der Code Inspector als zwei Beispiele vorgestellt, die nützliche Regex-Funktionalität bieten.

Regexe testen mit dem Regex Toy

SAP NetWeaver 7.0 bietet ein nützliches kleines Programm namens DEMO_REGEX_TOY, mit dem Entwickler ihre regulären Ausdrücke schnell und mühelos testen können.[12] Das interaktive Regex Toy zeigt an, ob und an welchen Positionen Übereinstimmungen für reguläre Ausdrücke in einem vom Benutzer interaktiv bereitgestellten Text gefunden werden. Nach der Eingabe eines Rege-

12 Falls dieses Programm in Ihrer Installation nicht vorhanden ist, müssen Sie das System auf das Support Package 7 aktualisieren. Ab den Releases 7.0, EhP2 und 7.1/7.2 gibt es auch ein kleineres Programm DEMO_REGEX, das für einfache Abgleiche ausreichend ist.

xes und eines Textes hebt das Programm die erste Übereinstimmung bzw. alle Übereinstimmungen für diesen Regex, wie in Abbildung 2.2 gezeigt, im Text hervor. Im unteren Bildschirmbildbereich werden die Untergruppen (siehe Abschnitt 2.3.1, »Arbeiten mit Untergruppen«) für die erste ermittelte Übereinstimmung angezeigt. Wenn Sie die Option REPLACE auswählen, können Sie die ermittelten Übereinstimmungen zudem durch einen beliebigen Text ersetzen.

Abbildung 2.2 Regex Toy

Analyse von ABAP-Programmen mit dem Code Inspector

Vielleicht haben Sie in Ihrer Laufbahn als ABAP-Entwickler schon einmal Code geschrieben, den Sie im Nachhinein lieber nicht zur Benutzung freigegeben hätten. Oder Sie möchten die Qualität von übernommenem Code verbessern, indem Sie nach Codemustern suchen, die die Performance verschlechtern oder Sicherheitsrisiken darstellen.

Für die LOOP AT-Anweisung gibt es beispielsweise zwei Hauptvarianten, die die Zusätze INTO wa und ASSIGNING <fs> verwenden. Die Letztgenannte ist im Allgemeinen vorzuziehen, da der Zugriff auf die aktuelle Zeile der durchlaufenen Tabelle nicht über einen Wertetransport, sondern effizient über eine Referenz erfolgt. Die Aufrufe von Systemfunktionen über CALL name können dagegen zu Sicherheitsproblemen führen, wenn sie nicht ordnungsgemäß durch ABAP-Code mit entsprechenden Authentifizierungsprüfungen geschützt werden.

Angenommen, Sie möchten nun anhand dieser Informationen Ihren Code verbessern und eine einfache Codeprüfung durchführen – wie sollten Sie vorgehen, um die relevanten Anweisungen zu finden? Der einfache Suchdialog des klassischen ABAP Editors weist diverse Einschränkungen auf, die den Nutzen dieses Dialogs für die vorliegende Aufgabe erheblich reduzieren:

- Die Suche erfolgt zeilenweise, sodass bei der Suche nach dem Muster LOOP*INTO keine mehrzeiligen LOOP-Anweisungen ermittelt werden.
- Doppelpunkt-Komma-Ausdrücke ändern den syntaktischen Aufbau von Anweisungen, sodass geschachtelte LOOP-Anweisungen hinter einem LOOP AT: nicht gefunden werden.
- Kommentare und Zeichenliterale enthalten möglicherweise die Suchbegriffe, was zu falsch positiven Ergebnissen führt.
- Negative Suchvorgänge das heißt eine Suche nach Wörtern, die nicht mit einem bestimmten Suchbegriff übereinstimmen, werden nicht unterstützt. Dies ist jedoch im zweiten Beispiel erforderlich (CALL name), um Anweisungen wie CALL SCREEN, CALL FUNCTION oder CALL METHOD auszuschließen, die in der Regel keine Gefahren darstellen.[13]

Der Code Inspector ist ein mächtiges Werkzeug zur Quellcodeanalyse und genau richtig für diese Art von Aufgabe. Das Programm ist im Lieferumfang sämtlicher Releases von SAP NetWeaver enthalten und löst ältere Werkzeuge wie die ohnehin nur zur internen Verwendung vorgesehene Transaktion SAMT ab. Der Code Inspector ist von keinem der genannten Probleme betroffen: Er verarbeitet die Anweisungen in einem normalisierten Quellcode nacheinander, das heißt mit aufgelösten Doppelpunkt-Komma-Ausdrücken, und ermöglicht den Ausschluss von Kommentaren. Eine ausführliche Einführung in das Werkzeug Code Inspector ist an dieser Stelle nicht möglich (siehe hierzu Kapitel 10, »Qualitätsüberprüfung mit dem Code Inspector« in Band 1 von *ABAP – Fortgeschrittene Techniken und Tools*), doch soll trotzdem anhand eines

13 Beachten Sie, dass bei der klassischen Suche nach CALL ' zwar Systemaufrufe ermittelt werden, ohne über CALL SCREENS und ähnliche Elemente zu stolpern; durch Variablen oder Konstanten definierte Systemaufrufe werden allerdings möglicherweise übersehen.

vollständigen Beispiels gezeigt werden, wie Sie sich dort ab den Releases 7.0, EhP2 und 7.1/7.2 Regexe zunutze machen können.

Nachdem Sie den Code Inspector mit Transaktion SCI oder über den Menüeintrag im ABAP Editor gestartet haben, definieren Sie zunächst einen neuen Objektsatz `REGEX_DEMO`, der sämtliche Programme enthält, die Sie untersuchen möchten. Anschließend erstellen Sie eine neue Prüfvariante, mit der alle Objekte innerhalb dieses Objektsatzes durchsucht werden (siehe Abbildung 2.3).

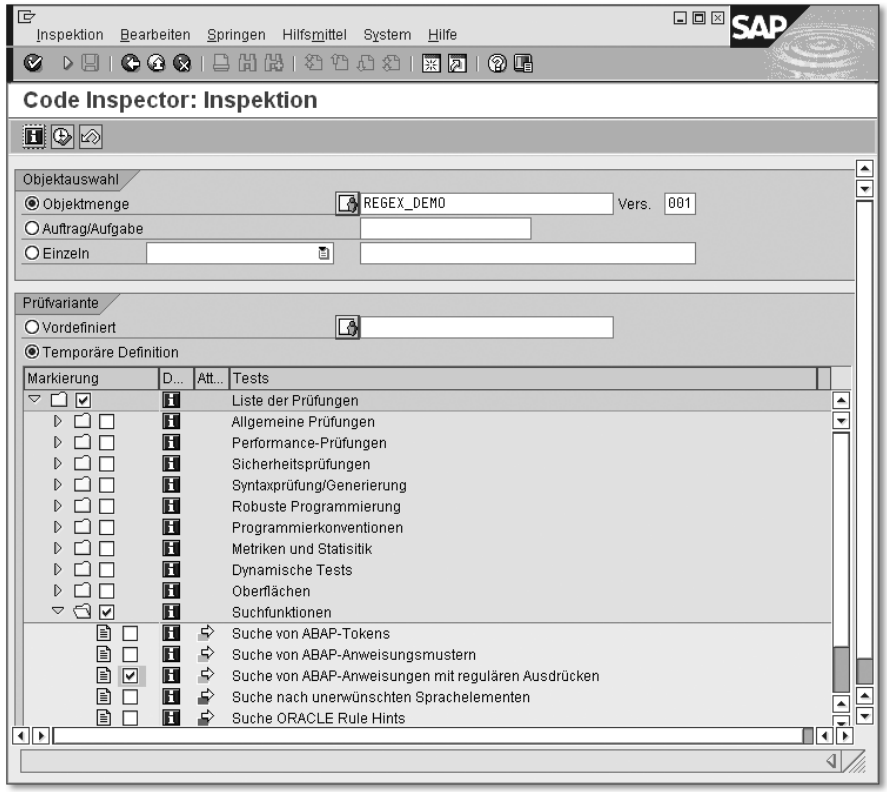


Abbildung 2.3 Erstellen einer neuen Inspektion im Code Inspector

Es stehen zahlreiche vordefinierte Analysen zur Auswahl.[14] Zum Zweck dieser einmaligen Ausführung entscheiden Sie sich jedoch für eine temporäre Definition.[15] Nachdem Sie den Teilbaum SUCHFUNKTIONEN aufgeklappt haben, klicken

14 Dank der umfangreichen vorhandenen Analysen ist es nur selten erforderlich, von Grund auf neue Analysen für den Code Inspector zu schreiben. Bei speziellen Anforderungen finden Sie in der Dokumentation zum Code Inspector weitere Informationen.

15 Hierfür müssen Sie im Einstiegsbild des Code Inspectors das Eingabefeld für den Namen der Inspektion leer lassen und ANLEGEN auswählen.

Sie auf den kleinen Pfeil neben dem seit den Releases 7.0, EhP2 und 7.1/7.2 vorhandenen Element SUCHE VON ABAP-ANWEISUNGEN MIT REGULÄREN AUSDRÜCKEN, um das in Abbildung 2.4 gezeigte Dialogfenster für die Regex-Suche zu öffnen.

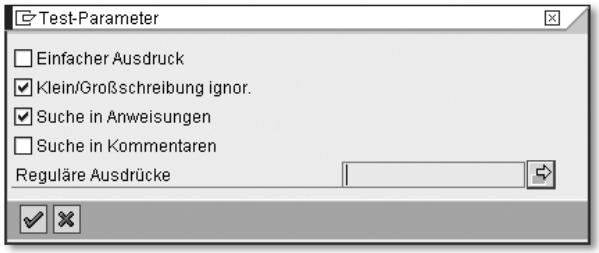


Abbildung 2.4 Dialogfenster für die Regex-Suche

Stellen Sie sicher, dass die Option SUCHE IN ANWEISUGNEN aktiviert ist, und geben Sie den folgenden Regex in das Texteingabefeld ein:

```
(\<LOOP\s+AT\>(?!.+\<ASSIGNING\>))|
(\<CALL\s+(?!FUNCTION|METHOD|SCREEN))
```

Dieses Muster sucht nach `LOOP`- und `CALL`-Anweisungen, die auf die Beschreibung der fraglichen Programmierkonstrukte passen. Um die Genauigkeit der Übereinstimmungen zu erhöhen, werden viele Wortgrenzen und Leerzeichenmengen verwendet. Der neue Operator (`?!...`), der hier bereits integriert wurde, wird als *negative Vorausschaubedingung* bezeichnet. Dieser Operator ermittelt eine Übereinstimmung, wenn der Text, der von dem in den Klammern angegebenen Teilausdruck repräsentiert wird, *nicht* gefunden wird. Dieser nützliche Operator wird in Abschnitt 2.3.2, »Weitere Operatoren und zukünftige Erweiterungen«, detailliert beschrieben. Das Regex-Eingabefeld kann erweitert werden, um Ausnahmen und Alternativen anzugeben. Abgesehen von einer verbesserten Lesbarkeit, wird dadurch jedoch keine neue Funktionalität zur Suche hinzugefügt, da diese Kombination von Regexen bereits durch Regexe selbst ausgedrückt werden kann.

Nachdem Sie das Dialogfenster über den Eingabeknopf ( ) geschlossen haben, kann der Inspektionslauf gestartet werden. Wenn Sie im oberen Bildschirmbildbereich auf den Knopf zum Ausführen klicken oder die Taste F8 drücken, führt das System den Lauf durch. Nach kurzer Zeit sollten eine Meldung über die erfolgreiche Ausführung und ein neuer Knopf zur Anzeige der Ergebnisse erscheinen (siehe Abbildung 2.5).

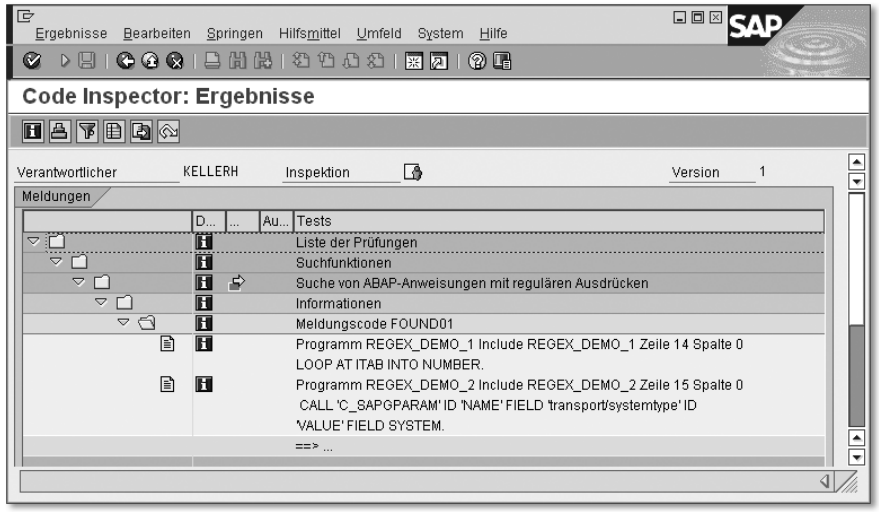


Abbildung 2.5 Ergebnisse der Inspektion

In diesem Beispiellauf hat der Code Inspector zwei Codezeilen ermittelt, die mit den Suchkriterien übereinstimmen. Durch Doppelklicken auf die Ergebniszeilen lässt sich der Quellcode anzeigen, um zu überprüfen, ob und wie sich der Code verbessern lässt.

2.3 Fortgeschrittene Eigenschaften regulärer Ausdrücke

Nach der vorangegangenen Einführung in die elementaren Eigenschaften von regulären Ausdrücken und ihre Verwendung sowohl in ABAP-Programmen als auch in Werkzeugen des SAP NetWeaver Application Servers ABAP wird nun auf einige fortgeschrittene Aspekte dieses mächtigen Konzeptes eingegangen.

2.3.1 Arbeiten mit Untergruppen

Untergruppen gehören zu den nützlichsten Aspekten von regulären Ausdrücken überhaupt. Um reguläre Ausdrücke effektiv zu nutzen, müssen Sie mit dem Konzept der Untergruppen vertraut sein.

Um die wichtigsten Eigenschaften von Untergruppen kurz vorwegzunehmen: Die bereits bekannten runden Klammern strukturieren nicht nur Ihren Regex, sondern speichern zudem eine interne Kopie der Unterfolge, die mit dem eingeklammerten regulären Ausdruck übereinstimmt, intern ab. Diese sogenannten *Untergruppen* können anschließend verwendet werden, um Teile der untersuchten Zeichenfolge

- kontextbasiert zu extrahieren oder
- wertebasiert zu transformieren.

Informationen zu den abgespeicherten Untergruppen können mit den Anweisungen `FIND` und `REPLACE` sowie mit `cl_abap_matcher` abgefragt werden. Darüber hinaus kann sowohl in regulären Ausdrücken als auch in Ersetzungstexten auf diese Untergruppen zugegriffen werden.

Untergruppen

Mithilfe von Klammern lassen sich reguläre Ausdrücke in *Unterausdrücken* strukturieren. Diese Unterausdrücke werden von links nach rechts nummeriert (einsetzend mit 1) und beginnen mit einer öffnenden Klammer. Auch eine Schachtelung ist möglich:

```
a+(b+)(c(e+)|d(e+))f
  <-->(----------)
  1   2
        <-->  <-->
        3     4
```

Wenn Sie Übereinstimmungen für diesen Regex innerhalb des Textes `aabbbdeef` ermitteln, erhalten Sie als Ergebnis die folgenden Übereinstimmungen:

```
Text:                ...aabbbdeef...
Übereinstimmung:        aabbbdeef
Untergruppe 1:            bbb
Untergruppe 2:               dee
Untergruppe 3:                           (leer)
Untergruppe 4:                ee
```

Die Übereinstimmungen mit den Unterausdrücken werden als Untergruppen abgespeichert. Um auf diese Untergruppen zuzugreifen, kann der Zusatz `SUBMATCHES` der Anweisung `FIND` verwendet werden:

```
FIND REGEX 'a+(b+)(c(e+)|d(e+))f'
     IN 'aabbbdeef'
     SUBMATCHES s1 s2 s3 s4 s5.
```

Diese Anweisung speichert die Werte `bbb`, `dee` und `ee` in den Variablen `s1`, `s2` und `s4` und initialisiert die Variablen `s3` und `s5`.

Der Zusatz SUBMATCHES füllt somit alle angegebenen Variablen mit der entsprechenden Zeichenfolge der Untergruppe. Falls weniger Unterausdrücke als Variablen vorhanden sind, werden die überzähligen Variablen initialisiert (wie s5 im Beispiel). Das Gleiche gilt für Variablen, für deren Unterausdrücke keine Übereinstimmungen gefunden wurden (wie s3 im Beispiel).

Um weitergehende Informationen zu erhalten, können Sie den Zusatz RESULTS verwenden, der, wie bereits in Abschnitt 2.2.4, »Reguläre Ausdrücke in ABAP-Programmen«, erwähnt, Ergebnisse der Typen match_result oder match_result_tab zurückgibt und auf deren Komponente submatches zugreift. Diese Komponente ist eine interne Tabelle vom Dictionary-Typ submatch_result_tab, deren Zeilen vom Typ submatch_result den Offset und die Länge für jede Untergruppe enthalten.

Alternativ kann auch wieder die Klasse cl_abap_matcher aus Abschnitt 2.2.4, »Reguläre Ausdrücke in ABAP-Programmen«, mit ihren Abfragemethoden eingesetzt werden, um dieselben Informationen zu erhalten (siehe Listing 2.11).

```
DATA: mymatcher TYPE REF TO cl_abap_matcher,
      str       TYPE string,
      off       TYPE i,
      len       TYPE i.
mymatcher =
    cl_abap_matcher=>create_matcher(
       pattern = 'a+(b+)(c(e+)|d(e+))f'
       text    = 'aabbbdeef' ).
mymatcher->find_next( ).
str = mymatcher->get_submatch( 1 ).
off = mymatcher->get_offset( 2 ).
len = mymatcher->get_length( 4 ).
```

Listing 2.11 Klasse cl_abap_match

Die Methode get_submatch( ) gibt ähnlich wie der Zusatz SUBMATCHES einfach die Zeichenfolge der *nten* Untergruppe zurück. Wenn Sie eine leere Untergruppe angeben, wird auch eine leere Zeichenfolge zurückgegeben. Übergeben Sie jedoch den Index einer *nicht vorhandenen* Gruppe, wird der Ausnahmefehler cx_sy_invalid_submatch ausgelöst. Der spezielle Index 0 bezieht sich auf die gesamte Übereinstimmung, sodass über get_submatch(0) der Text der gesamten Übereinstimmung abgerufen werden kann.

Die übrigen Abfragemethoden aus vorangegangener Tabelle 2.4 unterstützen ebenfalls einen optionalen index-Parameter, der die Zeile, den Offset oder die Länge der entsprechenden Untergruppe zurückgibt.

Wenngleich das Konzept der Untergruppen bisher recht unkompliziert ist, gibt es einige nennenswerte Feinheiten bei deren Verwendung innerhalb von Wiederholungen. Genau wie die wiederholten Unterausdrücke in jeder Iteration neu angewendet werden, wird auch jede potenzielle Untergruppe bei jeder Iteration neu gefüllt. Bei der Anwendung des Regexes

```
((ab])|(cd])|(ef]))*
<--------------------->
1
 <----> <----> <---->
 2      3      4
```

auf den Text `aebe` werden zum Beispiel für die Untergruppen 1 bis 4 die Übereinstimmungen `e`, `b`, `(empty)` bzw. `e` zurückgegeben, da jede Gruppe den letzten übereinstimmenden Zeichenfolgenwert beibehält.

Ein weiterer interessanter Aspekt von Untergruppen betrifft die »Gierigkeit« von regulären Ausdrücken, die auch für Unterausdrücke gilt. Während die Regel zur ersten längsten Übereinstimmung (siehe Abschnitt 2.2.3, »Suchen mit regulären Ausdrücken«) die zurückzugebende Übereinstimmung bei der Suche nach Regexen eindeutig definiert, bleibt zunächst unklar, wie Untergruppen bestimmt werden sollen. Betrachten Sie zum Beispiel folgende Anweisung:

```
FIND REGEX '([ab]+)([bc]+)' IN 'ooaabbccoo'
     SUBMATCHES s1 s2.
```

Die Regel für die erste längste Übereinstimmung legt nicht fest, ob `s1` der Wert `aa`, `aab` oder `aabb` zugewiesen wird. Da die Gierigkeit jedoch auch für Unterausdrücke, und zwar von links nach rechts, gilt, wird der Variablen `s1` der Wert `aabb` zugeordnet, und `s2` erhält den Restwert `cc`. Es ist jedoch wichtig zu verstehen, dass trotz gieriger Unterausdrücke primär der gesamte Regex übereinstimmen muss. Vergleichen Sie das vorhergehende Beispiel mit dem folgenden:

```
FIND REGEX '(a+)(ab)' IN 'ooaaboo'
     SUBMATCHES s1 s2.
```

Hier erhält trotz Gierigkeit `s1` nicht `aa` zugewiesen, da sonst der gesamte Regex nicht mehr übereinstimmen würde. Tatsächlich erfordert die einzige mögliche Übereinstimmung für den gesamten Regex, dass `s1` den Wert `a` und `s2` den Wert `ab` erhält.

Diese Analyse mag Ihnen zunächst recht theoretisch erscheinen. Es ist aber hilfreich, an dieses Verhalten zu denken, wenn Sie Untergruppen zum Extrahieren von Informationen verwenden, was im Folgenden vorgestellt wird.

Extrahieren von Teilinformationen mit Untergruppen

Im ersten Teil dieses Kapitels wurde gezeigt, wie Sie reguläre Ausdrücke für die kontextbasierte Suche nach Informationen verwenden (siehe Abschnitt 2.2.3, »Suchen mit regulären Ausdrücken«). Häufig sind diese Informationen jedoch zusätzlich unterstrukturiert und lediglich bestimmte Teile von ihnen interessant.

Im Folgenden wird dies wieder anhand eines Beispiels gezeigt: Angenommen, Sie schreiben eine Webanwendung, die eine URL wie die folgende mit den Anmeldeinformationen des Benutzers erhält:[16]

http://sap.com/handler?sess=1&login=ralph@geheim&lang=EN

Die Anmeldeinformationen werden über den `login`-Parameter angegeben, der Benutzername und Kennwort enthält, die durch ein `@`-Zeichen getrennt sind. Diese Informationen lassen sich problemlos mithilfe von `FIND REGEX` finden (siehe Listing 2.12).

```
DATA: login TYPE string,
      user  TYPE string,
      pass  TYPE string.
FIND REGEX '&login=\ w+@\ w+' IN url
     MATCH OFFSET off MATCH LENGTH len.
CHECK sy-subrc = 0.
login = url+off(len).
```

Listing 2.12 Anmeldeinformationen mit FIND REGEX ermitteln

Die Unterfolge `&login=ralph@geheim` mit den Anmeldeinformationen ist nun in der Variablen `login` gespeichert. Um den Benutzernamen und das Kennwort zu extrahieren, überspringen Sie zunächst den Namen des Parameters, indem Sie auf `login+7` verweisen und anschließend die restliche Zeichenfolge beim Trennzeichen abtrennen:

```
SPLIT login+7 AT '@' INTO user pass.
```

Dieser Code ist noch etwas umständlich, da Sie die Anzahl der zu überspringenden Zeichen selbst zählen müssen. Darüber hinaus müssen Sie einige der

16 Dieses konkrete Anwendungsbeispiel dient ausdrücklich nur zur Illustration und ist aufgrund seiner Sicherheitsmängel nicht für den Produktivbetrieb geeignet.

Eingaben zweimal durchsuchen, um an die interessanten Teile zu gelangen. Mithilfe von Untergruppen können Sie dieses Verfahren auf einen einzigen Vorgang reduzieren, der zudem weniger anfällig für Änderungen an der URL-Syntax ist:

```
FIND REGEX '&login=(\w+)@(\w+)' IN url
     SUBMATCHES user pass.
CHECK sy-subrc = 0.
```

Die erste Untergruppe des Regexes speichert den in der URL enthaltenen Benutzernamen und die zweite Untergruppe das Kennwort. Die Schlüsselwörter und Begrenzungszeichen, die zur Suche nach den Informationen eingesetzt werden, werden zwar für die tatsächliche Übereinstimmung benötigt, jedoch nicht als Teil des Ergebnisses zurückgegeben. Dies ist einer der entscheidenden Vorteile bei der Verwendung von Untergruppen.

Als weiteres, etwas kompliziertes Beispiel sollen Sie nun einen Pfadnamen wie */usr/sap/work/profile.txt* in *Verzeichnis*, *Basisname* und *Erweiterung* unterteilen:

```
FIND REGEX '(.*/)?([^.]*)(\..*)?' IN path
                                  SUBMATCHES dir base ext.
```

Den ersten Teil des Pfadnamens bildet das Verzeichnis, zu dem auch der letzte Schrägstrich (`/`) zählt. Die Erweiterung ist der letzte Teil des Pfades und beginnt mit dem ersten Punkt (`.`) von links, der nicht zum Verzeichnis gehört. Der übrige Teil zwischen dem Verzeichnis und der Erweiterung ist der Basisname. Jeder dieser Teile kann leer sein.

Der erste Unterausdruck `(.*/)?` stimmt mit dem Teil des Pfadnamens bis zum letzten Schrägstrich überein (sofern vorhanden, anderenfalls bleibt die zugehörige Untergruppe leer). Dieses Ergebnis entspricht dem Verzeichnisteil. Anschließend wird der Text bis zum ersten Punkt ermittelt, der dem Basisnamen der Datei entspricht. Falls kein Punkt vorhanden ist, handelt es sich bei dem übrigen Abschnitt des Pfades um den Basisnamen. Auch hier gilt, dass der Basisname leer sein kann. Zuletzt wird eine Übereinstimmung für den Punkt (falls vorhanden) und den übrigen Teil des Pfades ermittelt, der für die Erweiterung des Pfades steht.

In Fällen wie in diesem Beispiel, in denen Sie Informationen eigentlich nur unterteilen anstatt suchen wollen, sollten Sie einen Abgleich statt einer Suche durchführen, um auszuschließen, dass Sie falsch positive Ergebnisse für eine passende Teilfolge erhalten. Wenn Sie für diese Aufgabe nicht direkt ein Mat-

cher-Objekt verwenden möchten, können Sie für die FIND-Anweisung das Durchführen eines Abgleiches erzwingen, indem Sie den Regex in Zeilenanker ^ ... $ einschließen. In diesem Beispiel mit dem Pfadnamen ist dies allerdings nicht erforderlich, da der verwendete Regex so gierig ist, dass er stets entweder mit dem gesamten Text oder gar nicht übereinstimmt.

Ersetzen von Übereinstimmungen

Ein weiterer nützlicher Aspekt von Untergruppen ist, dass Sie innerhalb eines Ersetzungstextes[17] auf Untergruppen verweisen können. Sowohl die Anweisung REPLACE als auch die replace-Methoden kennen das spezielle Konstrukt $n, das die *nte* Untergruppe der aktuellen Übereinstimmung mit einem Regex bezeichnet.[18] Dies ist sehr hilfreich, um mit einer einzigen Anweisung Informationen hinzuzufügen oder vorhandene Inhalte zu transformieren.

Angenommen, Sie möchten Datumsangaben im amerikanischen Format MM/DD/YY in das europäische Datumsformat DD.MM.YY konvertieren. Mit dem richtigen Regex wird hierfür lediglich eine Zeile benötigt:

```
REPLACE ALL OCCURRENCES OF REGEX '(\d+)/(\d+)/'
        IN mytext WITH '$2.$1.'.
```

Zunächst wird nach den Datumsangaben im amerikanischen Format anhand der begrenzenden Schrägstriche gesucht, und die relevanten Teile mit den Monats- und Tagesdaten werden in Untergruppen extrahiert. Dann werden die alten Datumsangaben durch das neue Format ersetzt, bei dem die Reihenfolge von Tag und Monat umgekehrt ist und Punkte anstelle von Schrägstrichen verwendet werden.

Es ist sogar möglich, in einem Schritt Datumsangaben vom amerikanischen in das japanische Format (YY-MM-DD) und umgekehrt zu konvertieren, indem Sie von der Eigenschaft profitieren, dass Untergruppen ohne Übereinstimmung leer bleiben:

```
REPLACE ALL OCCURRENCES OF
        REGEX '(\d+)/(\d+)/(\d+)|(\d+)-(\d+)-(\d+)'
        IN mytext WITH '$2$6.$1$5.$3$4'.
```

17 Das ist der Text, der in der Anweisung REPLACE hinter WITH angegeben wird.

18 Für die ab den Releases 7.0, EhP2 und 7.1/7.2 verfügbaren eingebauten replace-Funktionen gilt das natürlich genauso.

Die Verarbeitung dieses Codes ist sogar etwas schneller als die Verwendung von zwei separaten REPLACE-Anweisungen für jeweils eine Umwandlungsrichtung.

Wie in der Einführung zu Untergruppen bereits beschrieben, steht die Untergruppe mit dem Index 0 gemäß Konvention für die gesamte Übereinstimmung. Daher können Sie auch $0 verwenden, um die gesamte aktuelle Übereinstimmung in den Ersetzungstext aufzunehmen. Dies ist insbesondere für das Hinzufügen von Informationen zum ursprünglichen Text hilfreich. Nehmen Sie beispielsweise an, dass die Fettschrift-Tags und zu allen Zahlen in einem HTML-Dokument hinzugefügt werden sollen. So einfach sieht das dann aus:

```
REPLACE ALL OCCURRENCES OF REGEX '\d+'
        IN mytext WITH '<b>$0</b>'.
```

Beachten Sie, dass die $n-Operatoren ausschließlich in Ersetzungstext verwendet werden können. Sie sind zwar nicht Teil der Regex-Syntax, es ist aber eine enge Verknüpfung vorhanden. So führt eine ungültige Angabe für Untergruppen, wie zum Beispiel $*, zum Ausnahmefehler cx_sy_invalid_regex_format. Wenn Sie ein literales Dollarzeichen in die Ersetzungstexte einfügen müssen, können Sie einfach das für Regexe übliche Fluchtsymbol verwenden: \$.

Um den Abschnitt über Ersetzungen abzuschließen, sollen Sie sich nochmals an Kapitel 1 erinnern: Beachten Sie beim Arbeiten mit REPLACE, dass die von REPLACE zurückgegebenen Informationen sich anders als bei FIND nicht auf die gefundenen Übereinstimmungen, sondern immer auf die eingefügte Zeichenfolge beziehen:

```
mytext = '-x-x-'.
REPLACE ALL OCCURRENCES OF REGEX '\w+'
        IN mytext WITH 'ooo'
        REPLACEMENT COUNT cnt REPLACEMENT OFFSET final_off.
WRITE: / mytext, cnt, final_off.
```

Als Ergebnis wird -ooo-ooo- 2 5 ausgegeben – das heißt, final_off weist bei der Fertigstellung den Wert 5 und nicht 3 auf.

Rückwärtsreferenzen

Ein weiterer recht ausgefeilter, jedoch eher selten verwendeter Operator für Untergruppen ist der *Rückwärtsreferenzoperator* (*n*), mit dem auf die *nte* unter-

geordnete Übereinstimmung im Regex selbst verwiesen werden kann. Oder mit anderen Worten: Jeder Untergruppe ist ein Operator `\1`, `\2`, `\3` ... zugeordnet, der hinter dem entsprechenden Unterausdruck angegeben werden kann und rückwärts auf die Untergruppe verweist. Er wirkt also im Regex als Platzhalter für die Zeichenfolge der Untergruppe. Dies mag zunächst seltsam klingen, aber nehmen Sie zum Beispiel an, dass Sie über eine einfache Datenbank mit Benutzernamen und Passwörtern verfügen, die als Zeilen aus *user:pass*-Einträgen in einer internen Tabelle `userdata` gespeichert sind. Dann können Sie die Anweisung

```
FIND ALL OCCURRENCES OF REGEX '^(\w+):\1$' IN TABLE userdata
     RESULTS res.
```

verwenden, um alle nachlässigen Benutzer zu ermitteln, deren Kennwort mit dem Anmeldenamen übereinstimmt.

An dieser Stelle ist es nochmals wichtig, den grundlegenden Unterschied zwischen *nicht übereinstimmenden* und *nicht vorhandenen* Untergruppen herauszustellen. Eine Rückwärtsreferenz auf eine Untergruppe ohne Übereinstimmung ist möglich und steht für eine leere Zeichenfolge, das heißt, dass sie praktisch ignoriert wird. Der Regex

```
regex:          (['"])?\w+\1
found here:     This "is" 'not" 'it'!
```

stimmt beispielsweise mit Wörtern überein, die auch in Anführungszeichen stehen können. Dabei müssen die Anführungszeichen jedoch konsistent sein. Wird der Fragezeichenoperator innerhalb des Unterausdrucks platziert, ändert sich zwar die Funktionsweise innerhalb dieses Ausdrucks geringfügig, dies hat allerdings keine Auswirkungen auf das Endergebnis der Übereinstimmung.

Das Verweisen auf nicht vorhandene Untergruppen oder auf Gruppen, die erst hinter dem Rückwärtsreferenzoperator kommen, ist allerdings nicht zulässig und löst die Ausnahme `cx_sy_invalid_regex` aus. Als Faustregel gilt, dass Sie stets überprüfen sollten, ob der *nten* Rückwärtsreferenz mindestens *n* schließende Klammern nachgestellt sind.

Im Gegensatz zu den meisten anderen Regex-Implementierungen ist die Anzahl an Rückwärtsreferenzen in ABAP nicht auf `\1` bis `\9` begrenzt. Stattdessen ist eine beliebige Anzahl von Referenzen, wie zum Beispiel `\69`, zulässig. In anderen Sprachen ist dies nicht möglich, da solche Ausdrücke in der Regel als Steuerungscodes für ASCII-Zeichen interpretiert werden.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z