Searching

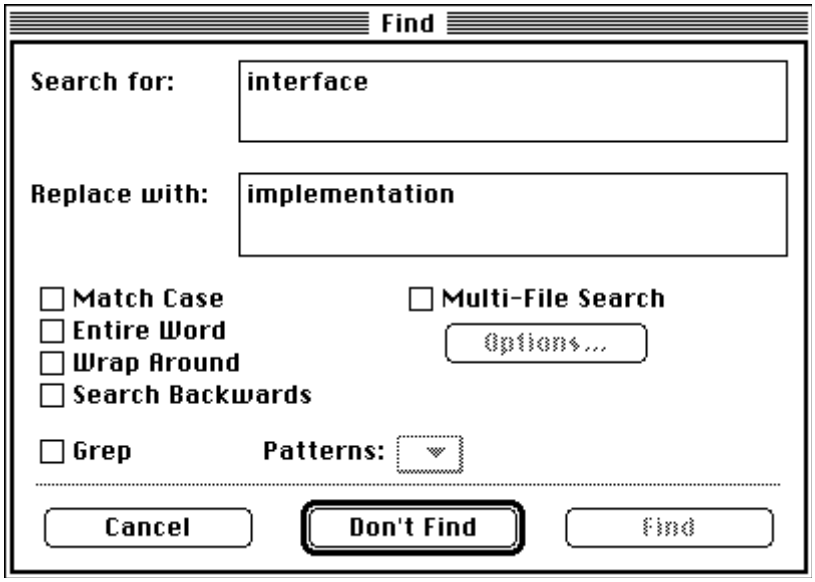
Introduction

This section covers BBEdit's searching capabilities, including:

- Searching for plain text
- Multi-File Searching
- Multi-File Replacing
- Regular-expression matching ("grep")

Searching for Plain Text

BBEdit's gives you the ability to search for strings of characters within the current document, or within multiple files, whether they're currently open in BBEdit or not. When you choose "Find…" from the **Search** menu, BBEdit will present this dialog:



The edit field to the right of "Search For:" contains the string of characters that you wish to search for. If the "Grep" check box is checked, the string in this edit field is a regular expression. See "Searching With Grep", below, for more details.

The edit field to the right of "Replace With:" contains the string of characters that will replace the current selection whenever you choose "Replace", "Replace and Find Again", or "Replace All" from the **Search** menu.

The "Match Case" check box determines whether the search is case-sensitive or not. If "Match Case" is checked,only text which has the same combination of upper and lower case letters as the Search For string will be found. For example:

Search String	Match Case On	Match Case Off

`interface`	`interface`	`interface`
		`Interface`
		`INTERFACE`
		`interFaCe`
		…

The "Entire Word" check box determines whether the text being searched must be bounded by word breaks.

Search String	Entire Word On	Entire Word Off

`Gestalt`	`Gestalt`	`NewGestalt`
		`Gestalt`
		`DeleteGestalt`
		`GestaltGoofBall`
		…

The "Wrap Around" check box will cause the entire document to be searched, regardless of where the current insertion point or selection range lies. Ordinarily, only the text from the start of the selection range to the end of the document is searched. If "Wrap Around" is turned on, and the search string isn't found between the start of the selection range and the end of the document, the search will automatically restart from the beginning of the document. If the search string is found in the document after wrapping around, BBEdit will blink the menu bar to alert you.

If the "Search Backwards" check box is checked, BBEdit will search backwards from the start of the insertion point to the start of the document, rather than forward to the end of the document. If "Wrap Around" is checked, and the search string is not found between the start of the document and the start of the selection range, the backwards search will resume from the end of the document.

After you have entered the search and replace strings and set the search options appropriately, you can click "Find", "Don't Find", or "Cancel". If you click "Find", BBEdit will immediately search for the current search and replace strings , using the current search options . If you click "Don't Find", BBEdit will accept the current search strings and options, but will not perform the search; you can then choose "Find Again" from the **Search** menu to start the search.

The following items on the **Search** menu are useful while searching within a document:

- **Find Again** repeats the search for the current search string, using the current settings. If you hold down the Shift key while choosing this command, the search direction will be reversed, so that you can search backwards even if you haven't turned on the "Search Backwards" switch in the Find… dialog.

- If a range of text is selected, **Find Selection** will make the selected text the search string, and then perform a "Find Again". If you hold down the Shift key while choosing this command, the search direction will be reversed.

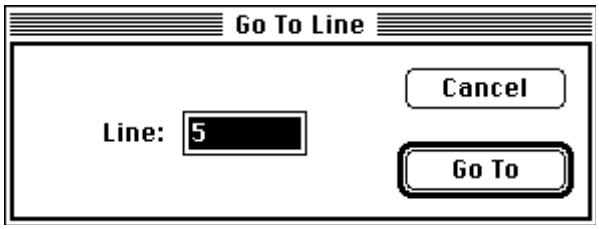
- **Enter Selection** makes the current selection the search string. You can then bring up the Find… dialog and change the search options or initiate a multi-file search.

- **Replace** replaces the current selection range with the replacement string (the string entered in the "Replace With:" edit field of the Find… dialog. If there is no selection range, the replacement string will be inserted at the insertion point.

- **Replace & Find Again** has the same effect as choosing Replace, followed by Find Again; it will replace the current selection range with the replacement string, and then search for the search string using the current search settings. If you hold down the Shift key while choosing this command, the search direction will be reversed.

- **Replace All** replaces all occurrences of the search string with the replacement string. If "Wrap Around" is turned on in the Find… dialog, the replacement will be done throughout the entire document; otherwise, only matching occurrences of the search string between the current selection range and the end of the document will be replaced.

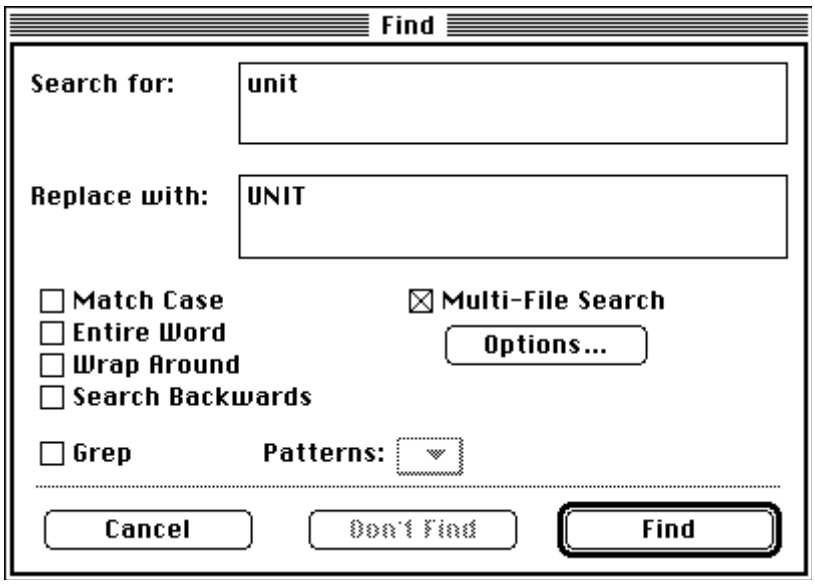
The "Go To Line…" command is useful for locating a line by number in the current document. When you choose "Go To Line…", the following dialog is presented:



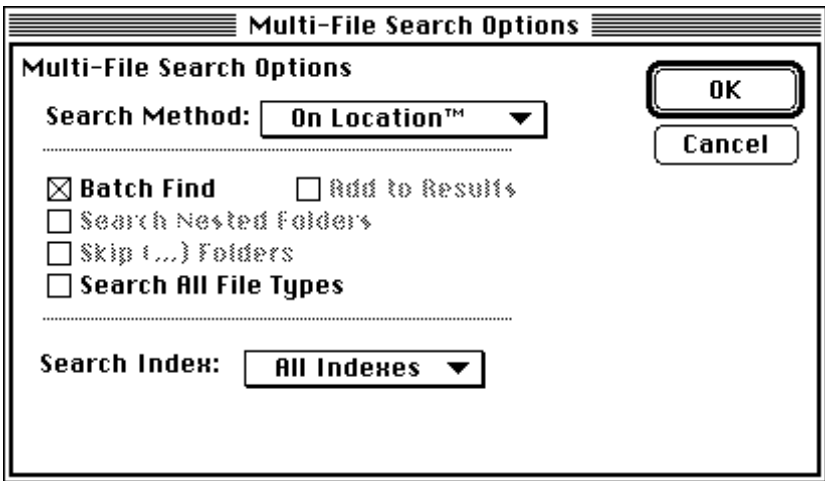
In the text field next to "Line:", enter the number of the line you wish to move to. Once the line number is entered, click "Go To", and the insertion point will be placed at the beginning of the desired line.

Multi-File Search

BBEdit provides a variety of means for searching through multiple text files at one time in order to locate the search string. To perform a multi-file search, check the "Multi-File Search" check box in the Find… dialog:



When you turn on "Multi-File Search", the "Options…" button is enabled; to set up the options for a multi-file search, click this button, and the following dialog will appear:



The popup menu next to "Search Method" determines how BBEdit will locate the files to be searched.

There are four ways to locate files:

- **On Location™.** If you have On Location 2.0 installed in your system, BBEdit will use On Location to search through all of the files that it knows about, and return to BBEdit all files which contain the search string. By default, BBEdit will search through all available On Location index files; you can select a single index to search from the "Search Index:" popup menu.

- **Directory Search.** When this search method is chosen, BBEdit scans through the folders starting at the one you choose, and each file that it encounters will be searched for the search string.

- **Open Windows.** When you choose this search method, BBEdit searches for the search string only in document windows that are currently open. This sort of search is very fast, and may be most convenient if you wish to limit the scope of your search to a few files.

- **Search Results.** This search method is only available when the "Search Results" window is open and contains the results of a previous Batch Find (see below).

The check boxes in the "Options…" dialog can be used to tailor the search to your needs:

- **Batch Find** accumulates the results of the search in progress and display them all at once in a Search Results window. If this check box is not checked, then the multi-file search will stop each time it encounters a match, and open the file that contains the match.

  Once the Search Results window is opened, you can double-click on entries in the window to display any given match, or select multiple matches from different files and display them all at once.

- **Add To Results** adds the results of the multi-file search to the existing Search Results window. This check box will be disabled if "Batch Find" is unchecked or if the Search Results window is not open.

- **Search Nested Folders** causes the Directory Scan search to search folders which are enclosed in the search's starting directory. If this check box is turned off, only the files in the starting directory will be searched.

- **Skip (…) Folders** causes the Directory Scan to skip folders whose names are enclosed in parentheses. This is useful if you have folders containing text files that you do not want to search for one reason or another; just enclose the folders' names in parentheses, and they will be skipped.

- **Search All File Types.** If this check box is checked, BBEdit searches files of all kinds, regardless of whether they contain actual text or not. If it's not checked, only text files will be searched.

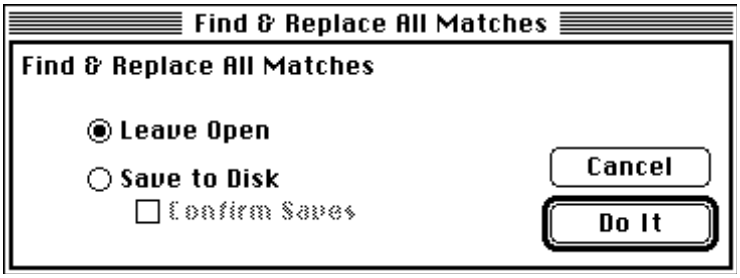
After you have set the options, click "OK" to save the settings and return to the "Find…" dialog. If you then click "Find", the multi-file search will start. If you click "Don't Find", the current settings will be saved, but the multi-file search won't start until you choose "Find in Next File" from the **Search** menu. (If Batch Find is selected, the "Don't Find" button is disabled.)

When BBEdit performs a multi-file search, it does so in two steps. First, it constructs a list of the files to be searched, using the search method specified in the Multi-File Search Options dialog. Second, it searches each file in the list for the search string. If "Batch Find" is selected, all occurrences in each file will be displayed in the Search Results window. Otherwise, each file will be opened to display the first occurrence of the search string; you can find subsequent occurrences of the search string in the same file by choosing "Find Again" from the **Search** menu. If you're not using Batch Find, you can locate the next file that contains the search string by choosing "Find In Next File" from the **Search** menu.

Easter Egg: If you hold down the Option key, "Find In Next File" becomes "Find All Matches". If you choose Find All Matches, BBEdit will perform the equivalent of Find In Next File until every file has been searched; each file that has an occurrence of the search string will be opened to show the first occurrence of the search string.

Multi-File Replacing

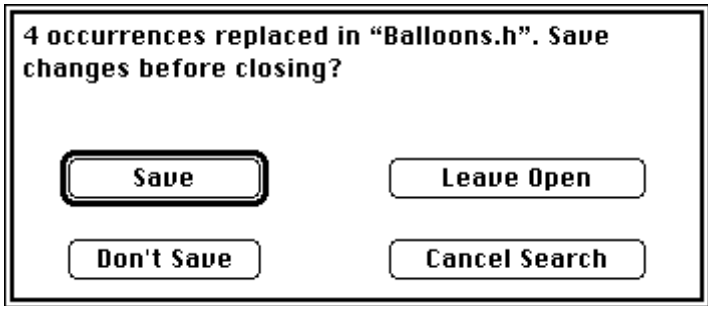
You can combine the capabilities of BBEdit's multi-file search with the Replace All command to perform multi-file replace operations. To do this, set up a multi-file search as desired, and un-check the "Batch Find" check box. When you return to the "Find…" dialog, click "Don't Find", and then choose "Find & Replace All Matches…" from the **Search** menu. You'll see the following dialog:



This dialog controls the behavior of a multi-file replace operation. There are three levels of safety that are available:

- Safest. Click on the "Leave Open" radio button. For each file that contains the search string, BBEdit will perform a "Replace All" on that file, and leave the file open so that you can inspect the changes.

- Less Safe. Click on the "Save To Disk" radio button, and make sure that the "Confirm Saves" check box is checked. BBEdit will perform a replace all on each file that contains the search string, and then ask you what to do:

  If you click "Save", BBEdit will save the changed file. If you click "Don't Save", BBEdit will throw away the changes that were just performed. If you click "Leave Open", BBEdit will leave the file open; this is the same behavior as the "Safest" case, above. If you click "Cancel Search", BBEdit will stop the multi-file replace operation.

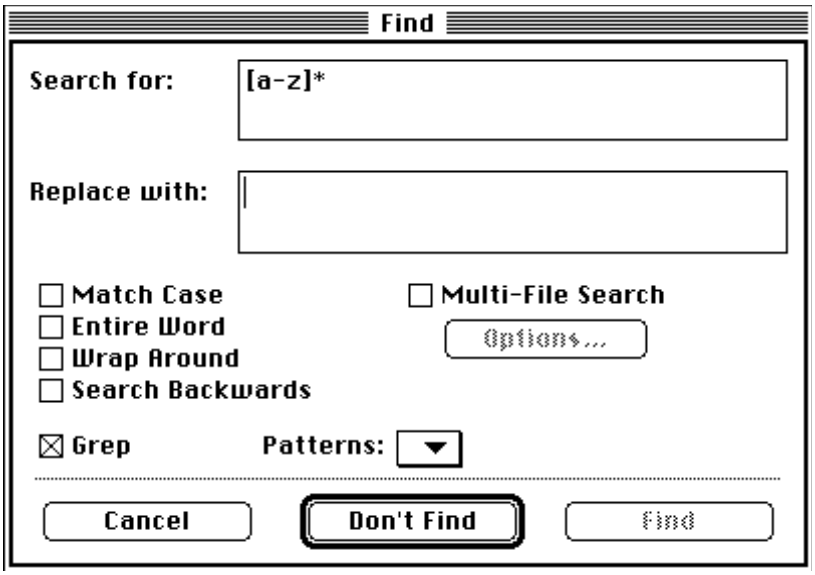
- Dangerous. Click on the "Save to Disk" radio button, and un-check the "Confirm Saves" check box. If you do this, BBEdit will perform a Replace All on each file that contains the search string, and then save the changed file to disk without asking. **You should only use these settings if you're absolutely certain of what you're doing, since the changes are irreversible.**

Regular-Expression Matching (Grep)

grep is a a method of pattern matching that derives from the Unix™ operating system. You are probably familiar with simple pattern matching from using word processors; when you ask a word processor to find all instances of the word "black", it is performing a simple pattern match, where each letter has to match literally. Matching strings in this manner is not very hard.

The ability to match strings in a more general manner is both more powerful and more complicated. It allows for sophisticated pattern matching operations, such as matching all words that begin with the letter "P" and end with the letters "er", or deleting the first word of every line. Grep provides a powerful means of doing this.

To use Grep for searching documents, just check the "Grep" check box in the Find… dialog:



The popup menu next to the "Patterns:" in the dialog contains a list of your most commonly-used Grep patterns. You can change this list in the "Grep Patterns" section of the Preferences… dialog.

How Grep Works

The "grep" mode of searching and replacing is a powerful tool. Although somewhat slower than normal text searching, grep allows the user to search for one of a set of many strings instead of a particular string. As a simple example, you can search for any occurrence of an identifier beginning with the letter `P`, or all lines that begin with a left brace.

A **pattern** is a string of characters that, in turn, describes a set of strings of characters. An example of a set of strings is the set of all strings that begin with the letter `P` and end with the letter `r`; the strings "Ptr" and "ProcPtr" are members of this set. We say that a string is **matched** by a pattern if it is a member of the set described by the pattern. Patterns are composed of sub-patterns which are patterns in themselves; this is how complicated patterns may be formed.

Some examples of grep patterns:

To replace a Pascal comment with a C comment, you would use

```
{\([^}]*\)}
```

to match the comment and

```
/*\1*/
```

to replace it.

To change all words that begin with the letter `P` to begin with the letter `Q`, you would use

```
\<P\([A-Za-z0-9]*\)\>
```

to match the word and

```
Q\1
```

to replace it.

To change a list of names; ie:

```
FrameRect
PaintRect
EmptyRect
```

to a list of names, followed by strings containing those names; i.e.

```
FrameRect,  "FrameRect",
PaintRect,  "PaintRect",
EmptyRect,  "EmptyRect",
```

you would use

```
\([A-Za-z][A-Za-z]*\)
```

to match the name and

```
\1,   "\1",
```

to replace it.

You don't have to understand how these work now; if you do, you may find that you don't have to read the rest of this chapter. The following section goes through the grep pattern matching and replacement rules step by step, so that by the end of it you should be able to understand how each of these grep patterns works and be able to make your own.

A note on notation: Writing about patterns and strings can be very confusing, since patterns and strings are made up of characters, as is this text. Therefore, we use certain typographical conventions to distinguish various usages.

All literal characters will be in the courier font; therefore, `a` and `xyz` refer to those literal strings of characters.

All patterns, when talked about in the abstract, will be italicized; therefore, *p* and *q* refer to abstract patterns.

All strings, when talked about in the abstract, will be Greek letters; therefore, ß and µ refer to abstract strings.

Sometimes we will be referring to parts of strings or patterns within longer ones. In these cases, the parts that are being referred to will be underlined. Therefore, in the string `xxaabx`, only the sub-string `aab` is actually being referred to; the other letters are used for context.

In the examples, a string that can occur anywhere in a line will be preceded and followed by an ellipsis (...) ; i.e. ... `xyz`.... If it can occur only at the beginning of the line, it will only be followed by an ellipsis; i.e., `xyz`.... Similarly, if it can occur only at the end of the line, it will be preceded but not followed by an ellipsis.

In some cases, the state of case sensitivity affects the results of a pattern match. In the examples we have noted when this is the case.

Pattern matching

Simple matching

1. Any character, with certain exceptions described below, is a pattern that matches itself.

Examples:

Pattern			Text	With case sensitivity
`X`	matches	...X...		
	doesn't match		...x...	on
	but matches		...x...	off

2. A pattern *x* followed by a pattern *y* forms a pattern *xy* that matches any string ßµ where ß can be matched by *x* and µ can be matched by *y*. We can, of course, take the compound pattern *xy* and concatenate yet another pattern *z* onto it, forming the pattern *xyz*.

Examples:

Pattern		Text	With case sensitivity
`XY`	matches	...XY...	
`Ptr`	matches	...Ptr...	
	doesn't match	...ptr...	on
	but does match	...ptr...	off

3. The character `.` is a pattern that will match any character.

Examples:

Pattern		Text
`P.r`	matches	...Ptr...
	and matches	...P.r...
`..`	matches	...ab...
	and matches	...a....

4. The character `\` followed by any character except `(`, `)`, `<`, `>`, or one of the digits `1-9` is a pattern that matches that character.

Examples:

Pattern		Text
`P\.r`	matches	...P.r...
	but doesn't match	...Ptr...
`P\\r`	matches	...P\r...

5. A string of characters *S* surrounded by square brackets (`[` and a `]`) forms a pattern `[`*s*`]` that matches a single instance of one of the characters in the string *S*. Note that the case sensitivity flag does not apply to characters between square brackets: letters must match exactly.

Examples:

Pattern		Text
`[abc]`	matches	...`ab`...
	and matches	...`xb`...
	but doesn't match	...`ab`...
`[abc][xyz]`	matches	...`ax`...
	but doesn't match	...`ab`...
`[abc]x`	matches	...`bx`...
	but doesn't match	...`Bx`...

5a. The pattern `[^ß]` matches any character that is not in the string ß. Special characters will be taken literally in this context. Again, case sensitivity doesn't apply to characters between square brackets.

Examples:

Pattern		Text
`[^abc]`	matches	...x...
	and matches	...A...
	but doesn't match	...a...
`[^abc]a`	matches	...xa...
	but doesn't match	...aa...
`[^.]a`	matches	...xa...
	but doesn't match	...`.`a...

5b. If a string of three characters in the form `[`*a*`-`*b*`]` occurs in in the pattern *p*, this represents all of the characters from *a* to *b* inclusive. All special characters are taken literally; i.e., `[!-.]` denotes the characters from `!` `to` `.`. Notice that the only way to include the character `]` in *p* is to make it the very first character. Likewise, the only way to include the character `-` in *p* is to have it either at the very beginning or the very end of *p*. Single characters and ranges may both be used between brackets.

Examples:

Pattern		Text
`[a-c]`	matches	...ac...
	and matches	...xc...
`[1x-z]a`	matches	...1a...
	and matches	...xa...
`[-x-z]a`	matches	...-a...
	and matches	…xa…

6. Any pattern *p* formed by any combination of rules 1 or 3-5b followed by a `*` forms the pattern *p*`*` that matches zero or more consecutive occurrences of characters matched by *p*.

Examples:

Pattern		Text	With case sensitivity
`[a-c]*`	matches	...`a`	
	and matches	...`acbca`	
	and matches	nothing	
`A[a-z]*`	matches	...`A`...	
	and matches	...`Abcb`...	
	and matches	...`abc`...	on
	but doesn't match	...`abc`...	off
`.*`	matches	anything from beginning of a line to the end of the line	
`[abc]*`	matches	...`b`	
	and matches	...`ab`	
	but doesn't match just (because it matches the longest string possible)	...`ab`	
`(.*)`	matches	...`(aaa)`...	
	and matches	...`()`...	

A closer example:

Let us examine more closely how the pattern `(.*)` matches text. This pattern will match any string that is enclosed in parentheses. This includes the string `()`, since the sub-pattern `.*` will match the empty string between the `(` and the `)`. But what about the string `(())`? Since the pattern `.*` will match any number of occurrences of all characters, won't it match the `(()` and cause the last `)` in the string to fail to match? Or conversely, won't the sub-pattern `(.*` match the whole string, leaving the `)` at the end of the pattern unmatched?

The answer to this is that any pattern of the form *p*`*` in a pattern *p*`*`*y* will match the largest number of occurrences of whatever *p* matches that still allows a match to *y*. Therefore, in matching `(())` against the pattern `(.*),` only the inner parentheses in the string `(())` will be matched by the sub-pattern `.*`.

Remembering sub-strings

We now have the ability to form patterns that are composed of sub-patterns, and will find it useful to "remember" sub-strings matched by sub-patterns and to be able to match against those substrings.

7. A pattern surrounded by `\(` and `\)` is a pattern that matches whatever the sub-pattern matches. This is useful for matching two or more instances of the same string and when doing replacements.

Example:

Pattern		Text
`\(abc\)`	matches	...`abc`
`\(ab(\)`	matches	...`ab(`

8. A `\` followed by *n*, where *n* is one of the digits `1`–`9`, is a pattern that matches whatever was matched by the sub-pattern beginning with the "nth" occurrence of `\(`. A pattern `\`*n* may be followed by an `*`, and forms a pattern `\`*n*`*` that matches zero or more occurrences of whatever `\`*n* matches.

Examples:

Pattern		Text
`\(abc\)\1`	matches	...`abcabc`...
`\(a.c\)\1`	matches	...`axcaxc`...
	but not	...`axcazc`...
	nor	...`axcaXc`...

Note that in this last pattern, the sub-pattern `\1` does not imply a re-application of the sub-pattern `a.c`, but what `a.c` matches. If `\(a.c\)` was matched with the string `axc`, then the sub-pattern `\1` would try to match the literal string `axc` against the remainder of the search string. Therefore, the pattern `\(a.c\)\1` will match `axcaxc`, but will not match `axcazc`.

Constraining matches

Sometimes it is useful to be able to "constrain" patterns to match only if certain conditions in the context outside the string matched are met.

9. A pattern surrounded by `\<` and `\>` is a pattern that matches whatever is matched by the sub-pattern, provided that the first and last characters of the matched string can be matched by `[A-Za-z0-9_]` and that the characters immediately surrounding the matched string cannot be matched by `[A-Za-z0-9_]` (i.e., can be matched by `[^A-Za-z0-9_]`).

This is used to match any string that matches the sub-pattern only if the matched string begins and ends on a "word" boundary (a "word" being a C identifier).

Examples:

Pattern		Text
`\<ab*\>`	matches	...+ab+...
	but doesn't match	...+ab+...
	and doesn't match	...+abc+

10. A pattern *p* that is preceded by a `^` forms a pattern `^`*p*. If the pattern `^`*p* is not preceded by any other pattern, it matches whatever *p* matches as long as the first character matched by *p* occurs at the beginning of a line. If the pattern `^`*p* is preceded by another pattern, then the `^` is taken literally.

Examples:

Pattern		Text
`^ab*`	matches	`ab`...
	but doesn't match	`xab`...
`ab^ab*`	matches	`ab^ab`...

11. A pattern *p* that is followed by a `$` forms a pattern *p*`$`. If the pattern *p*`$` is not followed by any other pattern, it matches whatever *p* matches as long as the last character matched by *p* occurs at the end of a line. If the pattern *p*`$` is followed by another pattern, then the `$` is taken literally.

Examples:

Pattern		Text
`ab$`	matches	...`ab`
	but doesn't match	...`abx`
`ab$ab`	matches	...`ab$ab`...
`^ab$`	matches	`ab`
	but doesn't match	`ab`...

Note that the characters ^ and $ constrain pattern matches to begin or end at line boundaries, and so can be combined to constrain a pattern to match an entire line only (as in the above example).

We mentioned at the beginning the ability to search for any identifier beginning with the letter P. This would be accomplished with the pattern \<[Pp][A-Za-z0-9_]*\>. Note that, if you have case sensitivity is off, then the patterns \<P[A-Za-z0-9_]*\> and \<p[A-Za-z0-9_]*\> would match the same strings. Also, if word-match is on, then any of these patterns with the \< and \> removed will match the same strings.

Replacement

Grep provides not only a more sophisticated method of searching, but a sophisticated method of replacing as well. In a replacement string, the following substitutions are made before any text replacement occurs:

1. Each occurrence of the character & is replaced with whatever was last matched by the pattern.

Examples:

"Find" string	"Replace" string	Original text	Result
abc	+&	...abc...	...+abc...
abc	&&	...abc...	...abcabc...

2. Each occurrence of a string of the form *n*, where *n* is one of the digits 1-9, is replaced by whatever was last matched by the sub-pattern beginning with the *n*th occurrence of \(.

Examples:

"Find" string	"Replace" string	Original text	Result
\(a*\)\(b*\)	\1\2	aabb...	aabb...
\(a*\)\(b*\)	\2\1	aabb...	bbaa...

3. Each occurrence of a string of the form *p*, where *p* is other than one of the digits 1-9, is replaced by *p*.

Examples:

"Find" string	"Replace" string	Original text	Result
\(a*\)\(b*\)	\1&\2\	aabb...	aa&bb...
\(a*\)\(b*\)	\\\2\1\\	aabb...	\bbaa\...

This allows you to not only be able to search for a string satisfying a complex set of conditions, but also to be able to do a subsequent replacement that varies depending on the string that is matched.

Some Examples

• Suppose that you have written a program that is to become a Macintosh application (i.e., it uses the Macintosh ToolBox instead of stdio for the user interface). Suppose also that you have discovered that you have forgotten to put a `\p` at the beginning of your string constants, so that your program is trying to pass C strings instead of Pascal strings to the ToolBox (which only knows how to deal with Pascal strings). You can easily change all your C strings to Pascal strings by specifying `"\(.*\)"` as the search pattern and `"\\p\1"` as the replacement string.

• Suppose you decided to reverse the two arguments of the function "foo". You might try the pattern `foo(\([^,]*\),\([^)]*\))` as the search pattern and `foo(\2, \1)` as the replacement pattern. How does the search pattern work?

Let's assume we're trying to match some text that looks like `foo(1,*bar)`

• `foo(\([^,]*\),\([^)]*\))` matches `foo(1,*bar)`
• `foo(\([^,]*\),\([^)]*\))` matches `foo(1,*bar)`
• `foo(\([^,]*\),\([^)]*\))` matches `foo(1,*bar)`
• `foo(\([^,]*\),\([^)]*\))` matches `foo(1,*bar)`
• `foo(\([^,]*\),\([^)]*\))` matches `foo(1,*bar)`

Since `\([^,]*\)` matched `1` and `\([^)]*\)` matched `*bar`, the two arguments to foo, the replacement pattern `foo(\2, \1)` will result in `foo(*bar, 1)`

This, unfortunately, won't work in the case of `foo(1,(*bar)+2)`, since `\([^)]*\)` will match only up to the first right parenthesis, leaving `+2)` unmatched. If we're sure that all calls to foo end with a semi-colon, however, we can change our pattern to `foo(\([^,]*\),\([^;]*\));` . In this pattern, instead of trying to match the second argument by matching everything up to the first right parenthesis, we match everything up to the `);` which terminates the invocation of foo.

In this example we showed how to analyze a grep pattern by examining sub-patterns. This is a good way of figuring out how to build a pattern as well. grep can be thought of as a small and rather cryptic programming language, with each pattern a program and sub-pattern a statement in this language. If you try to create a grep pattern by testing a small sub-pattern, then adding and testing additional sub-patterns until the complete pattern is built, you may find building complex grep patterns not nearly as daunting as you first thought.