Regular Expressions in Basic!

by Keith B. 2014-11-13

A write-up of notes I made for myself while learning to use regular expressions in rfo-Basic!

Introduction

The Basic! SPLIT and SPLIT ALL commands and the WORD$() function accept Regular Expressions as the delimiter specified in the test expression. This can cause problems if what was intended as a literal string gets interpreted as a pattern. However, using patterns can be both useful and powerful.

In these notes, strings are shown as Basic! constants. This means that all backslashes are doubled up, because \ is the escape character in Basic! strings as well as in regular expressions. For regular expressions in data files, or from user input with tget, input or text.input, only single backslashes are required.

Numbers in parentheses at the right hand margin cross-reference between examples in this note and those in the demonstration program, regex_examples.bas.

I've also posted a basic source file with various utility functions based on using regular expressions in word$(), regex_fns.bas

Pattern Matching

In general, characters in a regular expression match the same character in a target string. This applies to all single characters except for these 13 "metacharacters" which have special meanings in regular expressions:

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

If **| ^ $** or **.** is used as the delimiter in SPLIT or WORD$(), without a \\ prefix, then it will appear to work but the results will probably not be as intended.

Using **( ) [{ } * + ?** or **** as the delimiter in SPLIT or WORD$() will result in an error being raised:

"(is invalid argument" etc.

To use one of these characters as the delimiter in SPLIT or WORD$() prefix it with \\

For example, we could split a string that uses full stops as the delimiter like this:

```
SPLIT field$[], "first.second.third", "\\."                    (1)
```

To use \ as the delimiter, four backslashes are required, so that two will be passed to the regular expression parser:

```
SPLIT field$[], "first\\second\\third", "\\\\"                 (2)
```

Escape sequences comprising \\ followed by a letter are used for many shortcuts and codes. Any letters not currently used are reserved for future extension of the regular expression standard. It is an error to use the following letters after \\

g h i j k l m o q v y C F H I J K L M O R T V X Y

Quoting Literals

A simple and generally applicable way to avoid problems if you want to use longer literal strings as delimiters is to prefix your string with "\\Q". This will force all characters to be treated as literals until the end of the string or until "\\E". For example:

```
SPLIT field$[], "first..second..third", "\\Q.."
```

This works with any delimiter string that does not include "\\E" as a substring.

Metacharacters and Escape Sequences

A full stop . matches any one character (except newline, unless enabled in the flags setting described later)

The following escape sequences can be used to represent special characters in the delimiter:

Escape Sequence	Character Name	Equivalent chr$()
\\a	bell (alert) character	chr$(7)
\\e	escape character	chr$(27)
\\f	form feed	chr$(12)
\\n	newline	chr$(10)
\\r	carriage return	chr$(13)
\\t	tab character	chr$(9)

For example, to split a string at tab characters we could use:

```
SPLIT field$[], "first"+chr$(9)+"second"+chr$(9)+"third", "\\t"          (3)
```

Any control code (i.e. characters with ascii codes less than 32) can be represented as \\cx for control-x, where x is in the range '@' to '_'. For example, character 26, control-Z, would be \\cZ.

Any Unicode character can be represented as \\x{h…h} where h…h is the code in hexadecimal digits. Other ways of coding characters by their Unicode values and names are listed in the Annex.

Character Classes

Sometimes we may need to split a record where more than one delimiter is used. For example, we may want to divide a record at both commas and semicolons. We do this by using a character class as the regular expression. Character classes are specified by listing the required characters between square brackets, so in this case we would use:

```
SPLIT field$[], "first;second,third", "[,;]"                    (4)
```

In general the character class [abc...] represents any one of the characters listed inside the brackets; [a-k] represents any one of the characters between a and k inclusive; [^a-k] represents any one Unicode character *except* a to k inclusive. This could be useful, for example, to extract just the integers in a record:

```
SPLIT field$[], "first 10 second 200 third 300 ", "[^0-9]+"     (5)
```

(Here we are anticipating the section on Quantifiers below. For now, just note that the plus sign following the character class specification means that the delimiter comprises *one or more* characters in the preceding character class.)

To include a minus sign character, **-**, in the character class put it in first or last position:

```
[-09] matches any one of -, 0 or 9 and no other characters;
[0-9] matches any one of 0 to 9 inclusive. It does not match -
```

To include a **^** character in the character class put in in any position inside the [ ] except at the beginning.

Inside [ ], the metacharacters that require escaping are [\ and] (and depending on the context & and **-**) Other metacharacters can be used as literals. For example, to split text at commas and full stops we would use:

```
SPLIT field$[], "first.second,third", "[,.]"                    (6)
```

with no \\ required in front of the full-stop.

There are various pre-defined classes. These include:

```
\\d    any digit
\\D    any non-digit
\\s    any whitespace character
\\S    any non-whitespace character
\\w    any word character
\\W    any non-word character
```

These are Unicode classes, and may include characters you don't expect. For example, the digits class includes digits in non-latin scripts.

These class codes can be used inside [ ] combined with one-another or with regular characters. For example [\\s,] means any whitespace character *or* a comma.

There is a wide range of classes that can be referenced as \\p{classname} or \\p{property=value}. These match any one character in the named class, or for which the property has the specified value.

\\P{classname} or \\P{property=value} match any one character *not* in the named class or for which the property does *not* have that value.

See the Annex for lists of some of the available classes and properties, and links to find others.

For example, we could use the Unicode Block class for Greek letters, "inGreek" to extract all Greek characters from a text:

```
SPLIT field$[], "Greek from " + chr$(945,946,947) + " via " + ~
                chr$(955,956,957) + " to " + chr$(967,968,969) + ".", ~
                "\\P{inGreek}+"                                  (7)
```

Here, the delimiter is defined to be any one or more characters that are *not* in the Unicode Greek block. So the result array from the split will hold all the Greek words from the text, with an empty first element unless the text starts with a Greek letter. (This ignores any of the less common Greek characters that are in the Greek Extended block; an alternative that would include these is to split at any characters that are not in the Greek script, \\P{sc=Greek}.)

Intersections of classes are specified using the && operator, so [\\d&&\\p{ASCII}] would match any character that is *both* a digit *and* an ASCII character, though it would be simpler to use [0-9]!

There are three class operations:

Union	(implicit)	[a-dc-f] is equivalent to [a-f]
Intersection	&&	[a-d&&c-f] is equivalent to [cd]
Subtraction	--	[a-d--c-f] is equivalent to [ab]

Quantifiers

Not all delimiters will be single characters, or even fixed length. The regular expression syntax includes various quantifiers that we can use to handle this. These modify the meaning of the previous expression.

Quantifiers match some number of instances of the preceding item:

Quantifier	Number matched
*	Zero or more
?	Zero or one
+	One or more
{n}	Exactly n
{n,}	At least n
{n,m}	At least n but not more than m

Previous examples have used the + quantifier to get numbers and Greek-letter words.

Because the * and ? quantifiers allow zero-length matches, they may give unexpected results if used with single characters in SPLIT and WORD$. For example:

```
SPLIT field$[], "a,b", ",?"                                    (8)
```

results in 4 words: "", "a", "", "b"

However, these quantifiers can be useful when used in multi-character delimiters. For example, if we have some comma-separated values with additional spaces:

```
SPLIT field$[], "first, second, third, ", ", *"                (9)
```

Here the delimiter regular expression matches a comma followed by zero or more spaces.

If the gaps between fields included tab characters as well as spaces we would use:

```
SPLIT field$[], "first,\tsecond, third, ", ",[ \\t]*"          (10)
```

If it also included non-breaking spaces – and we did not want to keep them – we could use:

```
SPLIT field$[], "first,\tsecond,"+chr$(160)+" third, ", ~
                ",[ \\t\\x{A0}]*"                              (11)
```

where \\x{A0} means the character with hex code A0 = 160, the non-breaking space.

Another way of doing this is:

```
SPLIT field$[], "first,\tsecond,"+chr$(160)+" third, ", ~
                ",[\\p{javaSpaceChar}\\p{javaWhitespace}]*"    (12)
```

where [\\p{javaSpaceChar}\\p{javaWhitespace}] matches zero or more characters in the javaSpaceChar or javaWhitespace classes. (We use both because we need both tabs and non-breaking spaces to be matched.)

Concatenation and Alternation

To join together two regular expressions, simply set them next to one another. So, for example the expression a+b* means one or more a's followed by zero or more b's.

The alternation operator **|** means "or" and so, for example, a+|b+ means either one or more a's or one or more b's (but not a mixture of both). **|** has lower priority than concatenation, so abc|def means *either* abc *or* def.

| is interpreted as a literal (not the metacharacter) when it is inside [ ]

Groups and Backwards References

Parentheses **( )** can be used to group parts of a regular expression for use with quantifiers or to change how alternatives are applied. So if we wanted to specify a followed by bc or de and then f we could use a(bc|de)f.

An example using parentheses with quantifiers: If we have a record where the fields are divided by either a semicolon or a dot and a comma, in both cases followed by optional spaces then we could split it as follows:

```
SPLIT field$[], "first; second., third", "(;|\\.,) *"          (13)
```

The parentheses change the meaning of the expression. Without them, it would be ";|\\., *" Because concatenation is higher priority then the | operator, this would be equivalent to "(;)|(\\., *)". That is, the optional spaces would be attached to the ., alternative only.

As well as changing the order in which expressions are evaluated, parentheses "capture" the matched string, so that it can be referenced again later in the regular expression. These references take the form \\n or \\nm where m is any digit and n is any non-zero digit. (So there can be up to 99 groups.)

For example the pattern ([0-9])\\1\\1 would match any three consecutive identical digits, such as 333 or 999.

The order in which group numbers are assigned is the order of their opening parentheses. A group should not be referenced until after its closing parenthesis.

It is an error to use \\d where d is a single digit greater than the number of capturing groups. \\ followed by two digits is interpreted as a backwards reference if that number of groups is available, if not, the first digit is treated as a backwards reference and the second digit as a literal character.

There is a non-capturing form of group: (?:expression). This has the same effect as simple parentheses on the order in which the expression is evaluated, but the expression parser does not store the matched value. It may execute faster than the same expression using capturing groups, and allows expressions to be amended without having to change backwards reference numbers.

Greedy and non-Greedy Quantifiers

By default quantifiers are "greedy"; that means that the delimiter will use up as many characters as possible when matching the pattern. This may not be what we want. For example:

```
SPLIT field$[], "first:ignore:second:junk:third", ":.*:"       (14)
```

will return just two strings: "first" and "third". This is because the delimiter matches the first and last colons and all the characters in between. In this case, we probably want the shortest possible matches, and that means using

non-greedy quantifiers. Each quantifier has a non-greedy equivalent – just add a question mark after its symbol.

Greedy Quantifier	Non-Greedy Quantifier	Number matched
*	*?	Zero or more
?	??	Zero or one
+	+?	One or more
{n}	{n}?	Exactly n
{n,}	{n,}?	At least n
{n,m}	{n,m}?	At least n but not more than m

So the command:

```
SPLIT field$[], "first:ignore:second:junk:third", ":.*?:"          (15)
```

would return three strings, "first", "second", and "third" because the delimiter matches the shortest possible number of characters each time, so the first match is ":ignore:" and the second match is ":junk:".

Possessive Quantifiers and Atomic Groups

Quantifiers normally allow backtracking, so that, for example:

```
SPLIT x$[], "a0b", "\\d*[0 ]"
```

would put two values in x$[], "a" and "b", because the pattern "\\d*[0]" would match the string "0". Initially, "\\d*" (zero or more digits) would match the "0" but the pattern matcher would then backtrack to make the "0" available to match "[0]" (zero or space) which would otherwise be unmatched.

There is another form of quantifier – possessive quantifiers – that does not allow backtracking. These are the same as the normal quantifiers followed by a +.

```
SPLIT x$[], "a0b", "\\d*+[0 ]"
```

would put the whole string "a0b" into x$[1] because "\\d*+" (zero or more digits) matches the "0" and the "[0]" (zero or space) in the pattern is unmatched since there is no backtracking.

There is also a form of group that does not allow backtracking, the atomic group, which takes the form (?>expression). Atomic groups are non-capturing, so can't generate values to use as backwards references, though they can have a capturing group inside them as (?>(expression))

Start and End of Line

The metacharacters **^** and **$** match the start and end of line. For example, if we wanted to retrieve only text enclosed in < >, we could do as follows:

```
SPLIT field$[], "abc <first> xyz <second> p q r <third> ", ~
                "^.*?<|>.*?<|>.*?$"                                (16)
```

Here the delimiter matches any of:

- the start of the line up to and including the first < character
- a > character and all characters up to the next < character
- a > character and all characters up to the end of line

We use the non-greedy quantifier *? for 0 or more matches, so that the matching of the delimiters will stop at the first available >.

A null string will be returned in field$[1] followed by "first", "second" and "third" in the remaining elements of field$[]. (Note that this method won't handle nested < >'s.)

Often there is more than one way to specify equivalent regular expressions, and in this case an alternative way of getting the same result is:

```
SPLIT field$[], "abc <first> xyz <second> p q r <third> ", ~
                "(^|>).*?($|<)"                                    (17)
```

Yet another way of getting the same result is:

```
SPLIT field$[], "abc <first> xyz <second> p q r <third> ", ~
                "(^|>)[^<]*($|<)"                                  (18)
```

Here we have used [^<]* which matches zero or more characters other than <, so don't need to use the non-greedy quantifier. There is no "right" answer, though it may be worth doing some timings to find an efficient specification if the code is going to run many times.

Comments

Regular expressions can include comments taking the form (?# comment)

Any characters between (?# and) are ignored, including backslashes.

Flags

There are several options which affect how matching is carried out. These can be set using the following flags:

- i — Case insensitive matching.
- d — Only accept newline (chr$(10)) as a line terminator. Normally carriage return (chr$(13)) is also accepted.
- m — Allow ^ and $ to match beginning/end of any line, if there are line-breaks within the string. Otherwise they only match the start and end of the whole string.
- s — Allow . to match line break character ("s" for "single line"). Normally it doesn't.
- x — Allow and ignore whitespace and comments within the regular expression. Normally all text is treated as part of the pattern. In addition to (?#...) comments can start with a # and end with an end of line. To include spaces or # characters in the expression when the x flag is set, they must be preceded by \\

The syntax for setting or clearing these flags is:

(?on-off) where on and off are lists of flags to be set or cleared.

The lists can be empty and the - can be omitted if there are no flags to clear. For example (?i) sets case insensitive matching and (?-i) clears it.

If the fields in a record are terminated by the string "end.", but the case varies, we could use:

```
SPLIT field$[], "first End. second END. third end.", "(?i-)end\\. *"    (19)
```

To apply flags to part of a regular expression only, use (?on-off:a) where a is the part of the expression to which the flags will apply.

Look Ahead/Behind

The regular expression syntax provides a way of specifying that the matched text is preceded and/or followed by specified patterns that are not included in the match.

In Basic! this means that we can, for example, split a string at a particular character and include that character in the strings in the result array.

The syntax for doing this is:

(?<=a)	to specify that text matching a should precede the delimiters. ("Look behind")
(?=a)	to specify that text matching a should follow the delimiters. ("Look ahead")

There are also negative versions of these that specify what should *not* precede or follow the delimiter. These are

(?<=!a)	to specify that text matching a must not precede the delimiters.
(?=!a)	to specify that text matching a must not follow the delimiters.

The expressions for preceding text must have bounded length, so may not include * + or {n,} quantifiers which are unbounded, but can include {n,m} quantifiers since they have an upper limit of m repetitions. In theory any finite bound should be valid as a value of m; in practice there seems to be a limit of 10.

For example, if we wanted to split some text at every full stop, question mark or exclamation mark, to divide it into sentences, we could use the following:

```
SPLIT field$[], "First. Second? Third! ", "(?<=[.?!]) +"          (20)
```

This specifies that the delimiter is one or more spaces, and that it must be preceded by one of .?! (no \\ is needed for the full stop because it is inside the [ ])

The resulting array will hold entries "First.", "Second?", and "Third!".

If we wanted to allow for quotation marks possibly following the full stop etc. we could use:

```
SPLIT field$[], "First. \"Second?\" Third! ", ~
                "(?<=[.?!][\"\\p{Pf}]?) +"                        (21)
```

The term [\"\\p{Pf}]? matches zero or one straight double quote or any character in the Unicode final quote category. Note the *single* backslash in front of the quotation character. It is required for Basic! but not in the regular expression syntax, so only needs to be escaped once.

We may also want to allow for other whitespace characters including line breaks. So, an expression to split a block of text into sentences would be:

```
SPLIT field$[], "First. Second?"+chr$(10)+"Third! ", ~
                "(?<=[.?!][\"\\p{Pf}]?)[\\n\\t \\xA0]+"           (22)
```

The pattern here could be made easier to read by added spacing:

```
SPLIT field$[], "First. Second?"+chr$(10)+"Third! ", ~
          "(?x-) ( ?<=[.?!] [\"\\p{Pf}]? ) [\\n\\t\\ \\xA0]+"     (23)
```

Here, the new term (?x-) switches on the option to ignore spaces. The space character following the tab in the final [ ] group has to be escaped by \\ so that it *isn't* ignored.

We could also add comments:

```
SPLIT field$[], "First. Second?"+chr$(10)+"Third! ", ~
                "(?x-)\n" + ~
                "# SPLIT TEXT INTO SENTENCES\n" + ~
                "(?<=[.?!] # Sentence ends with . ? or !\n" + ~
                "[\"\\p{Pf}]?) # may be followed by quotes\n" + ~
                "[\\n\\t\\ \\xA0]+ # and 1 or more whitespace chars."
```

As a final example of look-ahead and -behind, suppose we have a string containing some html. We could split it into tags and text between tags as follows:

```
SPLIT field$[], "<h1>The main heading</h1><p>Some plain text and " + ~
          "<i>some in italic.</i></p>", "(?<=>)|(?<=.)(?=<)"          (24)
```

In this case, the pattern is entirely made up of look-behind and look-forward matches. There are two alternative patterns, separated by the **|** operator The first pattern will match immediately after a > character. The second will match immediately before a < and after any character – this prevents matching before a < that is the first character in the file. This allows the SPLIT command to keep the < > pairs as part of the tags. We have defined zero width delimiters, and so every character in the original string will go into one (and only one) of the array elements.

Using Word$() to check whether a string matches a regular expression

Basic! doesn't have a statement to check directly whether or not a string matches a regular expression, but we can achieve that result using the WORD$() function. Consider the expression:

```
WORD$("x"+s$, 1, "^x(?:" + p$ + ")")
```

where s$ is the string we want to test, and p$ is a regular expression we want to compare it with.

This will return an empty string if the whole of s$ matches p$, and in no other circumstances.

The choice of "x" as an additional character at the start of the string and the pattern is arbitrary; any normal character would do. The purpose of this is to ensure that if s$ is an empty string, word$() will only return an empty string if the pattern p$ matches an empty string.

The **^** means that the pattern can only match at the start of the string. If it does match the start, but does not match the whole string, there will be part of the string left over that will go into the WORD$() return value. As an example we look at using this to check that a string is a valid input to the HEX() function. (HEX() will cause a runtime error if called with an invalid string.)

A regular expression for a valid Basic! hexadecimal string is:

```
"[+-]?0*[0-9A-Fa-f]{1,16}"
```

[+-] matches either plus or minus and the ? quantifier means we have zero or one of these, so the pattern begins with an optional sign.

0* matches zero or more leading zeros

[0-9A-Fa-f] matches any hexadecimal digit, allowing a to f to be in upper or lower case.

{1,16} means that there can be between one and sixteen of these hexadecimal digits, (in addition to any leading zeros).

Here is some code that gets a string using TGET and uses WORD$() twice, firstly to remove any leading or trailing white space, then to check that it is a valid hexadecimal value:

```
DO
  TGET t$, "Hex: ", "Input a hexadecimal number"
  t$ = WORD$(t$, 1, "^\\s*|\\s*$")
  ishex = "" = WORD$("x" + t$, 1, "^x(?:[+-]?0*[0-9A-Fa-f]{1,16})")
  IF !ishex THEN PRINT "Not a valid hexadecimal number. Please re-input"
UNTIL ishex
PRINT "HEX(";t$;") = "; INT$(HEX(t$))
```

This will loop until a valid hexadecimal number is input.

References and Further Reading

Android uses a regular expressions package from ICU which replaces the normal Java classes RegexPattern and RegexMatcher. This is described in brief in "Regular Expressions - ICU User Guide" available at *userguide.icu-project.org/strings/regexp*

When reading this it is useful to be able to refer to Unicode Technical Standard #18 at *www.unicode.org/reports/tr18/* to understand what compliance with Level 1 and parts of Level 2 of that standard implies.

Also you may want to refer to the Java documentation at *docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html* since in part ICU describe their product by its similarities to and differences from the Java classes.

Other sites that may be useful:

regexlib.com is a large collection of regular expressions submitted by its users. You will need to check that the expressions there are compatible with Android.

www.regular-expressions.info has general information and tutorials about regular expressions on many different platforms. Be sure to check what is relevant to java, and bear in mind the differences between Android and generic java as a result of the use of the ICU library.

Annex: Regular Expression Syntax Summary

The syntax here is as required in Basic! literal strings; that is with additional backslash \ characters. If regular expressions are read from a data file; obtained via input, text.input or tget statements; or from an html form, then only single backslashes are required.

Regular expression syntax is case-sensitive.

Quoted Literals

\\Q	start literals
\\E	end literals

All text between a \\Q \\E pair (or from \\Q to the end of the text) is treated as quoted. i.e. metacharacters are treated as normal literals. Basic! requirements for using \ as an escape character in strings still apply.

Specific characters

*	a literal * and similarly for .|()[{}^$+?
\\\\	a literal \
\\a	a bell (alert) character
\\t	a tab character
\\n	a newline character
\\f	a form feed
\\r	a carriage return
\\e	an escape character
\\cX	control character CTRL-X
\\N{name}	The character with the specified Unicode name. e.g. \\N{GREEK CAPITAL LETTER SIGMA}. The name must be an exact match with the Unicode standard, including spaces, except that case is ignored. No escape sequences are allowed inside the name, except for \\Q...\\E quoted literals.

Character codes

\\0d \\0dd \\0ddd	octal codes 00 to 0377
\\x{h..h}	variable length hex code
\\xhh	2 digit hexadecimal code
\\uhhhh	4 digit hexadecimal code
\\Uhhhhhhhh	8 digit hexadecimal code

Metacharacters

\\	Modify the meaning of the next character
.	Match any one character (except \\n unless the s flag is set)
|	Alternation: a|b matches a or b
( )	Group and capture
[ ]	Define character class
{ }	Numeric quantifiers and class names

Anchors (zero width assertions)

^	beginning of text or of a line	} Depends on the
$	end of text or of a line	} m flag setting
\\b	a word boundary	
\\B	not a word boundary	
\\A	beginning of input	
\\Z	end of input; ignores any newline at the end of file	
\\z	end of input; treats a newline at the end of file as a separate character	
\\G	end of previous match	

Quantifiers

These apply to the preceding character, class or (group).

Greedy quantifiers match as many characters as possible.
Non-greedy quantifiers match as few as possible.
Possessive quantifiers match as many as possible and don't allow back-tracking.

Greedy	**Non-Greedy**	**Possessive**	
?	??	?+	Match 0 or 1 times
*	*?	*+	Match 0 or more times
+	+?	++	Match 1 or more times
{n}	{n}?	{n}+	Match exactly n times
{n,}	{n,}?	{n,}+	Match at least n times
{n,m}	{n,m}?	{n,m}	Match at least n but not more than m times

Comments

The content of comments, including any metacharacters is ignored.

(?#	Start comment
)	End of comment. Comments can not include)'s

Character class metacharacters

[ ]	Define character class
^	If the first character of a class, negates that class
-	Unless the first or last character, used for a range
&&	Intersection of two classes
--	Subtract second class from first.

The -- operator is an Android (ICU) extension of the Java Pattern class.
[a-d--c-f] is equivalent to [a-d&&[^c-f]]

Escape sequences within [ ]

\\\\	for \	
\\[	for [	
\\]	for]	
\\&	for &	} if context
\\-	for -	} requires

Escapes are not required for . | ( ) { } ^ $ * + ?

Character class shortcuts

These shortcuts can be used either on their own, or within a character class.

\\d	a digit
\\D	a non-digit
\\s	a whitespace character
\\S	a non-whitespace character
\\w	a ‘word’ character
\\W	a ‘non-word’ character
\\p{name}	any character in the named class
\\P{name}	any character not in the named class
[:name:]	any character in the named class (alternative syntax)

Groups and Backwards References

( )	A capturing Group
(?:)	A non-capturing Group
(?>)	An atomic (non-capturing) Group
\\d or \\dd	A backwards reference to the dth or ddth capturing Group

Flags

(?dimsux-dimsux:a)	Evaluate a with the given flags set/cleared. a is a non-capturing Group. Additional parentheses can be used to make it capturing.
(?dimsux-dimsux)	Evaluate the rest of the pattern with the given flags set/cleared

Meaning of the flags:

d	line feed is the only line terminator
i	Ignore case
m	^ and $ match at internal line breaks
s	Let . match \n
u	enable Unicode case folding
x	Allow whitespace and comments

Flags are off by default i.e.

- any combination of carriage return and/or line feed is accepted as a line teminator
- matching is case sensitive
- ^ and $ match at the start and end of the whole string only
- . does not match newline characters
- all text is interpreted as part of the pattern

The u flag is always set on Android, case-insensitive matching will always be Unicode-aware.

If the x flag is set, in addition to using (?#...), comments can start with a # and end with an end of line. To include spaces or # characters in the expression when the x flag is set, they must be preceded by \\

Look Ahead and Look Behind

```
  (?=pattern)     positive look-ahead.
  (?!pattern)     negative look-ahead.

  (?<=pattern)    positive look-behind.
  (?<!pattern)    negative look-behind.
```

These are non-capturing Groups. Additional parentheses can be used to make all or part of pattern capturing.

These match a specific pattern without including it in the text used as the delimiter. Positive assertions match when their pattern matches, negative assertions match when their pattern does not match.

Look-behind matches text up to the match position, look-ahead matches text following the match position.

The maximum length of possible matches for look-behind patterns must not be unbounded.

For example, "(?<=[:;])\t" matches a colon or semicolon followed by a tab, without including the punctuation mark in the matched delimiter.

Named Classes

The following are some of the class names that can be used in \\p{name} and \\P{name} (and in the alternative [:name:] syntax.)

Unicode general categories as follows:

Controls	C	Punctuation	P
Unassigned	Cn	Dash	Pd
Control	Cc	Start	Ps
Format	Cf	End	Pe
Private-use	Co	Connector	Pc
Surrogate	Cs	Initial Quote	Pi
		Final Quote	Pf
Letters	L	Other	Po
Uppercase	Lu		
Lowercase	Ll	Symbols	S
Title Case	Lt	Maths	Sm
Modifier	Lm	Currency	Sc
Other	Lo	Modifier	Sk
		Other	So
Marks	M		
Non-Spacing	Mn	Separators	Z
Enclosing	Me	Space	Zs
Combining Spacing	Mc	Line	Zl
		Paragraph	Zp
Numbers	N		
Decimal Digits	Nd		
Letter	Nl		
Other	No		

Examples:

\\p{Lu} or \\p{gc=Lu} or [:gc=Lu:] match any upper-case letter

Unicode Blocks:

the block name preceded by "in"; such as inGreek, inHebrew, inArrows etc.

or use block=name or blk=name.

Examples:

\\p{inGreek} or \\p{blk=Greek}

Unicode Scripts:

use script=name or sc=name; such as sc=Latin, sc=Common etc.

Example: \\p{sc=Latin}

Lists of blocks and scripts are available at www.unicode.org

Individual characters referenced by Unicode name:

\\p{name=character name} e.g. \\p{name=Exclamation Mark}.

This is subtly different to the \\N{character name} notation.

\\p(name=name} defines a class containing one character.
\\N{} defines the character itself and can be used in the specification of a range, which \\p{} can't.

The names of general categories, blocks, and scripts are "soft matched", that is spaces can be omitted or replaced by underlines, or extra spaces can be added, and case can be changed. Names of characters must be more tightly specified; case can be changed, and extra spaces can be inserted where there are already spaces, but spaces can not be omitted, and replacing spaces by underlines is not allowed.

POSIX classes as follows:

Alpha, Digit, XDigit, Alnum, Lower, Upper, Graph,
ASCII, Blank, Cntrl, Print, Punct

Examples:
\\p{ASCII} matches any character in range 0 to 127,
\\P{ASCII} matches any character with code 128 or greater

Character method names:

Names of methods for JAVA class Character starting with "is", with the "is" replaced by "java", including the following:

javaDigit, javaLetter, javaLetterOrDigit, javaLowerCase, javaUpperCase, javaSpaceChar, javaWhitespace

Example: \\p{javaDigit}

White-space Classes

The following tables show which characters are in which of the whitespace classes.

Character	Unicode	\\s	\\p{Zs}	\\p{Zl}	\\p{Zp}
horizontal tab	0009	y	-	-	-
line feed	000a	y	-	-	-
vertical tab	000b	y	-	-	-
form feed	000c	y	-	-	-
carriage return	000d	y	-	-	-
file separator	001c	-	-	-	-
group separator	001d	-	-	-	-
record separator	001e	-	-	-	-
unit separator	001f	-	-	-	-
space	0020	y	y	-	-
next line	0085	y	-	-	-
no-break space	00a0	y	y	-	-
ogham space mark	1680	y	y	-	-
mongolian birga	180e	y	y	-	-
en quad	2000	y	y	-	-
em quad	2001	y	y	-	-
en space	2002	y	y	-	-
em space	2003	y	y	-	-
3 per em space	2004	y	y	-	-
4 per em space	2005	y	y	-	-
6 per em space	2006	y	y	-	-
figure space	2007	y	y	-	-
punctuation space	2008	y	y	-	-
thin space	2009	y	y	-	-
hair space	200a	y	y	-	-
line separator	2028	y	-	y	-
para separator	2029	y	-	-	y
narrow no-break space	202f	y	y	-	-
medium maths space	205f	y	y	-	-
cjk ideographic space	3000	y	y	-	-

Character	Unicode	\\p{Blank}	\\p{javaWhitespace}	\\p{javaSpaceChar}
horizontal tab	0009	y	y	-
line feed	000a	-	y	-
vertical tab	000b	-	y	-
form feed	000c	-	y	-
carriage return	000d	-	y	-
file separator	001c	-	y	-
group separator	001d	-	y	-
record separator	001e	-	y	-
unit separator	001f	-	y	-
space	0020	y	y	y
next line	0085	-	-	-
no-break space	00a0	y	-	y
ogham space mark	1680	y	y	y
mongolian birga	180e	y	y	y
en quad	2000	y	y	y
em quad	2001	y	y	y
en space	2002	y	y	y
em space	2003	y	y	y
3 per em space	2004	y	y	y
4 per em space	2005	y	y	y
6 per em space	2006	y	y	y
figure space	2007	y	-	y
punctuation space	2008	y	y	y
thin space	2009	y	y	y
hair space	200a	y	y	y
line separator	2028	-	y	y
para separator	2029	-	y	y
narrow no-break space	202f	y	-	y
medium maths space	205f	y	y	y
cjk ideographic space	3000	y	y	y