DATABASE

Pattern Matching: the Gestalt Approach

By John W. Ratclif, July 01, 1988

Source Code Accompanies This Article. Download It Now.

- ratcliff.lst

String comparison routines are often limited to finding exact matches. John describes an algorithm (implemented in assembly language) that gives matches as percentages.

PATTERN MATCHING: THE GESTALT APPROACH

John W. Ratcliff, David E. Metzener

John Ratcliff has developed a number of educational packages at Milliken Publishing, most notably the Word Math series, and is currently doing cardiovascular research at St. Louis University and is developing a computer game for Electronic Arts.

David Metzener is currently a researcher in the cardiovascular division at the St. Louis University He has been writing educational software applications since 1982.

What is the gestalt approach to pattern matching? Gestalt is a word that describes how people can recognize a pattern as a functional unit that has properties not derivable by summation of its parts. For example, a person can recognize a picture in a connect-the-dots puzzle before finishing or even beginning it. This process of filling in the missing parts by comparing what is known to previous observations is called gestalt.

The Ratcliff/Obershelp pattern-matching algorithm uses this same process to decide how similar two one-dimensional patterns are. Since text strings are one dimensional, this algorithm returns a value that you can use as a confidence factor, or percentage, showing how alike any two strings are.

Because this pattern-matching algorithm can recognize matches in substrings quickly and easily, there are many applications for it. For example, a compiler using this algorithm would be able to determine what variable, keyword, or procedure name the programmer meant, even when the compiler encounters a spelling error. Educational software that can recognize a correct answer contextually (even when the answer contains a typing error) is another natural application. A command shell could finally recognize that SYMPONY doesn't exist---and do something intelligent with that information, such as pop up a menu of close alternatives like SYMPHONY. Text adventure games with their powerful parsers are an ideal application for this algorithm: the games could make broad assumptions in assimilating user input.

The Ratcliff/Obershelp pattern-matching algorithm was developed by John W. Ratcliff and John A. Obershelp in 1983 to address concerns about educational software. Often, educational software has consisted of multiple-choice questions only because the existing algorithms required an exact character-for-character match. The algorithm presented in this article is both forgiving and understanding of simple typing mistakes, and allows intelligent responses to erroneous input. To date, this algorithm has been implemented in a commercial spelling checker, a database search program, and a compiler.

Adding this algorithm to a compiler had some dramatic results. When this algorithm was implemented in a primitive C compiler, the compiler was able to make accurate assumptions when it encountered misspelled procedure names, keywords, and variables. When it couldn't find an identifier, it examined all of the currently defined names and collated the best matches. If the compiler could find no match better than 60 percent, then it produced a normal error message. The most common case, however, resulted in an accurate and unambiguous match: the compiler was able to continue with this assumption while producing both a warning message that indicated the assumption made and the line number that it occurred on. The result was that rather than getting a cascade of 50 warning messages because of one typing mistake the programmer now got a simple warning message and a successful compilation. After finding several strong matches, the compiler prompted the programmer for confirmation in apop-up window. The compiler could even go so far as to ask the programmer if it should automatically correct the source code as well. On the occasions when the compiler made a false assumption, it almost always generated errors due to mismatched arguments being passed to an assumed procedure. Even if an erroneous assumption results in a successful compilation, the programmer is still warned and knows not to run the executable that the compiler produced.

How the Algorithm Works

The best way to describe the Ratcliff/Obershelp pattern-matching algorithm, in using conventional computer terminology, is as a wild-card search that doesn't require wild cards. Instead, the algorithm creates its own wildcards, based on the closest matches found between the strings. Specifically, the algorithm works by examining two strings passed to it and locating the largest group of characters in common. The algorithm uses this group of characters as an anchor between the two strings. The algorithm then places any group of characters found to the left or the right of this anchor on a stack for further examination. This procedure is repeated for all substrings on the stack until there is nothing left to examine. The algorithm calculates the score returned as twice the number of characters found in common divided by the total number of characters in the two strings; the score is returned as an integer, reflecting a percentage match.

For example, suppose you want to compare the similarity between the word `Pennsylvania' and a mangled spelling as `Pencilvaneya.' The largest common group of characters that the algorithm would find is `lvan.' The two sub-groups remaining to the left are `Pennsy' and `Penci,' and to the right are `ia' and`eya.' The algorithm places both of these string sections on the stack to be examined, and advances the current score to eight, two times the number of characters found in common. The substrings `ia' and `eya' are next to come off of the stack and are then examined. The algorithm finds one character in common: a. The score is advanced to ten. The substrings to the left---'i' and `ey'---are placed on the stack, but then are immediately removed and determined to contain no character in common. Next, the algorithm pulls `Pennsy' and `Penci' off of the stack. The largest common substring found is `Pen.' The algorithm advances the score by 6 so that it is now 16.

There is nothing to the left of `Pen,' but to the right are the substrings `nsy' and `ci,' which are pushed onto the stack. When the algorithm pulls off `nsy' and `ci' next, it finds no characters in common. The stack is now empty and the algorith ready to return the similarity value found. There was a score of 16 out of a total of 24. This result means that the two strings were 67 percent alike.

Inside the Code

Now that you know how the algorithm works, you're ready to look at the code. This article includes an assembly language routine that is accessible as a function call for C programs. This assembly language routine has been optimized using techniques such as register optimization, algorithmic analysis, branch optimization, and instruction-cycle counts. Therefore, you may very well find this routine fast enough to be used as a basic string-companion function in your software. In that regard you should note that the variables in this routine are declared as static, rather than dynamic, to make the source code easier to follow.Example 1.

Example 1: An example C program calling the gestalt functions

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/***********************************************************************/
/*                          GESTALT.C                                 */
/*          written by John W. Ratcliff and David E. Metzener          */
/*                        November 10, 1987                           */
/*                                                                     */
/*  Demonstrates the Ratcliff/Obershelp Pattern Recognition Algorithm  */
/*  Link this with SIMIL.OBJ created from the SIMIL.ASM source file    */
/*  The actual similiarity function is called as:                      */
/*  int simil(char *str1,char *str2)                                   */
/*  where str1 and str2 are the two strings you wish to know their     */
/*  similarly value. simil returns a percentage match between         */
/*  0 and 100 percent.                                                 */
/***********************************************************************/
int      simil(char *stl1, char *str2);
void     ucase(char *str);

main ()
{
    char str1[80];
    char str2[80];
    int  prcnt;

printf("This program demonstrates the Ratcliff/Obershelp pattern\n");
```

```
printf("recognition algorithm. Enter series of word pairs to\n");
printf("discover their similarity values.\n");
printf("Enter strings of 'END' and 'END' to exit.\n\n);
do
     {
           printf("Enter the two strings separated by a space: ");
           scanf("%s %s" str1, str2);
           ucase(str1);
           ucase(str2);
           prcnt = simil(str1,str2);
           printf("%s and %s are %d\% alike.\n\n",str1,str2,prcnt);
     }     while (strcmp(str1,"END"));
}

void ucase(str)
char *str;
{
while (*str)
     {
      *str-toupper(*str);
      str++;
     }
}
```

It should be clear from the earlier discussion that the time-critical portion of the code is in the section that determines the maximum number of characters in common between two substrings. The worstcase scenario is when absolutely no characters are found in common between the two strings. When this happens, N x M number of comparisons are required, where N is the number of characters in the first string and M is the number of characters in the second string.

The comparison procedure is composed of two loops: an inner loop for string two and an outer loop for string one. At each character in the two strings the procedure checks to see how many characters are equal. Whenever any characters are found that are equal, the procedure then checks to see if this number is greater than the previous maximum number of characters found. If it is, then the procedure updates the variable maxchars and updates the substring to be returned. Whenever a new maxchars occurs you can shorten the search by the difference between the new maxchars and add that value. The reason for this is simply that once you have found for example, a five-character match, you do not need to waste time looking more than five characters from the ends of the two substrings because there is clearly no chance of finding more than five characters. Inside the inner loop, whenever the procedure finds any characters in common, whether they form a new maxchars or not, the procedure advances the inner loop past these characters. On exit from this procedure, the DX register contains the number of characters found in common, and the variables CLI, CRI, CL2, and CR2 are pointing to the left and the right of the character string found in common between the two source strings.

The main procedure, SIMIL, which calls the compare routine, "realizes" there is nothing to place on the stack if no characters were found in common. If there are no characters to the left of either of the two substrings, then you don't need to push anything to the left. If there is exactly one character to the left of both substrings, you don't need to push the characters on the stack because they cannot be equal (or the first character would have been included in the maxchars substring). These same rules apply to the right of the substring as well.

Performance Aspects

To evaluate the performance of the routines, the following tests were performed. First, a series of strings were created from 1 to 20 characters in length. Then 10,000 calls were made to the pattern-matching procedure for each of these 20 strings on an 8 MHz IBM AT. The time for each iteration was recorded in hundredths of seconds. Strings were created that were exactly equal, totally different, matching halfway at the beginning, matching halfway at the end, or matching hallway in the middle. The results of these tests are reported in Figure 1, page 50, as the number of comparisons performed per second. As you can see in the figure for exactly equal strings, the procedure found virtually no change as the strings became longer. For this case, the pattern-matching procedure acted as an ordinary string comparison function and performed approximately 8,000 comparisons per second. Totally different strings act as predicted, showing a quadratic curve in the form of N2 (from 8,000 comparisons/second for one character to 200 comparisons/second for two 20-character strings).

Strings that match at the beginning can exit quickly and those that match in the middle divide their search problem in half. (455 comparisons/second for 20-character strings.) Strings that match at the end model those that are totally unalike since nearly the entire strings are searched before the matching substring is located. (270 comparisons/second.)

Next, two 12-character strings were analyzed for every percentage that the procedure could return. At each percentage, a wide variety of combinations of substring matches were tried. These varieties included matches at the beginning, in the middle, and at the end, and those spread differently throughout the two strings. Figure 2, page 51, displays the average time for these tests at each percentage.

Figure 1: Effects of timing during different compare types

Final Thoughts

Implementing this algorithm in your application can dramatically improve your software, but it does require some judgment based on the environment. The first step in interpreting the ambiguous data should be by compiling a list of the most likely alternatives. Your program's action after this should be based on both how strong and how closely grouped the candidates are. For example, if the best match found is only a 50-percent match but all the other candidates are under 20-percent, the 50-percent match is quite likely the users' original intent. However, if there are six 90-percent matches or better, it would be best to provide the user with these matches in order of similarity, along with a convenient and rapid method of confirmation, such as a pop-up menu.

We hope this article has sparked some interest in the programming community, and we'd appreciate hearing about your applications of the algorithm. Adding pattern recognition to software has tremendous potential for improving all our lived programmers and users alike. We might finally make "user-friendly" something more than a marketing cliche.

Figure 2: Average timing for every percentage of a 12- to 12-character string match

```
Pattern Matching by Gestalt
by John W. Ratcliff

Listng One: Gestalt. article July 1988 issue

        TITLE   SIMIL.ASM written by John W. Ratcliff and David E. Metzener

; November 10, 1987
; Uses the Ratcliff/Obershelp pattern recognition algorithm.
; This program provides a new function to C on an 8086 based machine.
; The function SIMIL returns a percentage value corresponding to how
; alike any two strings are.  Be certain to upper case to two strings
; passed if you are not concerned about case sensitivity.
; NOTE:!!! This routine is for SMALL model only.  As an exerciese for
; the student, feel free to convert it to LARGE.
```

```
_TEXT   SEGMENT  BYTE PUBLIC 'CODE'
_TEXT   ENDS
CONST   SEGMENT  WORD PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT  WORD PUBLIC 'BSS'
_BSS    ENDS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
_DATA   ENDS
DGROUP  GROUP   CONST,  _BSS,   _DATA
        ASSUME  CS: _TEXT, DS: DGROUP, SS: DGROUP, ES: DGROUP



_DATA   SEGMENT

ststr1l dw   25 dup(?)              ; contains lefts for string 1
ststr1r dw   25 dup(?)              ; contains rights for string 1
ststr2l dw   25 dup(?)              ; contains lefts for string 2
ststr2r dw   25 dup(?)              ; contains rights for string 2
stcknum dw   ?                      ; number of elements on the stack
score   dw   ?                      ; the #of chars in common times 2
total   dw   ?                      ; total #of chars in string 1 and 2
cl1     dw   ?                      ; left of string 1 found in common
cr1     dw   ?                      ; right of string 1 found in common
cl2     dw   ?                      ; left of string 2 found in common
cr2     dw   ?                      ; right of string 2 found in common
s2ed    dw   ?                      ; the end of string 2 used in compare

_DATA   ENDS

        public _simil


_TEXT   SEGMENT

_simil  proc   near
; This routine expects pointers passed to two character strings, null
; terminated, that you wish compared.  It returns a percentage value
; from 0 to 100% corresponding to how alike the two strings are.
;                   +4        +6
; usage: simil(char *str1,char *str2)
; The similiarity routine is composed of three major components
; pushst    ---- pushes a strings section to be compared on the stack
; popst     ---- pops a string section to be examined off of the stack
; compare   ---- finds the largest group of characters in common between
;                any two string sections
; The similiarity routine begins by computing the total length of both
; strings passed and placing that value in TOTAL.  It then takes
; the beginning and ending of both strings passed and pushes them on
; the stack.  It then falls into the main line code.
; The original two strings are immediately popped off of the stack and
; are passed to the compare routine.  The compare routine will find the
; largest group of characters in common between the two strings.
```

```
; The number of characters in common is multiplied times two and added
; to the total score.  If there were no characters in common then there
; is nothing to push onto the stack.  If there are exactly one character
; to the left in both strings then we needn't push it on the stack.
; (We allready know they aren't equal from the previous call to compare.)
; Otherwise the characters to the left are pushed onto the stack.  These
; same rules apply to characters to the right of the substring found in
; common.  This process of pulling substrings off of the stack, comparing
; them, and pushing remaining sections on the stack is continued until
; the stack is empty.  On return the total score is divided by the
; number of characters in both strings.  This is mulitplied time 100 to
; yeild a percentage.  This percentage similiarity is returned to the
; calling procedure.

        push    bp                  ;save BP reg.
        mov     bp,sp               ;save SP reg in BP for use in program
        push    es                  ;save the ES segment register
        mov     ax,ds               ;copy DS segement register to ES
        mov     es,ax
        xor     ax,ax               ;zero out AX for clearing of SCORE var.
        mov     score,ax            ;zero out SCORE
        mov     stcknum,ax          ;initalize number of stack entries to 0
        mov     si,[bp+4]           ;move beginning pointer of string 1 to SI
        mov     di,[bp+6]           ;move beginning pointer of string 2 to DI
        cmp     [si],al             ;is it a null string?
        je      strerr              ;can't process null strings.
        cmp     [di],al             ;is it a null string?
        jne     docmp               ;neither is a null string so process them
strerr: jmp     donit               ;exit routine
docmp:  push    di                  ;save DI because of SCAS opcode
        push    si                  ;save SI because of SCAS opcode
        xor     al,al               ;clear out AL to search for end of string
        cld                         ;set direction flag to forward
        mov     cx,-1               ;make sure we repeat the correct # of times
        repnz   scasb               ;scan for string delimiter in string 2
        dec     di                  ;point DI to '$00' byte of string 2
        dec     di                  ;point DI to last character of string 2
        mov     bp,di               ;move DI to BP where it is supposed to be
        pop     di                  ;restore SI into DI for SCAS (string 1)
        repnz   scasb               ;scan for string delimiter in string 1
        not     cx                  ;do one's compliment for correct length of
        sub     cx,2                ;subtract the two zero bytes at the end of
        mov     total,cx            ;store string 2's length
        dec     di                  ;point DI to '$00' byte of string 1
        dec     di                  ;point DI to last character of string 1
        mov     bx,di               ;move DI to BX where it is supposed to be
        pop     di                  ;restore DI to what it should be
        call    pushst              ;Push values for the first call to SIMILIAR
main:   cmp     stcknum,0           ;is there anything on the stack?
        je      done                ;No, then all done!
        call    popst               ;get regs. set up for a COMPARE call
        call    compare             ;do compare for this substring set
        cmp     dx,0                ;if nothing in common then nothing to push
        je      main                ;try another set
```

```
        shl     dx,1                ;*2 for add to score
        add     score,dx            ;add into score
        mov     bp,stcknum          ;get number of entry I want to look at
        shl     bp,1                ;get AX ready to access string stacks
        mov     si,[ststr1l+bp]     ;move L1 into SI or L1
        mov     bx,cl1              ;move CL1 into BX or R1
        mov     di,[ststr2l+bp]     ;move L2 into DI or L2
        mov     cx,cl2              ;move CL2 into CX t emporarily
        mov     ax,[ststr1r+bp]     ;get old R1 off of stack
        mov     cl1,ax              ;place in CL1 temporarily
        mov     ax,[ststr2r+bp]     ;get old R2 off of stack
        mov     cl2,ax              ; save in CL2 temporarily
        mov     bp,cx               ;place CL2 into BP
        cmp     bx,si               ;compare CL1 to L1
        je      chrght              ;if zero, then nothing on left side string
        cmp     bp,di               ;compare CL2 to L2
        je      chrght              ;if zero, then nothing on left side string
        dec     bx                  ;point to last part of left side string 1
        dec     bp                  ;point to last part of left side string 2
        cmp     bx,si               ;only one character to examine?
        jne     pushit              ;no->we need to examine this
        cmp     bp,di               ;only one character in both?
        je      chrght              ;nothing to look at if both only one char
pushit: call    pushst              ;push left side on stack
chrght: mov     si,cr1              ;move CR1 into SI or L1
        mov     bx,cl1              ;move R1 into BX or R1
        mov     di,cr2              ;move CR2 into DI or L2
        mov     bp,cl2              ;move R2 into BP or R2
        cmp     si,bx               ;compare CR1 to R1
        je      main                ;if zero, then nothing on right side string
        cmp     di,bp               ;compare CR2 to R2
        je      main                ;if zero, then nothing on right side string
        inc     si                  ;point to last part of right side string 1
        inc     di                  ;point to last part of right side string 2
        cmp     bx,si               ;only one character to examine?
        jne     push2               ;no->examine it
        cmp     bp,di               ;only one character to examine in both?
        je      main                ;yes->get next string off of stack
push2:  call    pushst              ;push right side on stack
        jmp short main              ;do next level of compares
done:   mov     ax,score            ;get score into AX for MUL
        mov     cx,100              ;get 100 into CX for MUL
        mul     cx                  ;Multiply by 100
        mov     cx,total            ;get total characters for divide
        div     cx                  ;Divide by total
donit:  pop     es                  ;Restore ES segement register to entry valu
        pop     bp                  ;Restore BP back to entry value
        ret                         ;Leave with AX holding % similarity
_simil  endp

compare proc   near
; The compare routine locates the largest group of characters between string
;    and string 2.  This routine assumes that the direction flag is clear.
; Pass to this routine:
```

```
;   BX    = R1 (right side of string 1)
;   DS:SI = L1 (left side of string 1)
;   ES:DI = L2 (left side of string 2)
;   BP    = R2 (right side of string 2)
;
; This routine returns:
;   DX    = # of characters matching
;  CL1    = Left side of first string that matches
;  CL2    = Left side of second string that matches
;  CR1    = Right side of first string that matches
;  CR2    = Right side of second string that matches
; The compare routine is composed of two loops.  An inner and an outer loop.
; The worst case scenario is that ther are absolutely no characters in
; common between string 1 and string 2.  In this case N x M compares are
; performed.  However, whan an equal condition occurs in the inner
; loop, then the next character to be examinded in string 2 (for this loop)
; is advanced by the number of characters found equal.  Whenever a new
; maximum number of characters in common is found then the ending location
; of both the inner and outer loop is backed off by the difference between
; the new max chars value and the old max chars value for both loops.  In
; short if 5 characters have been found in common part of the way through
; the search then we can cut our search short 5 characters before the
; true end of both strings since there is no chance of finding better than
; a 5 character match at that point.  This technique means that an exact
; equal match will require only a single compare and combinations of other
; matches will proceed as efficiently as possible.

        mov    s2ed,bp              ;store end of string 2
        xor    dx,dx                ;Init MAXCHARS
forl3:  push   di                   ;Save start of string 2
forl4:  push   di                   ;Save start of string 2
        push   si                   ;Save start of string 1
        mov    cx,s2ed              ;Set up for calc of length of string 1
        sub    cx,di                ;get length of string 1 -1
        inc    cx                   ;make proper length
        push   cx                   ;Save starting length of string 1
        repz   cmpsb                ;compare strings
        jz     equal                ;if equal, then skip fixes
        inc    cx                   ;inc back because CMPS decs even if not equ
equal:  pop    ax                   ;get starting length of string1
        sub    ax,cx                ;get lenght of common characters
        jnz    newmax               ;more than 0 chars matched
        pop    si                   ;get back start of string 1
        pop    di                   ;get back start of string 2
reent:  inc    di                   ;Do the next character no matter what
reent2: cmp    di,bp                ;Are we done with string 2?
        jle    forl4                ;No, then do next string compare
        pop    di                   ;get back start of string 2
        inc    si                   ;next char in string 1 to scan
        cmp    si,bx                ;Are we done with string 1?
        jle    forl3                ;No, then do next string compare
        ret                         ;MAXCHARS is in DX register
; We branch downwards for both newmax and newmx2 because on the
; 8086.. line of processors a branch not taken is faster than
```

```
; one which is.  Therefore since the not equal condition is to be
; found most often and we would like the inner loop to execute as quickly
; as possible we branch outside of this loop on the  less frequent
; occurance.  When a match and or a new maxchars is found we branch down to
; these two routines, process the new conditions and then branch back up
; to the main line code.
newmax: cmp    ax,dx                  ;greater than MAXCHARS?
        jg     newmx2                 ;yes, update new maxchars and pointers
        pop    si                     ;get back start of string 1
        pop    di                     ;get back start of string 2
        add    di,ax                  ;Skip past matching chars
        jmp short reent2              ;re-enter inner loop
newmx2: pop    si                     ;get back start of string 1
        pop    di                     ;get back start of string 2
        mov    cl1,si                 ;put begin of match of string 1
        mov    cl2,di                 ;put begin of match of string 2
        mov    cx,ax                  ;save new maxchars
        sub    ax,dx                  ;get delta for adjustment to ends of string
        sub    bx,ax                  ;adjust end of string 1
        sub    bp,ax                  ;adjust end of string 2
        mov    dx,cx                  ;new maxchars
        dec    cx                     ;set up for advance to last matching char
        add    di,cx                  ;advance to last matching char string 2
        mov    cr2,di                 ;put end of match of string 2
        add    cx,si                  ;advance to last matching char string 1
        mov    cr1,cx                 ;put end of match of string 1
        jmp short reent               ;re-enter inner loop
compare endp

pushst  proc near
; On entry:
;    BX    = R1 (right side of string 1)
;    DS:SI = L1 (left side of string 1)
;    ES:DI = L2 (left side of string 2)
;    BP    = R2 (right side of string 2)

        mov    cx,bp                  ;save R2
        mov    bp,stcknum
        shl    bp,1                   ;*2 for words
        mov    [bp+ststr1l],si        ;put left side of string 1 on stack
        mov    [bp+ststr1r],bx        ;put right side of string 1 on stack
        mov    [bp+ststr2l],di        ;put left side of string 2 on stack
        mov    [bp+ststr2r],cx        ;put right side of string 2 on stack
        inc    stcknum                ;Add one to number of stack entries
        mov    bp,cx                  ;Restore R2
        ret
pushst  endp


popst   proc near
;    BX    = R1 (right side of string 1)
;    DS:SI = L1 (left side of string 1)
;    ES:DI = L2 (left side of string 2)
;    BP    = R2 (right side of string 2)
```

```
        dec     stcknum             ;point to last entry in stack
        mov     bp,stcknum          ;get number of stack entries
        shl     bp,1                ;*2 for words
        mov     si,[bp+ststr1l]     ;restore left side of string 1 from stack
        mov     bx,[bp+ststr1r]     ;restore right side of string 1 from stack
        mov     di,[bp+ststr2l]     ;restore left side of string 2 from stack
        mov     bp,[bp+ststr2r]     ;restore right side of string 2 from stack
        ret
popst   endp


_TEXT   ENDS

        END
```

Example 1: Gestalt article, July 1988 issue

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*****************************************************************************/
/*                            GESTALT.C                                      */
/*        written by John W. Ratcliff and David E. Metzener                  */
/*                         November 10, 1987                                 */
/*                                                                           */
/* Demonstrates the Ratcliff/Obershelp Pattern Recognition Algorithm         */
/* Link this with SIMIL.OBJ created from the SIMIL.ASM source file           */
/* The actual similiarity function is called as:                             */
/* int simil(char *str1,char *str2)                                          */
/* where str1 and str2 are the two strings you wish to know their            */
/* similiarity value.  simil returns a percentage match between              */
/* 0 and 100 percent.                                                        */
/*****************************************************************************/
int     simil(char *str1,char *str2);
void    ucase(char *str);

main()
{
  char  str1[80];
  char  str2[80];
  int   prcnt;

printf("This program demonstrates the Ratcliff/Obershelp pattern\n");
printf("recognition algorithm.  Enter series of word pairs to\n");
printf("discover their similarity values.\n");
printf("Enter strings of 'END' and 'END' to exit.\n\n");
do
  {
    printf("Enter the two strings seperated by a space: ");
    scanf("%s %s",str1,str2);
```

```
    ucase(str1);
    ucase(str2);
    prcnt = simil(str1,str2);
    printf("%s and %s are %d\% alike.\n\n",str1,str2,prcnt);
  } while (strcmp(str1,"END"));
}

void ucase(str)
char *str;
{
while (*str)
  {
   *str=toupper(*str);
   str++;
  }
}
```

Previous 1 2 3 4 **5**

Related Reading

- News
- Commentary

- **SmartBear Goes Even More Mobile**
- **Developer Internet of Things App Starter Kit**
- **Tuning Up Developer/Database Dashboards**
- **Red Hat and the CentOS Project Join Forces**
  More News»

- Slideshow
- Video

- **Jolt Awards: The Best Testing Tools**
- **Developer Reading List**
- **Developer's Reading List**
- **Developer's Reading List**
  More Slideshows»

- Most Popular

- **2013 Developer Salary Survey**
- **Graphics Programming Black Book**
- **Even Simple Floating-Point Output Is**