The CSV data parser plugin
PRINTED MANUAL

CSV data parser plugin

Printed: 11/2/2024

Publisher

AGG Software

Production

© 1999-2024 AGG Software
http://www.aggsoft.com

Table of Contents

1 Introduction

This module can parse a data flow which contains a tab or comma delimited text data. You can define any delimiter or quote characters in the module. Th module can easily and automatically parse all column in a CSV row to parser items. These variables can be post-processed and exported later as you want.

2 System requirements

The following requirements must be met for "CSV data parser" to be installed:

Operating system: Windows 2000 SP4 and above, including both x86 and x64 workstations and servers. The latest service pack for the corresponding OS is required.

Free disk space: Not less than 5 MB of free disk space is recommended.

Special access requirements: You should log on as a user with Administrator rights in order to install this module.

The main application (core) must be installed, for example, Advanced Serial Data Logger.

3 Installing CSV data parser

1. Close the main application (for example, Advanced Serial Data Logger) if it is running;
2. Copy the program to your hard drive;
3. Run the module installation file with a double click on the file name in Windows Explorer;
4. Follow the instructions of the installation software. Usually, it is enough just to click the "Next" button several times;
5. Start the main application. The name of the module will appear on the "Modules" tab of the "Settings" window if it is successfully installed.

If the module is compatible with the program, its name and version will be displayed in the module list. You can see examples of installed modules on fig.1-2. Some types of modules require additional configuration. To do it, just select a module from the list and click the "Setup" button next to the list. The configuration of the module is described below.

You can see some types of modules on the "Log file" tab. To configure such a module, you should select it from the "File type" list and click the "Advanced" button.

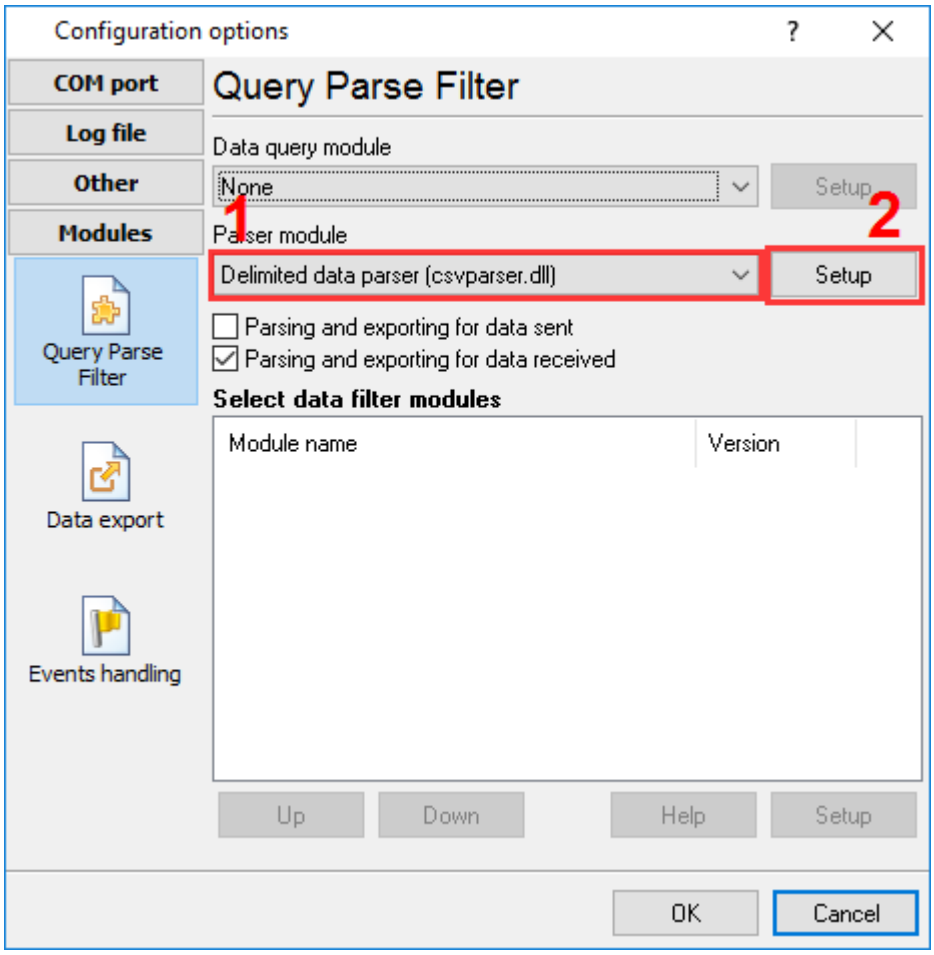

Fig. 1. Example of installed module

4 Glossary

Main program - it is the main executable of the application, for example, Advanced Serial Data Logger and asdlog.exe. It allows you to create several configurations with different settings and use different plugins.

Plugin - it is the additional plugin module for the main program. The plugin module extends the functionality of the main program.

Parser - it is the plugin module that processes the data flow, singling out data packets from it, and then variables from data packets. These variables are used in data export modules after that.

Core - see "Main program."

5 Analyzing and parsing data

To export the data received from the port, you should configure the parser. The parser allows you to single out data rows from the overall flow consisting of ASCII characters and parse these rows into variables that are exported into various destinations after that. You should define the rules the parser will use to process the incoming data flow on the "Data packet" tab (fig.1).

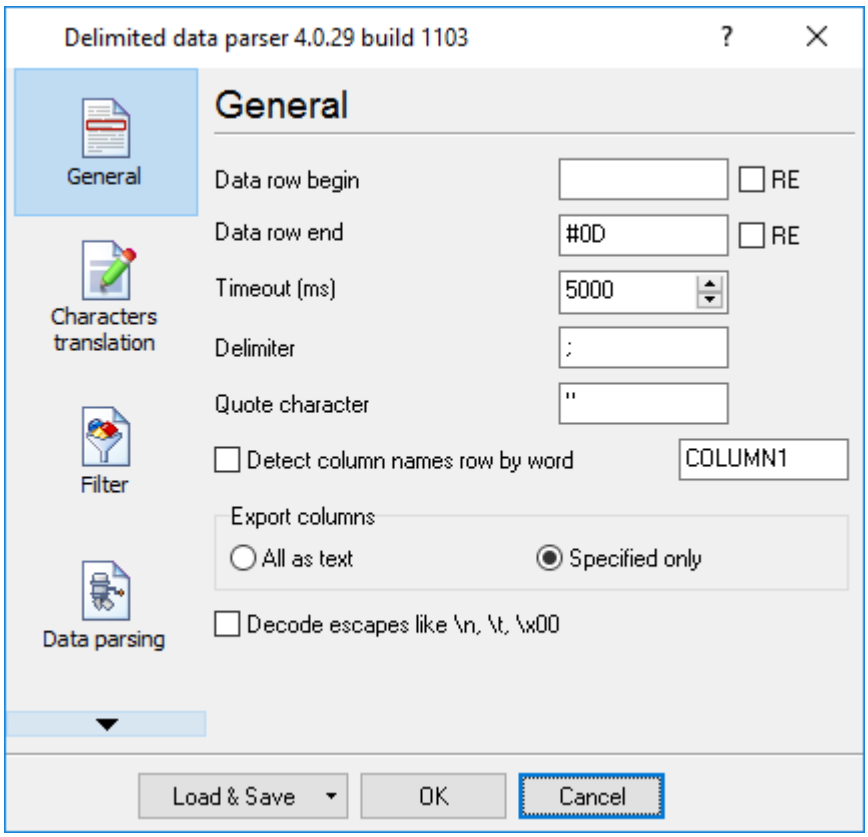

Fig.1. Data formatting.

Use beginning/ending signs to split data packets

Usually, an ASCII data flow contains a data row ending sign. Usually, the 0x0D, 0x0A characters or their combination are used. You can see what characters are the ending signs in your case. Just enable displaying characters with the code < 0x20h in the main window of the program and try to receive some data. If your data block contains nonprinting characters with the code < 0x20h, you should specify them in the module as #XX#XX (for example, #0D#0A).

- **Data row begin** – optional parameter. For example: #02;
- **Data row end** – required parameter. For example: #0D#0A;

Sometimes the beginning and the ending of a data packet vary. For example, the beginning of your data packet contains the value of time, like 00:01:55, but this value is constantly changing. Then you can create such a regular expression as \d{2}\:\d{2}\:\d{2} in order to determine the beginning of a packet and select the **RE** checkbox next to the corresponding field.

Timeout – if you do not specify the beginning of a data packet, it may happen that the module will infinitely wait for the ending of some data packet. The timeout value is used to prevent it. It defines the maximum interval the module can wait for a packet to end.

The "Load & Save" button allows you to load some presets for some types of the data flow, for example a CSV file or a plain text file.

Additional parameters on this tab

The following parameters are used in both methods.

- **Delimiter** - specify a delimiter which is used in your delimited data. Usually: comma, semicolon, tab. You may define tab character as #09.
- **Quote character** - specify one or more quote characters which are using in your CSV data flow. The format of this field: *<Opening quote 1><Closing quote 1><Format delimiter><Opening quote 2><Closing quote 2><Format delimiter>....<Opening quote N><Closing quote N>.* Where *Opening quote* and *Closing quote* are any characters. If the *Closing quote* isn't defined then *Opening quote = Closing quote. Format delimiter* = |. For example the following string "|[] defines two pairs of quotes " and [].
- **Detect column names row by word** - if you'll specify any value in this field, then the module will ignore all data rows which will contain this word.
- **Decode escapes like \n, \t, \x00** – some fields in your data flow may contain non-printable characters, which are encoded like: \t = ASCII TAB character, \n - ASCII line feed character and etc. This option allows you to decode these encoded characters automatically;
- **Export columns** - this options allow you to specify parsing options individually for each column in the data row. If you'll select the "**All as text**" then the module will export all columns from your data row with the "String" data type. The name of these exported variables will begin with the "COLUMN" word. Otherwise if you'll select the "**Specified only**" you can define columns which you want to export and a data type of each column on the "**Data parsing**" tab (fig.2).
- **Add a time stamp to each data row** – the parser will add the new "DATE_TIME_STAMP" variable to each data row that will be parsed into variables;
- **Add a data source ID to each data row** – the parser will add the new "DATA_SOURCE_ID" variable to each data row that will be parsed into variables; It will allow you to identify data during export if you collect data from several devices at the same time.

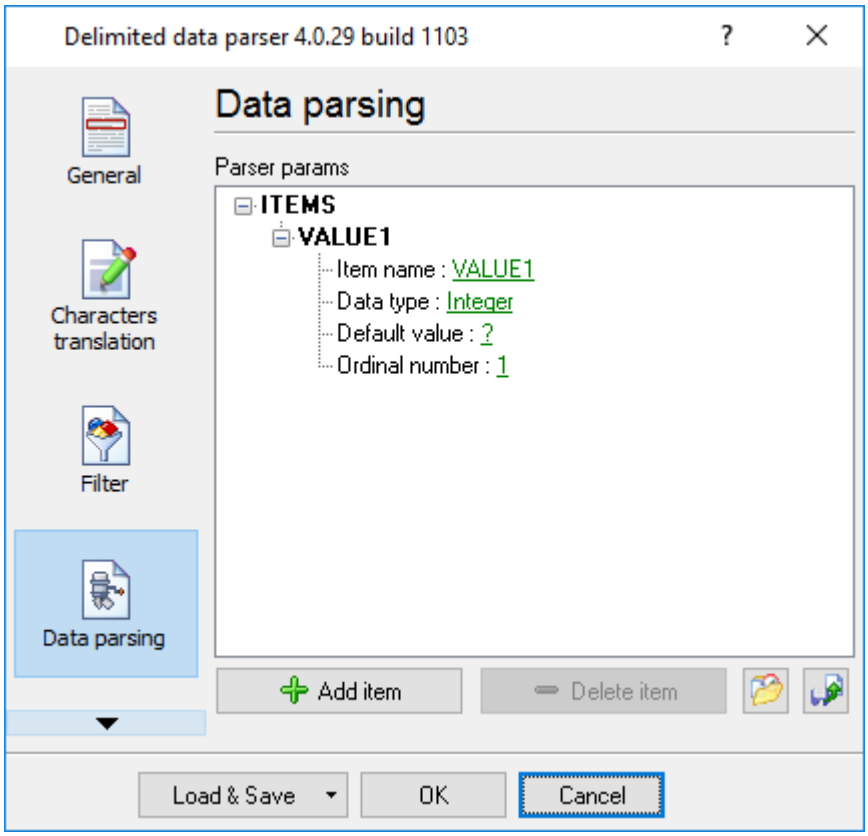

Fig.2. Data parsing

1. **Item name** - name of the parser variable. This item will be used in data export modules;
2. **Data type** - a data type of a column
3. **Default value** - the parser will use this value if can't extract the specified column and can't convert the extracted value to the specified data type;
4. **Ordinal number** - column number in your delimited data row.

Data parsing example

You can find several real-life examples on our site where different data parsing methods are used. It can help you understand how the parser works. https://www.aggsoft.com/serial-data-logger/tutorials.htm

Example:

Please take a look at the example below (we've received this data from a PBX):

```
223;741;09127;5:21:20;00:05:28;O;**********************************
5;13134;013;15:53:30;0:05:28;O;**********************************
```

Strings with data have different lengths in this case. But all variables are located in a certain order separated with a semicolon delimiter. In this example, you can single out data by their ordinal number (1..7) using the separator.

Data types

- String - String value - Character array with length from 1 to 65535 characters;
- Memo - String value - Character array with length from 1 to 2^32 characters;
- Bytes - Binary value;
- Blob - Binary Large Object field (bytes set);
- Boolean - Logical value (True/False) - 0 or 1;
- Float - Real number - value range: -2.9 x 10^-39 .. 1.7 x 10^38
- Smallint - Signed small integer - value range: 32768..32767;
- Word - Word (unsigned small integer) - value range: 0..65535;
- Integer - Signed integer value: -2147483648..2147483647;
- Date - Date;
- Time - Time;
- DateTime - Date and time.

6 Data format

You can use the following tab to specify the data format for some data types (see figure below).

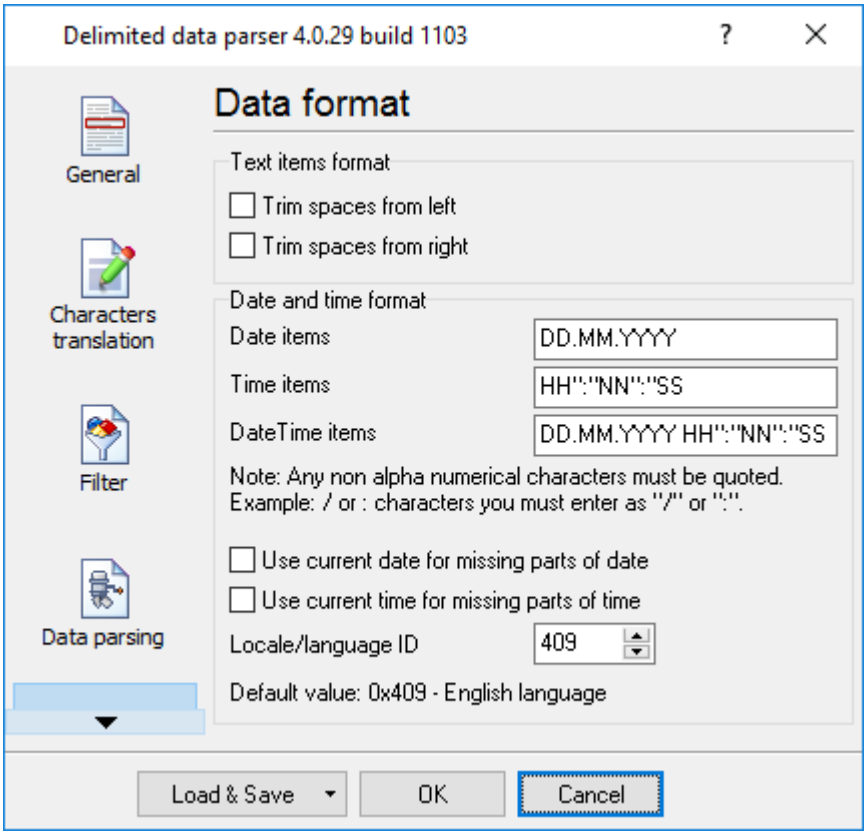

Fig. 2. Data format

Text data format – this group of options allows you to remove leading or trailing spaces in variables of the String type.

Date and time format – it often happens that the date or time format the external device sends does not coincide with the format in which this data is stored in the database. To convert a date into the necessary format, you should specify the format of the received date and time.

The same format is used to specify the date and time as the one that is used in the name of the log file in the main program (for example, Advanced Serial Data Logger). So please read about it in the help file of the main program. Note: the "/" and ":" characters are enclosed in quotation marks in the template (see figure above).

If some part is missing in the received date and time (for example, year), you can take the missing part from the current date and time:

Use the current date for missing parts;
Use the current time for missing parts;

Sometimes devices send dates with month names (for example, "Jan 10, 2024"), and this name can be in a language (in this example, it is in English) different from the language of your operating system. The language ID a date is sent in is specified in the "**Date language ID**" field in this case.

7 Characters translation

Character translation (fig.4) is used when you want to remove or replace some characters in a data packet. For example, remove nonprinting ASCII characters.

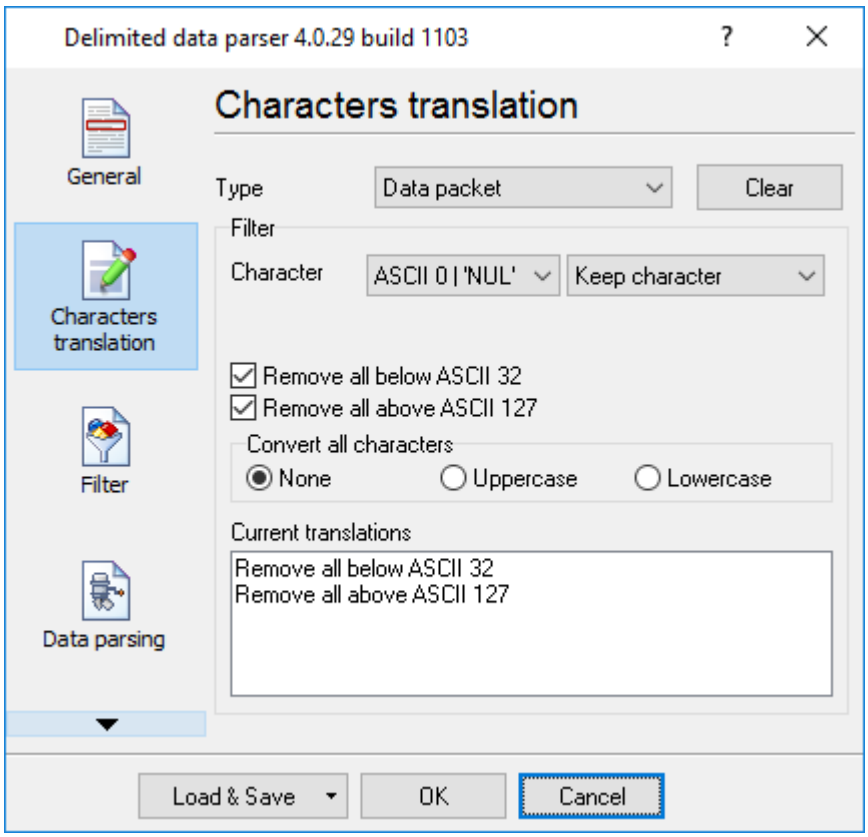

Fig.4 Character translation

8 Filter

The filter is used to ignore some data packets that you do not want to export with the help of other modules.

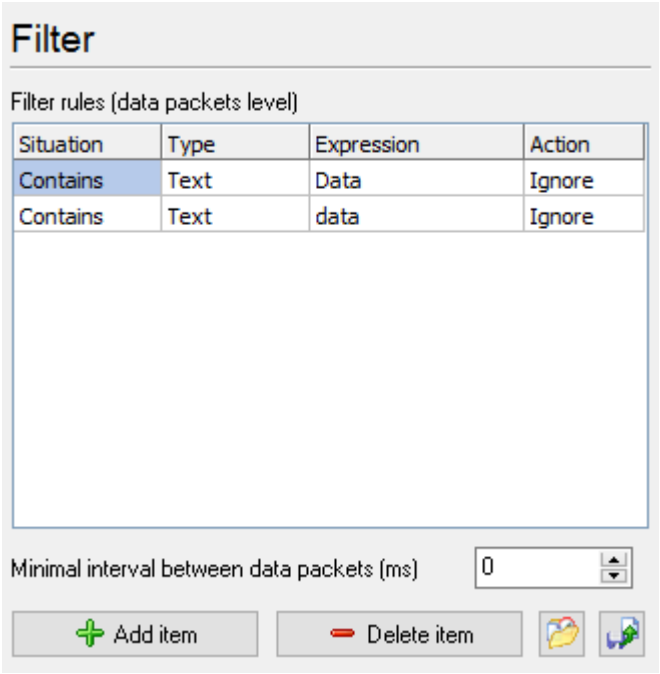

Fig. 3. Filtering rules

You should define one or several filtering rules. If some condition is true, the program does what is specified in the "Action" field with the matching packet.

Action types

- **Ignore** – the current data packet will be ignored and will not be exported;
- **Parse** – the current data packet will be parsed and exported.

Several condition types can be specified in the "Status" field.

Rule status types

- **Disabled** – this rule is disabled and not used for filtering purposes;
- **Contains** – this rules checks whether the string/expression from the "Expression" field is present in the data packet;
- **Does not contain** – this rule checks whether the string/expression from the "Expression" field is absent in the data packet.

Expressions types – Expressions in the "Expression" field can be of 2 types:

- **Text** – the module will search the data packet for a string specified in the "Expression" field. The search is case-sensitive.
- **Regular expression** – the module will use the regular expression specified in the "Expression" field in its search. The search is case-sensitive.

9 Syntax of Regular Expressions

Introduction

Regular Expressions are a widely-used method of specifying patterns of text to search for. Special metacharacters allow you to specify, for instance, that a particular string you are looking for occurs at the beginning or end of a line, or contains n recurrences of a certain character.

Regular expressions look ugly for novices, but they are a very simple, handy, and powerful tool.

Let's start our learning trip!

Simple matches

Any single character matches itself unless it is a metacharacter with a special meaning described below.

A series of characters matches that series of characters in the target string, so the pattern "bluh" would match "bluh" in the target string. Quite simple, eh?

If you want to use metacharacters or escape sequences or literally, you need to 'escape' them by the "\" backslash character. For instance, the "^" metacharacter matches the beginning of the string, but "\^" matches the "^" character and "\\" matches "\" and so on.

Examples:

`foobar`	*matches the 'foobar' string.*
`\^FooBarPtr`	*matches '^FooBarPtr'.*

Escape sequences

Characters may be specified using an escape sequence syntax, much like that used in C and Perl: "\n" matches a newline, "\t" a tab, etc. More generally, \xnn, where nn is a string of hexadecimal digits, matches the character whose ASCII value is nn. If you need wide (Unicode) character code, you can use '\x{nnnn}', where 'nnnn' - one or more hexadecimal digits.

`\xnn`	*char with hex code nn*
`\x{nnnn}`	*char with hex code nnnn (one byte for plain text and two bytes for Unicode)*
`\t`	*tab (HT/TAB), same as \x09*
`\n`	*newline (NL), same as \x0a*
`\r`	*car.return (CR), same as \x0d*
`\f`	*form feed (FF), same as \x0c*
`\a`	*alarm (bell) (BEL), same as \x07*
`\e`	*escape (ESC), same as \x1b*

Examples:

`foo\x20bar`	*matches 'foo bar' (note the space in the middle).*

`\tfoobar` *matches 'foobar' preceded by the tab character.*

Character classes

You can specify a **character class** by enclosing a list of characters in [], which will match any **one** character from the list.

If the first character after the "[" is "^", the class matches any character **not** in the list.

Examples:

`foob[aeiou]r` *finds strings 'foobar', 'foober' etc. but not 'foobbr', 'foobcr' etc.*
`foob[^aeiou]r` *find strings 'foobbr', 'foobcr' etc. but not 'foobar', 'foober' etc.*

Within a list, the "-" character is used to specify a **range**, so that a-z represents all characters between "a" and "z", inclusive.

If you want "-" itself to be a member of a class, put it at the start or end of the list, or escape it with a backslash. If you want ']' you may place it at the start of the list or escape it with a backslash.

Examples:

`[-az]` *matches 'a', 'z' and '-'*
`[az-]` *matches 'a', 'z' and '-'*
`[a\-z]` *matches 'a', 'z' and '-'*
`[a-z]` *matches all twenty six small characters from 'a' to 'z'*
`[\n-\x0D]` *matches any of #10,#11,#12,#13.*
`[\d-t]` *matches any digit, '-' or 't'.*
`[]-a]` *matches any char from ']'..'a'.*

Metacharacters

Metacharacters are special characters which are the essence of Regular Expressions. There are different types of metacharacters, described below.

Metacharacters - line separators

`^` *start of a line*
`$` *end of a line*
`\A` *start of a text*
`\Z` *end of a text*
`.` *any character in a line*

Examples:

`^foobar` *matches the 'foobar' string only if it's at the beginning of a line*
`foobar$` *matches the 'foobar' string only if it's at the end of a line*
`^foobar$` *matches the 'foobar' string only if it's the only string in a line*
`foob.r` *matches strings like 'foobar', 'foobbr', 'foob1r' and so on*

By default, the "^" metacharacter is only guaranteed to match at the beginning of the input string/text, the "$" metacharacter only at the end. Embedded line separators will not be matched by "^" or "$".

You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any line separator within the string, and "$" will match before any line separator.

The "." metacharacter by default matches any character.

Note that "^.*$" (an empty line pattern) does not match the empty string within the sequence \x0D\x0A, but matches the empty string within the sequence \x0A\x0D.

Metacharacters - predefined classes

`\w`	*an alphanumeric character (including "_")*
`\W`	*a non-alphanumeric*
`\d`	*a numeric character*
`\D`	*a non-numeric*
`\s`	*any space (same as [\t\n\r\f])*
`\S`	*a non space*

You may use \w, \d, and \s within custom **character classes**.

Examples:

`foob\dr` *matches strings like 'foob1r', "foob6r' and so on but not 'foobar', 'foobbr' and so on*

`foob[\w\s]r` *matches strings like 'foobar', 'foob r', 'foobbr' and so on but not 'foob1r', 'foob=r' and so on*

Metacharacters - iterators

Any item of a regular expression may be followed by another type of metacharacters - **iterators**. Using this type of metacharacters, you can specify the number of occurrences of the previous character, **metacharacter**, or **sub-expression**.

`*`	*zero or more ("greedy"), similar to {0,}*
`+`	*one or more ("greedy"), similar to {1,}*
`?`	*zero or one ("greedy"), similar to {0,1}*
`{n}`	*exactly n times*
`{n,}`	*at least n times ("greedy")*
`{n,m}`	*at least n but not more than m times ("greedy")*
`*?`	*zero or more ("non-greedy"), similar to {0,}?*
`+?`	*one or more ("non-greedy"), similar to {1,}?*
`??`	*zero or one ("non-greedy"), similar to {0,1}?*
`{n,}?`	*at least n times ("non-greedy")*
`{n,m}?`	*at least n but not more than m times ("non-greedy")*

So, digits in curly brackets of the form {n,m}, specify the minimum number of times to match the item n and the maximum m. The form {n} is equivalent to {n,n} and matches exactly n times. The form {n,} matches n or more times. There is no limit to the size of n or m, but large numbers will chew up more memory and slow down an execution time of a regular expression.

If a curly bracket occurs in any other context, it is treated as a regular character.

Examples:

`foob.*r` *matches strings like 'foobar', 'foobalkjdflkj9r' and 'foobr'*

`foob.+r` *matches strings like 'foobar', 'foobalkjdflkj9r', but not 'foobr'*

`foob.?r` *matches strings like 'foobar', 'foobbr' and 'foobr', but not 'foobalkj9r'*
`fooba{2}r` *matches the string 'foobaar'*
`fooba{2,}r` *matches strings like 'foobaar', 'foobaaar', 'foobaaaar' etc.*
`fooba{2,3}r` *matches strings like 'foobaar', or 'foobaaar', but not 'foobaaaar'*

A little explanation about "greediness". "Greedy" takes as many as possible, "non-greedy" takes as few as possible. For example, 'b+' and 'b*' applied to string 'abbbbc' return 'bbbb', 'b+?' returns 'b', 'b*?' returns empty string, 'b{2,3}?' returns 'bb', 'b{2,3}' returns 'bbb'.

Metacharacters - alternatives

You can specify a series of **alternatives** for a pattern using "|" to separate them, so that fee|fie|foe will match any of "fee", "fie", or "foe" in the target string (as would f(e|i|o)e). The first alternative includes everything from the last pattern delimiter ("(", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next pattern delimiter. For this reason, it's common practice to include alternatives in parentheses, to minimize confusion about where they start and end.
Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching foo|foot against "barefoot", only the "foo" part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)
Also remember that "|" is interpreted as a literal within square brackets, so if you write [fee|fie|foe], you're really only matching [feio|].

Examples:
`foo(bar|foo)` *matches strings 'foobar' or 'foofoo'.*

Metacharacters - subexpressions

The bracketing construct (...) may also be used to define a sub-expressions of the regular expression.

Sub-expressions are numbered based on the left to right order of their opening parenthesis. The first sub-expression has the number '1'.

Examples:
`(foobar){8,10}` *matches strings which contain 8, 9 or 10 instances of 'foobar'*
`foob([0-9]|a+)r` *matches 'foob0r', 'foob1r' , 'foobar', 'foobaar', 'foobaar', etc.*

Metacharacters - backreferences

Metacharacters \1 through \9 are interpreted as backreferences. \<n> matches previously matched **subexpression** #<n>.

Examples:
`(.)\1+` *matches 'aaaa' and 'cc'.*
`(.+)\1+` *also matches 'abab' and '123123'*
([' "] ?) (\ d+) \ 1 matches "'13" (in double quotes), or '4' (in single quotes) or 77 (without quotes) etc

Modifiers

Modifiers are used for changing the behaviour of the parser.

There are many ways to set up modifiers.
These modifiers may be embedded within the regular expression itself using the (?...) construct.

i
Do case-insensitive pattern matching (using installed in your system locale settings).
m
Treat string as multiple lines that change "^" and "$" from matching at only the very start or end of the string to the start or end of any line anywhere within the string.
s
Treat string as a single line that change "." to match any character whatsoever, even a line separators, which it normally would not match.
g
Non-standard modifier. It switches off all following operators into non-greedy mode (by default this modifier is On). So, if modifier /g is Off then '+' works as '+?', '*' as '*?' and so on.
x
Extend your pattern's legibility by permitting white-space and comments (see explanation below).

The modifier /x itself needs a little more explanation. It tells the parser to ignore whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The # character is also treated as a metacharacter introducing a comment, for example:

```
(
(abc) # comment 1
  |   # You can use spaces to format a regular expression. - parser ignores it
(efg) # comment 2
)
```

It also means that if you want real whitespace or # characters in the pattern (outside a character class, where they are unaffected by /x), that you'll either have to escape them or encode them using octal or hex escapes. These features go a long way towards making regular expressions text more readable.

How to change modifiers

`(?imsxr-imsxr)`
You may use it in a regular expression to change modifiers on-the-fly. If this construction is inlined into subexpression, it affects only into this subexpression.

Examples:

`(?i)New-York`	*matches 'New-york' and 'New-York'*
`(?i)New-(?-i)York`	*matches 'New-York' but not 'New-york'*
`(?i)(New-)?York`	*matches 'New-york' and 'new-york'*
`((?i)New-)?York`	*matches 'New-York', but not 'new-york*

`(?#text)`

It is a comment. The text inside brackets will be ignored. Note that the parser closes the comment as soon as it sees the ")" character, so there is no way to put a literal ")" in the comment.