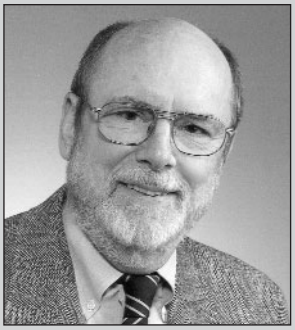
Get Ready to Exploit Regular Expression Pattern Matching to Examine and Manipulate Complex Data in Your ABAP Programs

David F. Jenkins



David Jenkins became an SAP consultant in 1996 after 29 years as a Consulting Systems Representative at IBM. He currently develops and teaches classes in beginning and advanced ABAP programming, ABAP performance and tuning, and Java, and consults with clients on all aspects of ABAP development, most recently at ChevronTexaco.

(complete bio appears on page 82)

Searching, replacement, parsing, extraction, and validation of data represent a significant portion of many ABAP development efforts. Such activities take place in many common contexts, including:

- Checking manual input data
- Searching and modifying external data files
- Examining database tables for the occurrence of troublesome or incorrect data
- Scanning internal table data
- Parsing out portions of data to access specific elements of information
- Parsing field data from files
- Validating interface file data (both format and value)

In this article, I describe in detail a few common data manipulation scenarios, highlighting the difficulties (from an ABAP coding standpoint) involved in accomplishing such efforts using only the tools currently on hand.

The use of *regular expression* pattern matching can make complex searches and replacements feasible, fast, and effective. There is no built-in support for regular expressions in SAP as of today, but there will be in an upcoming SAP NetWeaver release, which will provide regular expression search functionality in a number of different SAP contexts, including full integration into the ABAP language (enhancements to the `FIND` and `REPLACE` commands), and new classes (`CL_ABAP_MATCHER` and `CL_ABAP_REGEX`) that can be used

in custom programs. To give *SAP Professional Journal* readers a leg up on that release, this article provides a brief introduction to the concept of regular expression pattern matching, and shows how this functionality can provide useful results in standard SAP searching scenarios. (A future *SAP Professional Journal* article will discuss SAP's upcoming native support of regular expressions in detail.)

Since I have used regular expression pattern matching in other contexts, I have been anxiously awaiting their use in conjunction with SAP. In anticipation of the upcoming native support of regular expressions in SAP, I have found a stopgap/workaround method, also described in this article, that I easily implemented to utilize the regular expression functionality currently provided by SAP Web Application Server (SAP Web AS) 6.20. Whether you are a developer or a manager, you should find some information in this article that you can use to enhance the development of efficient, functional ABAP programs.

If, after reading this article, you are interested in reading further about regular expressions, I have included some references at the end of the article that you may find useful. A Google or Google Groups search for "Regular Expressions" will return a wealth of additional information on the subject, ranging from simple introductory-level information to erudite discussions of grammars, compilers, backtracking, regular language, and syntaxes. I've also included a selection of Web sites that offer valuable information on writing and using regular expressions, tutorials, and free/fee regular expression tools.

Using ABAP to Search, Replace, and Extract Text Strings

ABAP provides a number of commands to facilitate searching/replacement/validation of data:

- `FIND`
- `SEARCH` (superseded by `FIND` in 6.20 and later)
- `REPLACE`
- `SPLIT`
- `OVERLAY`

When invoking current `SEARCH` (or `FIND`) and `REPLACE` functionality, searching is limited to rather straightforward string matching. That is, so-called "wildcard" search arguments are not supported, and little flexibility is provided, other than provisions for honoring the case of the search pattern (or not), and starting the search at specified positions in the target string — that's pretty much it. (The ABAP statement `IF`, of course, provides limited support for wildcards via the `CP`, `CO`, `CA`, `CN`, and `NP` operators, but their employment would imply additional programming effort beyond that of simple `SEARCH/REPLACE` use.)

While this simple `SEARCH/REPLACE` functionality is undoubtedly useful, other facilities that give the developer tighter control over the actual matching operations might be even more useful. One approach, of course, is to develop custom ABAP code to accomplish more advanced searching/matching/replacing. In the sections that follow, I describe circumstances in which we might want to employ more enhanced searching/replacement services, and then go on to provide a very brief introduction to the subject of regular expressions, and how their use can address searching problems while markedly reducing the need for custom ABAP coding. I finish by describing how some such services might be easily incorporated into your current day-to-day ABAP development environments, in anticipation of the forthcoming SAP NetWeaver release.

Some [Erstwhile] Simple Data Entry Validation/Extraction Scenarios

Under what circumstances might we wish to utilize advanced searching/matching/replacing/extraction? Here are some common data manipulation scenarios

that appear rather simple on the surface, but on deeper analysis reveal significant development headaches. Experienced ABAP developers will quickly recognize these scenarios and the challenges they pose for solutions based on custom ABAP code.

> ✓ ***Note!***
>
> *In the following discussions, the term "validation" refers to the process of determining whether a particular bit of data could be an actual representation of the data it purports to contain. Therefore, we would "validate" 979/835-0123 as a phone number, because it could be a real phone number. Whether it is a real phone number is not part of the validation process in this context.*

Phone Numbers

Assume for the moment that you're developing the user interface for a program that will prompt the user for a US telephone number, and you'd like to do a couple of things with the entered data:

- Validate that the entered data does, indeed, represent a US phone number.
- Extract the area code portion of the phone number (if it exists in the data) for some further processing.

Without placing onerous data entry restrictions on the user (but at the same time imposing some reasonable data entry constraints), you decide you'll accept strings with the following characteristics (where *aaa* represents the area code, *ppp* represents the prefix, and *nnnn* is the actual number):[1]

1. The user can enter a three-digit area code *aaa*, a three-digit prefix *ppp*, and a four-digit number *nnnn* (subject to the rules below); while *aaa* is optional, *ppp* and *nnnn* are both obligatory.
2. *aaa* (if present) can be preceded by a string of zero or more blanks, followed by an optional left parenthesis.
3. The first digit of *aaa* (if present) may not be 0 or 1.
4. If present, *aaa* can be separated from *ppp* by a single blank, a dash, or a right parenthesis (if a left parenthesis precedes *aaa*).
5. *ppp* can be separated from *nnnn* by blanks or by a single dash.
6. No characters other than blanks may follow *nnnn*.
7. The first digit of *ppp* must be greater than zero.

Furthermore, we would like the matching operation to return *aaa*, if it exists. And we could go on, getting more complex with each succeeding rule. (Even at that, some clever user would probably find a way to enter a valid phone number that doesn't pass the previously listed edits, so let's stick with just these restrictions for the time being.)

These phone numbers should pass the edits:

```
2223334444
(222)3334444
  (222)333 4444
222333-4444
222-333 4444
222 333 4444
 222-333-4444
222-333-4444
(800)333 4444
123-4567
```

But these should be considered invalid entries:

```
0123456789     (violates rule 3)
-222-333-4444  (violates rule 2)
800/456/7890/  (violates rule 6)
(222)33-4444   (violates rule 1)
```

[1] The parsing of phone numbers is notoriously difficult (as is the parsing of email addresses). Issues such as country number format, presence/absence of extensions, presence/absence of area code, and various types of delimiters complicate the problem regardless of the method (custom code or regular expression) employed. For purposes of this example, then, I have assumed a rather structured format for the input format.

The point is, we want to make the entry of the data as easy as possible for the user, while making sure that we can extract an ostensibly valid number from what the user *does* enter; we want to let all valid phone numbers get though, and we want to reject all invalid phone numbers. Furthermore, we would like to be able to parse out the area code portion of the entered number (if it exists).

Straightforward matching/searching/replacement using `FIND/REPLACE` and a single matching string pose a number of limitations in this scenario:

- It makes no provision for alternatives (rule 4).
- It makes no provision for advanced logic to be applied during the search (rule 1).
- There is no facility to extract a desired portion or portions of the matching text (*aaa*).

Are these real problems in everyday development? Ask any experienced ABAP developer who has struggled with exactly these problems. (For that matter, ask any e-marketer who has lost a sale to a user who struggled with restrictive data entry requirements and abandoned a purchase in frustration!)

US Social Security Numbers (SSNs)

The verification of SSNs (especially if manually entered) can provide an interesting coding/debugging experience.[2] Let's say that we'd like to allow the data to be formatted as *aaacggcssss*, where *aaa* is a three-digit area code, *c...c* is a sequence of optional separator characters ("-", "/", ". ", or one or more blanks), *gg* is a two-digit group number, and *ssss* is some four-digit serial number. In addition, we note that the highest-numbered area currently in use is "772", "000" is an invalid area, "00" is an invalid group, and "0000" is an invalid serial number. Thus the following would constitute valid input:

```
222334444
222/33/4444
222.334444
222 33 4444
```

But these would be considered invalid:

```
774 22 4444
222//33/4444
222-00-4444
```

URLs, Email Addresses, and IP Addresses

Applications are increasingly required to use URLs, email addresses, and IP addresses. Email addresses in particular can be especially troublesome to validate, since they can contain wildly varying strings of characters.[3]

Examples of valid email addresses include:

```
First_name.last_name@yada.com
First_name Last_name
  <name1.name2@yada.com>
"yada. yada"@yada.org
```

Addresses that should *not* be considered valid are:

```
First_name last_name@yada.com
First_name."last_name"@yada.com
```

Currency Data

Users often get creative when furnishing an interface with currency data. Negative amounts can be shown by a preceding minus sign, a trailing minus sign, or surrounding the data in question with parentheses. Thousands separators may be periods, commas, or nonexistent. Dollars may be indicated (in the case of

[2] Readers interested in getting more information on SSN formatting and validation should refer to www.ssa.gov; the highest area and group number combinations in use at a given time can be found at www.ssa.gov/employer/highgroup.txt.

[3] The complexities of email address validation are far too voluminous to address here. A Google for "RFC 2822," "RFC 1034," or "RFC 1035" will return sources of additional information on Internet message formats, and will provide you with more than enough reading to fill some cold, winter evening….

US currency amounts) by a preceding dollar sign, in which case a minus sign may either precede or follow the dollar sign.

Validation can be further complicated by cents amounts — in the case of whole dollars, the cents may be missing, in which case the decimal point may be either present or missing. We may choose to enforce a standard that says that valid amounts can have zero, one, or two decimal places, but no more.

Before passing currency data on to a processing application, we would like to validate that the data does, in fact, contain an amount that can be processed by that application, so that we can forestall any possibilities of a crashed application due to ill-formed data. The logic to do this (in the absence of vendor-supplied conversion routines) can quickly become burdensome.

File Scanning

Data arriving via interface files can often present interesting problems, not the least of which is wasted resources due to nonproductive finger pointing. ("You sent us bad data!" "No, we sent it the way we've always sent it — your interface program is too dumb to understand what we sent!") Here's a sample of some common processing problems encountered when processing interface files.

Corrupted Data

Files containing interface data sometimes end up corrupted — records in the file may end up containing spurious control characters (CR/LF/Tab etc.), or character fields may contain illegal ASCII characters. These cases often result in two problems:

- *Recognizing* that a record contains spurious data
- *Correcting* the spurious data

Programming to search and replace control characters can be error prone and time consuming.

Field Parsing

Interface data files arrive in a wealth of differing formats and descriptions. Files often contain data in fields that are separated by some specific character, such as a comma. Processing problems can occur when fields contain strings of text, which may or may not include field separator characters. When it is expected that the strings may include field separator characters, the strings are often *delimited* — surrounded by delimiting characters (such as double quotation marks) that indicate the entire string between the delimiters must be treated as a single string. In the example that follows, we'll assume that the file contains fields separated by commas.

Here's a simple input file (note that record numbers have been added for readability, and are not part of the input data):

```
1  LAST NAME, FIRST NAME, PHONE,
     COMMENT, STATUS
2  Smith, Joe, 123 345 7890,
     SAP is great, Active
3  Jones, Bill, 111 222 3333,
     You're right…, Inactive
```

Parsing the data in this file would present no massive technical problems, since each field is unambiguously delimited by a comma.

What would happen, however, were the comment data in the record to contain a comma? Take a look at the following example:

```
1  LAST NAME, FIRST NAME, PHONE,
     COMMENT, STATUS
2  Smith, Joe, 123 456 7890,
     I do like SAP, I truly do, Active
3  Jones, Bill, 111 222 3333,
     Me, too, Inactive
```

Parsing these records will present a problem, since we can no longer rely on the commas to provide an unambiguous definition of the fields.

Recognizing this, many systems surround the text in fields that may contain a field separator character with some delimiting character. If that were done to

our problem file using a double quotation mark as such a delimiter, we might have:

```
1  LAST NAME, FIRST NAME, PHONE,
     COMMENT, STATUS
2  Smith, Joe, 123 456 7890,
     "I do like SAP, I truly do", Active
3  Jones, Bill, 111 222 3333,
     "Me, too", Inactive
```

With the inclusion of the string delimiters (double quotation marks), we now have a mechanism in place that will allow us to figure out where to split each record into its component fields. All that remains to do is to process the data, from left to right, only doing field splits if the next comma found does not appear inside a set of matching double quotation marks. Scanning from left to right, we need to keep track of how many double quotation marks follow the comma — if the number is even, then the comma is held to be a field separator; if it's odd, then the comma is apparently inside some single string.

While not at all impossible to do in ABAP, this processing can become unwieldy — perhaps involving multiple indexes, field symbols, intermediate variables, etc.

The Pattern Matching Challenge

We've seen that straightforward, string-matching searches using the ABAP commands `FIND/REPLACE` may not appreciably reduce our custom coding requirements. There is no native provision for:

- Negative searching logic — for example, if we want to insert a comma following a city unless no state has been entered
- `OR` logic
- Isolation of parts of the matching string to be used — in a replacement operation, or *even in subsequent matching tests in the same line of text*

All of these problems stem from the rather simple matching/replacement algorithm that the current `FIND/REPLACE` uses to search text; it would be extremely useful if developers could take advantage of advanced searching technologies to satisfy more demanding pattern matching requirements. In the case of replacement, we are further constrained by the fact that the matching pattern may not include any so-called "wildcard" characters.

In addition, there are other, even more useful searching facilities we might like to take advantage of:

- Turning case-sensitivity off or on from within the matching pattern itself, so that we could search for the word "FOO" in uppercase only, to be followed by "bar" in uppercase or lowercase
- Matching any number of a repeating character or expression — for example, match "foo" followed by any number of blanks (including, perhaps, zero) followed by "bar"
- Returning to the matching requestor the exact string that matched the pattern
- Selectively matching from sets of characters — to match an ABAP identifier, for example, one must find a string of nonblank characters that begin with a letter, followed by a string of alphanumeric characters (with the addition, possibly, of "." and "_")
- Matching a string of characters *only* if it appears at the start or end of a line

In summary, we need a search/replace facility that offers all the capabilities already mentioned while maintaining two additional overarching characteristics:

1. It does not fail to find the records/lines we *are* searching for.
2. It does not find records/lines that we *are not* searching for.

Fortunately, a time-tested pattern matching technology already exists that provides all of these features, has the requisite characteristics, and more: the use of *regular expressions* to define matching patterns, and matching engines that operate on those patterns to

find matching lines in a set of text lines. In the next section, I'll briefly introduce the concept of regular expressions. (Readers already familiar with this subject may wish to skip this section and move directly to the section titled "Regular Expression Pattern Matching in the SAP Environment," where I describe some useful rudimentary implementations of regular expression pattern matching in the world of ABAP development.)

Regular Expressions: A Brief Introduction

A *regular expression*[4] (also referred to as a *regexp*, *regex*, or *regxp*) is a single string that describes a set of strings according to a collection of syntactical rules; regular expressions (*regexes*) comprise a powerful notation for describing string-matching *patterns*. A "match" is the sequence of bytes or characters that corresponds to the regex pattern, as determined by the regex processing software (or "engine").

While regular expressions have not yet been fully embraced by the SAP/ABAP community, they have gained wide acceptance in all kinds of text searching scenarios (both within and without the Unix community), and have been incorporated into important standards, such as POSIX. It will take a relatively small investment of your time to gain a useful knowledge of regexes — this investment will pay you (and your installation) back many times over.

Regular expressions are not a tool in and of themselves, and their use is generally implemented as part of a larger utility. A classic example is the Unix-based "grep" tool. These days, provisions for using regular expressions can be found everywhere, including:

- Scripting languages (JavaScript, Perl, Tcl, awk, and Python)
- Editors (Emacs, vi, and Nisus Writer)
- Programming environments (Delphi, Visual C++, and SAP NetWeaver Developer Studio)
- Specialized tools (grep, egrep, lex, Expect, and sed)
- Configuration file processors and mail filters

While many of these tools originated in the Unix world, they are now available for a wide variety of platforms, including Microsoft Windows. Additionally, many favorite programming languages (including Java) offer regular expression libraries, so that programmers can include support for regex pattern matching in their own programs.

Most modern text editors contain regex capabilities. As of this writing, however, SAP has not included this functionality in its ABAP Workbench and utility program-searching repertoire.

So, what is a regular expression? It is a text string used as a search argument or matching pattern. It is typically used in searching situations where you are examining a text file, and want to locate every record (line) in the file containing a string of characters that meets your particular search requirements. Regexes define a *set of strings of characters* that constitute a satisfactory match. Regular expression searching, therefore, is done on a character-by-character basis (see the sidebar on page 73 for more on this).

Unfortunately, in many search situations, it is not enough to provide the search engine with a simple string of characters to be found in the text file. We may wish to find, for instance, all lines in a program text file that contain a "token" — a string of characters delimited on the left by whitespace (any combination of tabs and/or blanks) or by the beginning of the line, and on the right by whitespace or by the end of the line — of the form `C_FULL_PRICE` or

[4] For those who have a burning desire to know, they're called "regular expressions" stemming from the work of an American mathematician by the name of Stephen Kleene, who developed regular expressions as a notation for describing what he called "the algebra of regular sets." Kleene's work eventually found its way into some early efforts with computational search algorithms, and from there to some of the earliest text-manipulation tools on the Unix platform (including "ed" and "grep"). In the context of computer searches, the asterisk ("`*`") is formally known as a "Kleene star." (Source: Stephen Ramsay, at http://etext.lib.virginia.edu/helpsheets/regex.html)

C_DISC_PRICE. To complicate our searching requirements even further, we may know already that many lines of the file we wish to search contain the token C_NET_PRICE, which we are not interested in. Clearly, a simple ABAP search string of C_*PRICE will not do the job.

Regular expression pattern strings may contain special characters (*metacharacters*) that drive a powerful pattern matching algorithm, and allow sophisticated searches to be defined using a single string and a single call to the searching/matching program. In the scenario described in the preceding paragraph, metacharacters and search text have been combined into the following regex to produce the required results:

```
(\s+|^)C_(FULL|DISC)_PRICE(\s+|$)
```

Ignoring the rest of this regex for a moment, the part that reads FULL|DISC limits satisfactory matches to only those lines containing FULL or DISC nestled between C_ and _PRICE (and hence those reading C_NET_PRICE will be ignored). Bear in mind that in its current form, this regex will also match a C_FULL_PRICE string that is part of a comment, which may not be what's intended (or required) if we were searching a program file.

As an additional simple example, suppose you were examining a text file and wanted to identify every record in the file that contains a token representing the number zero. Bear in mind that a simple search for the character "0" may match many unwanted lines, and that the data could contain "0", "0.00", "000", "00.000", "0.", or perhaps some other form of zero. It could also contain "50", "1.00", ".01", or other strings that we *don't* want to match. The regular expression:

```
(\s+|^)(00*(\.0*)?)(\s+|$)
```

could be used to minimize the return of unwanted lines from your search.

A regular expression can be as simple as a single literal character: to search for lines containing a "Z", the regex would simply consist of the character "Z". Or, the regex could consist of a string of characters: to search for the string "Houston", the regex would simply consist of the string "Houston". Generally, however, such simple expressions are usually subexpressions of a more complex regex. The power of regular expression pattern matching derives from the use of combinations of simple search strings with special characters to provide flexibility and power in the search argument in order to (a) match all *required* lines, and (b) refrain from matching *unrequired* lines. In the sections that follow, I'll describe basic regex expression building blocks;[5] bear in mind that an "expression" may mean a single character, a group of characters, or a class of characters.

Special Characters (Metacharacters)

Regular expressions give special meaning to certain characters. Some of these characters are operators, and some show grouping. Other characters allow for the inclusion of nonprinting characters (such as the tab character) within a regex. All other characters simply represent themselves, as they would in an ordinary search string.

Characters in regular expressions that have a special meaning are called *metacharacters*. **Figure 1** lists many commonly used metacharacters and gives their usual meanings.

Escape Sequences

There will be times when you'll want to search for strings containing regex metacharacters without ascribing any special meaning to the character that its use in the regular expression would normally denote. For example, you may wish to look for the occurrence of an actual period in some string. To prevent the regex engine from treating the period in the regex as

[5] Only a basic subset of regular expression constructs is discussed here. For a more complete description of the behavior of regular expressions, see *Mastering Regular Expressions, 2nd Edition*, by Jeffrey E. F. Friedl (O'Reilly and Associates, 2002). Many Web sites also contain useful introductions to regular expressions, including http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html and www.regular-expressions.info.

Figure 1 Commonly Used Metacharacters and Their Meanings

Character	Meaning
`.` (period)	Matches any single character, except the newline character (this exception can be overridden by other matching options).
`*` (asterisk)	Matches any number (including zero) of occurrences of the expression that immediately precedes it. The expression can be a single character, a group of characters, or any character from a class of characters. (See the note on the next page.)
`+` (plus sign)	Matches one or more occurrences of the expression that immediately precedes it.
`?` (question mark	Matches zero or one occurrence of the expression that immediately precedes it.
`[` and `]` (opening and closing brackets)	Define the beginning and end, respectively, of a character class, which specifies a matching set of characters.
`(` and `)` (opening and closing parentheses)	Define the beginning and end, respectively, of a group of expressions; parentheses group expressions into larger units, and dictate precedence.
`|` (vertical bar)	Alternative specification that matches everything to the left of the operator, or everything to the right. (If you want to limit the reach of the alternation, you will need to use parentheses to indicate grouping and precedence: `a(bcd|efg)h` will match `aefgh` or `abcdh`.)
`$` (dollar sign)	Matches the end of a line.
`^` (caret)	Has one of two possible meanings in regular expressions: • Matches the beginning of a line. • When used as the first character in the specification of a character class, acts as a complement character — the expression then denotes the class of characters not included in the class specification.
`\` (backslash)	Used in escape sequences for escaping metacharacters and nonprinting characters, such as the tab character.
`\i` (backslash followed by some number)	Represents the i-th "capturing group." Capturing groups are numbered by counting their opening parentheses from left to right. (Group zero always stands for the entire expression.) In the expression `A((B)(C(D)))`, for example, five such groups are defined: Group 0: `A((B)(C(D)))` Group 1: `((B)(C(D)))` Group 2: `(B)` Group 3: `(C(D))` Group 4: `(D)` Capturing groups are so named because, during a match, each sub-sequence of the input sequence that matches such a group is saved. The captured sub-sequence may be used later in the expression, via forward or back references, and may also be retrieved from the matcher once the match operation is complete, facilitating replacement implementation.

✓ *"foo*" Matches "fox"*

When dealing with regular expressions, always bear in mind that the metacharacter "" has a different meaning than you may be used to. In other contexts, "*" often means "any string of characters"; in the regex context, "*" means "zero or more repetitions of the preceding regular expression."*

How might this be confusing? We may mistakenly write (in a regular expression) "foo", meaning that we wish to match the letters f-o-o followed by any other string of characters. While this may be appropriate for searching for Microsoft Windows file names, it has a distinctively different meaning in the regex context: it means "the letters f-o followed by zero or more occurrences of the letter o." It would match, therefore, not only foo itself and the foo in "foobar," but also the f-o in "fox" and "fo" (among others), and the f-o-o-o in "foooey!".*

a regular expressions metacharacter, you will need to *escape* the period in the regex — this is done by preceding it with a backslash. The search string to search for a literal period then becomes:

```
\.
```

All metacharacters are escaped the same way — by preceding their use in the search string with a backslash. **Figure 2** is a list of additional escape sequences supported by many search engines that implement regular expressions.

Figure 2 Escape Sequences

Escape	Meaning
`\n`	Newline character(s); generally, <CR><LF> or <LF>
`\t`	Tab
`\b`	Backspace (Ctrl+H) when used inside a character class (otherwise, an anchor)
`\r`	Carriage return
`\f`	Form feed
`\nnn`	Octal value between 0 and 0377
`\xnn`	Hexadecimal digit between 0x00 and 0xff
`\c`	The literal character "c", where c can be any character

Matching a Character

The basic matching unit of a regular expression is a single character. You can match a single character in one of three ways:

- **Literally:** Use the character itself (or an appropriate escape sequence) in the search string.
- **Ambiguously:** If matching the character literally is too limiting for your search situation, you may match ambiguously by using the period (" . ") metacharacter, which matches *any* character.
- **Using a character class:** If a literal character provides too limited a search, and an ambiguous match is too broad, a character class, which we'll discuss next, can be used to specify a matching set of characters.

Character Classes

A *character class* is specified using a set of characters enclosed between square brackets. It denotes a set of characters, *any one of which may match*. For example, the character class:

```
[AEIOUYaeiouy]
```

matches any vowel, either uppercase or lowercase.

Many metacharacters may be used in character classes without escaping. Exceptions are the opening and closing square brackets denoting a character class — if you want the character class to include square brackets, you must escape them using a backslash, as in:


```
[ABC\[]
```

Ranges of characters may be specified within the character class. This is done by placing a dash between the first character of the range and the last. Thus:

```
[0-9a-c]
```

will match any character that is one of the numbers from 0 to 9, or the lower case a, b, or c.[6]

The caret ("^") character has a special meaning when it appears as the first character of a character class — it *complements* the class. It is a shorthand way of saying, "Match any character *except* for the following," and avoids the necessity of specifying a large character class. When the caret appears in any position within the class other than the first, it merely adds the caret character to the class. For example:

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

matches anything *except* the 11 characters (which includes the caret character itself) following the caret.

Some character classes are "predefined." There are shorthand ways of referring to such classes:

- `\d` represents a digit; equivalent to `[0-9]`.
- `\D` represents a nondigit; equivalent to `[^0-9]`.
- `\s` represents a whitespace character; equivalent to `[ \t\n\x0b\f\r]`.
- `\S` represents a nonwhitespace character; equivalent to `[^\s]`.
- `\w` represents a word character; equivalent to `[A-Za-z_0-9]`.
- `\W` represents a nonword character; equivalent to `[^\w]`.

[6] How would one match a dash, then? If it's not in the character range you specify, just place it at the beginning or end of the character range, so that it's not between the start and end characters of the range. For example, `[0-9-]` will match any of the characters 0 through 9, or the dash character itself.

Anchors

Anchors set the position in the line where you want subsequent regex matching to take place:

- `\b`, when used outside of a character class, indicates a word boundary, and matches the position between a `\w` and a `\W`.
- `\B` indicates a position that is *not* a word boundary.
- `$` indicates the end of a line.
- `^`, when used outside of a character class, indicates the beginning of a line.

Beginning of Line; End of Line

You can use the metacharacters "`^`" and "`$`" to qualify your regular expression. We've already seen how the caret may be used to complement a character class, but it has another, quite different use outside of character classes. These metacharacters specify that, in order to match your regex, the text must appear at the beginning of a line ("`^`") or at the end of a line ("`$`"). Unlike the iteration qualifiers, however, these metacharacters can stand alone as regular expressions. That is, you can search for just the start of a line, or just the end of a line:

- `^Houston` searches for the characters "H-o-u-s-t-o-n" (Case-specific! In that order! Remember, regular expression matching is done on a character-by-character basis!) occurring at the start of a line.
- `\)$` searches for a right parenthesis that is at the end of a line (and hence not followed by any whitespace).
- `\)[ \t]*$` searches for a right parenthesis and

any following whitespace (some mixture of tabs or blanks, if any) at the end of a line.

Iteration Qualifiers

Iteration qualifiers[7] are metacharacters that are not regular expressions by themselves. Instead, they state how many iterations of the preceding expression there must be, or can be, in order to match. Commonly used iteration metacharacters are "`*`", "`+`", and "`?`". Referring to Figure 1, note that "`*`" matches any number of occurrences, *including zero*, of the preceding expression; "`+`" matches one or more occurrences of the preceding expression; and "`?`" matches exactly zero or one occurrence of the preceding expression. Therefore, if we're trying to find "Houston" preceded by one or more blanks, we could write our regex as "`  +Houston`", where the blank character is repeated one or more times (indicated by iteration factor "`+`"), followed by an "H", followed by an "o", followed by a "u", and so on.

Here are some examples of how iteration qualifiers might be used to match a pattern to an arbitrary length of whitespace, which may consist of any number of intermixed tabs and blanks:

- `\ *` will match any number of consecutive blanks, including none. At first blush, it may not seem useful to match none of something, but as we'll see, as part of larger expressions, it often becomes quite useful. For purposes of matching whitespace, however, the next example might be more useful.

- `\ +` will match one or more consecutive blanks. The "`+`" says, "match one or more of the previous." However, to match whitespace, we also need to match tabs, newlines, carriage returns, and form feeds. The problem is complicated by the fact that we don't know how many tabs, blanks, etc., there might be, and we don't know the order in which they may occur. We might attempt to use `\t*\n*\r*\f* *`, but that would only match if all the tabs (if any) preceded all the newline characters (if any), which preceded all the carriage returns (if any) … well, you get the picture. This application calls for a character class, as the next example illustrates.

- `[ \t\r\n\f]+` uses a character class (denoted by delimiting square brackets) containing a space, tab, carriage return, newline, and form feed. The "`+`" following it means that the expression will match any combination of the characters in the brackets, in any order, so long as there is at least one of them. Suppose we wanted to match whitespace at the beginning of a line if it's present, and otherwise just match the beginning of the line. Then we'd need to use something like the next formulation.

- `^[ \t\n\r\f]*` will match an arbitrary length (including a length of zero, a.k.a. no whitespace) of whitespace at the start of the line. The "`*`" iteration qualifier says, "match zero or more repetitions of the preceding," and the leading "`^`" says, "starting at the beginning of the line."

- `^[ \t\n\r\f]*Address:` could be used to match a line, provided it contained the string "Address:" at either the start of the line, or following an arbitrary length of whitespace at the start of the line.

- `^\s*Address:` uses the "`\s`" metacharacter (supported by many matching engines) to give us a shorthand way of writing the character class shown in the previous example.

There are many other qualifiers.[8] For instance, `regex{`*`n`*`}` specifies that expression "regex" is to be matched exactly *n* times; `regex{`*`n`*`,}` specifies that "regex" is to be matched at least *n* times; and `regex{`*`n`*`,`*`m`*`}` means that "regex" is to be matched at least *n* times, but not more than *m* times.

[7] Also known as *quantifiers*, *iteration factors*, *replication factors*, or *replication qualifiers*.

[8] See http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html for a description of many of these.

Regular Expression Matching Is Done on a Character-by-Character Basis!

Always keep in mind that when we specify a regular expression search string, we are setting out a *string of characters that must be matched on a character-by-character basis, in sequence, left to right*, in order for the line of text to match.

Suppose we're looking for lines in a text file that contain duplicate words. We might start by composing this regular expression to find a word in a record from our file:

```
\b\w+\b
```

This is our intent with this regex: position ourselves at a character that's at a word boundary ("`\b`") , find a string of one or more word characters ("`\w+`", which will include the character at the boundary position), and stop matching when we're positioned at another word boundary (the second "`\b`").

Suppose this is the record from our text file:

```
abc, def, (def), abcd
```

What will the regex match?

The regex matches "`abc`", which is the first, longest string matching the regex. But we're interested in finding subsequent matches to the word just found. If we surround that expression in parentheses, we will have created a *capturing group* (refer back to Figure 1), and its contents can be used later on in the matching process:

```
(\b\w+\b)
```

Since we're looking for lines with duplicate words, we modify the regex as follows:

```
(\b\w+\b)\1
```

(continued on next page)

Alternatives and Grouping

Alternatives and *grouping* go hand in glove. Alternatives often rely on grouping, and the use of alternatives forms one of the major needs for grouping.

Alternatives use the metacharacter "|" to denote that a match has been found if the text matches either of two regular expressions. The alternative operator may be repeated to indicate that the text may match any of several expressions. Grouping occurs when parentheses are placed around an expression that is part of a larger expression.

Alternation applies to entire expressions, if not limited by grouping.

Here are some examples of alternation:

- `Smith|Jones` searches for a line containing the string "Smith" *or* the string "Jones".
- `(Bill|Bob) (Smith|Jones)` searches for "Bill Smith" or "Bill Jones" or "Bob Smith" or "Bob Jones".

(continued from previous page)

where the "\1" (a "back reference") says we're looking for a string identical to the word that was matched by "(\b\w+\b)", and follows that word. Well, we won't find a second "abc" word (and that's a good thing), but we won't find the duplicate "def" words either (and that's a bad thing!).

We've asked the engine to isolate a word in capturing group 1, and if the record contains an identical string following the matched part of the record (the "abc") then we've got a match — we've found duplicate words. But it doesn't — in this example, ", def…" follows the "abc" word, not another "abc" word. Since the engine does its testing on a character-by-character basis, when it gets to the "," of the ", def…", it gives up and concludes that there is no duplicate "abc" immediately following the first "abc". And, by the same token, it won't recognize the duplicate "def", either.

So let's revise the regex one more time, to:

```
(\b\w+\b).*\1
```

Now we're asking the engine to find a word (the capturing group 1 data part of the matched data) followed by *anything* (including nothing!), followed by another string identical to that in capturing group 1. This should allow the engine to explore all possibilities, including those where capturing group 1 and its subsequent matching string are separated by some other intervening characters (in this case, ",").

Success — we found a match! But wait a minute — the engine reports that "abc" is a duplicate word, and it's not — what happened? Capturing group 1 contained "abc", and the engine found a second "abc" *string*. How can we tell the engine we want to find a second *word* that's equal to the one found in capturing group 1? Let's rewrite the regex to:

```
(\b\w*\b).*\b\1\b
```

Success at last! Now the regex results in finding duplicate "def" words, which is what we've been looking for all along.

Even the simplest regex can sometimes return unexpected results — you should keep the "character-by-character" concept firmly in mind as you deal with such situations.

Special Constructs

The power of regexes lies in their ability to describe highly complex matching operations in a pattern consisting of a single string of characters. For instance, the regex may contain a *back reference*, which is a reference to a specific group of characters already matched. Some matching engines also provide the capability to turn case-sensitivity on or off within the pattern string itself.

Here is one of these special constructs. It is called the *zero-width negative lookahead*, and is written as "(?!regex)". Here's how it works. The quantifier:

$\texttt{regex}_1\texttt{(?!regex}_2\texttt{)}$

returns a match condition if a match for $\texttt{regex}_1$ is found, but it is *not* followed by a match for expression $\texttt{regex}_2$. (It is called a "zero-width" operator because if it results in a match, no text satisfying the negative

matching for `regex2` is returned.) For example, the pattern "`q(?!u)`" will not match the "q" in "quickly", but will find a match in "qick brown fox…"

As a more complex example of the use of negative lookahead, if we wanted to find "q" *words* that were apparently misspelled, we might use a regex like this:

```
\b([Qq](?![Uu])\w+)\b
```

which will return the offending word to us via capturing group 1 (refer back to Figure 1).

Here are other (but by no means all other) useful special constructs:

- `regex1(?=regex2)` captures `regex1` if `regex1` matches and it is followed anywhere by a match for `regex2`.
- `regex1(?<=regex2)` captures `regex1` if `regex1` matches and it is preceded anywhere by a match for `regex2`.
- `regex1(?<!regex2)` captures `regex2` if `regex1` matches and it is not preceded anywhere by a match for `regex2`.

✓ *One Is Better Than None*

A common problem in creating useful regular expression patterns is using "." where it is not really appropriate. The ".*" expression says, "any number (including zero) of [any] character." The trouble is, this means that it will also match zero occurrences of no characters. This may often not be what you want, and can cause the search itself to fail, or even worse, get into an infinite loop. (There are an infinite number of zero occurrences of no characters on every line!)*

A better practice is to use ".+" rather than "." whenever possible, which says, "one or more of any characters." This is probably what you want, anyway.*

✓ *Specificity Limits Ambiguity*

When you create expressions using the "" and "+" replication operators, you lose some control over what they will really match. For instance, if the regex is "a+", what string from the following line results in a match?*

```
xaaaabc
```

Does it match "a", "aa", "aaa", or "aaaa"? If "aaa", which occurrence of "aaa" (since there are two of them)?

Whether a matching engine returns the first match, the last match, the leftmost longest match, etc., is very much a function of the engine implementation.[9] The POSIX standard dictates, for instance, that it will match the leftmost, longest matching string (which may not be, for your particular purposes, the "best" match). It is generally better to place expressions that use the "" and "+" qualifiers between expressions that don't employ them. For example, if we rewrote the regex above to read:*

```
xa+b
```

only the string "xaaaab" would match.

Some [Erstwhile] Simple Data Entry Validation/Extraction Scenarios: Reprise

Armed with a little regex knowledge, let's revisit some of the problems posed earlier, and see how regular expressions might be used to reduce ABAP coding requirements.

[9] For detailed discussions on matching engine behaviors, a Google for any combination of "DFA," "NFA," "Regular Expression Engine," "eager," "greedy," "backtracking," or "backreference" will point you to many sources. Friedl's *Mastering Regular Expressions, 2nd Edition* also contains a wealth of information regarding engine behavior.

Figure 3 Regular Expression for Validating a Phone Number

```
^ *(?:(?<1>[(])?(?<AreaCode>[2-9]\d{2})(?(1)[)])(?(1)(?<2>[ ])|(?:(?<3>[-
])|(?<4>[ ])))?)?(?<Prefix>[1-9]\d{2})((AreaCode)(?:(?(1)(?(2)[- ]|[-
]?))|(?(3)[-])|(?(4)[- ]))|[- ]?)(?<Suffix>\d{4}) *$
```

Figure 4 Regular Expression for Matching Multiple SSN Input Formats

```
(^|\s)(00[1-9]|0[1-9]0|0[1-9][1-9]|[1-6]\d{2}|7[0-6]\d|77[0-2])(-?|[\. ])([1-
9]0|0[1-9]|[1-9][1-9])\3(\d{3}[1-9]|[1-9]\d{3}|\d[1-9]\d{2}|\d{2}[1-
9]\d)($|\s|[;:,!\.\?])($|\s)
```

Phone Numbers

As I stated earlier, validation/parsing of phone numbers is notoriously difficult. Given the problem parameters stated earlier, **Figure 3** shows a regex that will validate an input string to see if it contains a phone number and will return (via capturing group "AreaCode"[10]) the parsed out area code.

US Social Security Numbers (SSNs)

Manual entry of SSNs can cause all sorts of problems, including invalid area values. The regex in **Figure 4** will match multiple input formats, and will honor a current Social Security Administration restriction that the highest area currently assigned is "772".

This pattern will match input such as:

```
123-45-6789
123 45 6789
123456789
772-12-1234
```

It will not match:

```
1234-56-7890
123-45-67890
773-12-1234
```

[10] Note that not all regex engines support the use of named capturing groups.

URLs, Email Addresses, and IP Addresses

Email addresses can be particularly difficult to validate, as they can contain an almost unlimited combination of characters. The regex in **Figure 5** (which uses conditionals, supported by PERL 5 and similar regex engines) was found at www.regexlib.com/DisplayPatterns.aspx, and has been modified to remove its .NET framework engine dependence. It does an admirable job of figuring out whether a string of text consists solely of an email address.

Note that it does *not* provide for the return of specific parts of the address (domain, display name, local name, etc.).

Currency Data

There are many different ways to enter currency, including negative amounts, separators, and currency symbols, which makes validating currency data quite challenging. Try the regex shown in **Figure 6** when validating file (or user, for that matter) input to establish that a valid US currency amount has been entered. This regex allows for an optional leading "$"; an optional leading/trailing "-" or "()" (one or the other, not both) to indicate the amount is negative; optional cents; and optional commas separating thousands (either all thousands must be separated by commas, or none at all). A "-" (if present) can be before or after

Figure 5 Regular Expression for Identifying an Email Address

```
^((?>[a-zA-Z\d!#$%&'*+\-/=?^_'{|}~]+\x20*|"((?=[\x01-\x7f])[^"\\]|\\[\x01-
\x7f])*"\x20*)*(<))?((?!\.)(?>\.?[a-zA-Z\d!#$%&'*+\-/=?^_'{|}~]+)+|"((?=[\x01-
\x7f])[^"\\]|\\[\x01-\x7f])*")@(((?!-)[a-zA-Z\d\-]+(?<!-)\.)+[a-zA-Z]{2,}|\[(((?
(?<!\[)\.)(25[0-5]|2[0-4]\d|[01]?\d?\d)){4}|[a-zA-Z\d\-]*[a-zA-Z\d]:((?=[\x01-
\x7f])[^\\\[\]]|\\[\x01-\x7f])+)\])(?(3)>)$
```

Figure 6 Regular Expression for Validating Currency Input

```
^\$?\-?([1-9]{1}[0-9]{0,2}(\,\d{3})*(\.\d{0,2})?|[1-9]{1}\d{
0,}(\.\d{0,2})?|0(\.\d{0,2})?|(\.\d{1,2})?)$|^\-?\$?([1-9]{1
}\d{0,2}(\,\d{3})*(\.\d{0,2})?|[1-9]{1}\d{0,}(\.\d{0,2})?|0(
\.\d{0,2})?|(\.\d{1,2})?)$|^\(\$?([1-9]{1}\d{0,2}(\,\d{3})*(
\.\d{0,2})?|[1-9]{1}\d{0,}(\.\d{0,2})?|0(\.\d{0,2})?|(\.\d{1
,2})?)\)$
```

"$", but "()" indicating negative value must enclose the dollar sign.[11]

The expression in Figure 6 would find matches in:

```
123.4
.42
-123.45
-$.10
123,456.78
```

but not in:

```
123.456
(-1.23)
.567-
123-456.78
```

[11] This is an example of a fine regex; it was written by Gregg Durishan and can be found at www.regexlib.com. It is also an example of a regex that is relatively hard to decipher, and hence is a good candidate for study using one or more of the regex tools identified in the sidebar "Debugging/Checking Regular Expressions" on page 79.

File Scanning

Remember that interface data can arrive corrupted, and in various formats that are difficult to parse. Here's how regular expressions can help.

Corrupted Data

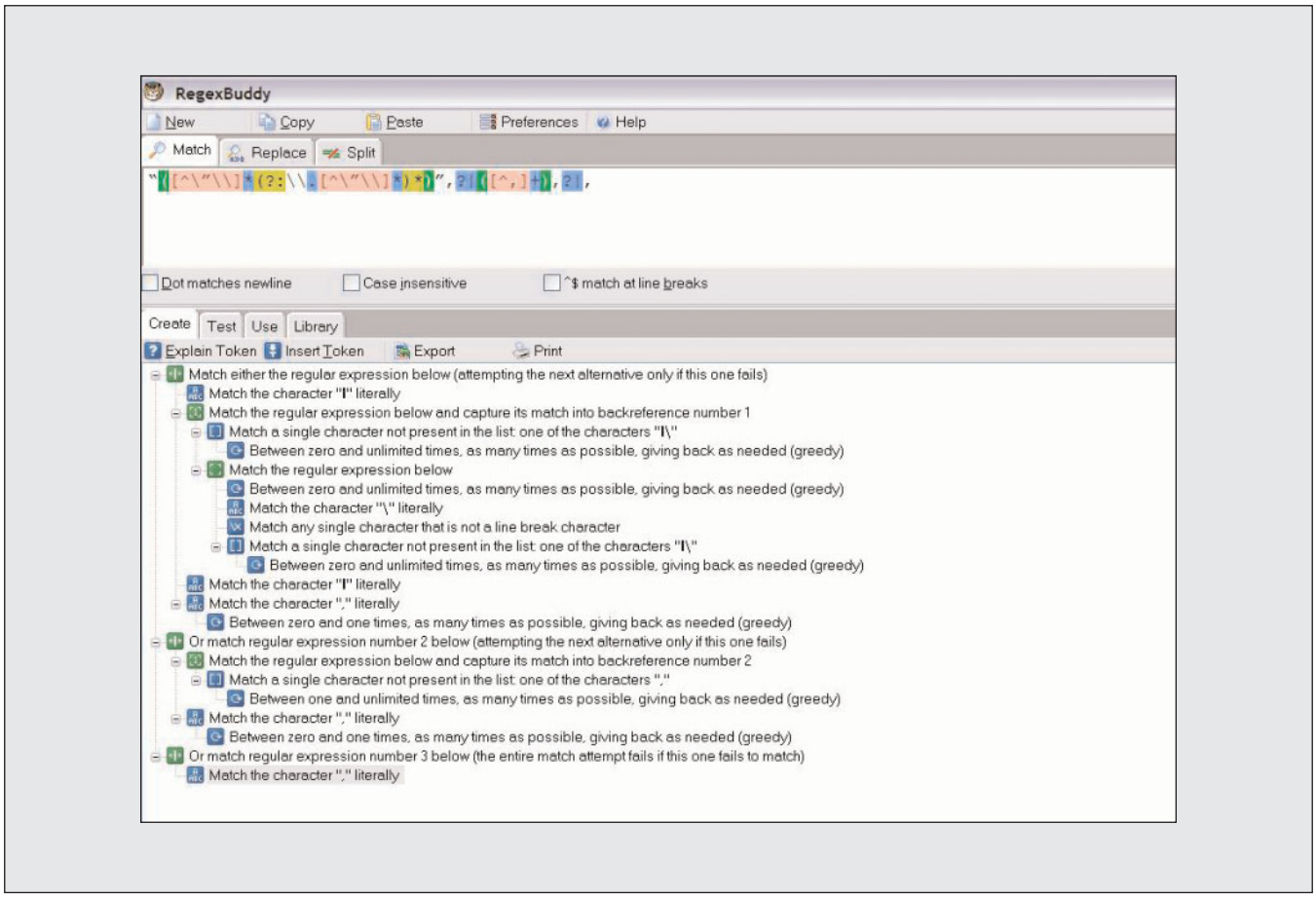
The following regex will determine whether an input line contains a string of one or more control characters, and return its location and length (via capturing group 1):

```
([\x00-\x1F\x7F]+)
```

Field Parsing

Provided the data is free of errors like unmatched double quotation marks or missing fields, the following

Figure 7 Parsing a Regular Expression Using the RegexBuddy Program



simple regex can be used repetitively to return succeeding fields from comma-separated data values, where any field can contain string data delimited by double quotation marks:

```
"([^\"\\]*(?:\\.[^\\"\\]*)*)",?|([^,
  ]+),?|,
```

Parsing a regex mentally can sometimes be daunting — **Figure 7** shows (in the Create pane) a parse tree of this regex generated by the RegexBuddy program (see the sidebar on the next page).

Figure 8 shows a part of the parse tree for the example regex in a different format, generated by the Regex Coach program (also see the sidebar on the next page).

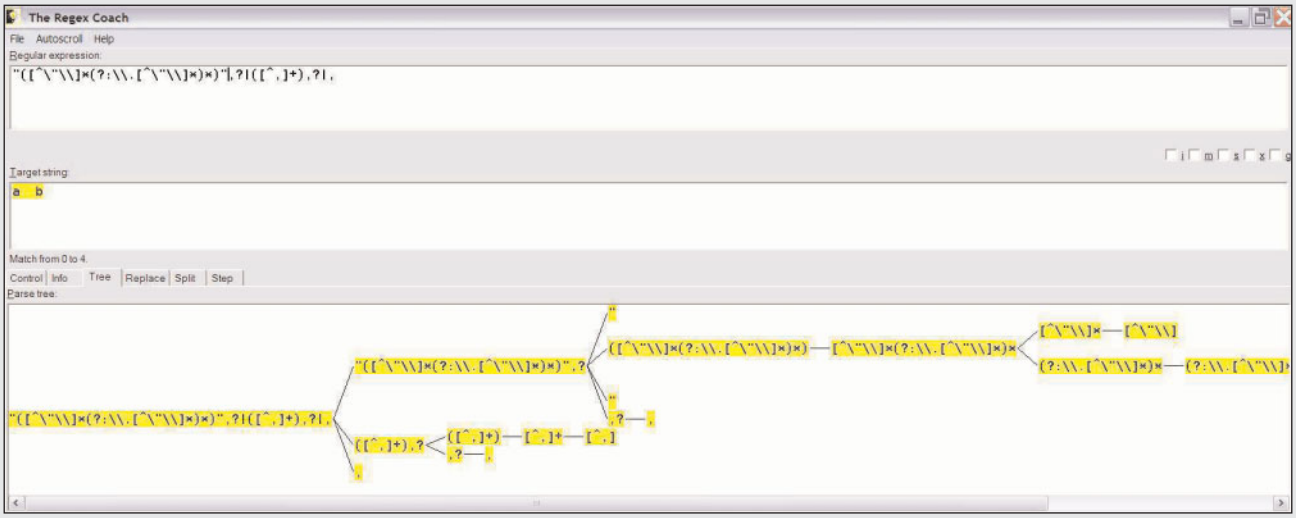
Regular Expression Pattern Matching in the SAP Environment

Assuming that you are convinced that the use of regular expressions could ease your ABAP development load, you are probably wondering how to best incorporate them into your SAP environment today, in anticipation of the mainstream support to be provided by SAP NetWeaver. In the next sections, I'll discuss a number of alternatives for doing just that.

Implementation Alternatives for a Regular Expression Pattern Matcher

One alternative for regex pattern matching is to

Figure 8 Parsing a Regular Expression Using the Regex Coach Program



Debugging/Checking Regular Expressions

There are a number of Internet sites that make available (most at no cost) quite useful facilities for entering regular expressions and testing these against sample text. Some of these run online, while others involve the installation of downloaded software. The facilities provided present varying sets of capabilities, ranging from a simple test of a regex against a single string of text, to graphical representation of the regular expression's parse tree and an attempt to describe the regex in English. Among those that you may want to investigate are:

- Visual REGEXP (requires Tcl) — http://laurent.riesterer.free.fr/regexp/
- Regex Tester — www.monster-submit.com/resources/regex/
- Online Regular Expression Syntax Check — www.bitesizeinc.net/demo.regexp.html
- The Regex Coach (extremely cool!) — http://weitz.de/regex-coach/
- RegexBuddy (tres cool; parsed regex tree display) — www.regexbuddy.com*
- Regular-Expressions.info (tutorial) — www.regular-expressions.info/tutorial.html

* Utilizes PCRE (Perl Compatible Regular Expressions).

Figure 9 Calling the ABAP Regex Pattern Matching Function Module

```
CALL FUNCTION 'Z_JS_REGEX_MATCH'
     TABLES
        t_statements    = <table of text lines to be searched>
     CHANGING
        search_string   = <pattern>
        flags           = <regular expression modifiers>.
```

implement a matching engine yourself. I would argue strenuously against this, for a number of reasons:

- There are a number of subtle functional and performance implications for any implementation that will require significant development effort.
- Regular expression pattern matching has been previously implemented successfully in many languages on differing platforms — you should be able to take advantage of these to satisfy your own particular installation requirements.
- There are a number of regex "languages," and the determination of which to implement will require significant effort. For instance, a complex language (such as POSIX) is rich in features, but requires implementations that can have a horrible worst-case performance.

Other alternatives include using one of the many public domain implementations on the Internet, and to deploy one of these as an RFC, callable from ABAP. For instance, it is not difficult to find a C regular expression matching engine that can be compiled and deployed onto your application server, so that it can be called as an RFC. (Note that Java now has regex class support that can be deployed on a server and called via an RFC.) One of the drawbacks of this approach is that it may incur maintenance costs to embrace functional changes in regex processing as they occur in the future. Another drawback is the potential performance overhead incurred by RFC processing.

As a stopgap measure (in anticipation of SAP's native regular expression implementation), a better alternative is to take advantage of the JavaScript interpreter included as part of the SAP Web AS 6.20 kernel. JavaScript 1.2 and later provide fairly extensive support for regular expression processing as part of `String` and `RegExp` classes; this support is made accessible in ABAP through the use of a number of SAP-supplied ABAP classes.[12] In the next section I'll discuss the implementation of an ABAP function module you can use to access JavaScript regex functionality.

Implementing an ABAP Regex Pattern Matching Function Module

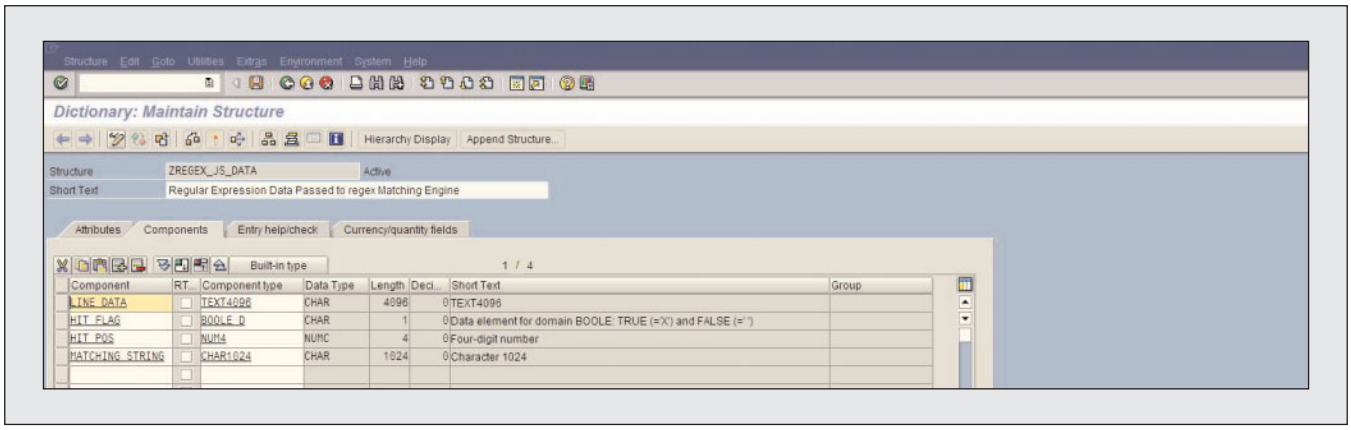
The appendix to this article contains the complete code of a prototypical ABAP function module that can be called to perform regex pattern matching against a table of text lines. Calling this function requires only the ABAP statement shown in **Figure 9**.

Arguments to the function have the following definitions:

- `search_string` is a regular expression matching pattern of type `STRING`.
- `flags` is a matching modifier of type `STRING`, which is a string of up to four characters (in JavaScript) that can be used to modify the matching operation. Probably the most useful of these is the "i" modifier, which dictates that matching will be case-*in*sensitive.

[12] For a detailed discussion of the SAP class support, see the article "Web-Enable Your SAP Applications with the Power of JavaScript" (*SAP Professional Journal*, March/April 2002). For extensive discussions of JavaScript `RegExp` and `String` class regex support, Google for "regular expression Javascript."

Figure 10 The ZREGEX_JS_DATA Structure of the Table



- `t_statements` is a table of lines (type `ZREGEX_JS_DATA`) against which matching is to take place. **Figure 10** shows the `ZREGEX_JS_DATA` structure of the table passed to the matching function.

Components of the `ZREGEX_JS_DATA` structure have the following meaning:

- `line_data` is a line of text against which matching is to take place. (Note that in this implementation, an artificial limitation on source statement length has been set at 4,096 characters; this limit might need to be modified to suit your particular application requirements.)
- `hit_flag` indicates whether a match is found in the line ("X" shows a match was found).
- `hit_pos` is the position of the first character in the matching string (relative to 0).
- `matching_string` is the matched string. (Note that in this implementation, an artificial limitation on the maximum length of any matching string is set at 1,024 characters; this limit might need to be modified to suit your particular application requirements.)

SAP Future Plans

SAP has recognized the benefits of regular expression matching and its applicability throughout multiple areas of SAP processing where text searching is required. To that end, SAP will provide native regex matching in an upcoming SAP NetWeaver release. This functionality will include regular expression support for the ABAP `FIND` and `REPLACE` commands, and perhaps other places where searching takes place. In addition, native classes will be made available that will provide regex support that can be called from custom ABAP, so that you can incorporate regex matching in your own applications. Look for a future *SAP Professional Journal* article that will describe the upcoming native regex facility.

Summary

In this article, I've tried to make a number of major points:

- ☑ Under many circumstances, standard ABAP string matching facilities do not provide the functionality demanded by today's applications, or their use implies difficult ABAP coding exercises.
- ☑ The use of regular expression pattern matching provides facilities for sophisticated searching and replacement that can reduce your requirements for ABAP development resources.
- ☑ You can prepare for SAP's implementation of regex support by implementing your own interim

References

- A. Aho, B. Kernigan, and P. Weinberger, *The AWK Programming Language* (Addison-Wesley Publishing Company, 1988).
- Ben Forta, *Sams Teach Yourself Regular Expressions in 10 Minutes* (Sams, 2004).
- J.E.F. Friedl, *Mastering Regular Expressions, 2nd Edition* (O'Reilly & Associates, Inc., 2002).

facility through the use of JavaScript 1.2 `String` and `RegExp` classes.

- ☑ ABAP coding of a function providing access to SAP Web AS kernel JavaScript functionality is straightforward and simple.
- ☑ The complexities inherent in native regex implementation discourage trying such development yourself. Await with bated breath, as I do, for SAP to provide this functionality as a part of SAP NetWeaver.

This article has barely scratched the surface of regular expression language definition and capability — a future article will explore in depth the native regex capability that will be part of SAP NetWeaver. In the hope that I've piqued your interest sufficiently to pursue the subject further, the sidebar above lists a few sources of additional information on the subject, as will a Google search using "regular expression" as a search argument. Regular expression tutorials abound on the Internet, and multiple news groups are dedicated to this subject. If you're new to this technology, regular expressions may appear at first blush to be a write-only language, but after some practice you'll find that you'll be able to decipher regexes with ease. To smooth that path, there are many useful tools available that allow you to test and debug regular expressions on sample data before putting them to productive use (see the sidebar on page 79 for a sampling).

David Jenkins has been involved with IT since 1957, when he started in the business as a punched card machine operator. Dave worked for a Houston bank for nine years before receiving his B.S. in Math from the University of Houston. He then joined IBM, where he worked in various marketing support positions, supporting contractors at the NASA Johnson Space Center. While at IBM, Dave spent a year teaching at New Mexico Highlands University as part of IBM's Faculty Loan Program. Since leaving IBM, Dave has received a master's in Management, Computing and Systems from Houston Baptist University, and has finished coursework for a Ph.D. in Management Information Systems at the University of Houston. Since 1996, Dave has been a consultant specializing in ABAP development — his most recent client assignment has been at ChevronTexaco, supporting their installation of SAP IS-Oil Production and Revenue Accounting. Dave is married with four children, and two beautiful grandchildren. He and his wife Joy live in the country 80 miles west of Houston, where they enjoy the wide-open spaces and fresh air, and their dog, cats, fish, and especially their colony of purple martins. Dave can be reached at davidfjenkins@earthlink.net.

Appendix: Source Code for Implementing an ABAP Regex Pattern Matching Function Module

```
FUNCTION z_js_regex_match.
*"----------------------------------------------------------------------
*"*"Local interface:
*"  TABLES
*"      STATEMENTS TYPE  ZREGEX_JS_DATA_TABLE
*"  CHANGING
*"     REFERENCE(SEARCH_STRING) TYPE  STRING
*"     REFERENCE(FLAGS) TYPE  STRING
*"----------------------------------------------------------------------
* local javascript data
  DATA:
    l_js_processor       TYPE REF TO cl_java_script,
    l_js_script          TYPE string,
    l_temp_stmt_tbl      TYPE zregex_js_data_table.

* build javascript context
  l_js_processor = cl_java_script=>create( ).

* bind to caller's args
  CALL METHOD l_js_processor->bind
    EXPORTING
      name_obj  = 'abap'
      name_prop = 'js_search_string'
    CHANGING
      data      = search_string.
  CALL METHOD l_js_processor->bind
    EXPORTING
      name_obj  = 'abap'
      name_prop = 't_statements'
    CHANGING
      data      = l_temp_stmt_tbl.
  CALL METHOD l_js_processor->bind
    EXPORTING
      name_obj  = 'abap'
      name_prop = 'flags'
    CHANGING
      data      = flags.
* build js program here
```

```
  CONCATENATE  'var reg = new RegExp( abap.js_search_string, abap.flags );'
               'for (i = 0; i < abap.t_statements.length; i++) {'
               '     var ref_struc = abap.t_statements[i];'
               '     var ar = reg.exec( ref_struc.line_data );'
               '     if (ar) {'
               '        ref_struc.hit_pos          = ar.index;'
               '        ref_struc.hit_flag         = ''X'';'
               '        ref_struc.matching_string  = ar[0];'
               '        }'
               '     }'
       INTO l_js_script SEPARATED BY cl_abap_char_utilities=>cr_lf.

  l_temp_stmt_tbl[] = statements[].

* evaluate
  l_js_processor->evaluate( java_script = l_js_script ).
  IF l_js_processor->last_condition_code <> cl_java_script=>cc_ok.
    WRITE: l_js_processor->last_error_message.
  ENDIF.

  statements[] = l_temp_stmt_tbl[].

ENDFUNCTION.
```

