Pattern Matching

Pattern Matching

Introduction

During the Spring and Summer of 1996 I was working on a project (a still unreleased text editor) which needed better search facilities than the simple literal search engine which I had devised for it. While there are many good search engines with freely available source code on the internet, the vast majority of these are written for use in a Unix environment using Unix memory management, command line interface and so forth and are not the ideal thing to use as is on the Mac. Over the course of those 5-6 months, I set out to port as many of these as I could to the Mac so that I could run comparative performance testing on all of them and choose the appropriate ones for my use.

At the same time I devised a series of memory management schemes for interchangeable use by each of these search engines where needed including an alloca like scheme which tracked memory allocation and ensured that no memory leaks occured (memory blocks that get allocated but end up forgotten in the confusion and do not get released from use). These memory management schemes allow each of these search engines to use the standard Mac relocatable memory allocation calls which help to prevent memory fragmentation unlike the Unix malloc call (which still remains an option). Surprisingly, this also resulted in slightly slower performance, although its ability to better coexist with the Mac system probably more than offsets this loss.

By the time the dust settled, I ended up with 18 different search engines working on the Mac including my literal search engine and wildcard engine, 4 BMG search engines (FAST), at least 8 regular expression engines, an approximation engine and a few other odd and end search engines. In the time that has passed, I no longer remember the exact origin and characteristics of each of the search engines although I hope to be able to piece it back together in time. This document is intended to bring together as much of this information which I can find into one place with the intent of bringing some kind of order to this state of chaos and to provide user documentation for these search engines. It will be an ongoing work with additional info added over time as needed.

Most of the source code for these search engines exists in the public domain for free use. However some of the regular expression matchers are from the GNU packages distributed by the Free Software Foundation. Due to an unusual licensing policy of the FSF, these particular engines could not be released to the public (even as freeware executables) unless I were to also release to the public domain ALL of my own supporting source code which is used in association with the modified FSF libraries. This source code represents my own blood and toil over the past two decades and I am not about to just give it away - no developer would do so other than those whose work is done exclusively for others. While I do not necessarily agree fully with the FSF license, I will abide by it and will limit the search engines used in my released software to those not covered by the FSF license. In the end, it is you the user who ends up on the short end of the stick because there have been several very useful programs or program features which I have wanted to release as freeware but which I have been unable to do because of this restriction.

A wide variety of pattern matchers is described here, each having its own unique capabilities and characteristics. For the novice user, the wildcard matcher is probably simpler to understand than the regular expression matchers and should be the first to be looked at. It does not provide the total versatility and control that is offered by regular expressions but comes close enough that it will allow you to perform most, if not all of the same tasks for which you might otherwise use regular expressions.

Definitions

Expression	An expression is an operation or a quantity stated in symbolic form. It may use symbols to represent numbers or abstract concepts for use in operations similar to arithmetic. An expression can also be thought of as any legal combination of symbols that represents a value. What is legal and illegal depends on the particular application and context of the expression. For example, x+5 is an expression, as also is the character string "Donkeys."

Expressions are used in programming languages, database systems, and spreadsheet applications. For example, in database systems, you use expressions to specify which information you want to see. These types of expressions are called queries.

Every expression consists of at least one operand and can have one or more operators. Operands are values, whereas operators are symbols that represent particular actions. In the expression

x + 5   x and 5 are operands, and + is an operator.

Expressions are often classified by the type of value that they represent. For example:

Boolean expressions :	Evaluate to either TRUE or FALSE
integer expressions:	Evaluate to whole numbers, like 3 or 100
Floating-point expressions:	Evaluate to real numbers, like -3.141
String expressions:	Evaluate to character strings

In our case we will be dealing exclusively with string expressions. These expressions consist of any combination of literals (the string equivalent of values) and string operators. For example we might encounter the wildcard expression of:

*dog?   where * and ? are wildcard operators and “dog” is a literal

Subexpression	As used in this document, a subexpression is defined to be any part of an expression which is in itself also a separate expression.
Literal	A literal is a string which is treated as the text which it contains without any special interpretation of its contents as operators or special characters of any kind.
Character	A single alphanumeric character. A single character is commonly expressed as the character surrounded by single quotes, for example: ‘s’.
String	A string is simply a series of zero or more alphanumeric characters which is treated as an entity or combined object. A string is commonly expressed as the string text surrounded by double quotes, for example: “catfish”.
Token	Something serving as an indication or an expression of something else. As used in this document, each individual element of an expression is considered to be a token. This includes all wildcard operators, all regular expression operators and all literals. Each literal string in such expressions is treated as a single token. This is my own term for lack of a better word.

Wildcard Expression	A wildcard expression as used in this document is a Find Pattern consisting of any combination of literal strings and wildcard tokens as defined in the wildcard syntax table shown below.
Regular Expression	A regular expression is a Find Pattern consisting of any combination of literal strings and regular expression operators as defined in the Regular Expression Syntax table shown below.
Search Engine	A program function which examines a block of text (a text document for instance) looking for matches to a specific string (the Find Pattern) which the user is trying to find within that block of text. The simplest (and most crude) method of doing this is to examine each character in sequence to see if it matches the first character in the Find Pattern. If it matches, then the entire pattern is compared to the succeding characters in the block of text. If not, then the next character is examined until the end of the document is reached. The Find Pattern may be a simple literal string as in most cases or it may be a regular expression or other complex pattern, from which comes the generalized term "pattern matching".
Matcher	A matcher is simply that portion of a search engine which compares the Find Pattern with the input text, taking into account the desired nature of the Find Pattern. As used by MacNetTools, the term matcher has a secondary meaning. Since MNT's use of search engines is primarily concerned with comparing file names with a user provided pattern ratehr than locating that pattern in a block of text, the matcher is considered to be a separate program function which combines the search engines to make the text comparison with an additional check to determine if the compared text spans the entire length of the filename. See "all or none restriction" below.
BMG Search Engine	Boyer-Moore-Gosper search engine - an extremely fast literal search engine which achieves its speed by preprocessing the find string into a table which allows multiple simultaneous comparisons. The BMG engine does not look at each character of the input text sequentially but jumps by n characters after each comparison where n is the length of the find string. Its inherent simplicity also leads to further speed increases.
Input Text	The text being examined (which in all cases as used by MNT is filename text).
Find Pattern	Pattern to be matched against the Input Text. This pattern may be any combination of plain text (literals) and tokens listed above. See examples below.
Repl Pattern	Pattern to be used as a replacement pattern in the context of the Input Text and the Find Pattern. This pattern may be any combination of plain text (literals) and tokens listed above. See examples below.
Repl Text	The replacement text built from the Input Text based on the comparison of the Find Pattern and the Repl Pattern.
Find Engine	The function(s) involved in matching the Find Pattern to the Input Text. Also referred to as the matcher. The Find Engine can always match any of the above tokens (assuming of course that the corresponding text exists in the Input Text) and is totally independent of the Repl Engine.
Repl Engine	The function(s) involved in generating the Repl Text from the Input Text based on the contents of the Repl Pattern. In MNT, text replacement operations are used only with the File Tools Rename command.

All or none restriction

All of the matchers as used by MacNetTools impose an all or none restriction - the entire input text string must be matched by the pattern provided or else it is considered to be a bad match. While the underlying search engines will easily find a specified pattern anywhere within the input string, our purpose here is in matching filename text with a specific pattern, looking for a go/no go result as a condition for determining another action (moving a file for example). For this reason the matchers all impose this restriction. We can find subpatterns within the input text by adding * tokens to each end of the Find Pattern so that this pattern will match the entire input string. This use of the * tokens applies to the wildcard matcher only; the specific tokens used for this purpose are different with the other matchers.

Wildcard Matcher

The wildcard pattern matching syntax is really nothing more than a greatly expanded form of the same syntax as used by MS-DOS for its wildcard matching of filenames and creation of replacement names. The matching tokens available and their meanings are described below. Note that with so many tokens to deal with, it becomes somewhat difficult to remember them all. For this reason, wherever there are edittext boxes for entry of such patterns, a set of either 8 or 10 buttons is also provided to facilitate entry of the desired tokens wherever the insertion point is located.

Wildcard Syntax

Button	Token	Hex	KeyBrd	Meaning
One	?	3F	?	Match any one character
Any	*	2A	*	Match any string of zero or more characters
Num	¢	A2	Option-4	Match any one digit character
Let	≈	C5	Option-X	Match any one letter character
Pun	§	A4	Option-6	Match any one punctuation character
Wht	∆	C6	Option-J	Match any one white character (space, tab, return)
Wst	«	C7	Option-\	Match any string of zero or more white characters
Dec	.	2E	.	Match any decimal character
ExN	±	B1	Opt-Shft-=	Match and extract any one digit in this character position if it exists
ExL	»	C8	Opt-Shft-\	Match and extract any one letter in this character position if it exists
Literal	N/A	N/A	N/A	Match actual text entered - any text other than the above tokens is considered to be a literal.

EditText Boxes and Checkboxes where applicable

Find Patn	Box for user entry of Find Pattern.
Repl Patn	Box for user entry of Repl Pattern.
Case Sensitive	If checked, the pattern matching will be performed in a case sensitive manner.
Use MetaChars	If checked, certain control characters will be input as a two character pair with a preceding caret ‘^’ indicating that it is a metacharacter pair. This feature of my text engine is not implemented in MNT because all Input Text is filename text and no filenames contain those characters which are supported as metacharacters (primarily return, line feed and tab).
Wild White	If checked, then white characters will be treated as wild characters (a rather obscure option).
Wild Decimal	If checked, then the decimal point character will be treated as a wild character. This is another obscure but slightly more useful option that was created to allow the pattern matching to more closely resemble that found in MS-DOS where matches pivoted around the decimal point separating the filename from the extension.
Sync Find/Repl	No longer applicable, needed or wanted in this version.
Lit Wild Repl	If checked, the Repl Pattern will be treated as a literal and will be directly substituted for the matched text with no replacement text being generated.

Replacement Operations

In version 1.2.1 the Wildcard Replacement Engine has been rewritten to be much simpler and less confusing while offering greatly enhanced capabilities. Gone now are the “Sync Find/Repl” option and the 25 replacement rules. The new Wildcard Replacement Engine has an interface very similar to that used by those regular expression matchers which have replacement provisions (see sections below). Basically, each token of the Find pattern is matched to a specific span of characters in the Input Text and that span of characters is assigned an index represented by the letters a-z which is used to represent it. These indices are assigned in order from a-z based on the position of the corresponding token in the Find Pattern and thus also the characters which they represent in the Input Text. They are not case sensitive and you may also use the letters A-Z or any combination of upper and lower case. The replacement method works exactly like that of the Regexp matcher except that letters are used rather than numbers allowing up to 26 tokens to be represented rather than only 9 as in Regexp. We will call these letter designations the replacement tokens in the description that follows. The Replacement Pattern may consist of any combination of escaped (preceded by the backslash char ‘\’) replacement tokens and/or literals in any order and the replacement engine will reassemble the text represented by those tokens in the order specified. There is room for a great amount of variability here, limited only by your imagination. Words and even individual letters within words can be reversed or rearranged in any order or even omitted from the replacement output. Here are a few examples which should show how it works better than can be described using words.

Wildcard Matching Examples:

T = Text being searched
F = Find Pattern
R = Replace Pattern
S = Replacement or Substitution Text generated by program
A = Array index letters

The first few examples will be fully broken down to show how this scheme works. Try following along using MNT's match check dialog while plugging in the T, F and R values as appropriate and it should help you to understand how each of these examples work.

```
T    "The lazy dog slept"
F    "*lazy*"
R    "\acrazy\c"
S    " The crazy dog slept "

     ----------------------------------------------------------------
T    |T|h|e| |l|a|z|y| |d|o|g| |s|l|e|p|t| | | | | | | | | | | | | |
     ----------------------------------------------------------------
F    |   *   |l a z y|         *         |
     ----------------------------------------------------------------
A    |   a   |   b   |         c         |
     ----------------------------------------------------------------

The text is then reassembled according to the replacement pattern of:

\a + “crazy” + \c or

“The “ + “crazy” + “ dog slept”
to give
“The crazy dog slept”
```

Now lets take a look at another example that looks more like a file name, although unusual (which is where this will all be used anyhow). Note that the Δ token in the Find Pattern below below matches any white character - we could also have used a space character in those same token positions. This is simple example of word reversal.

```
T    "dog eat cat.jpg"
F    "*Δ*Δ*.jpg"
R    "\e\b\c\d\a\f"
S    "cat eat dog.jpg"

     ----------------------------------------------------------------
T    |d|o|g| |e|a|t| |c|a|t|.|j|p|g| | | | | | | | | | | | | | | | |
     ----------------------------------------------------------------
F    |  *  |Δ|  *  |Δ|  *  |. j p g|
     ----------------------------------------------------------------
A    |  a  |b|  c  |d|  e  |   f   |
     ----------------------------------------------------------------

The text is reassembled as:

\e + \b + \c + \d + \a + \f or

“cat“ + “ ” + “eat” + “ ” + “dog” + “.jpg”
to give
“cat eat dog.jpg”
```

We can change this same example slightly by addition of a literal to totally alter the meaning of the result. We will also use those space characters mentioned above rather than the Δ token as described to show the alternate ways of matching text with various patterns.

```
T    "dog eat cat.jpg"
F    "* * *.jpg"
R    "\e\bloves\d\a\f"
S    "cat loves dog.jpg"


     -----------------------------------------------------------------
T    |d|o|g| |e|a|t| |c|a|t|.|j|p|g| | | | | | | | | | | | | | | | | |
     -----------------------------------------------------------------
F    |  *  | |  *  | |  *  |. j p g|
     -----------------------------------------------------------------
A    |  a  |b|  c  |d|  e  |    f  |
     -----------------------------------------------------------------

The text is reassembled as:

\e + \b + “loves” + \d + \a + \f or

“cat“ + “ ” + “loves” + “ ” + “dog” + “.jpg”
to give
“cat loves dog.jpg”
```

Here are some other examples of possible find and replace combinations. Please lets have no comments regarding my totally unusual fabrication of nonexistent words in some of these test cases which I came up with 8 years ago. I happened to spot an earwig on the floor on the day when these were written so the words frogwig and catwig seemed to somehow make sense at the time. I will try to think of a few new examples to go along with the old ones when time permits.

```
T    "Fox frogwig$ Dog fox"
F    "frog*§"
R    "cat\b\c"
S    ""
     Match Fails
```

The above attempt will return a failed match. See the paragraph above named 'All or none restriction" to find out why. We can change this example slightly by adding * tokens to each end of the Find Pattern so that we get the type of match desired:

```
T    "Fox frogwig$ Dog fox"
F    "*frog*§*"
R    "cat\c\d"
S    "catwig$"
```

Note also in above case that DOS wildcard syntax would not be able to match a pattern of "frog*?". The additional tokens added to MNT's wildcard syntax enables it to separate the § or punctuation character from all other characters enabling it to make matches that might not otherwise be possible. The same applies in many other cases as well.

```
T    "Fox frogwig$ Dog fox"
F    "*frog*§*"
R    "catdog\d"
S    "catdog$"

T    "Fox frogwig$ Dog fox"
F    "*frog*wig§*"
R    "cat\d\e"
S    "catwig$"
```

The above case was once considered invalid with the old version of the wildcard replacement engine. Note also how we can now separate parts of words using the * token as a null token which is used to match a point in the input text string but no actual characters. In this case we are separating frogwig into its frog and wig components.

```
T    "Fox frogwig$ Dog fox"
F    "*frog*§*"
R    "cat\c!"
S    "catwig!"

T    "Fox frogwig$ Dog fox"
F    "*frog*$*"
R    "cat\c\d"
S    "catwig$"
```

The above examples are relatively complex. Most of your typical usages are likely to be much simpler. Take for example the task of shortening the filenames of a group of images named as follows. In reality, the 44 character filenames involved are too long for the Mac OS file system... but not for OS X... but this is just an example:

```
Extremely Elegant Elephant Ear Leaves.01.jpg
Extremely Elegant Elephant Ear Leaves.02.jpg
Extremely Elegant Elephant Ear Leaves.03.jpg
Extremely Elegant Elephant Ear Leaves.04.jpg  etc

T    "Extremely Elegant Elephant Ear Leaves.01.jpg"
F    "Extremely Elegant Elephant Ear Leaves*"        Note the * at the end
R    "Elephant\b"
S    "Elephant.01.jpg"
```

Using these parameters with the Rename Matching command would rename this list of files to:

```
Elephant.01.jpg
Elephant.02.jpg
Elephant.03.jpg
Elephant.04.jpg
```

We could also do the same thing with a bit less writing with the following Find/Replace parameters, but this would also rename files such as the following:

```
Elephant Circus Show.01.jpg

T    "Extremely Elegant Elephant Ear Leaves.01.jpg"
F    "*Elephant*.*¢¢*"
R    "\b\d\e\f\g"
S    "Elephant.01.jpg"
```

Regular Expression Matchers

Regular expressions are a pattern matching feature of Unix which allow the construction of extremely precise and detailed patterns that can be designed to match either very specific portions of text or a very wide range of possible text patterns. A regular expression (commonly shortened to “regex” or “regexp”) consists of a string of ordinary text characters interspersed with the special characters which control he behavior of the regular expression. There are several different regex syntaxes available for use including grep, egrep, awk, emacs and others. Both of the regex matchers used in MNT make use of the egrep syntax. The table below is commonly used to describe the meanings of these characters. It applies to all regex described unless noted otherwise. Operator precedence is (highest to lowest) ?, *, and +, concatenation, and finally |. All other constructs are syntactically identical to normal characters. For links to more in depth (and complicated) descriptions of regular expressions, see the Credits section.

Regular Expression Syntax

(grep)	(egrep)	(explanation)
.	.	matches any single character except newline
\?	?	postfix operator; preceeding item is optional
*	*	postfix operator; preceeding item 0 or more times
\+	+	postfix operator; preceeding item 1 or more times
\\|	|	infix operator; matches either argument
^	^	matches the empty string at the beginning of a line - has different meaning when used in a list of characters
$	$	matches the empty string at the end of a line
\<	\<	matches the empty string at the beginning of a word
\>	\>	matches the empty string at the end of a word
[chars]	[chars]	match any character in the given class; if the first character after [is ^, match any character not in the given class; a range of characters may be specified by <first>-<last>; for example, \W (below) is equivalent to the class [^A-Za-z0-9]. To include the] character in such a list it must occur as the first character after the opening [.
\(\)	( )	parentheses are used to override operator precedence
\<1-9>	\<1-9>	\<n> matches a repeat of the text matched earlier in the regexp by the subexpression inside the nth opening parenthesis
\	\	any special character may be preceded by a backslash to match it literally

The following are for compatibility with GNU Emacs

\b	\b	matches the empty string at the edge of a word
\B	\B	matches the empty string if not at the edge of a word
\w	\w	matches word-constituent characters (letters & digits)
\W	\W	matches characters that are not word-constituent

Regexp Engine

Description

The Regexp engine was written by Dr. Henry Spencer of the University of Toronto. It was designed to be compatible with the Bell V8 regexp but was independently created and not derived from Bell source code. It is freely and commonly available in many internet archives under the name of regexp. Regexp is not particularly sophisticated and lacks many of the advanced configuration features of other regex engines but it is simple and easy to use while offering reasonably good performance. As such it is the ideal choice for the novice user who is just learning to use regular expressions. It follows the syntax listed in the table above exactly. Unlike most other regex engines, regexp includes a substitution feature for generating replacement text for text which has already been matched. This is where the greatest difference is likely to be found between various regex engines.

Regexp Replacement Ops

The Regexp replacement syntax is very similar to that used by the wildcard matcher (and for a good reason too - I patterned the wildcard matcher's substitution syntax after that of Regexp using letters instead of numbers and dropping the requirement for parentheses).

With Regexp, parentheses are used in the Find pattern to group portions of the pattern which are later to be used for substitution. These are given the invisible numbers 1-9 in the order in which they occur. In the Replacement pattern, the input text matched by these grouped portions of the Find pattern can then be invoked using an escaped (preceded by backslash) digit from 1-9 or \1, \2, ... \9. With Regexp only there is also one additional token that can be used in the Repl Patn. This is the ampersand & character which tells the replacement engine to use the entire matched portion of the input string as the substitution string which it outputs. You will see this in most of the test cases listed below but otherwise it will not be of any particular use in MNT. Take the example:

```
T    "word reverse"
F    "([a-z]+)([^a-z]+)([a-z]+)"
R    "\3\2\1"
S    "reverse word"
```

In the above example, the subexpression [a-z] tells the matcher to match any letter ranging from ‘a’ to ‘z’ in the input text. Adding the ‘+’ to the end to form [a-z]+ specifies that this range is to be matched anywhere from 1 to n times. Placing this subexpression in parentheses as ([a-z]+) groups this operation in terms of operator precedence but more importantly causes the input text matched by this subexpression to be saved in the first “save slot” for later reference by the substitution generator as item “\1”. This subexpression matches the word “word” in the input text. The second subexpression ([^a-z]+) is identical to the first except that the range specified is preceded by the ‘^’ symbol. This causes the subexpression to match anything except for the characters specified in the range a-z and in this case it matches the space character between “word’ and “reverse” in the input text. It is also grouped by parentheses causing the input text matched by this subexpression to be saved in the second “save slot” for later reference as item “\2”. The third subexpression ([a-z]+) is once again identical to the first and is also grouped in parentheses causing the input text matched by this subexpression to be saved in the second “save slot” for later reference as item “\3”. It matches the word “reverse in the input text. The Repl Patn specifies that the input text is to be reassembled as the text matched by the third subexpression (\3) followed by the text matched by the second subexpression (\2) followed by the text matched by the first subexpression (\1). This means that we put it back together as follows:

```
\1 = “word”
\2 = “ ”         (space character)
\3 = “reverse”

\3            +    \2    +    \1
“reverse”     +    “ “   +    “word”      =     “reverse word”.
```

It is never necessary to specify a substitution string (or Repl Patn) except with the Rename Matching command which is the only MNT command which uses it. In most cases you will only be concerned with providing a Find Patn to match one or many filenames for which you wish to perform the command specified (Move Matching for example). In those cases where a substitution string is unused, it is simply ignored.

Regexp Examples

As always, the best way to understand such constructs is to see them in action through use of examples. Regular expressions can look very complex and intimidating at first, but by looking at each of the features alone, they can be made simpler. As with the wildcard matcher, try following along using MNT's match check dialog while plugging in the T, F and R values as appropriate and it should help you to understand how each of these examples work.

```
T    "abbbbc"
F    "ab+bc"
R    "&"
S    "abbbbc"
```

First lets take a look at the simple example above. In this case we first encounter the a in the regular expression of ab+bc. This matches the initial character a in the input string. The b+ which follows matches 1 or more instances (thats what the + signifies) of the character b. We have 4 of them to be matched, but at this point the regex engine looks ahead and sees that it must match yet another b in the token that follows so it only matches 3 of them with the b+. Then the tokens bc match the remaining bc in the input text so this is a good (and full) match. With a Repl Patn of "&", all of the matched text gets sent to the output as "abbbbc".

```
T    "abbbxcd"
F    "ab*xc"
R    "&"
S    ""
```

The above case proceeds identically to the previous case except that the * operator causes the first b in the regular expression to be matched zero or more times (the same expression could also match an input string of axc as well as abbbxc). We get a partial match since the abbbxc portion of the input string is matched by the entire expression but my MNT's rules requiring full matches, this gets treated as a failed match as far as any action being taken on this item and the substitution output is suppressed.

```
T    "abcde"
F    "(ab|cd)*e"
R    "&-\1"
S    "abcde-cd"
```

In the above case we have the grouped subexpression (ab|cd) which will match either ab OR cd in the input text. This group is followed by the * operator which causes the entire group to be repeated zero or more times until it fails to make a match, so the first time around it matches the ab, then the second time around it mtches the cd. The grouping parentheses also cause the matched text to be sent to the first "save slot" for later reference by the substitution code as \1. However, you may notice that we have done this twice. The "save slot" cannot make a decision as to which set of matched text it prefers so it will always contain the last text matched by that group, or cd in this case. Then finally we match the e by the last token in the expression. We get a full match and the substitution text gets assembled as follows. The & represents the entire matched input text which is followed by a literal hyphen '-' and that is followed by the text in the \1 "save slot" or cd.

```
& = "abcde"
- = "-"         (space character)
\1 = "cd"

&           +     -     +     \1
"abcde"     +    "-"    +    "cd"   =    "abcde-cd".
```

Example of Move Matching - Regexp

Now lets take a look at an actual real world example in detail as it might be encountered in MNT, first by the Move Matching and then by the Rename Matching commands. Lets say that we have a series of files with names similar to the following list and contained within a larger list of files:

catpaw02.gif	dogpaw01.jpg	tigerpaw003.jpg	
catpaw05.gif	dogpaw02.jpg	tigerpaw004.jpg	
catpaw08.gif	dogpaw05.jpg	tigerpaw007.jpg	etc

We wish to move all of these files into a separate folder without all of the other files contained in the original folder. Obviously we must first set up our Move To folder as described under the Dest Fldr and Folder Button items in the File Tools section. Next we must devise a regular expression which will match those files which we wish to move but which will not match any others. On looking at the pattern of the filenames we see that there is one item which is consistent acros all of the files to be matched. This is the string "paw" which we may simply include in the regular expression as a literal. We will ignore the T, S and R lines of our previous TFRS charts used up to now and use just the F line to piecee together the parts of our regular expression. So far we know that we must match the string "paw" as a literal, thus:

```
F    "paw"
```

Now we also want to match a variety of letter characters (we will limit ourselves to lowercase letters for simplicity). Furthermore we do not know a specific number of characters to be matched because it varies with the filenames. To match this a range of [a-z] would work just fine and since we must match a variable number of characters greater than zero, the + postfix operator is just what we need giving us [a-z]+ as the desired subexpression. So now we have the following:

```
F    "[a-z]+paw"
```

Next we want to match a variable number of digits following the "paw" literal. A subexpression similar to the above case will also work just fine but one which describes a series of digits rather than letters. Whis would give us [0-9] and once again since the number of digits varies and is greater than zero, we can once again use the + operator to give [0-9]+ as our added subexpression to give:

```
F    "[a-z]+paw[0-9]+"
```

Next we must match that ‘.’ character. We could include it as is as a single character subexpression of ‘.’ but the ‘.’ character is itself an operator which just happens to match any character except newline. This would serve the purpose but would also match filenames such as “catpaw02tgif” which is not similar to others on our list of desired files. The correct way to do this is to escape the ‘.’ character so that it matches only the ‘.’ character. To do this we precede it with the escape operator giving us “\.” as our added subexpression. So now we have:

```
    F    "[a-z]+paw[0-9]+\."
```

Finally we must match either the gif or jpg extensions which we can do directly with the | operator giving us a subexpression of (gif|jpg) which will match either the string “gif’ or the string “jpg”. It is placed in parentheses so that it is treated as a single entity. This is probably not necessary but helps with the readability. Since this is to be matched only once, there is no need for any postfix operators such as * or + following it. Adding this to our regular expression gives a final result of:

```
    F    "[a-z]+paw[0-9]+\.(gif|jpg)"
```

Try using this regular expression in MNTs Match Check Dialog with any of the original filenames and you will get a full match (don’t do as I did and forget to select the Regexp matcher on the matcher popup). If this regular expression is used with the File Tools Move Matching (and with Regexp selected), the desired files and no others will be selectively moved to the chosen Move To folder.

Example of Rename Matching - Regexp

Now lets take the same example files which we used above and bulld the expressions which would be needed in order to rename these files using the Rename Matching command. Since we are talking about animals in all cases, lets rename the files to animalpawsxx.yyy where xx is the same 2 (or 3) digit number in each of the filenames and yyy is the extension used in each of the originals. It will not happen with the filenames chosen, but if for some reason we were to generate identical filenames for two or more of the renamed files, those following the first file would simply have a suffix of .00, .01 etc appended to the final filename. First of all let repeat our input names to keep from having to flip pages and also our original regular expression.

catpaw02.gif	dogpaw01.jpg	tigerpaw003.jpg	
catpaw05.gif	dogpaw02.jpg	tigerpaw004.jpg	
catpaw08.gif	dogpaw05.jpg	tigerpaw007.jpg	etc

```
    F    "[a-z]+paw[0-9]+\.(gif|jpg)"
```

The only change which we need to make in this case is the addition of one set of parentheses in order to provide grouping and a save slot assignment. The only thing that will change is the name of the animal being changed to the literal “animal” and [a-z]+ is the subexpression that matches the animal name. So we will simply group everything else in order to retain the text matched by the rest of the regular expression as shown below. Note the extra open parenthesis added before “paw” and the extra close parenthesis added at the very end of the expression. The save slots are assigned in the order in which the opening parentheses are encountered in the expression so slot 1 would represent all of the text matched after the animal name. This is exactly what we need in this case.

```
    F    "[a-z]+(paw[0-9]+\.(gif|jpg))"
```

We will also need a replacement expression in order to instruct the regex substitution code how to reassemble the output fragments. Since we want to change each animal name to the string “animal” we will begin with that string as a literal.

```
    R    "animal"
```

Now we need to tack on the rest of the text which we have saved in save slot 1 so we simply add the escaped number 1 or \1 to invoke this saved text as follows:

```
     R    "animal\1"
```

This is all that we need. Repeating our expressions, we can use the following set of Find and Repl expressions with Rename Matching (and Regexp) to rename all of the files listed (and only those files) to the name animalpawsxx.yyy which we devised.

```
     F    "[a-z]+(paw[0-9]+\.(gif|jpg))"
     R    "animal\1"
```

Regexp Test Cases

The following is a list of test cases used to validate the matcher. These may be used for further examples to help in learning the various syntactical features of the matcher. Note that in some of the Regexp cases, a lone hyphen character “-” signifies an empty string and not the hyphen character itself. Some of the test cases shown below indicate that they will generate a match (match OK = y) but that match will only be a partial match. This will be indicated by the MNT wildcard match dialog. Although the Regexp engine has passed this full range of tests under automated conditions, I have not attampted to perform each of these tests manually in MNT in order to flag those which are only partial matches. Those tests for which match OK = c are conditional cases used to test various alternative flags in the Regexp engine and should probably not be used as examples for learning.

Input Text	Find Patn (Regular Expr)	Match OK?	Repl Patn (Sub String)	Substitution Text Generated
T	F		R	S
abc	abc	y	&	abc
xbc	abc	n	-	-
axc	abc	n	-	-
abx	abc	n	-	-
xabcy	abc	y	&	abc
ababc	abc	y	&	abc
abc	ab*c	y	&	abc
abc	ab*bc	y	&	abc
abbc	ab*bc	y	&	abbc
abbbbc	ab*bc	y	&	abbbbc
abbc	ab+bc	y	&	abbc
abc	ab+bc	n	-	-
abq	ab+bc	n	-	-
abbbbc	ab+bc	y	&	abbbbc
abbc	ab?bc	y	&	abbc
abc	ab?bc	y	&	abc
abbbbc	ab?bc	n	-	-
abc	ab?c	y	&	abc
abc	^abc$	y	&	abc
abcc	^abc$	n	-	-
abcc	^abc	y	&	abc
aabc	^abc$	n	-	-
aabc	abc$	y	&	abc
abc	^	y	&	
abc	$	y	&	
abc	a.c	y	&	abc

Input Text	Find Patn	Match	Repl Patn	Substitution
Text				
	(Regular Expr)	OK?	(Sub String)	Generated
T	F		R	S
axc	a.c	y	&	axc
axyzc	a.*c	y	&	axyzc
axyzd	a.*c	n	-	-
abc	a[bc]d	n	-	-
abd	a[bc]d	y	&	abd
abd	a[b-d]e	n	-	-
ace	a[b-d]e	y	&	ace
aac	a[b-d]	y	&	ac
a-	a[-b]	y	&	a-
a-	a[b-]	y	&	a-
-	a[b-a]	c	-	-
-	a[]b	c	-	-
-	a[	c	-	-
a]	a]	y	&	a]
a]b	a[]]b	y	&	a]b
aed	a[^bc]d	y	&	aed
abd	a[^bc]d	n	-	-
adc	a[^-b]c	y	&	adc
a-c	a[^-b]c	n	-	-
a]c	a[^]b]c	n	-	-
adc	a[^]b]c	y	&	adc
abc	ab|cd	y	&	ab
abcd	ab|cd	y	&	ab
def	()ef	y	&-\1	ef-
-	()*	c	-	-
-	*a	c	-	-
-	^*	c	-	-
-	$*	c	-	-
-	(*)b	c	-	-
b	$b	n	-	-
-	a\	c	-	-
a(b	a\(b	y	&-\1	a(b-
ab	a\(*b	y	&	ab
a((b	a\(*b	y	&	a((b
a\b	a\\b	y	&	a\b
-	abc)	c	-	-
-	(abc	c	-	-
abc	((a))	y	&-\1-\2	a-a-a
abc	(a)b(c)	y	&-\1-\2	abc-a-c
aabbabc	a+b+c	y	&	abc
-	a**	c	-	-
-	a*?	c	-	-
-	(a*)*	c	-	-
-	(a*)+	c	-	-
-	(a|)*	c	-	-
-	(a*|b)*	c	-	-
ab	(a+|b)*	y	&-\1	ab-b
ab	(a+|b)+	y	&-\1	ab-b
ab	(a+|b)?	y	&-\1	a-a

Input Text	Find Patn	Match	Repl Patn	Substitution
Text				
	(Regular Expr)	OK?	(Sub String)	Generated
T	F		R	S
cde	[^ab]*	y	&	cde
-	(^)*	c	-	-
-	(ab|)*	c	-	-
-	)(	c	-	-
abc		y	&	
	abc	n	-	-
	a*	y	&	
abbbcd	([abc])*d	y	&-\1	abbbcd-c
abcd	([abc])*bcd	y	&-\1	abcd-a
e	a|b|c|d|e	y	&	e
ef	(a|b|c|d|e)f	y	&-\1	ef-e
-	((a*|b))*	c	-	-
abcdefg	abcd*efg	y	&	abcdefg
xabyabbbz	ab*	y	&	ab
xayabbbz	ab*	y	&	a
abcde	(ab|cd)e	y	&-\1	cde-cd
hij	[abhgefdc]ij	y	&	hij
abcde	^(ab|cd)e	n	x\1y	xy
abcdef	(abc|)ef	y	&-\1	ef-
abcd	(a|b)c*d	y	&-\1	bcd-b
abc	(ab|ab*)bc	y	&-\1	abc-a
abc	a([bc]*)c*	y	&-\1	abc-bc
abcd	a([bc]*)(c*d)	y	&-\1-\2	abcd-bc-d
abcd	a([bc]+)(c*d)	y	&-\1-\2	abcd-bc-d
abcd	a([bc]*)(c+d)	y	&-\1-\2	abcd-b-cd
adcdcde	a[bcd]*dcdcde	y	&	adcdcde
adcdcde	a[bcd]+dcdcde	n	-	-
abc	(ab|a)b*c	y	&-\1	abc-ab
abcd	((a)(b)c)(d)	y	\1-\2-\3-\4	abc-a-b-d
alpha	[a-zA-Z_][a-zA-Z0-9_]*	y	&	alpha
abh	^a(bc+|b[eh])g|.h$	y	&-\1	bh-
effgz	(bc+d$|ef*g.|h?i(j|k))	y	&-\1-\2	effgz-effgz-
ij	(bc+d$|ef*g.|h?i(j|k))	y	&-\1-\2	ij-ij-j
effg	(bc+d$|ef*g.|h?i(j|k))	n	-	-
bcdd	(bc+d$|ef*g.|h?i(j|k))	n	-	-
reffgz	(bc+d$|ef*g.|h?i(j|k))	y	&-\1-\2	effgz-effgz-
-	((((((((((a))))))))))	c	-	-
a	(((((((((a)))))))))	y	&	a
uh-uh	multiple words of text	n	-	-
multiple words, yeah	multiple words	y	&	multiple words
abcde	(.*)c(.*)	y	&-\1-\2	abcde-ab-de
(a, b)	\((.*), (.*)\)	y	(\2, \1)	(b, a)
ab	[k]	n	-	-

Regexs Engine

Description

The Regexs engine was written by a company named English Knowledge Systems, Inc and was distributed as part of another archive under the filenames of sr.c and sr.h. I know very little else regarding this engine except that like Regexp, it is fairly simple in its approach and that it also includes a substitution feature for generating replacement text for text which has already been matched. In terms of matching, it does follow the same syntax as that of Regexp but is somewhat less reliable and can give the user an occasional “hiccup”.

Regexs Replacement Ops

With Regexs, a $ character followed by a single letter is used in the Find pattern to identify those portions of input text matched by the most recent subexpression in the Find pattern as a unique entity which may later be referenced by that letter (just as the parentheses are used in Regexp). In the Replacement pattern, the input text matched by these portions of the Find pattern can then be invoked using a letter surrounded by the angle bracket characters. For example:

```
T    "word reverse"
F    "[a-z]+$a[^a-z]+$b[a-z]+$c"
R    "<c><b><a>"
S    "reverse word"
```

The syntax here is identical to the example given for Regexp except that there are no parentheses used. Grouping with parentheses is not required, although it may be used to assign more than just the most recent subexpression to the specified letter indentifier (see Regexs version of “Real World Example for Rename Matching" that follows several pages below). With Regexs the text put into each save slot is identified by &a, &b etc rather than by use of parentheses.

In the case shown above substitution, works as follows: The $a is used to specify that the input text matched by the first subexpression [a-z]+ is placed into save slot a which will contain the string “word”. The $b is used to specify that the input text matched by the second subexpression [^a-z]+ is placed into save slot b which will contain the string “ ”. The $c is used to specify that the input text matched by the third subexpression [a-z]+ is placed into save slot c which will contain the string “reverse”. The Repl Patn specifies that the input text is to be reassembled as the text matched by the third subexpression <c> followed by the text matched by the second subexpression <b> followed by the text matched by the first subexpression <a>. This means that we put it back together as follows:

```
<a> = “word”
<b> = “ ”              (space character)
<c> = “reverse”

<c>          +     <b>    +     <a>
“reverse”    +     “ “    +     “word”      =     “reverse word”.
```

Example of Move Matching - Regexs

With Regexs, this example is identical to the Move Matching example for Regexp. See that example for full details and description.

Example of Rename Matching - Regexs

See Regexp example for full details and description. With Regexs, the Rename Matching example is very nearly the same as that for Regexp. We only need to add the $a identifier to the end of the series of subexpressions grouped by the outer set of parentheses in order to assign the text matched by that group to save slot a. If you remember from the section on Regexs replacement operations above, we previously stated that grouping was not required for the Regexs substitution code to work properly, but that grouping could be used to change the subexpression(s) representing the text to be saved in each of the letter designated save slots. This is where we use this form of grouping. Without the outer parentheses shown below, the $a would only save the text matched by the last subexpression (gif|jpg) which would be either the string “gif” or “jpg” as applicable (try it and see for yourself). By grouping the subexpressions as shown we are able to assign the text matched by the entire expression excluding the first subexpression to the save slot as we did in the Regexp example. Again, this is just what we want. In the Repl Pattern we only need to change the form of the designation to retrieve the text in the save slot to <a> instead of \1 as used with Regexp. The final pair of expressions which we need are as follows:

```
F    "[a-z]+(paw[0-9]+\.(gif|jpg))$a"
R    "animal<a>"
```

Regexs Test Cases

The following is a list of test cases used to validate the matcher. These may be used for further examples to help in learning the various syntactical features of the matcher. I am uncertain as to the meaning of the third column in these test cases which are somewhat limited in scope as compared to the Regexp tests. Since both matchers follow the same syntax you can always use the Regexp test cases with the Regexs engine as long as no substitution is invoked. As mentioned before the primary difference between these two engines is in the substitution syntax.

Input Text	Find Patn (Regular Expr)	???
T	F	
aabada	ba	ca
aabada	b.d	ca
aabada	b\.d	ca
aabada	[acb].d	ca
aabada	[^acd].d	ca
aabada	[a-d].d	ca
aabada	[-acb].d	ca
aabada	[^-ac].d	ca
aabada	[acb-].d	ca
aabada	<2-4>	ca

Input Text	Find Patn (Regular Expr)	???	
T	F		
aabada	<2,4>	ca	
aabada	<2-~2>	ca	
aabada	<2>	ca	
aabada	<~2>	ca	
aabada\n	<~2>	ca	
aabaaada	ba+	ca	
aabda	ba+	ca	
aabaaada	ba*	ca	
aabda	ba*	ca	
aababada	(ba)*	ca	
aababada	aa(ba)*	ca	
aababada	(ba)+	ca	
aabada	aa(ba)*	ca	
aabada	(ba)+	ca	
aada	aa(ba)*	ca	
aada	(ba)+	ca	
aabada	(c|b)	ca	
aabada	c|b	ca	
aabada	(b|c)	ca	
aabada	b|c	ca	
aabada	(c|e|b|d|f)	ca	
aabababababada	aa(ba){2,4}	ca	
aabababababada	aa(ba){4,2}	ca	
aabababababada	aa(ba){4}	ca	
aabababababada	aa(ba){2}	ca	
aabababababada	aa(ba){2,}	ca	
aabababababada	aa(ba){,2}	ca	
aabababababada	aa(ba){4,}	ca	
aabababababada	aa(ba){,4}	ca	
aabada	(ba)$c	c<c>a	
word reverse	[a-z]+$a[^a-z]+$b[a-z]+$c	<c><b><a>	
abbaac	[a-zA-Z]$a<a>	<a>	
remove double double words		[a-z]+$a[^a-z]+<a>	<a>

Match Check Dialog

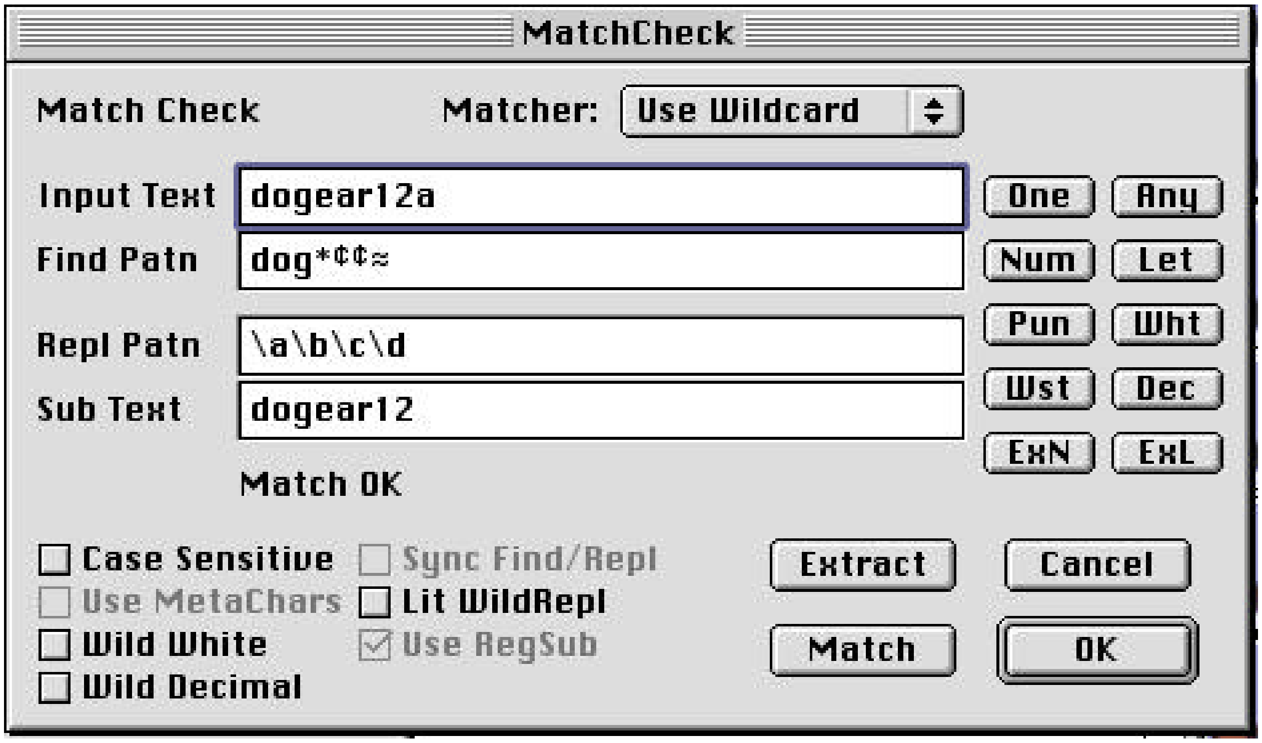
Presents a dialog which allows the user to experiment with various patterns which are to be input to the wildcard matcher to see what works prior to using those patterns with other MNT command which use them. In this way you can be assured of getting the desired results **before** you go moving all those files to places unseen. This can be a help because on rare occasions the wildcard matcher may give unpredictable results depending on the patterns which you feed to it.

Buttons - One, Any, Num, Let, Pun, Wht, Wst, Dec, ExN, ExL - all are described in document on Pattern Matching

Find Patn, Case Sensitive, Use MetaChars, Wild White, Wild Decimal, Sync Find/Repl, Lit Wild Repl - all are described in document on Pattern Matching

Match Dialog used to test Find and Replace matching via the Match button

The user enters text to be searched in the "Input Text" box which simulates the filename to be matched by the various commands. The wildcard pattern to be matched is then entered into the "Find Patn" box. If a replacement name is to be generated based on the above two items, then the replacement pattern is also entered into the "Repl Patn" box. Then click the Match button to invoke the pattern matcher. The success or failure of the match operation and of the replace operation, if any, will be displayed in text below the "Sub Text" box. If a replace operation is called for and if this operation succeeds, then the replacement text will be inserted into the "Sub Text" box.


MatchCheck

Match Check

Matcher: Use Wildcard

Input Text: dogear12a

Find Patn: dog*¢¢≈

Repl Patn: \a\b\c\d

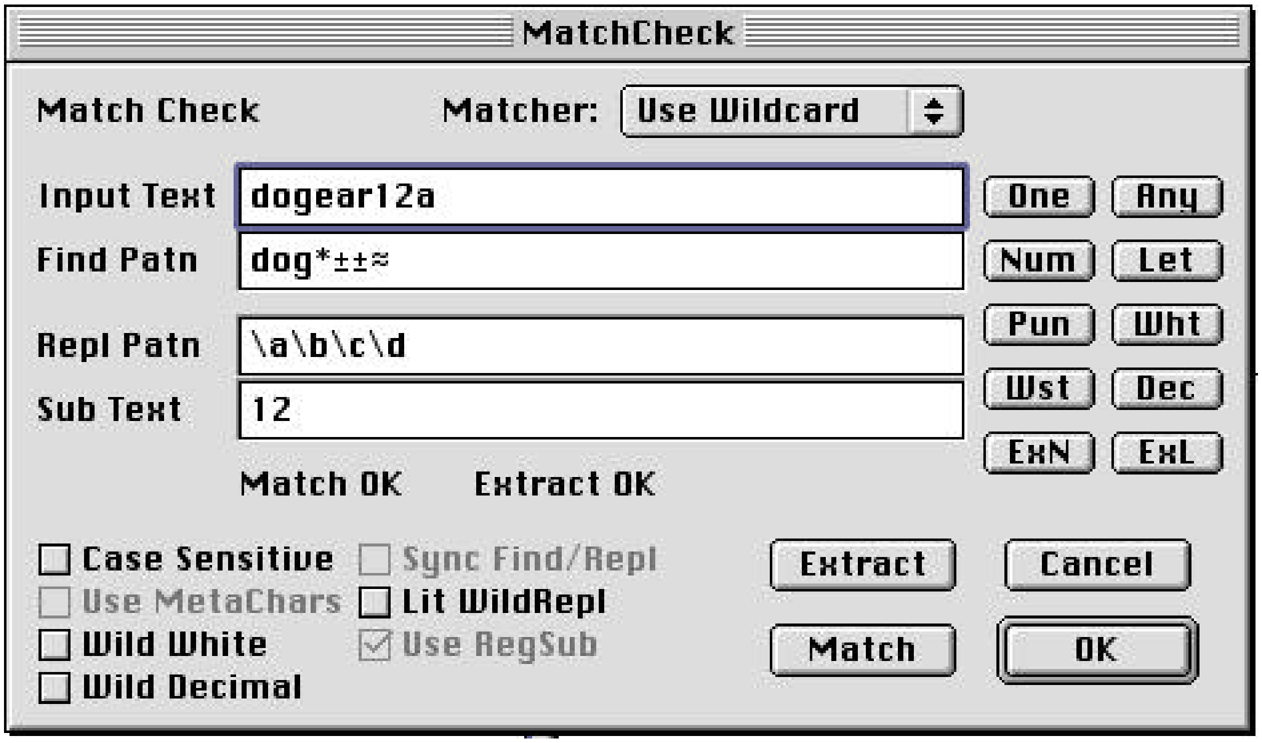
Sub Text: dogear12

Match OK

☐ Case Sensitive
☐ Use MetaChars
☐ Wild White
☐ Wild Decimal
☐ Sync Find/Repl
☐ Lit WildRepl
☑ Use RegSub

Match Dialog used to test numeric extraction via the Extract button

The Extract function is used only in the List Files command for MPG Patn List and MPG Patn Text formats. The user enters text to be searched in the “Input Text” box which simulates the filename to be matched by the various commands. The wildcard pattern to be matched is then entered into the “Find Patn” box. Included in this pattern should be one of the two extraction tokens (± or » for numeric or letter characters respectively - see document on Pattern Matching). The contents of the “Repl Patn” box are ignored in this case. Then click the Extract button to invoke the pattern matcher. The success or failure of the match operation and of the extract operation, if any, will be displayed in text below the “Sub Text” box. If the extract operation succeeds, then the extracted text will be inserted into the “Sub Text” box.



Matcher: Use Wildcard

Input Text	dogear12a
Find Patn	dog*±±≈
Repl Patn	\a\b\c\d
Sub Text	12

Match OK Extract OK

☐ Case Sensitive
☐ Use MetaChars
☐ Wild White
☐ Wild Decimal
☐ Sync Find/Repl
☐ Lit WildRepl
☑ Use RegSub

Credits and References

Thanks, credits, acknowledgments and references to other useful associated or similar products and documents are as follows:

Apple Computer Inc. for the wonderful Macintosh itself

The following referenced items are trademarks of Apple Computer Inc.
Apple , Macintosh, MacOS, Mac OS X

MetroWerks Inc. for the Codewarrior Pro Development System used to create MNT

UNIX is a trademark of Unix System Laboratories, Inc.

MS-DOS is a trademark of Microsoft Corp.

All other trademarks are held by their respective owners

Thanks also go to The Walt Disney Company for creating the many fine cartoon characters whose names served as sample filenames in this documentation.

Portions of definition for the term "expression" come from:
http://www.pcwebopedia.com/

Regex Matchers

Regexp
Copyright (c) 1986 by University of Toronto.
Written by Dr. Henry Spencer @ U of Toronto Zoology
Both code and manual page were written at U of T.
They are intended to be compatible with the Bell V8 regexp(3), but are not derived from Bell code.

Regexs
Copyright 1989 by English Knowledge Systems, Inc. All Rights Reserved.
Original Filenames were SR.C, SR.H which were part of some unknown archive
English Knowledge Systems, 408-438-6922 ????

Regular Expression Syntax (try the following links for more info)

http://py-howto.sourceforge.net/regex/regex.html
http://wks.uts.ohio-state.edu/unix_course/intro-73.html
http://sunsite.ualberta.ca/Documentation/Gnu/rx-1.5/html_node/regex_toc.html
http://cclib.nsu.ru/projects/gnudocs/win/gnudocs/regex/regex_toc.html
http://sunland.gsfc.nasa.gov/info/regex/Top.html
http://activedeveloper.dk/iishelp/jscript/htm/jsgrpregexpsyntax.htm
http://webdocs.caspur.it/ibm/web/vacpp-5.0/lpex/ref/rlrgxsyn.htm

Final Comments & Contact info

Phil Rogers

nettools@nym.alias.net	Primary email address
macnettools@hotmail.com	Alternative email address - mail to this one is less often checked
http://ksinksw.tripod.com/	MacNetTools Web Site and Primary Distribution Source

--