Usage notes for Windows Browser 1.4

Hippocrates Sendoukas
University of Southern California
Dept. of Finance & Bus. Economics
January 15, 1993

Introduction

This document summarizes the use of WBR which is a browser for text files under MS-Windows. It requires Windows 3.0 or later and works only on standard or enhanced modes. All program functions can be accessed through standard menus or accelerator keys that are shown after each menu entry. The program operation is quite simple; it reads a text file into memory and diplays it on the screen. You can move around the file by using the scrollbars or the arrow keys. You can also use regular expressions for convenient searching. Finally, you can use any installed font to display your files. All options as well as window position and size are recorded in the "win.ini" file for your convenience.

If you specify a filename upon invokation, the browser will open and display that file; in this way, you can create associations using the File Manager or by editing "win.ini". You can also read a file by selecting the "Open" entry in the "File" menu. The browser remembers the last four files that you opened and appends their names to this menu; in this way, you can open any of these files just by clicking on their name. There are cases where you specify a filename upon invokation, but you do not want the browser to remember that filename. In that case, you can use the "-l" switch before the filename; this technique is used by the DVI driver to display the temporary log files. When using the "-l" switch, the browser checks for previous invokations of itself and terminates them. In this way, you do not need to manually terminate each instance of the browser.

Searching for text

The browser supports searches using "regular expressions". This is a very powerful method that lets you perform "approximate" searches. Regular expressions are patterns that can match more than one string. They are composed of normal and special characters. In their simplest form they contain only normal characters and can match only a single string: for example the expression "abc" will match the substring whether it occurs by itself or within a word. It will match the strings "abc", "abcd", "0abcd"; it will not match the string "Abc" or "aBCd". Such simple patterns are useful, but there are many cases where one needs something more powerful; suppose for example that we want to find the keyword "if" in a Pascal program: Pascal is not case sensitive, so we have to check all possible spelling combinations; furthermore, we do not want the pattern to occur within other words. Regular expressions provide many such capabilities which are controlled by special characters embedded within the pattern. These characters and their functions are:

Character	**Function**
.	Matches any character
[...]	Matches a character from those inside the bracket
*	Matches zero or more instances of the preceding character
+	Matches one or more instances of the preceding character
?	Matches zero or one instances of the preceding character
{n1,n2}	Matches n1 to n2 instances of the preceding character
^	Matches the beginning of a line

$	Matches the end of a line
<...>	Matches an entire word
(...)	Brackets a regular expression

The first group of special characters involves the specification of single characters. A dot (.) denotes any character; for example, the pattern "i." matches the strings "if", "in" or "i7". A set of brackets enclosing some characters denotes a character class and the expression matches any single character from the character class; the pattern "[Ii]f" matches the strings "if" and "If". You can also abbreviate the character class by using a dash: the pattern "[a-z]" will match any lowercase letter. You can also specify a negative character class, where the expression will match any character except those in the character class; this is accomplished by specifying a caret sign (^) immediately after the left bracket; the pattern "[^a-z]" will match any character except a lowercase letter.

There are also some special characters dealing with the number of instances of a character (this is called closure). An asterisk denotes zero or more occurrences of the preceding character; the pattern "go*d" matches the strings "gd", "god", "good", "goood", etc. A plus sign denotes one or more instances of the preceding character; the pattern "go+d" matches the strings "god", "good", "goood", but not the string "gd". A question mark matches zero or one instances of the preceding character; therefore, the pattern "go?d" will match only the strings "gd" and "god". The special characters {n1,n2} match n1 to n2 instances of the preceding character; the pattern "go{2,3}d" matches the strings "good" and "goood", but not the strings "god" or "gooood". There are also two variants of the last special characters. The special characters "{n1}" match exactly n1 instances of the preceding character, while the special characters "{n1,}" match at least n1 instances of the preceding character.

The third group of special characters deals with the location of the string. If the first character of the regular expression is a caret (^), the expression will be matched only at the beginning of a line. For example, the pattern "^abc" matches the string "abc" only if it occurs at the beginning of a line. Similarly the dollar sign ($) matches the end of a line; the pattern "abc$" matches the string "abc" only if it occurs at the end of a line. The pattern "^$" will match all empty lines. Note that these two characters are treated as special characters only if they occur in the beginning or the end of the regular expression. That is, the pattern "a$b^c" will match the string "a$b^c" regardless of its position on a line. Another set of special characters are "<" and ">". These facilitate the matching of entire words, ignoring any matching substrings embedded in other strings. For example, the pattern "<abc>" will match the string "abc" by itself, but not the string "abcd".

Sometimes we need to specify parts of the regular expression. For this reason, we use the special characters "(" and ")" to bracket a part of the expression. The matching behavior is not affected, but we can refer to these parts of the expression by the notation "\N" where N is a digit between 1 and 9. Suppose that we want to find all sequences of two identical characters; this would appear quite difficult since we do not know these characters in advance. Regular expression bracketing solves this problem quite elegantly: the pattern "(.)\1" will find the desired characters.

Since the characters . [ ] * + ? ^ $ ( ) { } have a special meaning in the context of regular expressions, we need another special notation when we want to search for them literally: in order to suppress the special meaning of any character ("quote" it), we can precede it by a backslash. Therefore if we want to find an asterisk, the search string will be "*"; if we want to find a left parenthesis, the search string will be "\(". To find a backslash by itself, the search string will be "\\". For the user's convenience, the program also accepts some standard escape sequences. The sequence "\n" indicates the newline character (^J), "\r" indicates the carriage return character (^M), "\t" indicates the tab character (^I), "\b" indicates the backspace character (^H), "\a" indicates the bell character (^G), "\f" indicates the formfeed character (^L), and "\xNN" indicates the character with code NN where NN are two hexadecimal digits. These escape sequences can be used anywhere in the search patterns.

Another fundamental rule is that regular expressions try to match as many characters as possible. While this is usually desirable, there are cases where it can be surprising. Suppose for example that the search pattern is "\(.*\)" (the parentheses are quoted to indicate that we want to match them literally), and our text is "(one) and (two)". In this case, the pattern will match the entire line, since it will find the last right parenthesis at the end. If we wanted only the first pair of parentheses, we should specify "\([^)]*\)" as our search pattern. In this way, it will match the substring "(one)". The right parenthesis inside the square brackets does not need to be quoted: almost all characters in a character class are taken literally.

Another point that we should keep in mind is that the matching of closures can be sometimes surprising. Suppose for example that the search string is "s*". Since the search string specifies zero or more occurrences of the letter s, it will match anything, including the empty string. The problem is that the search string consists only of a closure, and therefore it matches anything. A simple solution to this problem, is to avoid using such search strings.

For a more thorough discussion of regular expressions, one can read any book describing the operation of the Unix "ed", "sed", "grep", "awk" or "lex" commands. In general, regular expressions are quite powerful means of manipulating text. They have however several limitations: for example, the construction of some compound patterns can be complicated. Furthermore, they are line oriented: that is, they cannot match expressions spanning more than one lines.

Regular Expression Errors

It should be obvious by now that not all regular expressions are valid. The program checks each expression and complains if it is illegal. The possible error messages are:

"**Invalid number**": This means that the program did not find an expected number. This can happen when you do not specify a number incide a pair of braces.

"**Invalid subexpression number**": This means that you specified an undefined subexpression. The expression "(.)\1" is valid because "\1" corresponds to the subexpression "(.)". However, the expression "(.)\2" is invalid because you did not specify two subexpressions (using parentheses).

"**Unbalanced parentheses**": This indicates that the left and right parentheses for bracketing subexpressions are unbalanced.

"**Too many (**": This means that you tried to bracket more than 9 subexpressions.

"**More than 2 numbers in { }**": You can specify one or two numbers within a pair of braces. This message indicates a violation of this rule.

"**} expected**": This message arises from an ill-formed pair of braces. The only valid tokens inside the braces are numbers or a comma after the first number.

"**First number exceeds second in { }**": Two numbers inside a pair of braces indicate the minimum and maximum number of times that the previous character should be matched. Obviously, the minimum cannot be more than the maximum.

"**Unbalanced square brackets**": This message indicates that the left and right brackets for a character class do not match.

"**Expression too complicated**": The program checks a regular expression for errors and "compiles" it to an intermediate form for quick searching. This error can arise if the intermediate form is too large, which typically happens if you specify too many character classes (above 15).

"**Invalid hexadecimal number**": As mentioned previously, the program lets you specify any 8-bit character by using a hexadecimal notation "\xNN" where NN is a valid hexadecimal number. All such numbers should consist of two hexadecimal digits exactly: that is, use \x08 instead of \x8. This error occurs if any of the two characters following \x is not a valid hexadecimal character.

"**Unknown error**": This is a "catch all" that should never occur. If you see it, you can be sure that there is a bug in the program and I would appreciate if you let me know about it, so I can fix it.

Selecting a font

The program lets you use any installed font in your system; it can deal with variable pitched, bitmap, vector or TrueType fonts. When you select the "Font" entry in the main menu, you are presented with a dialog box containing all the installed fonts at all sizes and styles. Your selection is automatically recorded in "win.ini", so you do not need to set it more than once.

Packing List

Make sure that you have all the relevant files:

Filename	**Description**
wbr.exe	Text file browser
wbr.hlp	Help file
wbr.wri	Printable documentation
miscwin.dll	Utility routines
ctl3d.dll	More utility routines
commdlg.dll	Common dialog routines (required only for Windows 3.0)

The only required files for the browser are "wbr.exe", "miscwin.dll" and "ctl3d.dll". The file "ctl3d.dll" must reside on you windows system directory. All other files except for the documentation must reside either in a directory specified by the PATH environment variable, or the base directory of Windows.

Caveats

The program reads a file into memory and just displays it; it will complain if there is not enough memory to load the file (which is unlikely since Windows has virtual memory). In general, it can handle quite large files (unlike Notepad) without excessive demands on memory. It has proven quite useful to me, and I hope that it serves you equally well. I have tested and debugged the program extensively, but I cannot afford to make any guarantees; anybody who uses it assumes all risks. On the other hand, if you find any bug or have any suggestion for improvements, I will be more than happy to hear about it and I will do my best to fix it. You can contact me via e-mail at "sendouk@scf.usc.edu", or regular mail at "3230 Overland Ave. #201, Los Angeles, CA 90034".

Licensing Agreement