James E. McDonough

RegEx in ABAP®: Pattern Matching with Regular Expressions

Rheinwerk® Publishing

Bonn • Boston

What You'll Learn

Get started on the journey of using regular expressions to facilitate pattern matching in ABAP programs! First you will familiarize yourself with regular expression language syntax. Then, learn the fundamental principles of RegEx, review pattern matching techniques with ABAP, discover beginner-friendly tools, and more!

1 Getting Started with Regular Expressions

Regular expression, hereafter referred to as *RegEx*, is a technique used in programming for finding and modifying character strings conforming to a specified pattern in a body of text. Since its origination in the 1950s, RegEx processing has evolved to having its own matching pattern language and syntax. In fact, RegEx is independent of programming language, execution environment, and hardware platform. It even has its own industry standards: Perl and POSIX. The implementation of RegExes in ABAP conforms to the POSIX standard.

A simple example of the use of RegExes is as an active spell checker for a text-editing program, where commonly misspelled words are replaced in the text as the user continues typing, such as replacing the string "perfrom" with the word *perform*. In this case, the text the user is creating represents the body of text to be examined for misspelled words, the string "perfrom" represents the pattern of characters to be found in the text, and the word *perform* represents the replacement text to be applied to those locations where "perfrom" has been found.

1.1 First Impressions

For many programmers, their first exposure to the concept of RegExes is through the initial shock of learning that what had been purported to be a simple introduction for beginners contains what appears to be an incomprehensible matching pattern, making their first step into this realm more difficult to take. For instance, the following matching pattern, which was taken from the ABAP example program identified as Listing 5.15 starting on page 312 of *ABAP Objects: ABAP Programming in SAP NetWeaver* (Keller, Krüger; 2nd ed., SAP PRESS 2007), can be used to determine whether a string of characters is in compliance with the format for an email address:

```
\w+(\.\w+)*@(\w+\.)+(\w{2,4})
```

RegEx matching patterns such as this one have been known to instill fear and intimidation into the hearts of some programmers, cause inquisitive facial expressions in others, and leave many grumbling about its cryptic nature. We can diagnose these types of reactions to RegExes as symptoms of *irregularity*, a malady all too common in the software industry.

1.2 A Treatment for Irregularity

Fortunately, there is a simple treatment for irregularity, which is to approach the subject of RegExes slowly and methodically, abandoning any attempts to digest the entirety of its pattern language in one gulp. This E-Bite's aim is to take you by the hand and gradually introduce the basic principles behind RegExes until eventually you understand the concepts underpinning the email address matching pattern shown previously. After all, some of you might be satisfied with learning the bare bones of the RegEx pattern required to get the immediate job done. Others may want to delve further into this subject so they can approach any job requiring the use of RegExes with confidence.

2 Other Pattern-Matching Techniques Used with ABAP

As a portent of things to come, we'll disclose here that the RegEx pattern language facilitates fuzzy matches; that is, a match where only some of the characters of a matching string need to be an explicit match. For example, a pattern may indicate text strings starting with the string "br" and ending with the string "ke" and having any single character intervening between these two strings. This pattern will match the words *brake* and *broke*. The letters "a" and "o", respectively, in these words are the fuzzy part of the match—their actual values are inconsequential to the fact that they merely represent a single character between the explicit strings "br" and "ke".

We typically indicate the presence of these indeterminate values through the use of *wildcard* characters, which is a concept familiar to ABAP programmers.

The RegEx pattern language has its own syntax for indicating such wildcard characters, and it's not the same as the wildcard characters we've come to know and love in ABAP programming. Accordingly, we'll take stock of what we already know about pattern matching with other ABAP techniques so we won't be surprised when these familiar fuzzy matching characters behave differently when used with RegEx patterns. We'll also review the logical expression operators used to identify characters or substrings within strings.

ABAP patterns you're already familiar with fall into these three categories:

- Using wildcard characters with SQL
- Using selection tables
- Using logical expression operators

2.1 Using Wildcard Characters with SQL

The Open SQL set of statements available with ABAP enables writing relational database access statements irrespective of the actual underlying database. The support for native SQL through Open SQL also extends to using the SQL wildcard characters percent sign (%) and underscore (_). These characters can be interspersed with other actual characters to construct a fuzzy matching pattern for use with SQL queries. The percent sign (%) indicates zero or more characters at that location, and the underscore character indicates exactly one character at that location. Typically, you'll use these with the `LIKE` operator on the `WHERE` clause of a `SELECT` statement.

Neither the percent sign nor the underscore is considered a wildcard indication when used with RegExes.

2.2 Using Selection Tables

ABAP can also use selection tables to control whether values conform with matching criteria. Most ABAPers are familiar with these tables through their implicit definition as a result of the `SELECT-OPTIONS` statement defining a field on a selection screen, and later with the introduction of the `RANGES` statement to explicitly define these selection tables as variables.

Selection tables are internal tables defined using a specific four-field structure:

- The first field is of type data element `ddsign` (length 1) and accommodates whether the corresponding range is to be *included* or *excluded* from the search.
- The second field is of type data element `ddoption` (length 2) and facilitates the operator to be used to compare values. Examples of these operators are `EQ` for *equal*, `LT` for *less than* and, applicable to wildcard searches, `CP` for *contains pattern* and `NP` for *does not contain pattern*.
- The third and fourth fields represent the low and high values limiting the applicable range. Whereas both of these last two fields use the same data element as each other, they take on the data element associated with the field for which they are defined, meaning that two different range tables for two different fields may not have the same structures as each other.

These selection tables can be used either with SQL retrievals, through the `IN` operator of the `WHERE` clause of a `SELECT` statement, or with conditional logic, such as through the `IN` operator used with an `IF` statement.

Selection tables also provide wildcard character counterparts to the percent sign and underscore applicable to SQL. The asterisk (*) is the selection table counterpart to the SQL percent sign, indicating zero or more characters at that location; and the plus sign (+) is the selection table counterpart to the SQL underscore, indicating exactly one character at that

location. These characters are applicable as wildcards only when the `CP` and `NP` pattern operators are used. These wildcard characters are (or should be) familiar to SAP users because this is how users specify fuzzy matches when specifying values for screen fields defined via the `SELECT-OPTIONS` statement.

Again, neither the asterisk nor the plus sign is considered a wildcard indication when used with RegExes.

2.3 Using Logical Expression Operators

Here we cover the comparison operators applicable to a logical expression, which is restricted to operating upon character-like data. Table 1 shows that there are four basic operators, each of which has its negation counterpart, for a total of eight operators.

Operators	Description
`CO`; `CN`	Contains only; does not contain only
`CA`; `NA`	Contains any; does not contain any
`CS`; `NS`	Contains string; does not contain string
`CP`; `NP`	Conforms to pattern; does not conform to pattern

Table 1 Character and String Operators Used with Logical Expressions

The first two pairs of operators are character-oriented; that is, they consider each character independently in the set of characters found in the value following the operator when determining the conformance of the value preceding the operator. The last two pairs are string-oriented; that is, they consider the entire string or pattern specified by the value following the operator when determining the conformance of the value preceding the operator.

The same wildcard characters found with selection tables—asterisk and plus sign—also are applicable here with the `CP` and `NP` pattern operators.

As you'll see, RegExes can also consider individual characters or entire strings during its pattern matching processing; however, it doesn't use operators to achieve this.

3 ABAP Support for Regular Expressions

In this section we introduce the ABAP statements, functions, and classes for which SAP provides support with the use of RegExes. These present us with the various options we have at our disposal for incorporating RegExes into ABAP programs.

3.1 FIND and REPLACE

Of those statements supporting RegExes in ABAP, the `FIND` and `REPLACE` statements have been around the longest. In its simplest forms, their respective syntax for accommodating RegExes is shown here:

```
FIND REGEX regular_expression_pattern IN text_to_be_scanned.
REPLACE REGEX regular_expression_pattern IN text_to_be_scanned
        WITH replacement_pattern.
```

The return code `sy-subrc` is set to zero if a match is found with the RegEx pattern. These statements throw the following catchable class-based exceptions related to RegExes:

- `CX_SY_INVALID_REGEX`
- `CX_SY_REGEX_TOO_COMPLEX`
- `CX_SY_INVALID_REGEX_FORMAT`

The last of these exceptions applies only to the replacement pattern associated with the `REPLACE` statement. Accordingly, it's advisable to implement proper exception processing when using these statements (refer to the ABAP statement documentation for more details, available via the Statement Documentation feature in the ABAP editor).

3.2 String Functions

String functions supporting RegExes are more recent additions to the ABAP language than the `FIND` and `REPLACE` statements. These functions are listed here:

- `COUNT`
- `CONTAINS`
- `FIND` and `FIND_END`
- `MATCH`
- `MATCHES`
- `REPLACE`
- `SUBSTRING_AFTER`, `SUBSTRING_BEFORE`, `SUBSTRING_FROM`, and `SUBSTRING_TO`

As you saw with the `FIND` and `REPLACE` statements, these statements also will throw catchable class-based exceptions associated with RegEx processing.

3.3 SAP-Supplied System Classes

SAP provides the object-oriented `cl_abap_regex` and `cl_abap_matcher` classes, which work in concert with each other to facilitate RegEx processing. Later, you'll see how these classes collaborate with each other to fulfill the task of finding those strings in the text-matching RegEx patterns.

4 Regular Expression Utilities

This section provides the first taste of RegExes with ABAP by writing a very simple RegEx testing utility through which to try various ABAP statements supporting RegEx processing.

Also featured here are four comprehensive RegEx testers—one SAP-based and the other three available on the Internet. You should try each of these at least once to get a feel for what they can provide in terms of productivity. RegexToy, the only SAP-based tester, will give you an accurate assessment of what you might expect when using RegExes with ABAP programs because the implementation of RegExes in ABAP is based on the POSIX standard. At least one of the Internet-based RegEx testers is based on the Perl standard, and though most of the elements composing the RegEx language syntax are and behave the same between these two standards, there are subtle differences that could manifest themselves with more complex RegEx patterns. So, as a word of caution, if you use the Internet-based RegEx testers to formulate your pattern, make certain to test that the pattern elicits the same behavior with the ABAP-based RegEx tester to avoid unpleasant surprises.

4.1 A Simple Regular Expression Tester Using ABAP

You can learn a lot about a programming concept by writing a small program and playing with it, which enables you to touch and feel the code and to see how it behaves as different variations are explored. So, let's dive right in and start coding with RegExes. Copy the ABAP program source code shown in Listing 1 to create your own report object for testing RegExes.

```
report zregex_playground.
*---------------------------------------------------------------------
* Properties -
*   Title                   : Regex playground
*   Type                    : Executable program
*   Package                 : $TMP
*   Unicode checks active : checked
*   Fixed point arithmetic: checked
* This program is loosely modeled on the example ABAP program
* shown as Listing 5.15, starting on page 312 of the book ABAP
* Objects: ABAP Programming in SAP NetWeaver (Horst Keller,
* Sascha Krüger; 2nd ed., 2007, SAP PRESS).
*---------------------------------------------------------------------
* Selection screen definition
```

```
*---------------------------------------------------------------------*
selection-screen : begin of screen 100 as window.
parameters       : pattern      type string      lower case
                 , text         type string      lower case
                 .
selection-screen : end   of screen 100.
*---------------------------------------------------------------------*
* Class regex_playground
*---------------------------------------------------------------------*
class regex_playground                  definition.
  public section.
    class-methods: play
                 .
endclass.
class regex_playground                  implementation.
  method play.
    constants    : information    type symsgty   value 'I'
                 , warning        type symsgty   value 'W'
                 , error          type symsgty   value 'E'
                 , conforms_to_pattern type string
                   value 'conforms to RegEx pattern'
                 , does_not_conform_to_pattern type string
                   value 'does not conform to RegEx pattern'
                 .
    data         : notification   type string
                 , severity       type symsgty
                 , exception      type ref to cx_root
                 .
    do.
      call selection-screen 100 starting at 02 02.
      if sy-subrc ne 00.
        return.
      endif.
      try.
        find regex pattern in text.
      if sy-subrc le 00.
        notification             = conforms_to_pattern.
        severity                 = information.
      else.
        notification             = does_not_conform_to_pattern.
        severity                 = warning.
      endif.
      concatenate text
                  notification
                  pattern
             into notification
                    separated by space.
  catch cx_sy_invalid_regex into exception.
```

```
        notification               = exception->get_text( ).
        severity                   = error.
      endtry.
      message notification type information
              display like severity.
    enddo.
  endmethod.
endclass.
*---------------------------------------------------------------------*
* Event start-of-selection
*---------------------------------------------------------------------*
start-of-selection.
  regex_playground=>play( ).
```

Listing 1 ABAP Program Source Code for Simple Regular Expression Tester

Now, follow these steps:

1. Activate the object such that you're able to execute it using your favorite SAP transaction for doing so (SA38, SE80, SE38, etc.). Execute this program using only a single left parenthesis in the pattern field, for which you should receive a failure popup message indicating the RegEx pattern is invalid. A single left parenthesis by itself is an invalid RegEx pattern, so this confirms the ability of the program to intercept the class-based exception `cx_sy_invalid_regex` thrown by the `FIND` statement.
2. Execute this program using the two-character pattern `34` and the five-character text `12345`, for which you should receive an information popup message indicating the text conforms to this pattern.
3. Execute using the same text but with pattern `32` to receive a warning popup message indicating the text doesn't conform to this pattern.

We'll be using these same two patterns and one text repeatedly as we test changes to the program to illustrate the behavior of other ABAP statements supporting RegExes. This text and these simple patterns hardly do justice to the power available with RegExes, but, for now, they are useful in helping you understand the ABAP statements facilitating RegExes. In subsequent sections, we'll explore further the RegEx language syntax and create more sophisticated matching patterns worthy of use with RegExes.

Using Regular Expressions with the FIND Statement

As you can see in the source code of the RegEx playground program in Listing 1, the `FIND` statement uses the pattern we supply as the operand for its RegEx-based `REGEX` clause and uses the text we supply as the operand for its `IN` clause in which to find any matching patterns. `FIND` is a statement that sets the `sy-subrc` system variable depending on the success or failure of the activity: any value greater than zero is an indication of failure. The statement following `FIND` is to check the value of the `sy-subrc` system variable, and, depending on its value, the notification text and message severity are set accordingly.

Using Regular Expressions with the FIND Function

Change the code by commenting out the two lines containing the `FIND` statement and the test of the `sy-subrc` system variable following it, and place the following line of code after these commented lines:

```
if find( val = text regex = pattern ) ge 00.
```

Next, activate and execute the program using the same patterns and text used with the preceding version, for which you should receive similar results.

In this version, we used the `FIND` function, for which success is measured by a return value greater than or equal to zero. As you can see, the `FIND` function doesn't require inspecting the `sy-subrc` system variable as was necessary with the `FIND` statement.

Using Regular Expressions with the COUNT Function

Change the code by commenting out the one line containing the `FIND` function, and place the following line of code after this commented line:

```
if count( val = text regex = pattern ) gt 00.
```

Next, activate and execute the program using the same patterns and text used with the preceding version, for which you should receive similar results.

In this version, we used the `COUNT` function, for which success is measured by a return value greater than zero. This also doesn't require inspecting the `sy-subrc` system variable as was necessary with the `FIND` statement.

Using Regular Expressions with the CONTAINS Function

Change the code by commenting out the one line containing the `COUNT` function, and place the following line of code after this commented line:

```
if contains( val = text regex = pattern ).
```

Next, activate and execute the program using the same patterns and text used with the preceding version, for which you should receive similar results.

In this version, we used the `CONTAINS` function, for which success is implicit and not measured by any return value we can check. Accordingly, there is no check for any return value with this statement.

Using Regular Expressions with the MATCHES Function

Change the code by commenting out the one line containing the `CONTAINS` function, and place the following line of code after this commented line:

```
if matches( val = text regex = pattern ).
```

Next, activate and execute the program using the same patterns and text used with the preceding version, for which you should receive warnings with both patterns.

In this version, we used the `MATCHES` function, for which success is implicit and not measured by any return value we can check. Accordingly, there is no check for any return value with this statement. However, with the `MATCHES` function, the only successful result is when the entire value conforms to the pattern. In our simple tests, we subjected the text `12345` to the patterns `34` and `32`. Although the pattern `34` is embedded within the text `12345`, the text also contains extraneous leading and trailing characters, which causes the `MATCHES` function to fail. You can prove it

works by providing the same string of characters for both pattern and text.

Using Regular Expressions with the SAP-Supplied System Classes

Change the code by commenting out the one line containing the `MATCHES` function, and place the following three ABAP statements after this commented line:

```
create object regex_object exporting pattern = pattern.
matcher_object =
  regex_object->create_matcher( text = text ).
  if matcher_object->find_next( ) is not initial.
```

In addition, change the data statement so it includes the following two additional fields following the definition of the exception data field. The already existing lines are shown in Listing 2 in italics.

```
data          : notification   type string
              , severity       type symsgty
              , exception      type ref to cx_root
              , regex_object   type ref to cl_abap_regex
              , matcher_object type ref to cl_abap_matcher
              .
```

Listing 2 Additional Fields Following the Exception Data Field

Next, activate and execute the program using the same patterns and text used with the preceding version, for which you should receive results similar to those when using the `FIND` statement.

In this version, we used the SAP-supplied classes to facilitate the RegEx processing. The first of these new executable statements creates an instance of the `cl_abap_regex` class, for which we need to provide the RegEx pattern to be used. The next statement creates an instance of the `cl_abap_matcher` class, for which we need to provide the text to be subjected to the RegEx pattern. The next statement invokes the `find_next` method of an instance of the `cl_abap_matcher` class, with a returning value other than initial signifying that the RegEx pattern was found within the text.

When using these classes, only the minimum work is done to satisfy the request. In this case, we used the `find_next` method to locate the next occurrence of the string in the text matching the pattern. That is all it does. If you want to know anything more about what was found, this requires the use of yet another method. So, let's say you want to know where within the text this pattern was located. You could use the `get_offset` method of the `cl_abap_matcher` class to retrieve this information, as illustrated in the snippet of code in Listing 3, which can be copied into the program between the existing `message` and the `enddo` statements (shown in italics), to issue a message indicating the character position within the string where the pattern was found to match.

```
    message notification type information
                    display like severity.
    if severity eq information.
      data     : match_offset type int4
               , match_position type char10
               .
      match_offset = matcher_object->get_offset( ).
      add 01 to match_offset.
      write match_offset to match_position left-justified.
      concatenate 'Match found at position'
                  match_position
             into notification
                    separated by space.
      message notification type information
                   display like information.
    endif.
  enddo.
```

Listing 3 Additional Lines to Show Character Position Where String Is Found

A single instance of the `cl_abap_regex` class can be used to create unlimited instances of the `cl_abap_matcher` class, so long as each of the text values provided for the creation of each new instance of `cl_abap_matcher` is to be subjected to the same pattern used to create the instance of `cl_abap_regex`. In the case of our RegEx playground program, we created new instances of both classes with each iteration through the `do` statement, simply because the pattern itself is being supplied by the user with each iteration.

Indeed, creating an instance of the `cl_abap_regex` class results in the creation of an object known as a *finite automaton*, also known as a *finite state machine*. If you were to browse and follow the source code for the constructor of the `cl_abap_regex` class, you would see that it uses some programming statements that aren't commonly available to the average ABAP programmer. Accordingly, the creation of a finite state machine triggers some sophisticated processing that parses the RegEx pattern and builds the set of finite states required to facilitate the processing of text to find pattern matches. Establishing a finite state machine presumably takes place with any of the statements supporting RegEx processing, but these classes offer you a glimpse into the complex processing necessary to build one in order to satisfy subsequent matching requests.

Now that you have a good understanding of the ABAP statements supporting RegExes, let's dive into the specific tools that are used in testing the validity of RegEx patterns.

4.2 RegexToy

RegexToy is the only SAP-based RegEx tester featured here, the source code for which is available in ABAP Report DEMO_REGEX_TOY.

As you can see from Figure 1, an upper INPUT section provides a REGEX field accepting a RegEx pattern, and a lower TEXT section accepts the text to be subjected to the RegEx pattern. Matching results are shown in the MATCHES section following the INPUT section, but requires the user either to select a radio button corresponding to one of the commands in the OPTIONS section or to press the Enter key while the cursor is positioned in the REGEX or REPLACEMENT fields preceding the OPTIONS section. The matching results area has a white background; those strings matching the RegEx pattern are shown using a bold red foreground text font, while nonmatching text is shown with a nonbold black foreground text font.

If you click the DOCUMENTATION ABOUT REGULAR EXPRESSIONS button at the top of the screen, you're taken to the standard SAP online documentation on RegExes.

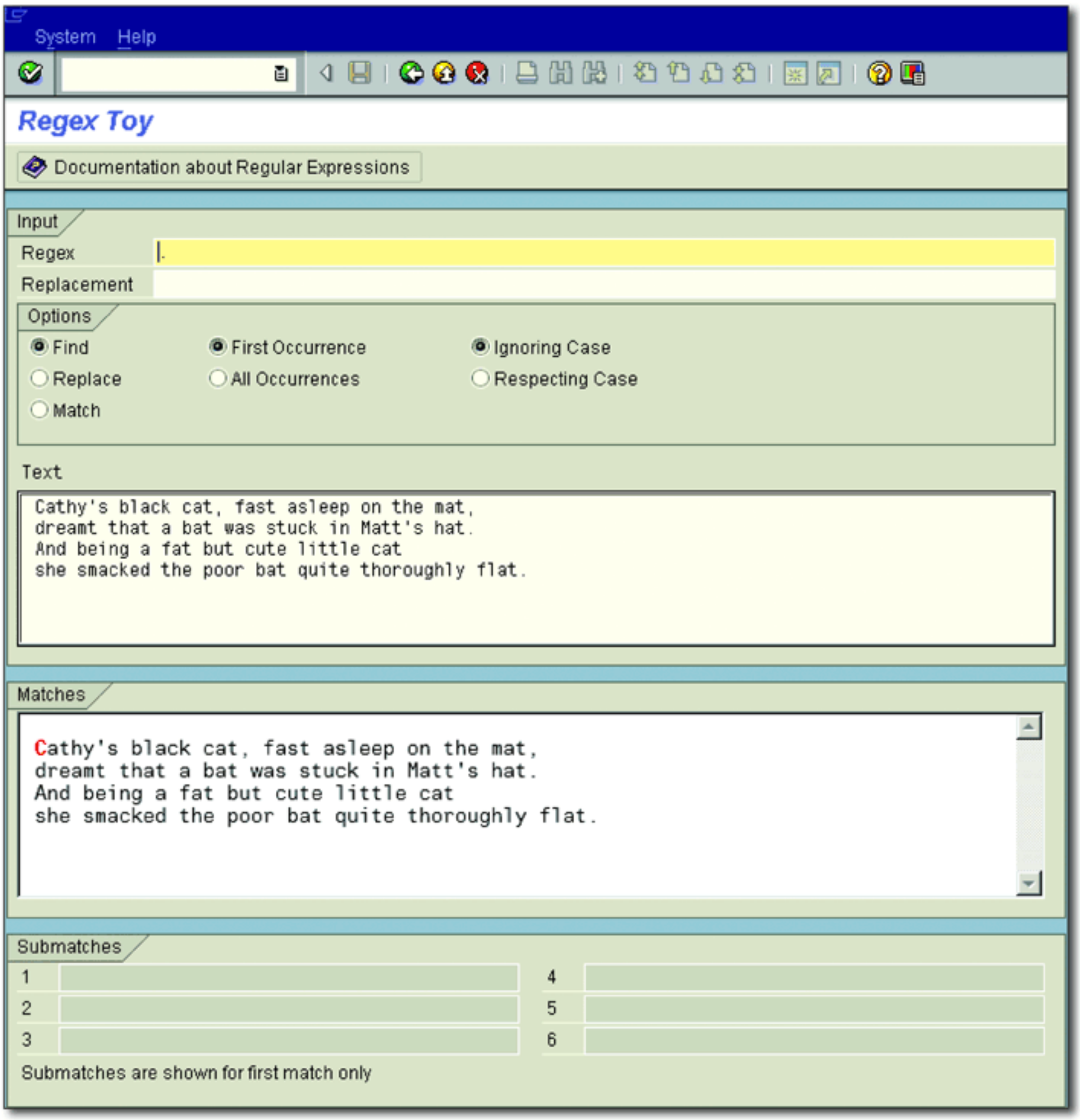

Figure 1 RegexToy Screen Format

This is a magnificent tool by which the nuances of RegEx behavior in ABAP can be tested prior to applying a pattern to productive code under development. Far superior to the meager testing utility we had constructed in the previous section, this utility not only indicates whether a string of text has any matches with the specified RegEx pattern but also shows in context where these matches occur.

Those ABAP programmers who are adventurous enough will find a wealth of information in the code, not the least of which is the ability to see examples of the `FIND` and `REPLACE` statements using their `REGEX` clauses. An article published by the author of RegexToy includes a disclaimer that it isn't so much a tool but merely a toy. However, after you try it, you might agree that the author is being too modest regarding its usefulness and applicability as a development aid.

4.3 RegexPal

RegexPal is an Internet-based JavaScript RegEx tester available at http://regexpal.com/.

Figure 2 shows that an upper text box in RegexPal accepts a RegEx pattern, and a lower text box accepts the text to be subjected to the RegEx pattern. Matching results are shown immediately as the user changes the content of either the pattern or text box. Using the default options, the RegEx pattern is background color-coded as you type it, revealing the segments of the pattern and signifying whether it's syntactically correct, which helps in making sure your RegEx pattern is sound. All matching strings found in the text are identified using alternating yellow and blue background highlighting.

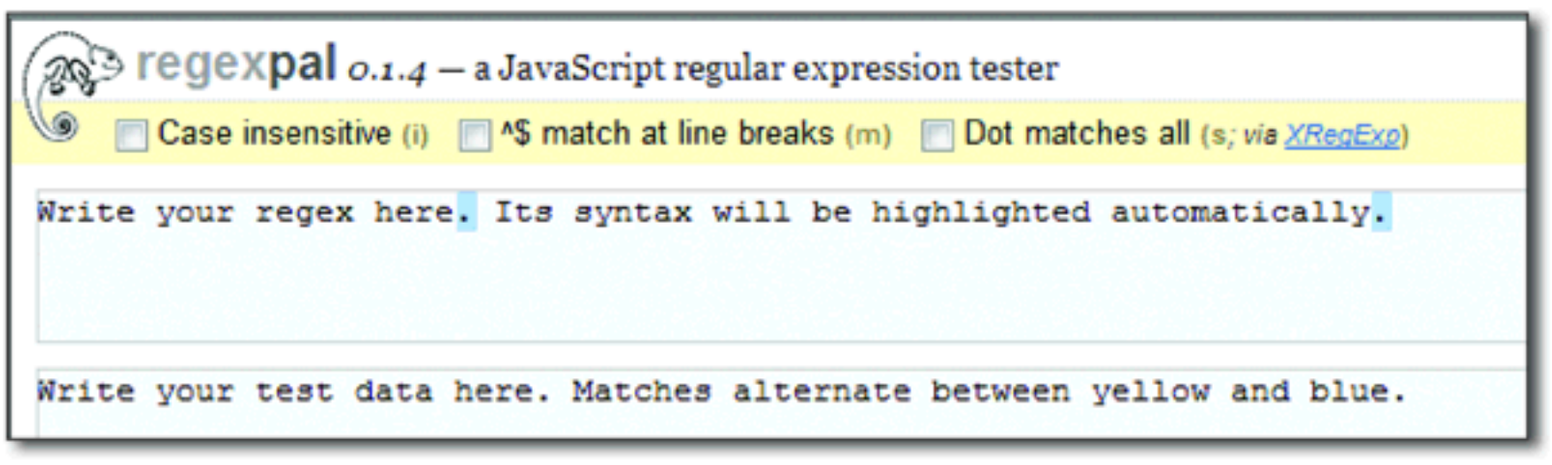

Figure 2 RegexPal Screen Format

This tester has no facility to test the replacement of text, but it does have a selection for a fairly good quick reference and nearly complete reference describing the elements of the RegEx language syntax.

4.4 Rubular

Rubular also is an Internet-based Ruby RegEx tester available at http://rubular.com.

An upper text box accepts a RegEx pattern and a lower text box accepts the text to be subjected to the RegEx pattern, as shown in Figure 3. Matching results are shown in a separate box immediately as the user changes the content of either the pattern or text box. All text is shown using white foreground font against a black background, with the exception of the matching text box, which shows nonmatching text the same way as with the pattern and text boxes but shows matching text with dark blue foreground font against light blue background. Consecutive matches for the pattern are interspersed with a blank character in the matching results box, which can be difficult to discern because the background color of the interspersed blank character is the same as the adjacent matching strings.

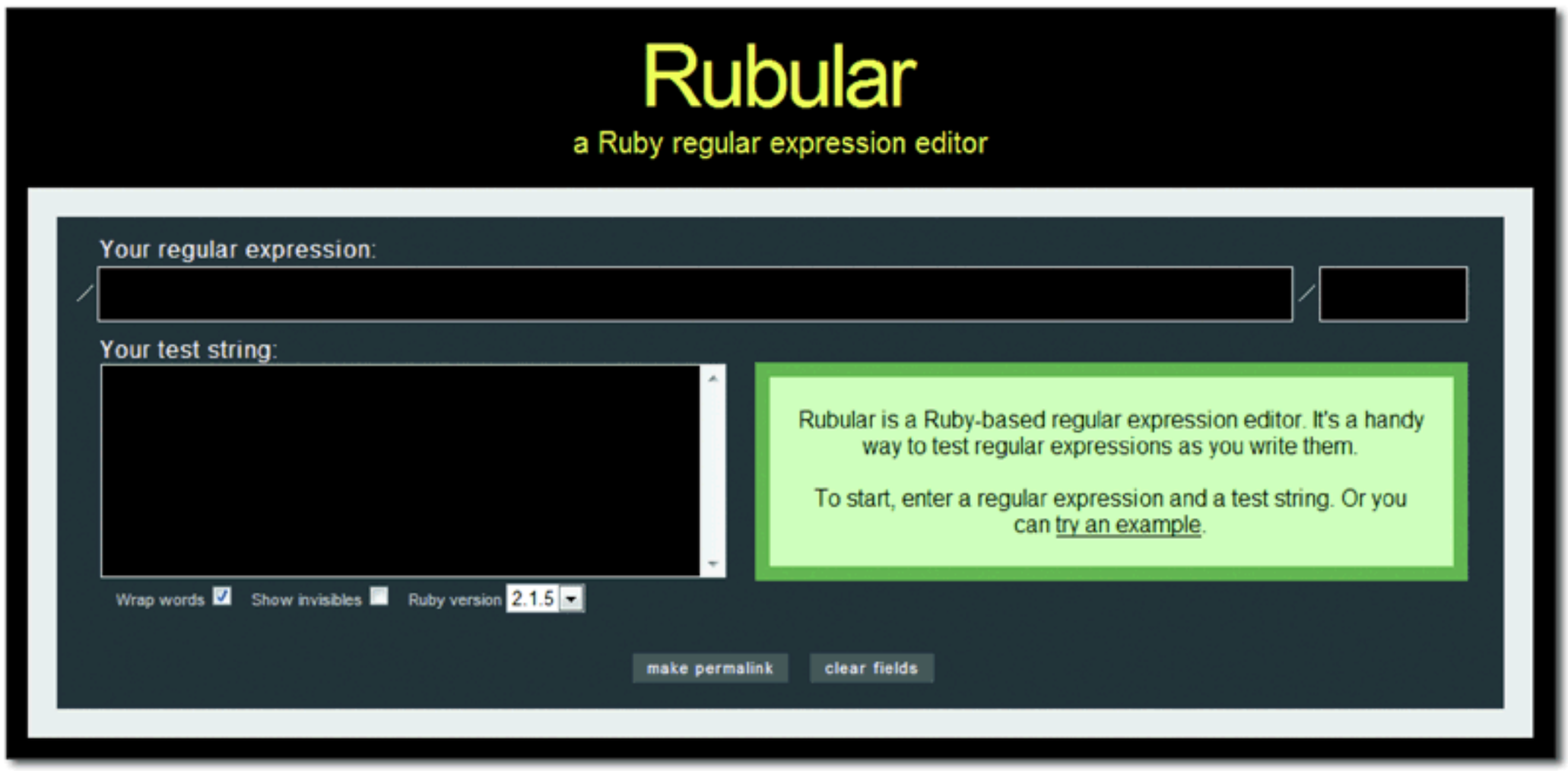

Figure 3 Rubular Screen Format

This tester also has no facility to test the replacement of text, but just below the input areas, it does have a RegEx quick reference describing the elements of the RegEx language syntax.

4.5 RegexStorm

RegexStorm is yet another Internet-based .NET RegEx tester, powered by AJAX and available at http://regexstorm.net/tester.

Similar to the other testers discussed already, Figure 4 shows that an upper text box accepts a RegEx pattern, and a lower text box accepts the text to be subjected to the RegEx pattern. Matching results are shown immediately as the user changes the content of either the PATTERN or INPUT text box. Although there is no color-coding associated with the RegEx pattern, as found with RegexPal, it's similar to RegexPal in that all matching strings found in the text are identified using alternating green and blue background highlighting.

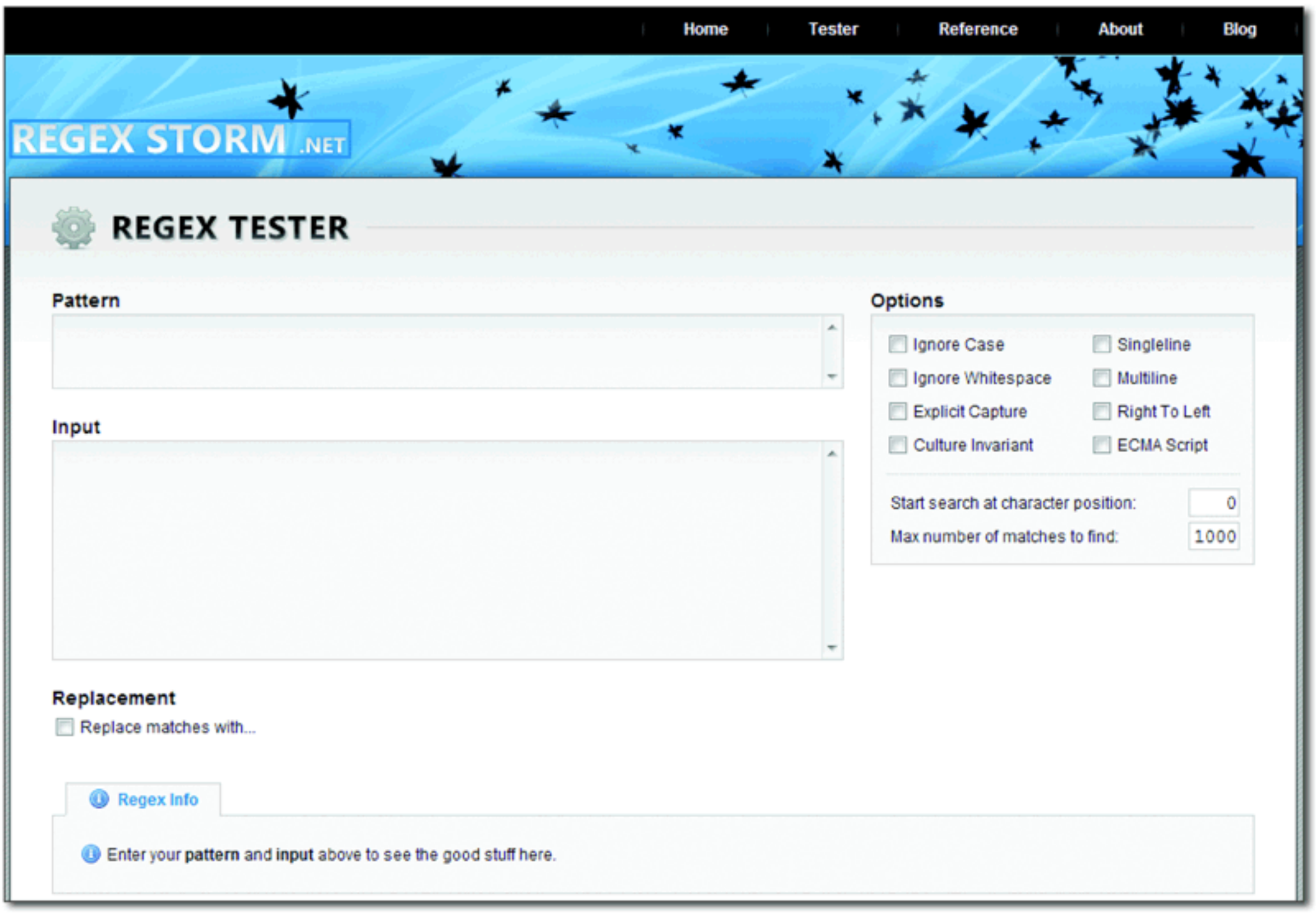

Figure 4 RegexStorm Screen Format

This tester seems to be the most helpful of all the Internet-based testers featured here. It has the capability to perform matching string replacement and provides much more detail about the results of pattern matching. The replacement text box doesn't appear by default, but beneath the INPUT text box is a REPLACEMENT section, with a REPLACE MATCHES WITH checkbox. Clicking on this checkbox places a checkmark in the checkbox, and opens a text box to receive the replacement string. The actual text replacement result can be viewed by selecting the CONTEXT tab at the bottom of the screen (see Figure 5). It also has a selection for a robust reference describing the elements of the RegEx language syntax, including links to see modifiable examples of each language element in context using the tester itself.

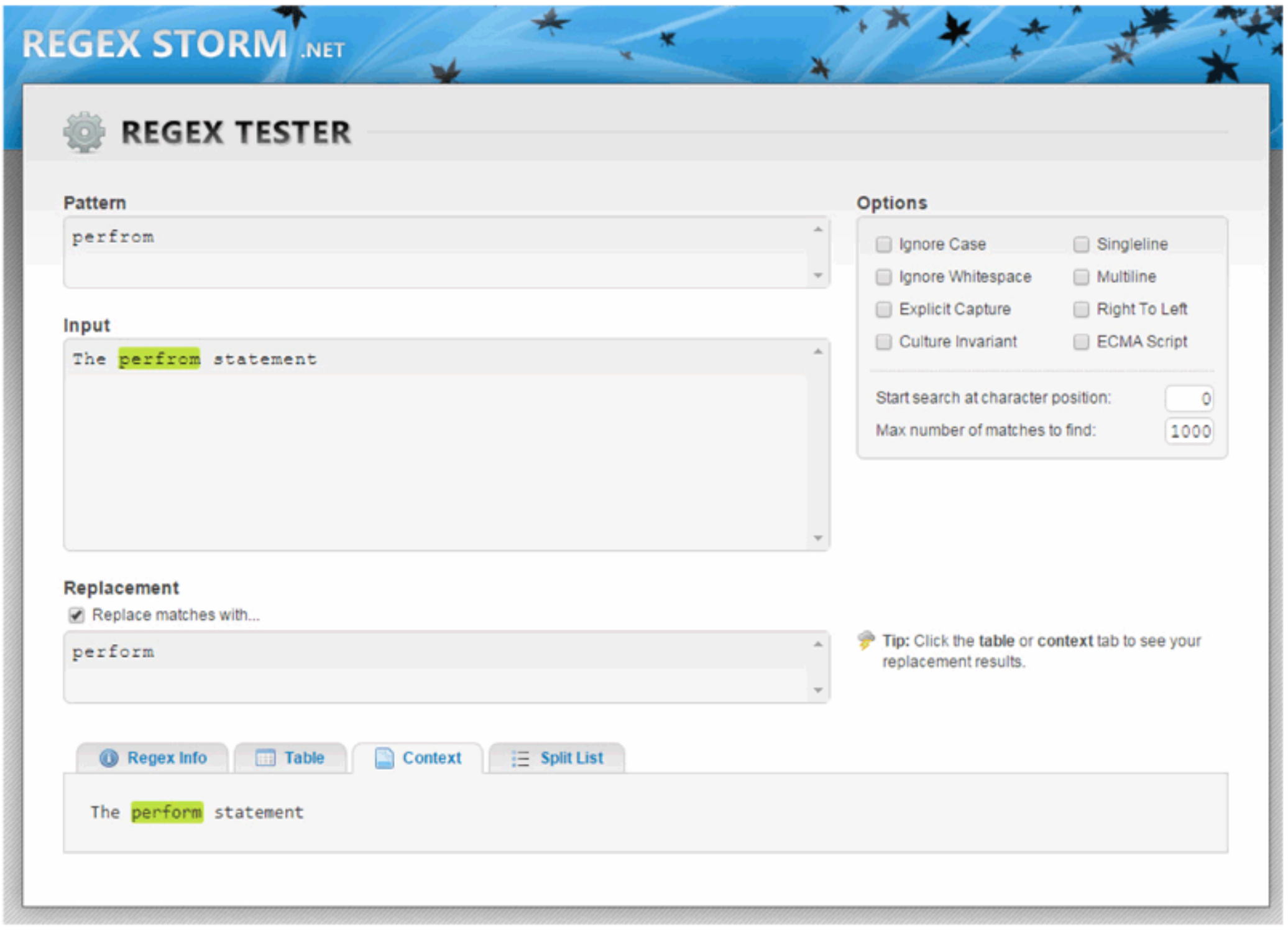

Figure 5 RegexStorm Screen Format with the Context Tab Selected

5 Fundamental Principles of Regular Expressions

Now we'll explore the RegEx language syntax and learn how to use it to create patterns in ABAP to find matching strings in a body of text. Although we don't cover every nuance of the language syntax for RegExes here, we'll go over the basics so that you can begin writing RegEx patterns to solve your processing requirements.

5.1 A Fragrant Introduction to Pattern Matching

Invoke one or all of the RegEx testers presented in the preceding section. Place the cursor in the test data area text box of the tester and type in (or copy and paste) the following famous tongue twister:

A skunk sat on a stump. The skunk thunk the stump stunk and the stump thunk the skunk stunk.

Be careful to type the string exactly the way it appears here, with capitalization at the beginning of the sentences, periods at their ends, and two spaces separating the end of the first sentence and the start of the second. Now place the cursor in the RegEx pattern text box above it, and type the lowercase letter "u". Under the OPTIONS section in RegexToy, select the FIND, ALL OCCURRENCES, and RESPECTING CASE radio buttons.

With both RegexPal and RegexStorm, the text in the lower box immediately becomes highlighted with the locations where the letter "u" appears, alternating between two different colors of background highlighting with each newly found location. With RegexToy and Rubular, the text is copied to a corresponding match result text box, with RegexToy showing the matches using red text foreground highlighting, and Rubular showing them using dark blue text foreground and light blue background highlighting. With all of the testers, you should find that the letter "u" has been found at 10 different locations within the text: it's identified three times within each instance of the word *skunk*, three times within each

instance of the word *stump*, two times within each instance of *thunk*, and two times within each instance of *stunk*.

Congratulations! You've just taken your first step toward understanding RegExes by writing a simple, single-character string-matching pattern. Accordingly, this illustrates the first of the most basic principles of matching text through RegExes.

Basic Principle

Unless otherwise qualified, all alphabetic characters, all digit characters, and most special characters used in RegExes simply match themselves in the target text.

Let's take this a bit further. Type a lowercase letter "n" following the lowercase "u" in the RegEx text box. Now the text box is highlighted with those locations where the string "un" appears, for which there are seven. Accordingly, this illustrates the second of the most basic principles of matching text through RegExes.

Basic Principle

Unless otherwise qualified, consecutive characters used in RegExes will match those locations where the same consecutive characters appear in the target text.

Accordingly, the foundation for using RegExes is for *string matching*.

Continuing with this process, place a lowercase letter "h" at the front of the current RegEx string. Now the text box is highlighted with those locations where the string "hun" appears, for which there only are two. Replace the leading "h" with "k"—now there are three locations where this string "kun" appears. Replace the leading "k" with "t", and now there only are two locations where this string "tun" appears. Finally, replace the leading "t" with "b", and now there is no longer any highlighting of the text because the string "bun" doesn't exist within it.

Remove the "b" from the front of the string so the matches for strings "un" reappear, and then change the first character from a lowercase "u" to its uppercase equivalent. Suddenly again there are no longer any matches in the text. This illustrates the third of the most basic principles of matching text through RegExes.

Basic Principle

Unless otherwise qualified, RegExes are case-sensitive.

5.2 Specifying Character Sets within Regular Expression Patterns

At this point, you may feel a bit more comfortable with constructing RegEx matching strings, but we've only scratched the surface. Our adventures in string matching through the RegEx "un" have located the words *thunk*, *skunk*, and *stunk*, which, as we saw above, locates seven occurrences of the string "un" within the text. Subsequently, qualifying this RegEx string with leading "h", "k", or "t" has resulted in identifying the locations of two, three, and two occurrences, respectively, of these strings. That is, RegEx "hun" identifies two locations where it appears in the word *thunk*, RegEx "kun" identifies three locations where it appears in the word *skunk*, and RegEx "tun" identifies two locations where it appears in the word *stunk*.

Suppose we want to find where the string "hun" *or* the string "kun" appears in the text. One way to do this is to define two separate RegExes—one containing the string "hun" and the other containing the string "kun"—and to use these, one after the other, to identify the locations where either string appears. Most of us might regard this as a rather inelegant solution. However, RegExes provide more than just simple string-matching capabilities—they enable you to define a *pattern* for matching a multitude of different strings to be found in the target text.

In the case of matching "hun" or "kun", we want to match those strings in the text that are three characters in length, begin either with "h" or "k", and immediately are followed by the string "un". RegExes facilitate this by providing a way to identify a *set of characters*, any one of which can be used to satisfy the matching pattern. Such a set of characters is specified between matching square brackets in the RegEx pattern:

```
[hk]un
```

Notice the first character of this string is a left square bracket, which initiates the specification of a set of interchangeable characters. The next two characters are `h` followed by `k`. The next character is a right square bracket, which closes the set of interchangeable characters. The next two characters are `u` followed by `n`.

This pattern effectively intends to match any three-character string starting with either "h" or "k" and ending with "un". It will locate the strings "hun" *and* "kun" appearing in the target text. It's regarded as a three-character string-matching pattern because the first four characters of the pattern—`[hk]`—collectively represent a single character position in the text for string-matching purposes, while the `u` and `n` characters of the pattern simply match themselves when found in the text.

With the characters `h` and `k` appearing as consecutive characters within the square brackets, this becomes our first example of the "unless otherwise qualified" phrase used previously. The bounding square brackets qualify the characters appearing between them as interchangeable characters, which means the contiguous string of characters from the left through the right square bracket is *not* to be regarded as a single string to be matched in the text.

Try this pattern now with the RegEx testers. You should find it identifies five matches—the string "hun" is highlighted two times and the string "kun" is highlighted three times.

Let's have more fun with this and specify a RegEx pattern that, in addition to the strings "hun" and "kun", also is to include all the occurrences of the

string "tun". With what you've already learned, think about how you might alter the pattern to accommodate matching this additional string. The answer simply is to include the character "t" within the set of characters bounded by the square brackets:

```
[hkt]un
```

Try this pattern now with the RegEx testers. You should find it identifies seven matches—the string "hun" is highlighted two times, the string "kun" is highlighted three times, and the string "tun" is highlighted two times. This illustrates the next basic principle of matching patterns used with RegExes.

Basic Principle

Bounding square brackets specified in a RegEx pattern indicates a set of characters, any one of which can match at that character position in the target string.

Indeed, the bounding square brackets aren't regarded as characters to be matched in the text but as *RegEx language syntax*, enclosing those characters that are to be regarded as matching characters in the text. The left and right square bracket characters are two members of the set of special characters composing the RegEx language syntax, none of which, unless otherwise qualified, represents itself in the text.

For a set of characters where each character is identified, it doesn't matter the order in which each character appears within the set. The set "hkt" is equivalent to the set "tkh", which is equivalent to the set "kth". All that is necessary is that the set of interchangeable characters appear between bounding square brackets.

Recall how previously we provided the string "bun" to be located in our tongue twister text and found it resulted in no matches. Suppose we now were to include a lowercase character "b" with our latest set of characters for matching "hun", "kun", or "tun" within the text:

```
[hktb]un
```

How do you think this would change the results of the seven matches we found where the string "un" is preceded by one of the characters "h", "k", or "t" within the text?

The best way to find this answer is to try it with a RegEx tester. You should find that including additional characters within the set of characters to be found preceding "un" in the text has no effect at all—the pattern still finds the same seven matches it found before we included "b" in the set. It merely means that there were no locations in the text matching the string "bun".

Let's change our RegEx pattern so that we have more than one set of characters to match a given character position, as illustrated in the following pattern:

```
[hktb]u[n]
```

Notice we've surrounded the lowercase `n` in the pattern with bounding square brackets. Based on what you've learned so far, how will this change the results we get with the RegEx tester? Keep in mind that now we have not just one, but two sets of characters specified in the pattern—one set is located at the first character position of the three-character string, and the other set is located at the last character position. Again, the best way to know is to try this pattern with a RegEx tester. You should find that this modification also results in no change from the previous pattern. This is because the set of characters representing the third character position of the three-character string includes only the character `n`, meaning that only a three-character string ending with "n" can satisfy a match for the pattern.

Let's now include a lowercase "m" along with "n" as the set of characters that can match at the third character position:

```
[hktb]u[nm]
```

Now when we use this with a RegEx tester, there are 10 matches due to the inclusion of the string "tum" found in the word *stump*, a word appearing three times in the text.

Suppose we were to move the "b" of the first character set from preceding the closing set square bracket to following it:

```
[hkt]bu[nm]
```

By moving this character out of the first character set to appear ahead of the `u`, we've changed the pattern from a three-character string to a four-character string, which requires the string "bu" to occupy the second and third character positions for a match. Because this string doesn't appear within the text, we get no matches for this pattern.

Is there a way to change this pattern, other than moving the "b" back into the first character set, where the alphabetic characters continue to appear in their current sequence, but which will result in matches found within the text? One answer is to include both the "b" and "u" together into a single character set representing matching characters for the second character position. We can do this simply by surrounding "b" and "u" with bounding set square brackets:

```
[hkt][bu][nm]
```

As you'll see when you try this, it results in finding the same 10 matches we found before moving "b" out of the set of characters matching the first character position. When it comes to character sets, the RegEx pattern syntax makes no assumptions about characters composing the set other than that the set may not be empty. A set may contain only one character, or it may contain a multitude of characters. For example, the following two RegEx patterns will get identical matching results:

```
hun
[h][u][n]
```

Exercise 1

Define a RegEx matching pattern that will identify all of the lowercase vowels in the target text. Test this pattern using a RegEx tester with the tongue twister as text.

Answer

```
[aeiou]
```

5.3 Specifying Character Ranges for Character Sets

Consider a scenario where we found a need to identify those locations in some text consisting of any three consecutive lowercase characters ending with "un" *except* for the string "tun". Our tongue twister text includes "tun" as well as "hun" and "kun", so we can continue to use it as our model text, but we'll define the RegEx matching pattern to include all other matching strings as well. One way to do this is to define the pattern where the set of characters to match the first character position are all indicated explicitly in the set, as in the following:

```
[abcdefghijklmnopqrsuvwxyz]un
```

Notice that "t" doesn't appear in the character set illustrated by this example. Whereas this matching pattern will identify the occurrences of "hun" and "kun" in our tongue twister, it also would match strings "bun", "fun", "gun", "nun", "pun", and so on in any other target text that might contain these strings.

Although it satisfies the requirements we've specified, what programming challenges might arise with creating such a long pattern? Some would point to the number of characters the programmer would be required to type accurately to define the set of characters to match the first character position, arguing that reducing keystrokes during development saves time. Yes, whereas this is true, perhaps a much more insidious challenge is the fact that during subsequent maintenance, it's easy to miss that this set of characters contains only 25 of the possible 26 lowercase characters, and it would take some careful scrutiny to reach the conclusion that "t" is the only lowercase character not included among them. Fortunately, there is a better way.

Anytime you encounter a situation in which a consecutive series of characters needs to be specified for a set, you can use a shorthand method by indicating the lower and upper bounding characters of the series separated by a hyphen. Accordingly, the series of lowercase alphabetic characters excluding "t" can be specified using the following set:

```
[a-su-z]
```

Here, the set of characters is defined as a series of lowercase alphabetic characters from `a` (lower bounding character) through `s` (upper bounding character) and from `u` (lower bounding character) through `z` (upper bounding character), using two series of characters to identify the entire set. Although we saved some time during development with the time it takes to type this string instead of the entire alphabet, it now becomes far easier during maintenance to see that this set represents all the alphabetic characters with the exception of "t", which, had it not been excluded from the set, would have enabled us simply to define the set with the single "a-z" range. Combining this set into a RegEx pattern to represent the first character position of a three-character string ending with "un" gives the following:

```
[a-su-z]un
```

Notice the use of the hyphen to denote a range of characters. The hyphen is a character normally representing itself in the matching text, but it takes on a different role when it appears between two characters included in a character set, as illustrated in the preceding example. It's regarded as a RegEx language syntax character *only* when it appears in a square bracket-bounded character set between two characters where the first character is considered lower than the second character in accordance with the natural collating sequence of the characters. Unless it satisfies this unique role, it's regarded simply as a hyphen to be found in the text. Indeed, the character set specified for the preceding pattern could include a hyphen to represent a character to be matched along with the two ranges of characters the other hyphens help to define, as in the following:

```
[a-su-z-]un
```

As illustrated, we simply have followed the `z` in the character set with a hyphen. Accordingly, the first two hyphens act as range indicators because each one appears between two characters representing an ascending collating sequence, but the third hyphen simply represents itself because it doesn't appear between such characters. In other words,

this RegEx would not find "tun", but would find "hun", "kun", and also "-un".

This format for specifying consecutive character ranges using a hyphen separating the low value and high value of the range is applicable to these series of characters:

- Uppercase alphabetic characters (A–Z)
- Lowercase alphabetic characters (a–z)
- Digit characters (0–9)

Exercise 2

Define a RegEx matching pattern using a single range to specify the first character position that will identify all occurrences of "hun" and "kun" but not of "tun" in the target text. Test this pattern using a RegEx tester with the tongue twister as text.

Answers

Pattern with narrowest first character range is `[h-k]un`.

Pattern with widest first character range is `[a-s]un`.

5.4 Specifying String Sets within Regular Expression Patterns

If you wanted to find those strings where the letters "k" or "t" but not the letter "h" precedes the letter "u" in the tongue twister text, one way is to define a RegEx using a character set with "k" and "t" to represent the first character position, followed by the letter "u" as the second character, as in the following:

```
[kt]u
```

We'll find that this RegEx pattern locates eight matching strings: three for the "sku" in *skunk*, two for the "stu" in *stunk*, and three for the "stu" in *stump*. In each of these cases, the located string is preceded by the letter

"s", so the following RegEx pattern also identifies the same matching locations:

```
s[kt]u
```

Let's expand the tongue twister so the pattern `[kt]u` will find something the pattern `s[kt]u` won't find, with changes highlighted with bold:

A **young spunky** skunk **slunk up**on a stump **of a mature tree trunk. Naturally, t**he skunk thunk the stump stunk and the stump thunk the skunk stunk.

We should find now that pattern `[kt]u` will locate 10 matches, with the 2 additional matches being found in the additional words *mature* and *Naturally*. Indeed, we should observe that the letter `u` is preceded by the following sets of two-character values in those cases where it's surrounded by other alphabetic characters:

- yo (young)
- sp (spunky)
- sk (skunk)
- sl (slunk)
- at (mature; Naturally)
- st (stump; stunk)
- tr (trunk)

Confirm this list using a simple RegEx tester with the pattern `u` against the expanded tongue twister to show all the locations where "u" appears.

This expanded text offers some new possibilities for string searching. Suppose we want to find those locations where the letter "u" is preceded either by "sp" or "at" but by none of the other two-character combinations in the preceding list. The use of the pattern `s[kt]u` no longer suffices because it *will* find the string "stu" in the words *stump* and *stunk* and *won't* find the string "atu" in the words *mature* and *Naturally*.

RegExes facilitate this by providing a way to identify a *set of strings*, any one of which can be used to satisfy the matching pattern. Such a set of strings is specified between matching parentheses in the RegEx pattern, with each string separated from the next by a vertical bar:

```
(sp|at)u
```

The vertical bar separating the strings effectively acts as a Boolean OR operator, indicating that only one of the multiple strings composing the set will be used to identify a matching pattern. This illustrates the next basic principle of matching patterns used with RegExes.

Basic Principle

Bounding parenthesis with embedded vertical bar characters specified in a RegEx pattern indicates a set of strings, any one of which can match at that position in the target string.

The bounding parenthesis and vertical bar aren't regarded as characters to be matched in the text but as RegEx language syntax, enclosing those characters that are to be regarded as matching strings in the text. The left and right parenthesis and vertical bar are three members of the set of special characters composing the RegEx language syntax, none of which, unless otherwise qualified, represents itself in the text.

Notice the first character of this string is a left parenthesis, which initiates the specification of a set of interchangeable strings. The next two characters `sp` identify the first of the strings in the set. The next character is a vertical bar separating the first string from the next string. The next two characters `at` identify the second of the strings in the set. The next character is a right parenthesis, closing the set of interchangeable strings. This is followed by the `u` to follow either of the two strings specified in the string set.

This pattern effectively intends to match any three-character string starting with "sp" or "at" and ending with "u". It will locate the strings "spu" and "atu" appearing in the target text, but it will exclude "you", "sku",

"slu", "stu", and "tru". It's regarded as a three-character string-matching pattern because each of the strings in the set has a length of two characters, while the u character, which must be found following one of the strings in the set for the pattern to match, is only one character in length.

With parentheses enclosing a set of strings where each string is separated from adjacent strings by a vertical bar, this becomes our next example of the "unless otherwise qualified" phrase used previously. The bounding parentheses and the vertical bar qualify the characters appearing between them as interchangeable strings, which means the contiguous string of characters from the left through to the right parenthesis is not to be regarded as a single string to be matched in the text.

Try this pattern now with a RegEx tester. You should find it identifies three matches—the string "spu" is highlighted once and the string "atu" is highlighted twice.

As with the characters composing the character sets we've already covered, strings composing string sets also may be specified in any sequence. That is, the set "(ab|cd|ef)" is equivalent to the set "(cd|ef|ab)", which also is equivalent to the set "(ef|ab|cd)". Also, with string sets, there is no requirement for each string to be of equal length as the other strings in the same set. The following RegEx pattern, which uses strings of unequal length as members of the same set of strings, will find the words *bough*, *enough*, *tough*, *though*, *through*, and *thorough*, but it won't find the words *borough*, *cough*, *dough*, or *rough*.

```
(b|en|t|th|thr|thor)ough
```

Later, you'll see that paired parentheses play two other roles in the RegEx language syntax, but when accompanied by the vertical bar, as just illustrated, it specifies a set of strings.

Exercise 3

Define a RegEx matching pattern using a set of strings to indicate matches for the strings "you", "sku", "slu", "stu", and "tru" but omitting matches for

both strings "spu" and "atu". Test this pattern using a RegEx tester with the expanded tongue twister as text.

Answers

Pattern with fewest characters is `(yo|sk|sl|st|tr)u`.

Pattern with most characters is `(you|sku|slu|stu|tru)`.

5.5 Special Characters with Special Qualities

As stated in Section 5.1, unless otherwise qualified, all alphabetic characters and digit characters defined within the RegEx match themselves in the target text. This also applies to most but not all special characters, which are neither alphabetic nor numeric.

If you were to start at the left of the first row of keys on a 101-key standard US layout computer keyboard and move to the right across the keys, and upon reaching the end of a row of keys continue from the left of the next row down until reaching the end of the lowest row of keys, you would find there are 32 special characters available. For now, let's explore some of the capabilities afforded by the use of special characters in RegEx patterns.

Space

The first one of these is the space character. Some might argue whether this is to be considered a special character, but it's regarded here as a special character only inasmuch as it's neither alphabetic nor numeric. With regard to RegExes, it's not so special—that is, it matches itself in the target text.

Try this: Use the original tongue twister text from earlier as the target text with a RegEx tester, and then type a single space into the RegEx field. In RegexPal, Rubular, and RegexStorm, you should notice that all 19 spaces

between the words are highlighted, including each of the two consecutive spaces separating the two sentences from each other.

> **Note**
>
> RegexToy neither observes trailing spaces specified in the REGEX pattern text box nor highlights pattern-matching spaces appearing in the MATCHES box. See Section 5.8 for patches to be applied to a copy of this program to enable both these capabilities and more.

Square Brackets

The next two special characters are the left and right square brackets used to enclose a set of interchangeable characters that can occupy a specific character position. These special characters aren't regarded as matching characters for the target string but as RegEx language syntax. Accordingly, they control how other characters are to be interpreted when found in the target text.

Parentheses and Vertical Bar

The next three are the parentheses, both left and right, and the vertical bar used to enclose a set of interchangeable strings that can occupy a specific string position. These special characters aren't regarded as matching characters for the target string but also as RegEx language syntax. Accordingly, they control how other strings are to be interpreted when found in the target text.

Dot

The next one of these is the dot character. Recall our use of the following RegEx pattern:

```
[hkt]un
```

This indicates matches for the three-character strings "hun", "kun", and "tun" within the tongue twister text. Now, if we place the character "s" at the front of this pattern as follows

```
s[hkt]un
```

we restrict it to matching any four-character strings that start with "s", have "h", "k", or "t" in the second character position, and end with "un". Try this now, and you should find it locates three strings of "skun" and two strings of "stun". If, instead, the task requires simply that we find strings in the text where "s" is separated from "un" by any character, we can transform this into the following using a second position indicator covering the range of all lowercase characters:

```
s[a-z]un
```

This would work for our tongue twister, but it would not find those strings where "s" and "un" were separated by an uppercase character, a digit, a space, or any of the other special characters. An easy way to accommodate this is to specify a wildcard character in the second character position that will match any character. The dot *is* the wildcard character of RegExes; a dot appearing in a RegEx pattern will match any character at that character position. Accordingly, the following RegEx pattern accommodates this nicely:

```
s.un
```

Indeed, the dot character is the only special character that neither matches itself nor is regarded as part of the RegEx language syntax, such as the square brackets, parentheses, and vertical bar we've already covered.

It should be noted some implementations of RegEx interpreters don't recognize the newline character as one included in the set of characters a dot will match, so you need to be aware of the environment in which this character will be used, lest you find it not matching strings you intend it to match or matching strings you intend it not to match.

Escaping

Now we know a dot will match any character, and we know that left and right square brackets, left and right parentheses, and vertical bars do not indicate characters to be matched in the text but serve as an enclosure to

represent a set of interchangeable characters. Suppose you encounter a requirement where you need to find a dot, a square bracket, a parenthesis, or vertical bar in the text itself. The RegEx language syntax enables this by preceding a character normally not representing itself in the text with a backslash character (\), which denotes to the RegEx interpreter to regard the character following the backslash as a character to be recognized in the text. This is known as *escaping*—the backslash is the escape character, casting upon the subsequent character a quality of interpretation different than it would normally have.

Try this with the RegEx tester using the earlier tongue twister text and the RegEx pattern:

```
\.
```

This pair of characters changes the way the dot character is recognized, which normally would be interpreted as matching "any single character." Instead, the escape character preceding the dot indicates that the dot is to be regarded as a character to be located in the target text.

You should find that it highlights the periods at the end of each sentence. Indeed, the use of the escape character is the first time we've encountered two consecutive pattern characters to denote a single character to be matched in the target text. Remove the leading backslash, and every character position of the target text, including the periods terminating each of the sentences, is identified as matching the "any single character" pattern. Try it and see for yourself.

When the escape character precedes a character that otherwise would be interpreted as part of the RegEx language syntax, it causes its syntactical designation to be overridden and instead is interpreted as a character to be found in the text. The escape character even applies to the backslash itself; that is, if you require a RegEx pattern to include the backslash character as a character to be found in the target text, then specify this as a pair of backslash characters. The first backslash indicates that the next character should be regarded as the one to be found in the text, and the second one represents the backslash character to be found in the text.

Shortcuts for Specifying Character Sets

The escape character also has an effect on characters that are *not* otherwise interpreted as part of the RegEx language syntax. These characters, normally matching themselves in the text, take on a different role when preceded by the escape character, which now causes them to be interpreted as RegEx language syntax elements. These elements substitute for ranges of characters and have come to be known as *range shortcuts*. There are seven of these types of range shortcuts.

The four in Table 2, each designated by a lowercase alphabetic character following the escape character, represent range shortcuts denoting a set of characters to be recognized in the text.

Range Shortcut	Description
`\d`	This pair of characters indicates matches for any digit character. It represents the equivalent of the following range specification: [0-9].
`\w`	This pair of characters indicates matches for any word characters, which includes all letters, numbers, and also the underscore character. It represents the equivalent of the following range specification: [A-Za-z0-9_].
`\s`	This pair of characters indicates matches for any whitespace character. A whitespace character typically refers to any nonprintable character, which includes spaces, tab characters, line feed characters, and so on.
`\b`	This pair of characters indicates matches for a word boundary. A word boundary is denoted either by a leading or trailing whitespace or by a trailing period, comma, semicolon, and so on. Try this with the tongue twister and the RegEx `k\b`. You should find that the letter "k" is found at the end of the words *thunk*, *stunk*, and *skunk*. Specifically, it *doesn't* find the "k" following the "s" in the word *skunk*, and it does find the trailing "k" in the word *skunk* ending the second sentence with a period.

Table 2 Range Shortcuts Denoting a Set of Characters to Be Recoginized in the Text

The remaining three range shortcuts (Table 3), each designated by an uppercase alphabetic character following the escape character, denote the antithesis of a set of characters to be recognized in the text.

Range Shortcut	Description
`\D`	This pair of characters indicates matches for any *nondigit* character. It includes the set of all characters that aren't included in the range shortcut `\d`.
`\W`	This pair of characters indicates matches for any *nonword* characters. It includes the set of all characters that aren't included in the range shortcut `\w`.
`\S`	This pair of characters indicates matches for any *non-whitespace* character. It includes the set of all characters that aren't included in the range shortcut `\s`.

Table 3 Range Shortcuts Denoting the Antithesis of a Set of Characters to Be Recognized in the Text

Other Special Characters

Additional special characters are included in the set composing the RegEx language syntax. Among them are left and right braces, dollar sign, caret, asterisk, question mark, and plus sign. We'll defer explanation of these until they can be shown in context in the next section.

5.6 Continuous Pattern Matching and its Constraints

Return to one or all of the RegEx testers using the original tongue twister text and a RegEx pattern composed of a single dot. Notice how this RegEx identifies each character position as a match for the pattern it represents—the single occurrence of any character.

Change this single character to a pair of dots and the pattern to be matched now becomes 2 characters in length. The utility responds accordingly by highlighting alternating pairs of characters. However, notice that now the period at the end of the final sentence no longer has

any highlighting because the tongue twister contains 93 characters, an odd number, and we've indicated we want to match *pairs* of characters. Accordingly, the string runs out before the final period can be used with a character to match the pair of characters pattern. Even so, we see that the RegEx is matched 46 times, which is the number of times this RegEx pattern can be matched in the text before the text runs out of 2-character pairs, with the string highlighting alternating with each pair.

Change this again to now indicate a set of three consecutive dots. The highlighting is changed again to include a set of 3 characters before alternating to the other color. With this pattern, the entire tongue twister is highlighted again because its total number of characters, 93, is evenly divisible by 3, and we're left with 31 matches of this 3-character string.

Include a fourth dot in the pattern, and see how the highlighting changes to indicate the length of the tongue twister matching the pattern. Continue this process of adding a single dot to the RegEx pattern. The results should be as follows:

- 4 dots: 23 matches with 1 remaining unmatchable character
- 5 dots: 18 matches with 3 remaining unmatchable characters
- 6 dots: 15 matches with 3 remaining unmatchable characters
- 7 dots: 13 matches with 2 remaining unmatchable characters
- 8 dots: 11 matches with 5 remaining unmatchable characters
- 9 dots: 10 matches with 3 remaining unmatchable characters

Continue with this process, and you'll see the number of remaining unmatched characters continue to fluctuate with each new dot added to the pattern:

- 10 dots: 9 matches with 3 remaining unmatchable characters
- 11 dots: 8 matches with 5 remaining unmatchable characters
- 12 dots: 7 matches with 9 remaining unmatchable characters
- 13 dots: 7 matches with 2 remaining unmatchable characters

- 14 dots: 6 matches with 9 remaining unmatchable characters
- 15 dots: 6 matches with 3 remaining unmatchable characters
- 16 dots: 5 matches with 13 remaining unmatchable characters
- 17 dots: 5 matches with 8 remaining unmatchable characters
- 18 dots: 5 matches with 3 remaining unmatchable characters
- 19 dots: 4 matches with 17 remaining unmatchable characters
- 20 dots: 4 matches with 13 remaining unmatchable characters
- 21 dots: 4 matches with 9 remaining unmatchable characters
- 22 dots: 4 matches with 5 remaining unmatchable characters
- 23 dots: 4 matches with 1 remaining unmatchable character
- 24 dots: 3 matches with 21 remaining unmatchable characters

Eventually the series of dots representing the matching pattern reaches a length of 46. At this point, the pattern is matched twice in the text with a single remaining unmatched character. Increase the length of the pattern with yet another dot, and you'll see the pattern is matched only once in the text, leaving behind 46 unmatched characters trailing the matching first 47 characters. Adding a single dot character to the pattern from this point onward won't change the number of matches until the length of the pattern becomes 94, exceeding the length of the text itself, at which point the pattern no longer is matched in the text.

5.7 Text Replacement

RegExes not only can find matching strings but also can apply changes to those strings, which is known as *text replacement*. To use this feature, you simply provide the string of text to be used to replace the corresponding matched text. An example of using RegExes with text replacement is within a spell checker, where a misspelled word, such as *perfrom* is converted to its correct spelling *perform*. You'll see an example of text replacement in Section 6.1.

Let's try this with some tongue twister text, in which the word *thunk* appears as a nonstandard past participle of the word *think*, for which the standard past participle is the word *thought*. A simple RegEx text replacement to correct *thunk* with *thought* is one where the word *thunk* represents the string to be found and *thought* represents the string to replace it.

> **Tip**
>
> Use the RegexToy or RegexStorm testers for this because both facilitate text replacement. With RegexToy, the replacement input text box appears beneath the pattern input text box. With RegexStorm, the REPLACEMENT text box doesn't appear by default but needs to be opened explicitly, by clicking the REPLACE MATCHES WITH checkbox.

Provide the tongue twister text in the text box, enter "thought" as the replacement text, and enter "thunk" as the pattern to be matched. RegexToy shows the result immediately in the MATCHES section. With RegexStorm, the result is available in the text box associated with the CONTEXT tab. In both cases, the word *thunk* has been replaced with the word *thought*.

It may seem simple enough with this example, but it can get more complicated as the search requirement becomes more specific. For instance, your search pattern may need to indicate that the string "thunk" must be preceded by any whitespace character in order to ignore the string "thunk" appearing in the word *kerthunk*, a variant of the word *kerplunk*, and that it should ignore the string "thunk" appearing in the word *thunking*, which could be interpreted as a simple misspelling of the word *thinking*.

5.8 Patches for RegexToy

This section includes ABAP code enhancements to the RegexToy program. Make a copy of Report DEMO_REGEX_TOY *and* all of its corresponding objects, change the value of GUI title TITLE_100 to "Enhanced Regex Toy", and apply each of the following changes.

Patch 1

The REGEX field doesn't recognize trailing spaces specified in the pattern. Test this with the tongue twister as text and a two-character pattern composed of "k" with a following space. This pattern should only find "k" at the end of words but finds all "k" characters. This patch causes trailing spaces to be recognized when the cursor is located in the REGEX field at the point following the last trailing space to be observed. Place the series of lines shown in Listing 4 in the method `main` ahead of the line clearing `sub1` through `sub6` (first and last lines, shown in italic, already exist):

```
    ENDIF.
    " -------------------------------------
    " DEMO_REGEX_TOY enhancement #1
    " Enable explicit trailing spaces in pattern:
    data       : trailing_space type int4
               , pattern_length type int4
               .
    if sy-curow eq 02.
      trailing_space            = sy-cucol - 16.
      pattern_length            = strlen( pattern ).
      subtract pattern_length from trailing_space.
      while trailing_space gt 00.
        concatenate pattern
                    space
               into pattern
                      respecting blanks.
        subtract 01 from trailing_space.
      endwhile.
    endif.
    " -------------------------------------
    CLEAR: sub1, sub2, sub3, sub4, sub5, sub6.
```

Listing 4 RegexToy Patch #1

Patch 2

With the previous patch, only instances of "k" at the end of words are identified as matches, but using the same test as with the previous patch, you still can't see that the trailing space is part of the matching string. This patch (Listing 5) replaces the foreground text highlighting with alternating background text highlighting. Place this series of lines in the method `display` following the `REPLACE ALL OCCURRENCES OF` statement (first and last lines, shown in italic, already exist):

```
'@@tgr@@' IN TABLE result_it WITH '</b></font>'.
" ------------------------------------
" DEMO_REGEX_TOY enhancement #2
" Change highlighting from single foreground color to
" alternating background colors:
constants    : color_red       type string
                 value 'color="#FF0000"'
             , background_color_green
                               type string
                 value 'style=background-color:"#B6D840"'
             , background_color_blue
                               type string
                 value 'style=background-color:"#98CCE8"'
             .
data         : color_toggle    type int4
             , replacement_color
                               type string
             .
clear sy-subrc.
while sy-subrc is initial.
  color_toggle                = 01 - color_toggle.
  if color_toggle eq 00.
    replacement_color         = background_color_blue.
  else.
    replacement_color         = background_color_green.
  endif.
  replace first occurrence of color_red in table result_it
     with replacement_color.
endwhile.
" ------------------------------------
CLEAR result_html.
```

Listing 5 RegexToy Patch #2

With this patch, the space following the "k" at the end of words is now included in the matching string, and each match is shown with alternating green and blue background text highlighting.

Patch 3

Now search the tongue twister for the string "the skunk", ignoring case, and you'll see that only the first of the two occurrences is identified as a match. Strings straddling implicit line breaks inserted for window processing are excluded from match results, causing the second occurrence of "the skunk" to be missed. This patch preserves line integrity for matching

purposes. The first change is to be applied to the method `init`; subsequent changes are to be applied to the method `main`.

1. Change line

   ```
   wordwrap_to_linebreak_mode = cl_gui_textedit=>true.
   ```

   to

   ```
   wordwrap_to_linebreak_mode = cl_gui_textedit=>false.
   ```

2. Comment out the `SPLIT` statement following the `cl_gui_cfw=>flush( )` statement.
3. Change `IN` clauses of all `FIND` and `REPLACE` statements (six in all) in the `TRY-ENDTRY` block from

   ```
   ... IN TABLE result_it ...
   ```

   to

   ```
   ... IN text_wa ...
   ```

4. Copy the commented `SPLIT` statement (from the preceding) and place it ahead of the `display( )` statement as an active line.

Patch 4

The previous patch enables both occurrences of "the skunk" string to be found, but now the content appearing in the MATCHES result window extends beyond the designated default window width and requires horizontal scrolling to see it all. This patch restores text wrap to content appearing in the MATCHES window using line breaks at approximate default window width. Place the lines in Listing 6 in the method `display` ahead of the `CONCATENATE LINES OF` statement (first and last lines, shown in italic, already exist):

```
APPEND '<html><body><font face="Arial monospaced for ...
" ------------------------------------
" DEMO_REGEX_TOY enhancement #4
" Format result text using approximate window width
" specified by line_len:
constants    : html_space     type string    value ' '
             , html_special_character_start
```

```
                      type string    value '&'
              , html_special_character_end
                      type string    value ';'
              , html_format_start
                      type string    value '<'
              , html_format_end
                      type string    value '>'
              .
data          : visible_text_length
                      type int4
              , html_last_space_length
                      type int4
              , cause_line_break
                      type flag
                , html_string    type string
                , html_excess    type string
                , html_stack     type standard table
                                   of string
                .
loop at result_it
   into result_wa.
  clear: html_string, visible_text_length.
  while strlen( result_wa ) gt 00.
    case result_wa+00(01).
      when html_format_start.
        while result_wa+00(01) ne html_format_end.
          concatenate html_string
                      result_wa+00(01)
                 into html_string.
          shift result_wa left by 01 places.
        endwhile.
      when html_special_character_start.
        if strlen( result_wa ) ge 06.
          if result_wa+00(06) eq html_space.
            if visible_text_length gt line_len.
              cause_line_break  = abap_true.
            endif.
  endif.
endif.
while result_wa+00(01) ne
      html_special_character_end.
  concatenate html_string
              result_wa+00(01)
         into html_string.
  shift result_wa left by 01 places.
endwhile.
add 01 to visible_text_length.
if cause_line_break eq abap_false.
```

```
      html_last_space_length
                          = strlen( html_string ) + 01.
    endif.
  when others.
    add 01 to visible_text_length.
endcase.
concatenate html_string
            result_wa+00(01)
       into html_string.
shift result_wa left by 01 places.
if cause_line_break eq abap_true.
  clear html_excess.
  if html_last_space_length gt 00.
    html_excess =
            substring( val = html_string
                       off = html_last_space_length ).
      html_string =
            substring( val = html_string
                       len = html_last_space_length ).
    endif.
    append html_string
        to html_stack.
    clear: html_string, html_last_space_length.
    visible_text_length     = strlen( html_string ).
    cause_line_break        = abap_false.
    if strlen( html_excess ) gt 00.
      concatenate html_excess
                  result_wa
            into result_wa.
    endif.
  endif.
endwhile.
append html_string
    to html_stack.
endloop.
" -----------------------------------
CONCATENATE LINES OF result_it INTO result_wa ...
```

Listing 6 RegexToy Patch #4

Then change the `CONCATENATE LINES OF` statement from

```
CONCATENATE LINES OF result_it INTO result_wa ...
```

to

```
CONCATENATE LINES OF html_stack INTO result_wa ...
```

Patch 5

With the previous patch, the width of content appearing in the MATCHES window is restored to the approximate default window width, but this also causes explicit line breaks to be ignored. Enter the "A skunk sat on a stump." text with line breaks between each word, and specify the RegEx pattern any-character (a single dot); although all characters correctly show matching the any-character pattern, the explicit line breaks have been lost, so all words appear on the same line in the MATCHES window.

The patch shown in Listing 7 preserves explicit line breaks. Place these lines in the method `main` after the `TRY` statement (first and last lines, shown in italic, already exist):

```
TRY.
" ------------------------------------
" DEMO_REGEX_TOY enhancement #5
" Convert two-character carriage-return/line-feed into
" single-character form feed to prevent split during
" search for any character:
replace all occurrences
     of cl_abap_char_utilities=>cr_lf in text_wa
   with cl_abap_char_utilities=>form_feed.
" ------------------------------------
IF nocase = 'X'.
```

Listing 7 RegexToy Patch #5

Also, change the `AT` clause of the `SPLIT` statement, following the `ENDTRY` statement, from

```
... AT cl_abap_char_utilities=>cr_lf ...
```

to

```
... AT cl_abap_char_utilities=>form_feed ...
```

6 Regular Expressions in Practical Use

Now it's time to put the RegEx pattern matching and replacing features to practical use identifying and adjusting some common string formats in

ABAP. With each of these examples, the requirements for locating the information is explained, an example string of text representative of the format is presented for use, and we walk through the process of building a RegEx pattern, step by step, to satisfy the requirement. Upon completing this section, you'll understand not only how to build a RegEx pattern but also why it works.

Imagine our company enables its corporate customers to update their own customer profile information retained in the SAP system. Lately, the customers have been complaining that the current process allows too many mistakes entering credit card numbers and email addresses, and they would like to see more flexibility in providing these values along with more relevant validity processing associated with them. We've been asked to provide an enhancement to this process whereby a popup screen is presented through an ABAP user exit on which the customer can provide a credit card number and email address to be used with subsequent sales transactions.

We've been advised that both the credit card number and email address fields on the new popup screen could be made more flexible by subjecting these values specified by the user to simple `FIND` statements using RegExes for determining whether these specified profile values are in compliance with credit card numbers and email addresses, respectively. Writing the ABAP code to validate these fields with respective `FIND` statements is mostly familiar ground, but the corresponding RegExes to be used with each one are the subjects of the next two sections:

- Validating a credit card number
- Validating an email address

Afterward, we'll address how to facilitate the following with RegExes in ABAP:

- Assessing password strength
- Identifying text file format

6.1 Validating Credit Card Numbers

Many people have become comfortable using the Internet to purchase goods and services. One common way for providing payment for these purchases is through the use of a credit card. Of course, although we always look for the secure web page indication before divulging the precious credit card number, some web pages don't make it easy for us to comply with their request to enter these digits.

Have you ever found yourself in this situation? The instructions on a website ask you to enter your credit card number precisely how it appears on your card. You type in four groups of four digits, each group separated by a space—only to be met with a terse message indicating that you've entered an invalid credit card number. After a moment of fearing that your card has expired or been cancelled, you realize that the website is simply one that requires the credit card numbers be entered without any spaces. Remove the spaces, and all is well. Afterward you might have wondered, "How hard can it be to allow the user to type the credit card number with intervening spaces?" After all, that is how it appears on the card and how most folks would be inclined to provide it verbally when shopping via phone or when typing it into a field on a web page. This is where RegExes can help.

You've already seen where the RegEx pattern syntax two-character string `\d` represents any digit character. Now let's see how to build a RegEx that will accommodate typing a credit card number with or without intervening spaces. For this experiment, we'll use the following bogus credit card number:

```
1234 2345 3456 4567
```

This is a good sample credit card number because it uses most of the digits and also has within itself the benefit of a recurring pattern—each group of four digits contains consecutive ascending digits, and each group starts with the next consecutive digit beyond the first digit of the preceding group. This will make it easier to identify each of the four groups even when the digit groups have no intervening spaces to provide this visual

clue. So, let's place this value into a RegEx tester as the text to be scanned for pattern compliance.

Next, let's specify our RegEx matching pattern using the "any digit" range shortcut:

```
\d
```

At this point, you should find that each digit of the credit card number is marked as complying with the pattern—perhaps with each one marked in alternating highlight colors—but the intervening spaces aren't marked this way. So far, all is good.

Next, we want to indicate that there should be four digits to a group. One way to do this is to indicate a matching pattern in which the "any digit" designation occurs four times in succession:

```
\d\d\d\d
```

Although not wrong, it's rather inelegant, and it would become difficult to determine how many of these occur in succession after a certain point. A better way is to use a *quantifier* indication to represent a pattern of exactly four digits in succession, which we can do by following the "any digit" indicator with matching braces enclosing the number "4":

```
\d{4}
```

The left and right brace characters also are considered part of the set of RegEx language syntax, and, unless otherwise qualified by an escape character, they don't represent characters to be matched in the target text. Matching braces actually enables specifying both a low and high limit to the number of consecutive occurrences of the preceding element. The general format is the following:

```
{m,n}
```

In this format, `m` represents the fewest occurrences required to match the pattern, `n` represents the most occurrences that can still match, and a comma separates the two values. Either value may be omitted, with a missing low limit indicating zero, and a missing high limit indicating unlimited. When both the low and high limit values are the same, we can

dispense with the comma altogether and simply indicate the single number representing the exact number of occurrences required to match the pattern, as we've done with our latest version of the pattern. Using braces like this is one of many ways the RegEx language syntax provides for indicating the *quantification* of preceding elements.

Now you should see that the highlighting has changed and that each group of four digits appears in a single highlighting color, with the highlighting color now alternating with each new group of four digits. We're getting closer, but right now, this simply represents a set of independent four-digit strings but not necessarily a valid credit card number. Let's confirm this by replacing the space separating the second and third group of digits with the alphabetic character "z":

```
1234 2345z3456 4567
```

As you can see, we still get the same highlighting as before, so we need to adjust our pattern to permit spaces between the group of four digits but no other characters. Let's indicate this by suffixing our pattern with the any whitespace character `\s`:

```
\d{4}\s
```

Now we see that only the strings "1234" and "3456" are highlighted. Indeed, these are the only occurrences in our now-invalid credit card number that have four consecutive digits followed by a space. So, let's replace the "z" with a space and restore our credit card number to the one that has four groups of four-digit strings separated by single spaces:

```
1234 2345 3456 4567
```

Now we should see that the strings "1234", "2345", and "3456" are highlighted in alternating colors, but that the string "4567" remains without highlighting because the string "4567" isn't followed by a space. But we don't want each credit card number entered necessarily to be followed by a space. We can fix this by making the space character optional by appending a quantification identifier to indicate it may occur zero times or one time:

```
\d{4}\s{0,1}
```

This quantification required another five characters in our RegEx pattern. The RegEx language syntax provides yet another method of quantification, using only a single character to indicate that a preceding element may appear zero times or one time. We do this with the question mark following the character position we want to be optional, which in our case is the any whitespace character. As with braces, the question mark character also is part of the RegEx language syntax and, unless otherwise qualified by an escape character, doesn't represent a character to be matched in the target text. Our RegEx pattern now becomes the following:

```
\d{4}\s?
```

The single character `?` has replaced the longer string `{0,1}` to qualify the any whitespace character, but where both are syntactically equivalent. Sure enough, we now have a pattern that finds matches with each of the four digit groups.

So, will this pattern still match our credit card number if we were to remove the spaces between the digit groups? Let's try this and see what happens:

```
1234234534564567
```

You should find that it still works correctly and identifies our string of digits as a valid credit card number. Indeed, this consecutive string of digits is the format so many websites require when entering a credit card number. Many of us encounter some difficulty determining whether we've entered the number correctly when we can't see the digit groupings.

So will this pattern continue to work if we were to place a space between the "1" and the "2" of the first digit group? Let's try it.

```
1 234234534564567
```

We see from the results that the pattern merely matches a different set of four consecutive digits and no longer four groups of them. The matching groups now are "2342", "3453", and "4564". Now what?

One way to resolve this is to indicate a requirement for the pattern to begin matching with the very first character. We indicate this by placing a caret character (^) at the start of the pattern:

```
^\d{4}\s?
```

Try this now. You should find that none of the string is highlighted because a match can't be found starting at the beginning of the string for four consecutive digits that may or may not be followed by a space character. The caret is yet another special character considered part of the set of RegEx language syntax and, unless otherwise qualified by an escape character, doesn't represent a character to be matched in the target text.

Remove the space between the "1" and the "2" in the text, and now we should see that only the first group of digits is highlighted, but we need to accommodate 16 digits. So let's change the pattern to allow for 16 digits instead of only 4:

```
^\d{16}\s?
```

Great! That works nicely and highlights the entire credit card number, indicating conformance with the pattern. At this point, if we were to randomly introduce alphabetic characters into the credit card number string, the highlighting immediately would be turned off, confirming for us that the invalid character causes the credit card number no longer to comply with the pattern. Try this and see for yourself.

The problem with this pattern is that while it prevents alphabetic characters placed among the digits from conforming to the pattern, this same conformance failure now also applies to our use of the intervening spaces between the digit groups:

```
1234 2345 3456 4567
```

This is the format we want to be able to use, but when we do, the string no longer conforms to the pattern, just as when any alphabetic character randomly is inserted into the string.

One way to overcome this is to return to our indication that exactly four digits are to be recognized consecutively before a potential space charac-

ter, but then to indicate that this pattern itself may occur exactly four times.

To do this, we need to use *grouping*. With grouping, we indicate that a series of characters is to be regarded as a group. A group is denoted by surrounding those pattern characters identifying the group with bounding parentheses. We've already seen the use of bounding parentheses with intervening vertical bars to establish sets of strings, so we know that the left and right parenthesis characters also are considered part of the set of RegEx language syntax and, unless otherwise qualified by an escape character, don't represent characters to be matched in the target text. In our case, we want the group to be the four-digit string with optional following space:

```
(\d{4}\s?)
```

Notice the bounding parentheses surrounding the pattern characters. We want this group to be repeated exactly four times. We can specify this by using the same technique to indicate that a digit must be repeated exactly four times, applying a quantification factor to the entire group:

```
(\d{4}\s?){4}
```

This illustrates one of the two remaining uses of parentheses with RegExes. Here we're applying a quantification factor to the entire group by following the closing right parenthesis with a quantification modifier—specifically, this group is to occur exactly four times. When we use this pattern with the credit card number, we find that the entire string is highlighted in one color, indicating that the entire string is being used to match the pattern.

Notice that we had dropped from this pattern the leading caret character to indicate starting at the first character position of the string. Accordingly, we could insert as many random alphabetic characters to the front of our credit card number, and it still will conform to our pattern. We certainly don't want this, so let's again use the leading caret to indicate that the pattern must be matched starting with the first character position:

```
^(\d{4}\s?){4}
```

Now those random alphabetic characters at the front of our credit card number result in no pattern matching, and removing them again brings our string into compliance with our credit card number pattern.

You'll notice that our credit card number pattern allows for a trailing space after the final digit. Let's see whether this really works by typing a space after the "7" in our credit card number. You should find that, indeed, it does work, and that the trailing space is now included in the highlighted sequence.

We now have a RegEx pattern that will identify a credit card number that may or may not have intervening spaces between the four four-digit groups and may or may not have a trailing space. We could remove and reinsert these spaces wherever they occur, one by one, and each time our credit card string will continue to match the pattern.

Because we're allowing a trailing space after the last four-digit group, suppose we also wanted to accommodate a leading space ahead of the first four-digit group.

One way to do this is to place the `\s?` string sequence, which represents zero or one occurrences of any whitespace character, between the leading caret character and the left parenthesis starting the group indication:

```
^\s?(\d{4}\s?){4}
```

With this pattern, a leading space appearing in the first character position is considered acceptable for matching the pattern. Try it and see for yourself.

Let's revisit those optional spaces one more time. With our current pattern, we're allowing only a single space between the four-digit groups and also at the start and end of the string. If we want to allow any number of consecutive spaces at these locations, how could we change the pattern to accommodate this?

We could change the quantification indication for the any whitespace character positions from expecting zero or one spaces to expecting zero or more spaces. This is achieved by replacing the quantification character following the any whitespace indicators (`\s`) from question mark to *asterisk* (*):

```
^\s*(\d{4}\s*){4}
```

The asterisk is yet another special character considered part of the set of RegEx language syntax and, unless otherwise qualified by an escape character, doesn't represent a character to be matched in the target text. The single asterisk is the equivalent of the longer quantification expression `{0,}`, which indicates a minimum of zero and a maximum left unspecified. With a pattern like this to check the validity of a credit card format, the poor website user who has difficulty distinguishing whether a credit card number composed solely of digits is correct can use intervening spaces to assist with the entry, and not otherwise be penalized with an ominous message about a credit card number being invalid.

However, there still is one thing we've overlooked. We've prevented a user from typing random alphabetic characters at the front of a potential credit card number value, but we haven't taken steps to prevent these from appearing after the value. Using our current RegEx pattern, if a user were to follow the credit card number with random alphabetic characters, it still would be considered a valid credit card number. We can't have that. So how can we prevent trailing characters as we did with leading characters?

The answer is to use the end-of-string counterpart to the start-of-string leading caret. Its counterpart is the dollar sign character (`$`), which requires the pattern to match with the last character of text. The dollar sign is yet another special character considered part of the set of RegEx language syntax and, unless otherwise qualified by an escape character, doesn't represent a character to be matched in the target text. Adjusting accordingly, our pattern now looks like this:

```
^\s*(\d{4}\s*){4}$
```

With this as the matching pattern, a credit card number followed by random alphabetic characters now would be disqualified as conforming with the credit card pattern.

> **Note**
>
> If you were to look at the code for RegexToy, you would find a MATCH radio button in the OPTIONS section. Although there is no online documentation available with the utility to describe how to use the tester itself, clicking the MATCH button causes a leading caret and trailing dollar sign to be prefixed and suffixed, respectively, to the RegEx pattern you provided. The behavior with the MATCH button is to ensure that the entire string exactly matches the pattern, with no preceding or succeeding characters permitted, even when your RegEx pattern doesn't explicitly indicate this.

Reformatting a Credit Card Value Using Replacement Text

We now have a RegEx pattern by which we can validate a credit card value, and it accommodates the user placing a random series of spaces preceding, between, and following the four groups of digits. However, there is no reason we would need to retain all those extraneous spaces in subsequent displays of the credit card value, for instance, when it's to appear on a screen where the user is asked to confirm this and other values entered. There are programming statements that could do this, and ABAP programmers might choose the `CONDENSE` statement to eliminate the series of consecutive spaces down to single spaces. Although this would do the trick, such adjustments are limited to what the statement enables us to do, and we might find ourselves with the need to perform some more robust formatting on some strings we find in text.

Instead, we can use the more advanced *text replacement* feature RegExes provide to achieve this, which relies on the use of group designations appearing in the matching pattern. Recall that groups are identified by parentheses surrounding the sequence of characters that are to be grouped. Until now, we had created groups of characters to enable applying a replication factor to the series, as we did with our credit card valida-

tion pattern when we appended the quantifier `{4}` to the parenthetical group enclosing the string `\d{4}\s*`.

RegExes can *capture* the text of a matching group, and the captured text can be used in a pattern to alter the matching text. Capturing matching text is the third of the three ways paired parentheses are used in the RegEx language syntax. If we need to capture one or more characters for later use but have no reason to define a set of strings separated by a vertical bar (one use for using paired parentheses) and also have no reason for assigning a replication factor to a group (the other use for using paired parentheses), then parentheses simply surround those portions of the RegEx pattern to be captured. For example, suppose we wanted not only to find but also to capture the strings "think", "thank", and "thunk" so each could be replaced with itself along with bounding HTML bold highlight starting and ending marks preceding and succeeding it, respectively, we simply would enclose the matching pattern in a group:

```
(th.nk)
```

It now qualifies as a captured group. It should be noted that both of the other uses of paired parentheses—for denoting a set of strings and for denoting a quantifier applicable to a group—also will be regarded as captured groups.

The syntax for referring to these captured groups simply is the dollar sign (`$`) followed by a digit indicating the corresponding number of the group. Groups are numbered starting with 1, which refers to the first parenthetical group found in the matching pattern, and increment consecutively for each subsequent parenthetical group appearing in the matching pattern. Indeed, $0 also is a valid capture group, but it differs from the others in that it doesn't require enclosing parentheses to identify the boundaries of the group; instead, it always refers to the entire string found to match the pattern, regardless of any grouping that also may accompany the pattern.

Let's try this: Use the RegexToy or RegexStorm tester for this because they facilitate replacement of text. Using our credit card pattern and bogus credit card number as RegEx and text to be scanned, respectively, type the

following string into the REPLACEMENT text box starting at the first character position:

```
$1
```

With RegexStorm, click on the TABLE tab appearing beneath the REPLACEMENT box. As you can see from Figure 6, it will show, among others, the columns for MATCHED STRING, REPLACEMENT, and $1.

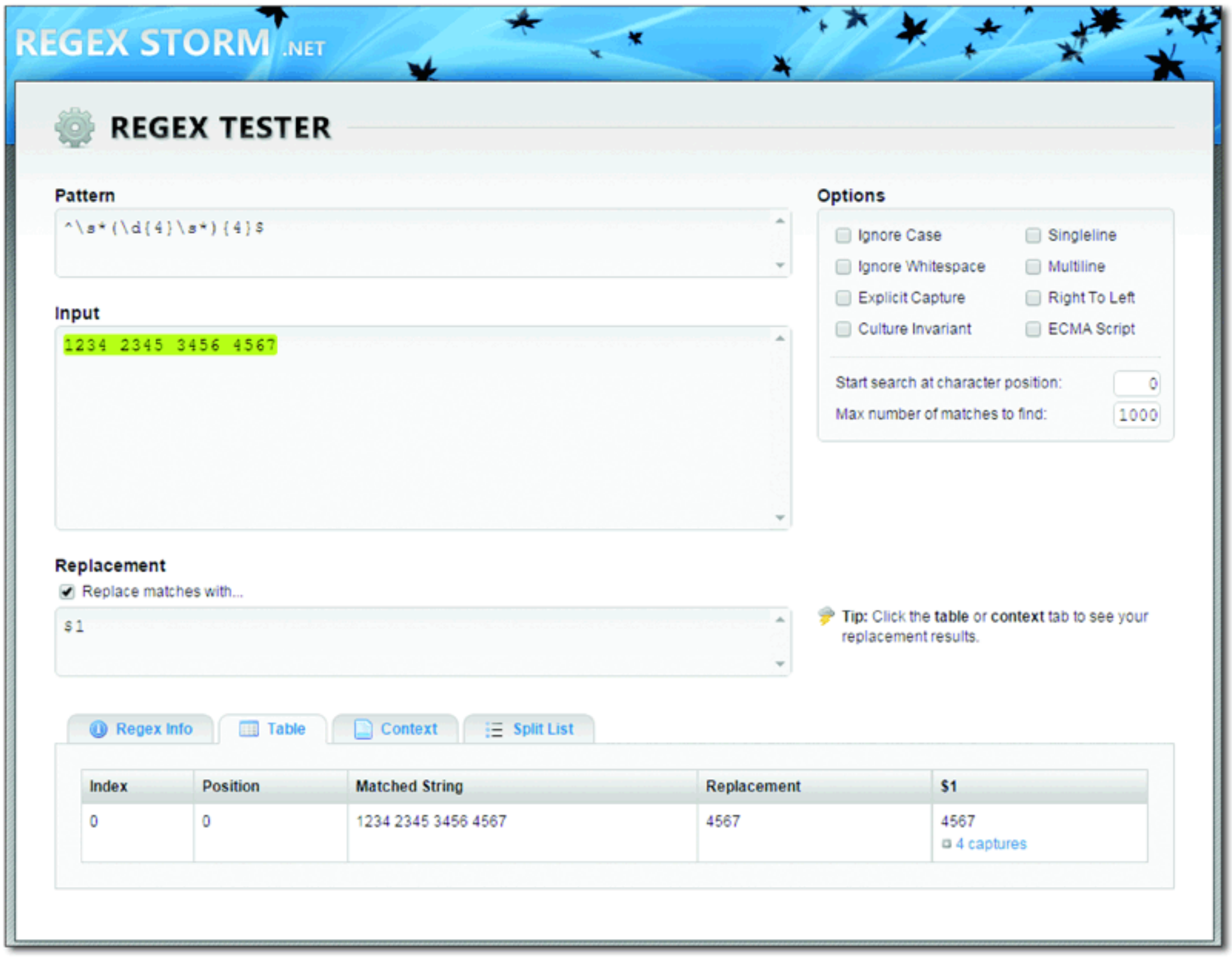

Figure 6 Replacement Box of RegexStorm

Notice in Figure 6 that both the REPLACEMENT and $1 columns show the value "4567". Notice also the $1 column shows 4 CAPTURES. Clicking on this shows each of the individually captured values. This illustrates a limitation with group captures—anytime the group has a quantifier indicat-

ing a replication factor, the captured value retained is from the last or only matching string.

This means that to capture each of the four individual four-digit series of numbers represented by our credit card value, we'll need to define a matching pattern that contains four individual groups. Accordingly, let's change our credit card matching pattern so that each group is explicitly defined:

```
^\s*(\d{4}\s*)(\d{4}\s*)(\d{4}\s*)(\d{4}\s*)$
```

Notice we've removed the quantifying replication factor `{4}` from the single parenthetical group we had in our former pattern and replaced it with three additional group indications, for a total of four. When we try this with our replacement string, we find that $1 now contains "1234", representing the capture value of the first group. Not only that, but we see that $2, $3, and $4 contain the values "2345", "3456", and "4567", respectively. So let's change the replacement string to indicate the use of all four capture groups:

```
$1$2$3$4
```

Upon trying this, we now find that our full credit card value will appear in the CONTEXT text box. Yes, this is more like it! So, providing we place spaces only between the digits of the four-digit groups, and not within them, we find our pattern is matched and the CONTEXT text box reflects the replaced values via the group designations.

Adjusting the Replacement Pattern

Upon further inspection, we find this to be somewhat incorrect. Yes, the replacement value contains the valid credit card number, but it also includes the spaces between them that we were trying to eliminate using the captured groups.

To figure out this problem, notice that each of the four individual capture groups represented by the four parenthetical segments includes the indication for zero or more spaces. Therefore, each of the capture groups is

capturing the trailing spaces along with the corresponding digits. We can change this by moving the space indication from inside to outside the capture group, as shown here. The first line shows the matching pattern with the four capture groups that incorrectly include capturing the spaces, and the second line shows the corrected version that will leave the spaces out of the captured groups, with the differences between them shown highlighted in bold:

```
^\s*(\d{4}\s*)(\d{4}\s*)(\d{4}\s*)(\d{4}\s*)$
^\s*(\d{4})\s*(\d{4})\s*(\d{4})\s*(\d{4})\s*$
```

Now within our credit card number, we can place spaces anywhere they still would permit matching the pattern, and we always see that the replacement string has no leading, trailing, or embedded spaces. This still may not be the easiest to read, so let's change this a bit more by specifying a replacement string that puts exactly one space between each of the groups:

```
$1 $2 $3 $4
```

Clearly, that is much better. Now we have a RegEx pattern for accepting credit card numbers where the user may enter the number with or without spaces at those locations where spaces are permissible, and the RegEx causes the value to be replaced with a nicely formatted value to be used in subsequent confirmation displays.

6.2 Validating Email Addresses

Continuing with the user exit for our corporate customer profile, we've been told by our business analyst that a valid email address is defined by the following segments of strings:

- A string composed of one or more alphabetic, digit, and underscore characters
- Zero or more strings composed of the following: a single dot character followed by a string of one or more alphabetic, digit and underscore characters

- One at-sign (@) character
- One or more strings composed of the following: a string of one or more alphabetic, digit, and underscore characters followed by a single dot character
- A string composed of at least two and at most four alphabetic, digit, and underscore characters

Build the Pattern

Now, let's build the RegEx pattern to validate email addresses one segment at a time and then concatenate all of the segments into one pattern:

1. **A string composed of one or more alphabetic, digit, and underscore characters**
   We can denote this with the following RegEx fragment:

   ```
   \w+
   ```

   The backslash acts as the escape character indicating to regard the subsequent character differently from its normal behavior in a RegEx pattern. The `w`, having been escaped by the preceding character, is shorthand denoting that any alphabetic, digit, or underscore character is valid in this position. The plus sign is a quantifier for the `w` indicating that the characters conforming to the escaped `w` may appear one or more times in succession.

2. **Zero or more strings composed of the following: a single dot character followed by a string of one or more alphabetic, digit, and underscore characters**
   We can denote this with the following RegEx fragment:

   ```
   (\.\w+)*
   ```

   The paired parentheses denote a group containing a string, and the trailing asterisk is a group quantifier indicating that the group may occur zero or more times. Within the group, the two backslash characters each act as escape characters indicating to regard their respective subsequent character differently from its normal behavior in a RegEx

pattern. The dot following the first escape character denotes that an actual dot character may appear at this location in the string. The `w` following the second escape character is shorthand denoting that any alphabetic, digit, or underscore character is valid in this position, and its trailing plus sign is a quantifier for the `w` indicating that the characters conforming to the escaped `w` may appear one or more times in succession.

3. **One at-sign (@) character**
   We can denote this with the following RegEx fragment:

   ```
   @
   ```

   The at-sign simply represents itself in the pattern.

4. **One or more strings composed of the following: a string of one or more alphabetic, digit, and underscore characters followed by a single dot character**
   We can denote this with the following RegEx fragment:

   ```
   (\w+\.)+
   ```

   The paired parentheses denote a group containing a string, and the trailing plus sign is a group quantifier indicating that the group may occur one or more times. Within the group, the two backslash characters each act as escape characters indicating to regard their respective subsequent character differently from its normal behavior in a RegEx pattern. The `w` following the first escape character is shorthand denoting that any alphabetic, digit, or underscore character is valid in this position, and its trailing plus sign is a quantifier for the `w` indicating that the characters conforming to the escaped `w` may appear one or more times in succession. The dot following the second escape character denotes that an actual dot character may appear at this location in the string.

5. **A string composed of at least two and at most four alphabetic, digit, and underscore characters**
   We can denote this with the following RegEx fragment:

   ```
   \w{2,4}
   ```

The backslash character acts as an escape character indicating to regard its subsequent character differently from its normal behavior in a RegEx pattern. The `w` following the escape character is shorthand denoting that any alphabetic, digit, or underscore character is valid in this position. The braces enclosing the `2,4` sequence of characters is a quantifier for the `w` indicating that the characters conforming to the escaped `w` may appear as few as two times and as many as four times in succession.

Concatenating Regular Expression Fragments

Concatenating these RegEx fragments together results in the following RegEx pattern, where the characters corresponding to each of the five fragments previously described are shown with alternating bold:

```
\w+(\.\w+)*@(\w+\.)+\w{2,4}
```

To experiment with this pattern, fire up one or all of the RegEx testers, and type this pattern into the RegEx pattern text box. Then, one by one, type the potential email address strings from Table 4 into the text box, and see how the tester identifies their compliance as valid email addresses.

As can be seen from the Valid Address column in Table 4, some of the email addresses indicate a qualified yes for their validity. This really means that a string was found in the text conforming to the RegEx pattern identifying an email address, but it doesn't guarantee that the entire string containing this pattern is indeed a valid email address. The Comments column of the table indicates how each one of these qualified yeses could be considered invalid.

Item	Potential Email Address	Valid Address	Comments
1	@.ludwig.von.beethoven@@9symphonies.org.@	No	Invalid consecutive at-signs following "beethoven"

Table 4 Valid and Invalid Email Addressing for Testing the Email Regular Expression Pattern

Item	Potential Email Address	Valid Address	Comments
2	@.ludwig.von.beethoven.@9symphonies.org.@	No	Invalid ".@" string following "beethoven"
3	@.ludwig.von.beethoven@9symphonies.org.@	Qualified yes	Valid email address with invalid leading "@." string and trailing ".@" string
4	@.ludwig.von.beethoven@9symphonies.org.	Qualified yes	Valid email address with invalid leading "@." string and trailing dot character
5	.ludwig.von.beethoven@9symphonies.org.	Qualified yes	Valid email address with invalid leading and trailing dot characters
6	.ludwig.von.beethoven@9symphonies.org	Qualified yes	Valid email address with invalid leading dot character
7	ludwig.von.beethoven@9symphonies.org	Yes	Valid email address with three name qualifiers preceding the at-sign
8	ludwig.beethoven@9symphonies.org	Yes	Valid email address with two name qualifiers preceding the at-sign
9	beethoven@9symphonies.org	Yes	Valid email address with only one name qualifier preceding the at-sign
10	ludwig.von.beethoven@9symphonies.or	Yes	Valid email address with two-character top-level domain name ("or")

Table 4 Valid and Invalid Email Addressing for Testing the Email Regular Expression Pattern (Cont.)

Item	Potential Email Address	Valid Address	Comments
11	ludwig.von.beethoven@9symphonies.o	No	Invalid top-level domain name (".o")
12	ludwig.von.beethoven@9symphonies.orga	Yes	Valid email address with four-character top-level domain name ("orga")
13	ludwig.von.beethoven@9symphonies.organ	Qualified yes	Valid email address with invalid trailing "n" character following domain name
14	lud@wig.von.beethoven@9symphonies.org	Qualified yes	Due to having two at-signs, is identified as two consecutive valid email addresses
15	ludwig.von.beethoven@9-symphonies.org	No	Invalid hyphen character separating "9" and "symphonies"

Table 4 Valid and Invalid Email Addressing for Testing the Email Regular Expression Pattern (Cont.)

Because usually we're interested in whether a specific string of characters is a valid email address, and not whether we can find what amounts to a valid email address within some text file, a more prudent approach to this validation is to ensure that the email address begins at the first character of text and ends at the last character of text. Only then could we be more reasonably certain that such an address doesn't accidentally appear as a substring in some larger string.

We know how to do this from our experiments with credit card patterns—prefix the RegEx pattern with a caret and suffix it with a dollar sign, indicating, respectively, that the matching string must begin with

the first character of text and end with the last character of text. So let's modify our email address validity RegEx pattern accordingly:

```
^\w+(\.\w+)*@(\w+\.)+\w{2,4}$
```

When again we try those email addresses listed in Table 4 with qualified yeses for their validity, we now should find that each of these indicates *No* for its validity; that is, extraneous leading or trailing characters and multiple at-signs otherwise denoting two consecutive email addresses no longer are considered acceptable for RegEx pattern-matching purposes.

A More Comprehensive Email Pattern

After thoroughly testing the program and placing it into production, within the first few days after release, the inevitable support call will come to inform us that email *ludwig.von.beethoven@9-symphonies.org*, which we had regarded as invalid due to the hyphen, is indeed valid! In fact, we're told, virtually all of the special characters are regarded as valid in email addresses at those locations where we had expected only letters, digits, and the underscore character.

Investigation will reveal that each location in our email address pattern where the *any word* character is used—the `\w`—actually should accommodate *any character except* those that designate nodes in the email address, specifically, the at-sign and the separating dots. The RegEx language syntax facilitates this by enabling us to specify the set of characters that *aren't* valid at that string position, instead of the entire set of characters that *are* valid at a particular string position, as we already know how to do with character sets.

Using this technique with our email address pattern, we could replace each of the any word character identifiers with the two-character set composed of the at-sign and the dot, which *are to be excluded* from matching at that position and meaning that all other characters are valid at that position. This constitutes a negative set, or *exception* set, of characters. Such a character set is identified by a caret character occupying the first character position within the character set. This caret indicates that all

subsequent member characters of the set will cause the pattern *not to be matched*; that is, all characters are regarded as matching the character position *except* those listed in the character set.

This is the second role we see being played by the caret character in the RegEx language syntax, the other being the start-of-string indicator discussed earlier. Accordingly, let's change our email validation pattern to replace the any-word indicators with the exception set composed of leading caret negation indicator and members at-sign and dot, as shown here (the changes are highlighted with alternating bold):

```
^\w+(\.\w+)*@(\w+\.)+\w{2,4}$
^[^@.]+(\.[^@.]+)*@([^@.]+\.)+[^@.]{2,4}$
```

Notice that none of the dots in the new character sets is preceded by the escape character, which causes each one to be recognized as a dot in the text and not as the any-character the dot normally matches. We might have expected the escape indicator to be required; however, when included in a character set, the dot needs no qualification to be regarded as itself. This is because its normal role of matching any character otherwise would nullify the entire set. An inclusion character set that includes a dot would mean the same thing as the dot by itself without a character set specification. Likewise, an *exclusion* character set with a caret as the first character in the character set would effectively mean no character possibly could match that character position. Even so, the escape character still can be used to offer the visual clue to suggest the dot doesn't function in its normal role. You can prove this by subjecting one of the preceding email addresses to the following two RegEx patterns:

```
[^.]
[^\.]
```

As you should find, both of them cause the dot characters in the email address to be regarded as characters not matching the pattern. Indeed, with the second variation, a backslash character appearing in the email address is regarded as a valid character, confirming that its role in the character set remains as an escape indicator cast upon the subsequent character.

6.3 Assessing Password Strength

Let's assume our company leaders have expressed interest in an ABAP utility into which users can type a potential password they are considering using, and the utility will assess its relative strength against the password being cracked. The security department has identified the set of criteria to which a potential password should conform to be rated secure. One strength assessment point will be awarded for each of the criteria with which the potential password conforms. The sum of the strength assessment points will be shown in an information message popup screen upon the user typing a password into the initial selection screen and clicking execute. This utility should enable the user to iteratively provide passwords and for each one receive its corresponding strength assessment points so long as the user continues to click the execute command, abandoning the activity via the back, exit, or cancel commands.

The security department has identified the following criteria as being representative of a strong password:

- At least one uppercase alphabetic character
- At least one lowercase alphabetic character
- At least one digit character
- At least one special character
- At least eight characters in length

Special characters, we've been told, are limited to those that share a digit character on the keyboard, that is, those characters that can be typed by pressing the Shift key and a digit character appearing on the upper row of the keyboard. The security department also has identified the following criteria as being representative of a *very strong* password:

- At least two uppercase alphabetic characters
- At least two lowercase alphabetic characters
- At least two digit characters
- At least two special characters
- At least 12 characters in length

Additionally, an *incredibly strong* password would be at least 16 characters in length.

The strongest password a user could provide would conform to each of the criteria and earn the user the maximum of 11 password strength points.

Our task is to design and write an ABAP utility that can accept a password string of characters as input and assess the merits of the password based on awarding one point for conforming to one of these criteria.

We'll leave the mechanics of the input screen and message response until later. For now, let's concentrate on how to assess the merits of the potential password against each of the criteria. Had our criteria been limited to those representing only a strong password, we would have be able to use a series of familiar ABAP statements such as the series of `CONTAINS ANY` operators in Listing 8 to assess the relative strength.

```
clear password_strength.
if password ca sy-abcde.
  add 01 to password_strength.
endif.
if password ca 'abcdefghijklmnopqrstuvwxyz'.
  add 01 to password_strength.
endif.
if password ca '0123456789'.
  add 01 to password_strength.
endif.
if password ca '!@#$%^&*()'.
  add 01 to password_strength.
endif.
```

Listing 8 ABAP Code Using the CONTAINS ANY Operator for Validating Password Strength

However, we also need to be able to identify two occurrences of uppercase, lowercase, digit, and special characters, which can occur at any character position. Considering what we need to manage to retain the offset where the first of any of these characters had been found and then continuing from there to find a second character within the same character set from the next character position to the end of the password, while also

accounting for the first character of a character set being found as the final password character isn't impossible, but it is cumbersome.

Here is where the use of RegExes begins to make our programming lives easier. We can define RegEx counterparts to the preceding statements and use these to locate both single and double occurrences of the character sets. Let's define RegEx patterns for finding at least one character from each of the character sets, starting with a pattern to match one uppercase character:

```
[A-Z]
```

That was simple. Now for a pattern to match one lowercase character:

```
[a-z]
```

Again, that wasn't difficult. Now for a pattern to match one digit:

```
\d
```

And, finally, a pattern to match one of the specified special characters:

```
[!@#\$%\^&\*\(\)]
```

That last one was a bit more complicated, not only because we needed to type the entire list representing the set of valid special characters but also because some of these characters naturally mean something else in the RegEx language syntax unless they are preceded by the escape character (\), which we needed to use to precede the special characters dollar sign, caret, asterisk, and both left and right parentheses to indicate these are to be regarded as characters that may appear in the text.

Now we can alter the previous code replacing the use of the `CONTAINS ANY` operator with the `FIND` function, as in Listing 9.

```
clear password_strength.
if find( val = password regex = one_upper_case_letter_pattern )
  ge 00. add 01 to password_strength.
endif.
if find( val = password regex = one_lower_case_letter_pattern )
  ge 00. add 01 to password_strength.
endif.
if find( val = password regex = one_digit_pattern ) ge 00.
  add 01 to password_strength.
endif.
if find( val = password regex = one_special_character_pattern )
```

```
    ge 00. add 01 to password_strength.
  endif.
```

Listing 9 ABAP Code of Listing 8, But with the FIND Function Replacing the CONTAINS ANY Operator

In addition, we also can define comparable RegEx patterns for finding at least two of the characters from the corresponding character set, again starting with a pattern to match two uppercase characters:

```
[A-Z].*[A-Z]
```

A pattern to match two lowercase characters:

```
[a-z].*[a-z]
```

A pattern to match two digits:

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

And, finally, a pattern to match two of the specified special characters:

```
[!@#\$%\^&\*\(\)].*[!@#\$%\^&\*\(\)]
```

In each of the preceding cases, we've taken the pattern for the single character and appended it to itself with an intervening indication for zero or more of any character—the dot followed by the asterisk. Recall that the dot indicates *any character* may occupy that position of the string, and the asterisk is a quantifier for the dot, indicating that the matching any-character may appear zero or more times at that location. These patterns now may follow those patterns checking for single-character compliance, as in Listing 10.

```
.         ..
          if find( val = password
                   regex = two_upper_case_letters_pattern ) ge 00.
            add 01 to password_strength.
          endif.
          if find( val = password
                   regex = two_lower_case_letters_pattern ) ge 00.
            add 01 to password_strength.
          endif.
          if find( val = password
                   regex = two_digits_pattern ) ge 00.
            add 01 to password_strength.
          endif.
          if find( val = password
```

```
            regex = two_special_characters_pattern ) ge 00.
  add 01 to password_strength.
endif.
...
```

Listing 10 ABAP Code for Checking Whether Password Contains Two Uppercase, Two Lowercase, Two Digits, and Two Special Characters

At this stage, you might be thinking that the `CONTAINS ANY` operator design wasn't very difficult, despite having to keep track of finding each successive character when it came to matching two characters from a particular character set. Perhaps this is true; however, let's consider that with RegEx patterns, we're able to build the two-character patterns by concatenating two one-character patterns with an intervening any-character pattern. Not only that, but with subsequent development, such as when the security department finally gets around to defining more resilient password criteria that might award points for a minimum of three characters from each character set, it would be a simple matter to create new three-character patterns using the same concatenation technique we used to create the two-character patterns and to include these patterns in the series of checks. Performing the equivalent surgery on the code using the `CONTAINS ANY` operator now becomes more complicated and prone to introducing errors into the code.

Indeed, at some point, we might conclude that placing the qualifying criteria, whether defined using the `CONTAINS ANY` operator technique or RegExes, might be better if the qualifications were contained within a table to be looped at. With this design, a single loop through the table containing each of the RegEx patterns would require only a single `IF` statement within the loop, while the corresponding `CONTAINS ANY` operator technique still would require code for parsing through the password string to find subsequent matches for each character set.

So, let's do some more RegEx programming. Create a password strength analyzer program using the source code shown in Listing 11 (this is a similar design to the simple RegEx tester program we used in Section 4.1, so you can copy that program as a starting point and alter it accordingly).

```
report zpassword_strength_analyzer.
*---------------------------------------------------------------------
* Properties -
*   Title                  : Password strength analyzer
*   Type                   : Executable program
*   Package                : $TMP
*   Unicode checks active  : checked
*   Fixed point arithmetic: checked
* This program is loosely modeled on the example ABAP program
* shown as Listing 5.15, starting on page 312 of the book ABAP
* Objects: ABAP Programming in SAP NetWeaver (Horst Keller,
* Sascha Krüger; 2nd ed., 2007, SAP PRESS).
*---------------------------------------------------------------------
* Selection screen definition
*---------------------------------------------------------------------
selection-screen : begin of screen 100 as window.
parameters       : password       type string      lower case
                 .
selection-screen : end   of screen 100.
*---------------------------------------------------------------------
* Class password_strength_analyzer
*---------------------------------------------------------------------
class password_strength_analyzer       definition.
  public section.
    class-methods: class_constructor
                 , analyze
                 .
  private section.
    constants    : one_upper_case_letter_pattern
                                 type string    value '[A-Z]'
                 , one_lower_case_letter_pattern
                                 type string    value '[a-z]'
                 , one_digit_pattern
                                 type string    value '\d'
                 " These special characters represent those
                 " sharing the digit keys on the top row of the
                 " keyboard.  Special characters dollar sign,
                 " caret, asterisk, left parenthesis, and right
                 " parenthesis all are part of the regular
                 " expression syntax.  In order to indicate
                 " these are characters to be found in the
                 " text, each one is preceded by the escape
                 " character: backslash.
                 , one_special_character_pattern
                                 type string
                                   value '[!@#\$%\^&\*\(\)]'
                 , any_intervening_characters
                                 type string    value '.*'
                 , minimum_length_08
                                 type string    value '.{8,}'
```

```
                    , minimum_length_12
                                    type string    value '.{12,}'
                    , minimum_length_16
                                    type string    value '.{16,}'
                    .
        class-data  : two_upper_case_letters_pattern
                                    type string
                    , two_lower_case_letters_pattern
                                    type string
                    , two_digits_pattern
                                    type string
                    , two_special_characters_pattern
                                    type string
                    .
endclass.
class password_strength_analyzer       implementation.
  method class_constructor.
    " Set all two-characters patterns:
    concatenate one_upper_case_letter_pattern
                any_intervening_characters
                one_upper_case_letter_pattern
          into two_upper_case_letters_pattern.
    concatenate one_lower_case_letter_pattern
                any_intervening_characters
                one_lower_case_letter_pattern
          into two_lower_case_letters_pattern.
    concatenate one_digit_pattern
                any_intervening_characters
                one_digit_pattern
          into two_digits_pattern.
    concatenate one_special_character_pattern
                any_intervening_characters
                one_special_character_pattern
          into two_special_characters_pattern.
  endmethod.
  method analyze.
constants   : information    type symsgty   value 'I'
            , error          type symsgty   value 'E'
            , password_strength_banner type string
              value 'Password strength is'
            .
data        : notification   type string
            , severity       type symsgty
            , exception      type ref to cx_root
            , password_strength
                             type n length 02
            .
do.
  call selection-screen 100 starting at 02 02.
  if sy-subrc ne 00.
```

```
  return.
endif.
clear password_strength.
try.
  if find( val   = password
           regex = one_upper_case_letter_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = two_upper_case_letters_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = one_lower_case_letter_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = two_lower_case_letters_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = one_digit_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = two_digits_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = one_special_character_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = two_special_characters_pattern ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = minimum_length_08 ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = minimum_length_12 ) ge 00.
    add 01 to password_strength.
  endif.
  if find( val   = password
           regex = minimum_length_16 ) ge 00.
    add 01 to password_strength.
  endif.
  concatenate password_strength_banner
              password_strength
```

```
                into notification
                       separated by space.
        severity                    = information.
      catch cx_sy_invalid_regex into exception.
        notification                = exception->get_text( ).
        severity                    = error.
      endtry.
      message notification type information
              display like severity.
    enddo.
  endmethod.
endclass.
*---------------------------------------------------------------------
* Event start-of-selection
*---------------------------------------------------------------------
start-of-selection.
  password_strength_analyzer=>analyze( ).
```

Listing 11 ABAP Program for Password Strength Analyzer

After the new program is created and activated, execute it using the potential passwords described in Table 5 to confirm that each one earns the number of strength points indicated.

Item	Potential Password	Strength Points	Comments
1	+	0	The plus sign character isn't a special character earning strength points, so no points are awarded.
2	+A	1	One point is awarded for including the first uppercase character.
3	+A+	1	Again, the plus sign character isn't a special character earning strength points, so no additional points are awarded.
4	+A+g	2	One more point is awarded for including the first lowercase character.
5	+A+g$	3	One more point is awarded for including the first special character earning strength points.

Table 5 Passwords to Be Used for Testing the Password Strength Analyzer

Item	Potential Password	Strength Points	Comments
6	+A+g$5	4	One more point is awarded for including the first digit character.
7	+A+g$5+	4	Again, the plus sign character isn't a special character earning strength points, so no additional points are awarded.
8	+A+g$5++	5	Although the additional plus sign character earns no strength points, one more point is awarded for pass-word length, which is now at least eight characters.
9	+A+g$5++K	6	One more point is awarded for including the second uppercase char-acter.
10	+A+g$5++K2	7	One more point is awarded for including the second digit character.
11	+A+g$5++K2+	7	Again, the plus sign character isn't a special character earning strength points, so no additional points are awarded.
12	+A+g$5++K2++	8	Although the plus sign character earns no strength points, one more point is awarded for password length, which is now at least 12 characters.
13	+A+g$5++K2++w	9	One more point is awarded for including the second lowercase char-acter.
14	+A+g$5++K2++w&	10	One more point is awarded for including the second special character earning strength points.
15	+A+g$5++K2++w&+	10	Again, the plus sign character isn't a special character earning strength points, so no additional points are awarded.

Table 5 Passwords to Be Used for Testing the Password Strength Analyzer (Cont.)

Item	Potential Password	Strength Points	Comments
16	+A+g$5++K2++w&++	11	Although the plus sign character earns no strength points, one more point is awarded for password length, which is now at least 16 characters.

Table 5 Passwords to Be Used for Testing the Password Strength Analyzer (Cont.)

Now try your own password strings with the analyzer. Determine that you get the correct password strength assessment based on what you know about the process by which the points are awarded.

6.4 Identifying the Text File Format

Suppose we work at an Internet-based retailer facilitating the electronic exchange of commerce documents with suppliers through text files formatted using XML to describe the transfer orders for merchandise moving between warehouses and using HTML to describe information such as assembly instructions, packing lists, maintenance manuals, warranties, sales pitches, and customer testimonials for the merchandise. Upon receiving a set of text files from a supplier, we would need the ABAP program handling the file exchange for our SAP logistics system to determine which types of files have been received and shuttle each one to the appropriate department for further processing.

This could be handled by a customized singleton global class defined with a public method accepting the text file as input and using a RegEx to identify whether the file format is XML, HTML, or indeterminate. This might be a good candidate for using the `cl_abap_regex` and `cl_abap_matcher` classes, as illustrated in Listing 12.

```
class zcl_file_type_identifier definition final create private
      public.
  public section.
    types         : file_type       type char1
                  , text_file_format
```

```
                                type string
                .
constants       : file_type_is_indeterminate
                                type file_type value space
                , file_type_is_html
                                type file_type value 'H'
                , file_type_is_xml
                                type file_type value 'X'
                .
class-methods: class_constructor
                , get_instance
                  returning
                    value(instance)
                                type ref
                                  to zcl_file_type_identifier
                .
  methods       : identify_file_type
                    importing
                      text_file  type text_file_format
                    returning
                      value(file_type)
                                type file_type
                .
  private section.
    constants   : xml_identifier type string value '<?x'
                , file_identifier_pattern
                                type string value ...
                .
    class-data  : singleton     type ref
                                  to zcl_file_type_identifier
                .
    data        : regex_object  type ref to cl_abap_regex
                .
    methods     : constructor
                .
endclass.
class zcl_file_type_identifier implementation.
  method class_constructor.
    create object singleton.
  endmethod.
  method constructor.
    create object me->regex_object
      exporting pattern           = file_identifier_pattern.
  endmethod.
  method get_instance.
    instance                    = singleton.
  endmethod.
  method identify_file_type.
    data          : matcher_object type ref to cl_abap_matcher
                  , matcher_offset type int4
```

```
                .
matcher_object =
  me->regex_object->create_matcher( text = text_file ).
if matcher_object->find_next( ) is initial.
  file_type                    =  file_type_is_indeterminate.
else.
  matcher_offset = matcher_object->get_offset( ).
  if substring( val = text_file
                off = matcher_offset
                len = 03 )    eq xml_identifier.
    file_type                  =  file_type_is_xml.
  else.
    file_type                  =  file_type_is_html.
  endif.
endif.
  endmethod.
endclass.
```

Listing 12 ABAP Global Class for Identifying the Text File Format

Notice we haven't yet provided a RegEx value for the private constant `file_identifier_pattern` of this class; we shall do this shortly. The methods of this class can be invoked by a program that has obtained a file in need of type identification, as in Listing 13.

```
  o
  o
  data           : received_file  type
                     zcl_file_type_identifier=>text_file_format
                 , file_type      type
                     zcl_file_type_identifier=>file_type
                 , file_type_identifier
                                  type ref
                                    to zcl_file_type_identifier
                   .
  o
  o
  " Identify type of text file received from supplier:
  file_type_identifier =
    zcl_file_type_identifier=>get_instance( ).
  file_type =
    file_type_identifier->identify_file_type(
      text_file = received_file ).
  case file_type.
    when zcl_file_type_identifier=>file_type_is_xml.
      ...
    when zcl_file_type_identifier=>file_type_is_html.
      ...
```

```
    when others.
      ...
  endcase.
  o
  o
```

Listing 13 ABAP Code Fragment Invoking ABAP Global Class in Listing 12, to Identify the Type of Text File Received

Now let's explore how to define a pattern for distinguishing between XML and HTML files, and use this pattern to complete the value for the private constant `file_identifier_pattern` of class `zcl_file_type_indentifier` from Listing 12. The protocol definitively identifying either HTML or XML occurs within the first few characters. This isn't to suggest that failure to find these protocol identifiers means the text doesn't comply, but simply that subsequent checks would be required to make such a determination.

To definitively be defined as HTML, the first nonblank character of text must be the left angle bracket (<), followed by an exclamation point, followed by the string `doctype` using any combination of uppercase and lowercase characters, followed by at least one space:

```
<!DOCTYPE
```

We can define a RegEx pattern for this identifier as follows:

```
\s*<![Dd][Oo][Cc][Tt][Yy][Pp][Ee]\s
```

The first three pattern characters indicate that any number of whitespace characters may appear, including none. The next two indicate a left angle bracket must appear followed by an exclamation point. These are followed by a series of character sets spelling the word *doctype* in which either the uppercase or lowercase character will satisfy the match. These are followed by the two-character indication for any whitespace character.

To definitively be defined as XML, the first character of text must be the left angle bracket (<), followed by a question mark, followed by the lowercase string `xml` followed by at least one space:

```
<?xml
```

We can define a RegEx pattern for this identifier as follows:

```
<\?xml\s
```

The first character indicates that a left angle bracket must appear, and, unlike its HTML counterpart pattern, there can be no leading spaces. This is followed by the two-character indication for a question mark character; two characters are used because the question mark, which normally indicates a quantification of zero or one for the preceding element, requires the escape character backslash to designate that it is to be regarded as a character to be found in the text. This is followed by the explicit three-character string `xml`, indicating there is no uppercase variation for this string. This is followed by the two-character indication for any whitespace character.

To create a RegEx capable of determining whether a file of text certainly conforms to either of these file formats, we can build the RegEx as a set of these two strings. Recall that a set of strings is composed of strings bounded by parentheses and separated from each other by a vertical bar, as in the following pattern:

```
(string1|string2|...)
```

Replacing `string1` and `string2` with our HTML and XML text identifiers, respectively, results in the following RegEx:

```
(\s*<![Dd][Oo][Cc][Tt][Yy][Pp][Ee]\s|<\?xml\s)
```

In addition, we want to indicate that this pattern must be found at the very beginning of the text file. The HTML matching pattern already accommodates any leading spaces that may precede the opening left angle bracket, and the XML format doesn't permit leading characters of any type. Unless we indicate that the pattern is to start with the first character of the file, then we may simply find either of these file format protocol identifiers embedded in some other type of text file, perhaps merely representing examples of file elements described in a white paper titled "HTML and XML File Format Protocols" and formatted using the OpenOffice document text format (.odt) or perhaps the Adobe PDF (.pdf). Accord-

ingly, we need to prefix this group with a leading caret, indicating that the matching string must be found starting with the first character of text found in the file:

```
^(\s*<![Dd][Oo][Cc][Tt][Yy][Pp][Ee]\s|<\?xml\s)
```

Now we would be able to subject a text file to a `FIND FIRST` statement using this RegEx to determine whether the text definitely conforms to either the HTML or XML file protocol formats.

Fire up one or all of the RegEx testers and provide the pattern just described into the RegEx pattern text box. Then enter the following string into the text box beginning at the first character position:

```
<!DOCTYPE html> <?xml version="1.0" encoding="UTF-8"?>
```

This string `<!DOCTYPE` along with its following space should be highlighted. Experiment with changing the letters of `DOCTYPE` to lowercase and then to a random mix of uppercase and lowercase characters. The string should remain highlighted with each change. Include a random series of leading spaces—the string should remain highlighted with each change. Then include leading characters—the string no longer should retain its highlighting.

Cut the `<!DOCTYPE html>` string along with its trailing space preceding the XML identifier and paste it afterward, essentially reversing the order of these strings in the text. Now the string `<?xml` and its trailing space is highlighted. Experiment with changing the letters of `xml` to uppercase and then to a random mix of uppercase and lowercase characters. The string no longer should retain its highlighting with any of these changes. Restore the string to lowercase `xml` so that it regains its highlighting. Include a single leading character—space or otherwise—and the string no longer should retain its highlighting.

This concept could be adapted to any other file formats where the corresponding protocol has rules enabling the identification of the format in the first few characters, whether proprietary or industry standard in nature.

7 Greedy and Lazy Matching

The implementation of RegExes in ABAP is based upon the *leftmost-longest rule*. This means that pattern matching will begin with the farthest left character of text and from that point moving to the right will observe the longest matching string possible. After a string in the text has been identified as matching the pattern using this leftmost-longest rule, the search continues with the next available character of text searching again for another leftmost-longest match. This process is repeated until the text has been fully scanned to its final character.

Another name for the leftmost-longest rule is *greedy matching*. An alternative to greedy matching is known as *lazy matching*, where the fewest characters possible are observed when determining a match for a pattern.

> **Note**
>
> Lazy RegEx matching isn't yet supported within ABAP.

Some RegEx implementations facilitate lazy matching by the use of a question mark following a quantifier, which is an indication that the pattern is to be matched using the fewest characters permissible as specified by the quantifier.

As an example, let's consider the following RegEx pattern:

```
.*\d
```

This pattern indicates matches for zero or more characters preceding a digit character. The dot indicates matches for any character, and the asterisk is a quantifier that indicates zero or more of these any-characters are to be considered a match. The backslash acts as an escape character for the next element, changing its normal behavior, and the `d`, having been escaped, indicates matches for any digit. Using this pattern with the following string composed of two sets of three alphabetic characters followed by three digits

```
abc123def456
```

would cause a match on the entire string. This is because the greedy behavior of the zero or more any-characters would identify a match on the string starting with "a" and ending with "5", leaving the final "6" to match the any digit character represented by the `\d` in the pattern.

We can invert the greedy behavior of the zero or more characters to show that we want the fewest characters that would cause a match by inserting a question mark following the asterisk:

```
.*?\d
```

This illustrates yet another behavior of the question mark character, which until now we had understood to indicate a quantifier of zero or one.

Using this RegEx pattern with the Internet-based RegEx testers, all of which accommodate this syntax for lazy pattern matching, we get not just one match but six matches. In fact, with this pattern, each digit in the string constitutes a match with the `\d` portion of the pattern, and from this point leftward, the fewest zero or more characters, represented by the `.*?` portion of the pattern, matches the characters from the preceding digit.

The difference between the two patterns is illustrated here with alternating bolding, where the first string, based on greedy matching, finds a single match, and the second string, based on lazy matching, finds six matches:

```
abc123def456
abc123def456
```

In the second string, each digit constitutes a match for the pattern, and each one is preceded by the fewest zero or more characters required to satisfy the match.

8 Summary

Here, we'll discuss what we've covered and look ahead to see what changes we might see in later releases.

8.1 What We Have Learned

We've come a long way from the revulsion we might have felt at the sight of the RegEx language syntax, and now we understand better how this works and how we can construct viable RegEx patterns. We also know that the wildcard characters with which we're familiar from SQL and selection tables don't behave the same as wildcards in the RegEx language syntax. We didn't cover every last element of the RegEx language, but we covered the basics, so you can easily adapt to using the unfamiliar elements now that you understand the following concepts:

- Unless otherwise qualified, all alphabetic characters, all digit characters, and most special characters used in RegExes simply match themselves in the target text.
- Unless otherwise qualified, consecutive characters used in RegExes will match those locations where the same consecutive characters appear in the target text.
- Unless otherwise qualified, RegExes are case-sensitive.
- Bounding square brackets specified in a RegEx pattern indicates a set of characters, any one of which can match at that character position in the target string.
- Bounding parentheses *with* embedded vertical bar characters specified in a RegEx pattern indicates a set of strings, any one of which can match at that position in the target string.
- Bounding parentheses whether *with* or *without* embedded vertical bar characters specified in a RegEx pattern indicates a group; groups are captured and available for use with replacement strings.
- Unless otherwise qualified, the dot character matches any character and is the only wildcard character recognized with RegExes.
- Single characters or groups can be assigned a quantification value indicating the number of times the single character or group may be repeated in succession.
- A range of characters can be specified by indicating the first and last character of the range separated by a hyphen.

Table 6 summarizes the roles played by some of the special characters composing the RegEx language syntax.

Character	Designation
.	Any single character
[...]	Set of interchangeable characters applicable to a single character position
(...|...|...)	Set of interchangeable strings applicable to a single string position
(...)	Captured group
{m,n}	Quantifier applicable to the preceding element, indicating minimum and maximum replications
?	Quantifier shorthand for {0,1}
*	Quantifier shorthand for {0,}
+	Quantifier shorthand for {1,}
^	Start of string; also, indicator for exception character set
$	End of string
\	Escape character
\s	Any whitespace character
\S	Any character other than whitespace character
\w	Any letter, number, or underscore character
\W	Any character other than letter, number, and underscore character
\d	Any digit
\D	Any character other than a digit
\b	A word boundary
-	Range indicator when appearing between two characters of a character set
$0	Reference in replacement string to entire matching pattern
$1, $2, $3 …$n	Reference in replacement string to *n*th capture group

Table 6 Roles of Special Characters in the RegExes Language Syntax

8.2 What the Future Might Hold

Although ABAP already provides a robust implementation accommodating RegExes, we may see new statements and/or functions supporting RegExes in future releases. We may even see support for both greedy and lazy pattern matching, which were discussed briefly in Section 7.

Although ABAP currently doesn't support lazy matching, the online documentation describing Special Characters in RegExes includes the three entries (Table 7) in the list of special characters for character string patterns:

Special Characters for Character String Patterns	Description
`{n,m}?`	Reserved for later enhancements
`*?`	Reserved for later enhancements
`+?`	Reserved for later enhancements

Table 7 Special Characters for Character String Patterns as Listed in ABAP Documentation

In each case, the characters preceding the question mark are quantifier indications:

- `{n,m}?`
  Shows a range of characters with minimum and maximum length limits bounded by braces and separated by a comma.
- `*?`
  Shows the asterisk character, a single-character quantifier denoting zero or more characters.
- `+?`
  Shows the plus sign character, another single-character quantifier denoting one or more characters.

This suggests that at some point in the future, there may be support for lazy pattern matching with RegExes in ABAP.

9 What's Next?

You've learned about RegEx and ABAP, become familiar with a number of tools, and used practical examples to improve your programming skills. Why not continue that pattern by learning the basic principles of object-oriented programming using ABAP Objects! Pick up *Object-Oriented Programming with ABAP Objects* by James Wood today, and discover exception handling, object debugging, unit testing, and more. It's never been easier to make the move from procedural programming to object-oriented programming.

In addition to this book, our editors picked a few other SAP PRESS publications that you might also be interested in. Check out the next page to learn more!

More from SAP PRESS

ABAP 7.4 Certification Guide—SAP Certified Development Associate: If you're about to take your ABAP C_TAW12_740 certification exam, this is the book for you. Answer sample questions, revisit key terminology, and implement recommended test-taking strategies. Visit www.sap-press.com/3792 for more information.

663 pages, 3rd edition, pub. 05/2015
E-book: $59.99 | **Print:** $69.95 | **Bundle:** $79.99
www.sap-press.com/3792

ABAP to the Future: Discover all the tips, tricks, and features of the ABAP universe! Learn about ABAP in relation to Web Dynpro ABAP, Floorplan Manager, SAPUI5, BRF+, BOPF, and more. Visit www.sap-press.com/3680 for more information.

727 pages, pub. 03/2015
E-book: $59.99 | **Print:** $69.95 | **Bundle:** $79.99
www.sap-press.com/3680

ABAP Workbench: Learn 100 things every ABAP developer should know about the programming language! Save time with helpful shortcuts, pick up new tips and tricks, and brush up on old skills. Visit www.sap-press.com/3127 for more information.

341 pages, pub. 07/2012
E-book: $44.99 | **Print:** $49.95 | **Bundle:** $59.99
www.sap-press.com/3127

SAP PRESS E-Bites

SAP PRESS E-Bites provide you with a high-quality response to your specific project need. If you're looking for detailed instructions on a specific task; or if you need to become familiar with a small, but crucial sub-component of an SAP product; or if you want to understand all the hype around product xyz: SAP PRESS E-Bites have you covered. Authored by the top professionals in the SAP universe, E-Bites provide the excellence you know from SAP PRESS, in a digestible electronic format, delivered (and consumed) in a fraction of the time!

Janet Salmon
SAP Simple Finance: How Do I Get Started without Migrating?
ISBN 978-1-4932-1284-2 | $9.99 | 40 pages

Eli Klovski
Configuring Funds Management for SAP Public Sector
ISBN 978-1-4932-1302-3 | $14.99 | 95 pages

Eric Du
SAP HANA Smart Data Streaming and the Internet of Things
ISBN 978-1-4932-1303-0 | $9.99 | 86 pages

The Author of this E-Bite

James E. McDonough has been a computer programmer for 33 years. Initially a mainframe programmer, he made the switch to ABAP almost 20 years ago, and now works as a contract ABAP programmer, designing and writing ABAP programs on a daily basis. An advocate of using the object-oriented programming features available with ABAP, he has been teaching private ABAP education courses over the past few years, where his background in education enables him to present and explain complicated concepts in a way that makes sense to beginners. James received his degree in music education from Trenton State College, and spent two years teaching music in the New Jersey public school system. He maintains an active presence as a freelance jazz bassist between New York and Philadelphia.

Usage, Service, and Legal Notes

Notes on Usage

This e-book is **protected by copyright**. By purchasing this e-book, you have agreed to accept and adhere to the copyrights. You are entitled to use this e-book for personal purposes. You may print and copy it, too, but also only for personal use. Sharing an electronic or printed copy with others, however, is not permitted, neither as a whole nor in parts. Of course, making them available on the Internet or in a company network is illegal.

For detailed and legally binding usage conditions, please refer to the section Usage, Service, and Legal Notes.

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: Hareem Shafi (hareems@rheinwerk-publishing.com).

We welcome every suggestion for improvement but, of course, also any praise! You can also share your reading experience via Twitter, Facebook, or email.

Supplements

Supplements (sample code, exercise materials, lists, and so on) are provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://

www.sap-press.com/3974. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at SAP PRESS, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.sap-press.com/ provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.sap-press.com/.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you

Imprint

This e-bite is a publication many contributed to, specifically:

Editor Hareem Shafi
Acquisitions Editor Kelly Grace Weaver
Copyeditor Julie McNamee
Cover Design Graham Geary
Layout Design Graham Geary
Production Nicole Carpenter
Typesetting III-Satz, Husby (Germany)

ISBN 978-1-4932-1307-8

1st edition 2015