What are regular expressions and how can I use them in Stata?

Title	Regular expressions
Author	Kevin S. Turner, StataCorp

Regular expressions use a notation system that allows for matching complex patterns of text with minimal effort. While there is no formal standardization of the syntax for a regular expression, there is a general consensus on the basic elements of the syntax. It is this core syntax that Stata implements in its regular-expression functions.

Regular expressions are simply strings that are a mix of literals and operators. For example, if you simply want to test whether a substring of "xyz" exists in another string, you can use the literal "xyz" as your regular expression. Granted, it is not very powerful, but it is a legal expression. Mixing in operators allows you to match more complex patterns. Here are the core operators that Stata's regular expression parser supports:

Counting	
*	Asterisk means "match zero or more" of the preceding expression.
+	Plus sign means "match one or more" of the preceding expression.
?	Question mark means "match either zero or one" of the preceding expression.
Characters	
a–z	The dash operator means "match a range of characters or numbers". The "a" and "z" are merely an example. It could also be 0–9, 5–8, F–M, etc.
.	Period means "match any character".
\	A backslash is used as an escape character to match characters that would otherwise be interpreted as a regular-expression operator.
Anchors	
^	When placed at the beginning of a regular expression, the caret means "match expression at beginning of string". This character can be thought of as an "anchor" character since it does not directly match a character, only the location of the match.
$	When the dollar sign is placed at the end of a regular expression, it means "match expression at end of string". This is the other anchor character.
Groups	
|	The pipe character signifies a logical "or" that is often used in character sets (see square brackets below).
[]	Square brackets denote a set of allowable characters/expressions to use in matching, such as [a-zA-Z0-9] for all alphanumeric characters.
()	Parentheses must match and denote a subexpression group.

Other popular regular-expression syntaxes include the POSIX standard and Perl's standard. Both expand on these basic operators by including counting operators (use of curly braces), metacharacters (usually of the form :alpha:, etc.), and other syntax-specific additions.

When presented with the choice of which regular-expression syntax to adopt, Stata has several options. Different operating systems offer their own regular-expression parsers for applications to use, but there is no guarantee that these parsers are consistent. Stata avoids this ambiguity by using its own parser. In doing this, Stata ensures a level of consistency across computer platforms even if there are some syntax features missing that are present in other regular-expression parsers. Most of these extra syntax elements, however, are not critical and can be represented, albeit in longer form, with Stata's current parser.

Now that we have talked a bit about regular-expression syntax, let's see some examples of expressions to match some common strings.

Example 1: Dates

A common use of regular expressions in general (not just Stata) is in matching time and data information. Let's assume we have a variable with string data of the following form:

```
12jan2003
1April1995
17may1977
02September2000
...
```

Assuming these dates are representative of the whole, we can see that all start with the date as a series of digits. We should always find a digit, and it should always be at the beginning. Knowing that, we can start our regular expression with

```
^[0-9]+
```

Following the day is the month, but it appears the month can be abbreviated or spelled out. Both uppercase and lowercase letters are used. We can look for a series of mixed-case, alphabetic characters next:

```
^[0-9]+[a-zA-Z]+
```

After the month, we have the year, another series of digits. We will read the year's digits until we reach the end of the string, hence the use of a dollar sign.

```
^[0-9]+[a-zA-Z]+[0-9]+$
```

This is not the only way to parse this string. We could have left out the ^ and $ characters that signify the beginning and end, respectively, of the string. However, if we had a string such as "**12Oct1996 4Jun1997**", without the ^ and $ characters, we would have gotten a positive match, but only against the first date. That behavior might be okay for the situation, but it might not.

You might have noticed that we also don't care how many digits we are reading for the year (or any of the parts). If we know we will always be looking for four digits, we could have used the following regular expression:

```
^[0-9]+[a-zA-Z]+[0-9][0-9][0-9][0-9]$
```

Any string that had a two-digit year would not have been matched. How restrictive or loose of a regular expression you use in matching your data is really up to you. Sometimes the data will force you to construct lengthy regular expressions so that you can match exactly what you want.

Although the previous regular expression is perfectly fine for telling us when we are parsing a date, it does not help when it comes to separating out the date, month, and year data components of the string. To do that, we use a regular-expression facility called "subexpressions".

Subexpressions are a way of grouping subpatterns of a regular expression so that the data they match against can be extracted as a substring. For example, if we wanted to retrieve the date, month, and year data, we would have enclosed their respective regular expression parts in parentheses like so:

```
^([0-9]+)([a-zA-Z]+)([0-9][0-9][0-9][0-9])$
```

Note: The expression added below is the same as the one above.

After confirming that this regular expression has matched a string, we could then use certain functions for retrieving the subexpressions of interest. Subexpression one would refer to the date, two to the month, and three to the year. Subexpression zero always returns the entire string matched by the regular expression as a whole. For example, matching against the string `"12jan2003"` would result in the following subexpressions:

Subexpression #	String returned
0	12jan2003
1	12
2	jan
3	2003

We can list the matched observations by

```
list var if regexm(var,"^[0-9]+[a-zA-Z]+[0-9]+$")
```

Example 2: Phone number

Assume we have a variable for phone numbers, and we would like to get the area codes. The data are in various formats:

```
(979) 123-1234
979-121-1231
...
```

We have two different formats that we have to parse. The regular expression will be a little shorter because we are concerned only with getting the area code, but the same principles would have applied if we had been going after the rest of the phone number.

```
^\(?[0-9][0-9][0-9]\)?
```

Breaking down this regular expression, we see the ^ that signifies we want to match at the beginning of the string, and then it is followed by `"\(?"`. The left parenthesis is preceded by a backslash because a left parenthesis, in normal regular-expression syntax, means to start a subexpression, and here we are looking for a literal left parenthesis. The backslash is used to turn any character that would otherwise be a regular-expression operator into a literal character.

The question mark operator denotes that we are looking for either a left or right parenthesis or none at all.

We do not match until the end of the string. We know the area code will always be the first three digits we see, so there is no need to complete the regular expression so that it matches the entire line. In fact, our last regular expression does not even need to test for the last parenthesis if we are confident that the area code is always the first three digits. It could be reduced to simply:

```
^\(?[0-9][0-9][0-9]
```

If we had other data mixed in that started with three digits but did not specify an area code, we would have to construct a more restrictive regular expression to distinguish between the two types of data, if possible.