INTRODUCTION TO GREP

grep is a a method of pattern matching that derives from the Unix™ 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.

But 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.

HOW GREP WORKS

The "grep" mode of searching and replacing is a powerful tool. At the expense of being somewhat slower than normal text searching, using 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; in fact, it would be surprising if you do. 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.

PATTERN MATCHING AND REPLACEMENT RULES

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 this 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 …`a`b…	
	and matches	…x`b`…
	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 …a…c…	
	and matches	…x…c…
`[1x-z]a`	matches …1a…	
	and matches	…x…a…
`[-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.