Making Regular Expressions Your Friends

Michael D. Galloy
Trainer/Consultant
Research Systems, Inc.

In this article:

Language shapes the way we think, and determines what we can think about.

—B. L. Whorf

1. Introduction

Regular expressions are a succinct, powerful syntax to describe a set of matching strings. The notation is a "little language" embedded in IDL and used through the *STREGEX* and *STRSPLIT* routines. This white paper attempts to describe this language and give examples of its use for IDL programmers.

A basic understanding of IDL syntax and an understanding of strings in IDL is required. While regular expressions themselves can be quite complicated and are considered an "advanced" topic, limited knowledge of the rest of IDL is required.

The syntax of regular expressions can pack quite complicated logic into a short string. This makes them powerful for all manners of string manipulation tasks, but requires caution by even the most experienced of users. Small mistakes in the specification can lead to very unexpected results. While regular expressions are more powerful than the simple pattern matching algorithms found in routines like *STRMATCH* and *FILE_SEARCH*, they are also slower.

Regular expressions in IDL are based on the regex package written by Henry Spencer, modified only to the extent required to integrate it into IDL. This package is freely available at `ftp://zoo.toronto.edu/pub.regex.shar`.

The regular expressions used by IDL correspond to Posix Extended Regular Expressions and are similar to the expressions accepted by such varied tools as Perl, Java, Python, the .NET framework, the Apache web server, and a host of UNIX utilities such as vi, egrep, sed, awk, and lex. Using regular expressions is complicated by the arbitrary differences in syntax in the various implementations of these tools.

This paper will stay strictly to the syntax used in IDL, though most examples will work in other tools with a few simple or no modifications. The IDL routines *STREGEX* and *STRSPLIT* use regular expressions; for most of our examples, we will use *STREGEX* to show the results, but check "Extended examples" on page 7 for examples using *STRSPLIT*. Once you master regular expressions in IDL, you may find yourself branching out to other uses. For example, use the UNIX `egrep` command in a C shell to find all ENVI *DOIT* routine calls in a directory full of IDL source code files:

```
egrep -n "[^\'\"]envi_[_[:alnum:]$]+_doit" *.pro
```

Or, to find all the calls of the *MEDIAN* or *MEAN* function:

```
egrep -n '[^[:alnum:]_$](median|mean)\(.*\)' *.pro
```

2. Basics

To perform a search using a regular expression, a string to be searched and a string specifying the search pattern are needed. In the simplest case, the search pattern is a sequence of literal characters which *STREGEX* will attempt to locate. For example, to find the sequence of letters “ik” in the string “Mike,” use:

```
IDL> print, stregex('Mike', 'ik')
           1
```

STREGEX returns the position of the search pattern within the first string or -1 if it doesn’t find it. Setting the boolean keyword EXTRACT would make *STREGEX* return the matching string itself instead of its position.

If there is more than one match in the given string, the regular expression will match the one that starts first in the string. If there are multiple substrings that match starting at the same position, the regular expression will return the longest match (“greedy” matching).

So far, we haven’t done anything that can’t be done with *STRPOS*. In fact, *STRMATCH* can do better. The next section, “Metacharacters”, will describe what gives regular expressions their power.

3. Metacharacters

Most characters, like numbers and letters, match themselves in a regular expression. These are called ordinary characters. A metacharacter is any of the characters that regular expressions do not interpret literally. The metacharacters are `\ [ ] ( ) ? * + { } ^ $ . |`. Metacharacters can be used as wildcards, to match certain groups of characters, to indicate a repeating pattern, and to match things not corresponding to a character such as the beginning or ending of a line.

. DOT

The period matches any character.

```
IDL> print, stregex(['eMike', 'iMike'], '.Mike', /extract)
eMike iMike
```

? * + { } REPETITION CHARACTERS

These are the various repetition characters. They can be used to indicate how many times the character or expression immediately to the left of the operators should be matched. Use parenthesis to indicate the expression that should be repeated if it is more than a single character.

Metacharacters	Matches	Example
`?`	Optional, matches 0 or 1 times	`ab?a` matches `aa` or `aba`
`*`	Matches 0 or more times	`ab*a` matches `aa`, `aba`, `abba`, `abbba`, etc.
`+`	Matches 1 or more times	`a+` matches `a`, `aa`, `aaa`, etc.
`{n}`	Matches exactly n times	`ab{2}a` matches `abba` only
`{m,n}`	Matches from m to n times	`ab{2,4}b` matches `abba`, `abbba`, `abbbba`

Table 9-1: Repetition metacharacters.

```
IDL> print, stregex(['abbcccde', 'bbccccd'], 'a?b{2}c{3,5}d+e*')
           0           0
```

^ $ ANCHOR CHARACTERS

These are the *anchor* characters. They match the beginning (^) and the end ($) of a string. These are zero-width "characters" and nothing is included in the result representing them.

```
IDL> print, stregex(['and two', 'one and two', 'one and', 'and'], '^and')
           0          -1          -1           0
IDL> print, stregex(['and two', 'one and two', 'one and', 'and'], 'and$')
          -1          -1           4           0
IDL> print, stregex(['and two', 'one and two', 'one and', 'and'], '$and$')
          -1          -1          -1          -1
```

The ^ will only produce a match if it is the first (or $ the last) character in the search pattern.

| ALTERNATION

The | symbol represents *alternation*. It specifies choices which are allowable at that point in the regular expression. Use parenthesis to indicate an alternation in a larger regular expression.

```
IDL> print, stregex('Jodie', 'George|Jane|Elroy|Jodie', /extract)
Jodie
IDL> print, stregex('Elroy Jetson', '(George|Jane|Elroy|Jodie) Jetson', /extract)
Elroy Jetson
```

[ ] BRACKET EXPRESSIONS

The [] characters are used to produce a *bracket expression*. These are used to allow a choice of different characters at that point in the regular expression. See "Bracket expressions" on page 3 for more information.

( ) SUBEXPRESSIONS

The () characters are used to indicate a *subexpression*. These are used in *alternation* (see the | metacharacter below) or in extracting subexpressions from a larger matching expression. See "Subexpressions" on page 5 for more information.

**** ESCAPE CHARACTER

The backslash is used to escape the meaning of a metacharacter (including itself). So to match the * character use * as in:

```
IDL> print, stregex('1*2=2', '1\*2', /extract)
1*2
IDL> file = 'C:\RSI\IDL56\examples\data\glowing_gas.jpg'
IDL> print, stregex(file, 'C:\\', /extract)
C:\
```

4. Bracket expressions

Bracket expressions provide a set of characters to match. These can be created from a list of characters, a range of characters, or a set of excluded characters.

The simplest case is a list of valid characters. For example, the following is an example of matching a sequence of one or more vowels:

```
IDL> print, stregex('Fountain', '[aeiou]+', /extract)
ou
```

The normal metacharacters lose their special meaning in a bracket expression, while he characters ^, -, and some combinations with [and] gain a special meaning.

4.1. Range expressions

Instead of listing each character in the character class, a range can be specified by a starting character and an ending character. The ordering is determined by the ascii lookup table (see Table 9-3 for details). So, to get the lowercase letters from `a` to `m`, the range `[a-m]` could be used. Of course, `[%-+]` is also legal, but not recommended. Multiple ranges can be combined to form expressions such as `[A-Za-z]`, which matches any letter. It is illegal to specify a range backwards (with the first endpoint of the range having a higher ASCII value than the second endpoint).

The one tricky thing to do with ranges is to specify a range which includes a character that is special in bracket expressions (ie. `-`, `[`, `]`, or `^`). To do this, use a *collating element* to specify the character of the range. Collating elements are specified by placing the given character within `[.` and `.]`. For example, to specify the range of characters from `-` to `0` (which is `-`, `.`, `/`, and `0`), use `[[.-.]-0]`. Normally, it would be simpler to just use an expression such as `[-./0]` to indicate this range.

4.2. Character classes

A character class is a named listing of characters which can be used in a bracket expression. There are character classes for common sets of characters such as letters, digits, whitespace, etc. There are twelve character classes available in IDL; they are listed in Table 9-2.

To use character classes in a bracket expression, include the name of the class inside `[:` and `:]`. The character class names are case-sensitive and must be specified in lowercase. For example, to match a sequence of one or more alphabetic characters, try:

```
IDL> print, stregex('Test expression', '[[:alpha:]]+', /extract)
Test
```

Note that there are nested bracket expressions, one to open the bracket expression and one to indicate the character class. More complicated bracket expressions could allow for multiple character classes, as in:

```
IDL> print, stregex('1+2=3 is true', '[[:digit:][:punct:]]+', /extract)
1+2=3
```

It would also be possible to mix ranges, characters, and character classes in a bracket expression, like:

```
IDL> print, stregex('aBE5C', '[A-BE[:digit:]]+', /extract)
BE5
```

Class	Description
`alnum`	alphanumeric characters, equivalent to `0-9A-Za-z`
`alpha`	alphabetic characters, regardless of case, equivalent to `A-Za-z`; avoid `A-z` as it matches the characters `[\]^_`` as well.
`blank`	tab and space characters
`cntrl`	control characters, ASCII characters 1-31 and 127
`digit`	decimal digits, equivalent to `0-9`
`graph`	graphable characters, in other words, the characters that will show up on the screen (the printable characters except space), ASCII characters 33-126
`lower`	lowercase letters, equivalent to `a-z`
`print`	printable characters, ASCII characters 32-126

Table 9-2: Character classes in IDL regular expressions. See Table 9-3 on page 10 for ASCII character references.

Class	Description
punct	punctuation characters; the following characters are considered punctuation: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
space	whitespace characters, ASCII characters 9-13, 32
upper	uppercase letters, equivalent to A-Z
xdigit	hexadecimal digits, equivalent to 0-9A-Fa-f

Table 9-2: Character classes in IDL regular expressions. See Table 9-3 on page 10 for ASCII character references.

4.3. Non-matching lists

When the first character of a bracket expression is a ^ (caret), then the rest of the bracket expression is a non-matching list instead of a matching list. So for example, [^ab] matches any character *except* a or b. Any valid bracket expression can be negated by slipping a ^ in right after the opening [(except for a bracket expression that is already negated—no double negatives in regular expressions).

A character class can be negated, using the same notation. So [^[:alpha:]] matches any non-alphabetic character and [^[:lower:][:digit:]] matches any character except lowercase letters or decimal digits.

A word of warning, if you want to specify a bracket expression which includes the literal character ^, make sure to *not* put it first accidently negating the bracket expression. So to get a match on common mathematical operators use [+-*/^] instead of [^+-*/].

4.4. Miscellaneous

It is also possible to match zero width "characters" such as the beginning or ending of a word with [.<.] and [.>.] in a bracket expression. A word is defined to be a sequence of word characters (alnum character class and _) where there are no word characters immediately before or after the sequence.

```
IDL> print, stregex('George and Mike', '[[:space:]]Mike', /extract)
 Mike
IDL> print, stregex('George and Mike', '[[:<:]]Mike', /extract)
Mike
```

The first expression specifically requires the matching string to start with a character from the space character class, while the second expression requires there is not word character *before* the matching string. So the first expression matches five characters (starting with a space) and the second matches only four characters. Note that first expression would not match if the line started with “Mike” while the second expression would match.

5. Subexpressions

Subexpressions are used in two contexts:

- with alteration or repetition operators
- to locate a subexpression (using *STREGEX*’s SUBEXPR keyword)

In the first case, for example:

```
IDL> print, stregex(['ac', 'bc', 'cc'], '(a|b)c')
           0           0          -1
IDL> print, stregex('ababab', '(ab)+', /extract)
ababab
```

In the second case, used in conjunction with the *STREGEX*'s SUBEXPR keyword, subexpressions can return the location of a subexpression within a larger matching regular expression.

For example, here both the full match and a subexpression of the match are returned:

```
IDL> print, stregex('Mike', 'M(.*)', /extract, /subexpr)
Mike ike
```

Note that even if the SUBEXPR keyword is set, when parentheses are used as a grouping mechanism for repetition, only the first match is returned as part of the result, not a match for each repetition. Another way of calculating this is that the number of subexpressions returned is equal to the number of pairs of parentheses in the regular expression. Also, parenthesis may be nested; the subexpressions are returned in the order of the location of the opening parenthesis.

```
IDL> print, stregex('abc', '((a)b)c', /subexpr, /extract)
abc ab a
```

6. STREGEX syntax reference

```
Result = STREGEX( StringExpression, RegularExpression [, /FOLD_CASE]
       [, /BOOLEAN | , /EXTRACT | , LENGTH=variable [, /SUBEXPR]] )
```

Result

By default IDL returns the location (or an array of locations, if *StringExpression* is a string array) of the match or `-1` if there was no match. If the BOOLEAN keyword is set, the return value will be `0` (if not found) or `1` (if found). If the EXTRACT keyword is set, it will return the matching string or the empty string, if no match is found.

StringExpression

The *StringExpression* is a string or string array to be matched. If a string array, the regular expression tries to match on each element of the array independently and returns its results (and through the LENGTH keyword) as an array.

RegularExpression

The *RegularExpression* is always a scalar string containing the regular expression to match.

BOOLEAN

Set this boolean keyword to have *STREGEX* return 0 (false) or 1 (true) indicating whether the regular expression was matched.

```
IDL> print, stregex(['Test sentence', 'Test was here'], 'was')
          -1           5
IDL> print, stregex(['Test sentence', 'Test was here'], 'was', /boolean)
   0   1
```

EXTRACT

The EXTRACT keyword changes the return value to the matched strings instead of their locations. The EXTRACT keyword cannot be used with the BOOLEAN or LENGTH keywords.

FOLD_CASE

Set this boolean keyword to have *STREGEX* perform case-insensitive matching. The default is case-sensitive matching.

```
IDL> print, stregex('Mike', '[a-z]+', /extract)
ike
IDL> print, stregex('Mike', '[a-z]+', /extract, /fold_case)
Mike
```

LENGTH

Set to a named variable to return the length in characters of the returned strings. Together with the result of this function, which contains the starting points of the matches in *StringExpression*, LENGTH can be used with the *STRMID* function to extract the matched substrings.

```
IDL> str = 'Mike'
IDL> pos = stregex('Mike', 'ik', length=len)
IDL> print, pos, len
           1           2
IDL> print, strmid(str, pos, len)
ik
```

This keyword can not be used in conjunction with the BOOLEAN or EXTRACT keywords.

SUBEXPR

See “Subexpressions” on page 5 for more information about using the SUBEXPR keyword to return the location of a subexpression. The size of the return value and the LENGTH keyword is determined from both the dimensions of the *StringExpression* parameter and the number of parenthesis in the *RegularExpression* parameter. The dimensionality of this result has a first dimension equal to the number of parenthesis in the regular expression plus one and the remaining dimensions are copied from the dimensions of the *StringExpression* parameter. So for example, for a string expression which was 2 by 3 and a regular expression with 3 sets of parenthesis:

```
IDL> str_expr = [['a', 'eabcd'], ['b', 'c'], ['abcde', 'abcde']]
IDL> result = stregex(str_expr, '(a)(bc)(d)', /subexpr)
IDL> help, result
RESULT          LONG      = Array[4, 2, 3]
```

Note that parenthesis used with quantity operators only return a single subexpression match (not one for each time they match). So `(a)+` would match only the first `a` in the subexpression result (but all the `a`’s in the full result).

7. Extended examples

To search for PNG or JPG files in the IDL distribution on the C: drive onWindows:

```
IDL> files = file_search(filepath(''), '*')
IDL> valid_name = '[[:alpha:]_.][[:alnum:]_.]*'
IDL> re = 'C:\\(' + valid_name + '\\)*' + valid_name + '\.(png|jpg)'
IDL> print, stregex(files, re, /extract, /fold_case)
```

The *STRSPLIT* routine excepts a regular expression for its second positional parameter when the REGEX keyword is set.

It is often useful to split a string on any number of white space characters. This is simple with STRSPLIT and regular expressions:

```
IDL> result = strsplit('Mike was here', '[[:space:]]', /extract, /regex)
IDL> help, result[0], result[1], result[2]
<Expression>    STRING    = 'Mike'
<Expression>    STRING    = 'was'
<Expression>    STRING    = 'here'
```

The end of word bracket expression can be used to split a string without loss of any of the characters of the original string.

```
IDL> result = strsplit('Mike was here', '[[:>:]]', /extract, /regex)
IDL> help, result[0], result[1], result[2]
<Expression>    STRING    = 'Mike'
<Expression>    STRING    = ' was'
<Expression>    STRING    = ' here'
```

8. Library of patterns

Most of these patterns have some drawbacks and are not robust enough to catch all the special cases involved in matching patterns of the specified type. Rather, they are intended to be examples of the type of uses to which regular expressions are put.

`<[^>]+>`

XML tag

`(...) (...) (..) (..):(..):(..) (....)`

You can easily pull out the pieces of the result of the *SYSTIME* function using the above regular expression. Note that this does not validate whether a string is a *SYSTIME* result very well.

`(.[[:digit:]]+|[[:digit:]]+.?[[:digit:]]*)([Ee][+-]?[[:digit:]]+)?`

floating point value, possibly in scientific notation, examples.

`;.*$`

comment on a line of IDL code (includes a `;` in a literal string)

`^[^@ ]+@([^. ]+\.)+(com|edu|gov)$`

Valid email, more valid top-level domain names, some restrictions characters allowed in domain names and usernames, IP addresses, etc. It would be easy to use *STREGEX* with the BOOLEAN keyword set to create an email validation.

`[[:alpha:]_][[:alnum:]_$]*`

IDL variable name

9. Search and replace

It is common to not only search for a particular pattern, but to replace the pattern with another. *STR_REPLACE* addresses that need. It is available at: `http://internal/~mgalloy/idl/mine/misc/str_replace.html`.

```
Result = STR_REPLACE( StringExpression, SearchPattern, Replacement, [, /FOLD_CASE]
       [, /GLOBAL ] [, /EVALUATE] )
```

StringExpression

A scalar string to be searched

SearchPattern

A regular expression which specifies which substrings are to be replced. Can reference subexpressions in the replacement.

Replacement

The string to replace the search pattern with. Use `$1`, `$2`, etc. to refer to subexpressions in the *SearchPattern*, while `$&` refers to the entire match.

FOLD_CASE

Indicating the searching will be case-insentive, but the replacement will not change any case.

GLOBAL

The default is to find the first match and stop; set the GLOBAL keyword to find and replace all occurances of the search pattern.

EVALUATE

Set the EVALUATE keyword to indicate that the *Replacment* expression is IDL code and not literal expression.

9.1. Examples

The following shows the simplest use of *STR_REPLACE*, to replace one literal string with another.

```
IDL> print, str_replace('Mike was here', 'was', 'was not')
Mike was not here
```

The `subexpression`s in the search pattern are used here in conjuction with the `$1` and `$2` variables in the replacement pattern to switch the order of the first two "words" in the phrase.

```
IDL> print, str_replace('Mike was here', '([^ ]*) ([^ ]*)', '$2 $1')
was Mike here
```

The following can place commas in a string representing an integer. This could be done without the loop if commas where placed every three digits starting from the front. But since the counting starts at the back, we have to put one comma in at a time.

```
IDL> str = '1874382735872856'
IDL> regex = '^[+-]?([[:digit:]]+)([[:digit:]]{3})'
IDL> for i = 0, strlen(str)/3 - 1 do str = str_replace(str, regex, '$1,$2')
IDL> print, str
1,874,382,735,872,856
```

The EVALUATE keyword allows a replacement pattern which is IDL code. The match (`$&`) and subexpression match variables (`$1`, `$2`, etc.) are still available.

```
IDL> print, str_replace('Mike5', 'Mike([0-9]+)', 'strtrim(fix($1) * 2, 2)', /evaluate)
10
```

In a more complicated example, we would like to convert to all caps each “word” (a capital letter followed by lowercase letters) following the string `Mike`.

```
IDL> str = 'MikeGeorgeHenryMikeBill'
IDL> regex = 'Mike([A-Z][a-z]*)'
IDL> print, str_replace(str, regex, '"Mike"+strupcase("$1")', /evaluate, /global)
MikeGEORGEHenryMikeBILL
```

10. ASCII reference

0	NUL	null	43	+		86	V	
1	SOH	start of heading	44	,		87	W	
2	STX	start of text	45	-		88	X	
3	ETX	end of text	46	.		89	Y	
4	EOT	end of transmission	47	/		90	Z	
5	ENQ	enquiry	48	0		91	[	
6	ACK	acknowledge	49	1		92	\	
7	BEL	bell	50	2		93	]	
8	BS	backspace	51	3		94	^	
9	TAB	horizontal tab	52	4		95	_	
10	LF	NL line feed, new line	53	5		96	‘	
11	VT	vertical feed	54	6		97	a	
12	FF	NP form feed, new page	55	7		98	b	
13	CR	carriage return	56	8		99	c	
14	SO	shift out	57	9		100	d	
15	SI	shift in	58	:		101	e	
16	DLE	data link escape	59	;		102	f	
17	DC1	device control 1	60	<		103	g	
18	DC2	device control 2	61	=		104	h	
19	DC3	device control 3	62	>		105	i	
20	DC4	device control 4	63	?		106	j	
21	NAK	negative acknowledge	64	@		107	k	
22	SYN	synchronous idle	65	A		108	l	
23	ETB	end of trans block	66	B		109	m	
24	CAN	cancel	67	C		110	n	
25	EM	end of media	68	D		111	o	
26	SUB	substitute	69	E		112	p	
27	ESC	escape	70	F		113	q	
28	FS	file separator	71	G		114	r	
29	GS	group separator	72	H		115	s	
30	RS	record separator	73	I		116	t	
31	US	unit separator	74	J		117	u	
32		space	75	K		118	v	
33	!		76	L		119	w	
34	"		77	M		120	x	
35	#		78	N		121	y	
36	$		79	O		122	z	
37	%		80	P		123	{	
38	&		81	Q		124	|	
39	’		82	R		125	}	
40	(		83	S		126	~	
41	)		84	T		127	DEL	
42	*		85	U				

Table 9-3: ASCII reference chart. Note the punctuation marks between the uppercase and lowercase letters. This is the standard ASCII values. Extended ASCII values are from 128 to 255.

11. References

Building IDL Applications, Chapter 9: String, Learning about regular expressions.

Basic regular expressions usage in IDL.

Friedl, Jeffrey E.F. *Mastering Regular Expressions*. O’Reilly & Associates, Inc., 2002.

Definitive reference for regular expressions in general.

Online help pages for *STREGEX*, *STRSPLIT* and “Learning about regular expressions.”

Details of the IDL syntax for regular expressions and syntax for STREGEX and STRSPLIT.

Robbins, Arnold. UNIX in a Nutshell.

Basic egrep usage and list of UNIX tools which use regular expressions.

Oram, Andy. “Marshall McLuhan vs. Marshalling Regular Expressions.” `http://www.oreillynet.com/pub/a/network/2002/07/08/platform.html`. July 8, 2002.

Interesting article about the power of plain text and regular expressions’ role in using text.

Michael Galloy *is a trainer and consultant specializing in IDL programming for Research Systems. Recently, he has worked on the Topics in Advanced IDL manual, the What’s New in IDL 5.6 webinar, the Performance: Code Tuning webinar, and the IDLdoc project. He enjoys playing Ulimate frisbee and Nethack.*