Appendix E - Implementation of conceptualizer

```
/*******************************   conceptualization.c  ********************************
 *
 *  Purpose: Program to walk through a domain and identify concepts at both
 *           document and element level according to heuristical rules
 *  Date:   June 30th-July 2nd, 2002
 *  Input:  List of files in the domain
 *  Output:  Prolog files on the form:
 *               node(software, file3.html, [agents, agent, user, ]).
 *               node(agents, file3.html#sub0, [software, ]).
 *               node(interface, file3.html#sub1, [user, intelligent, agents, ]).
 *               etc...
 *           Report files with all information (one file for each document)
 *  Uses:    ruleadt.h- implementation of rules
 *  Notes:   Temporary files are used due to easy manipulation with stringlists
 *           The functions ReportDocCandidates, ReportElmCandidates, InitPrologFile
 *           and ReportPrologFile, are used in main.
*****************************************************************************************/

#include <stdio.h>
#include <string.h>
#include "ruleadt.h"


/*****************************************************************************************
 *  Name:    ReportDocCandidates
 *  Purpose: Update reportfile with all candidates and values in a document
*****************************************************************************************/
ReportDocCandidates(CVR *record, char filename[NAME_SIZE],
                    char current[NAME_SIZE], int boundary){
  char tuple[NAME_SIZE];
  CVR *temp;
  FILE *reportfile;
  int i = 0;

  reportfile = fopen(filename, "a+");
  fprintf(reportfile, "\t\t\t Document candidates of %s \n", current);
  strcpy(tuple, "");
  temp=record;  //don't want to miss the array for good

  while (i++ < boundary){
    sprintf(tuple, "%s:  %d\n", temp->candidate, temp->value);
    fputs(tuple, reportfile);
    temp++;
  }//print
  fputs("**********************************\n\n\n", reportfile);
  fclose(reportfile);
}//ReportDocCandidates


/*****************************************************************************************
 *  Name:    ReportElmCandidates
 *  Purpose: Update reportfile with all candidates and values in an element
*****************************************************************************************/
ReportElmCandidates(CVR *record, char filename[NAME_SIZE],
                    char id[NAME_SIZE], char tag[TAG_SIZE], int boundary){
  char tuple[NAME_SIZE];
  CVR *temp;
  FILE *reportfile;
  int i = 0;

  reportfile = fopen(filename, "a+");
  fprintf(reportfile, "\t\t\t Element candidates of %s of elementtype: %s \n", id, tag);
  strcpy(tuple, "");
  temp=record;  //don't want to miss the array for good!

  while (i++ < boundary){
    sprintf(tuple, "%s:  %d\n", temp->candidate, temp->value);
    fputs(tuple, reportfile);
    temp++;
  }//print
  fputs("-----------------\n\n\n", reportfile);
  fclose(reportfile);
}//ReportElmCandidates


/*****************************************************************************************
 *  Name:    InitPrologfiles
 *  Purpose: Writes the values as a comment to the prolog-file.
```

```
*****************************************************************************************/
InitPrologfiles(char f[NAME_SIZE], char f2[NAME_SIZE],
                int H1, int H3, int H4, int H5a, int H5b, int H6, int H8){
  FILE *fp, *fp2;

  fp = fopen(f, "w");
  fprintf(fp, "%% ------------------------------------------------------------------\n");
  fprintf(fp, "%% The points associated with the Heuristics: \n");
  fprintf(fp, "%% \t HC-1  (DSL)         = %d \t\t HC-2  (term frequency)   = TF \n", H1);
  fprintf(fp, "%% \t HC-3  (emphasizers) = %d \t\t HC-4  (meta information) = %d \n", H3, H4);
  fprintf(fp, "%% \t HC-5a (headings)    = %d \t\t HC-5b (punish links)     = %d \n", H5a, H5b);
  fprintf(fp, "%% \t HC-6  (linked to)   = %d \t\t HC-8  (punish doc_con)   = %d \n", H6, H8);
  fprintf(fp, "%% Attributes: (concept, element type, location, candidate list)\n");
  fprintf(fp, "%% ------------------------------------------------------------------\n\n");
  fclose(fp);

  fp2 = fopen(f2, "w");
  fprintf(fp2, "%% ------------------------------------------------------------------\n");
  fprintf(fp2, "%% All hyper references in the domain \n");
  fprintf(fp2, "%% Attributes: (from, to, tag)\n");
  fprintf(fp2, "%% ------------------------------------------------------------------\n\n");
  fclose(fp2);
} //InitPrologfiles


/*****************************************************************************************
 *  Name:    ReportPrologFile
 *  Purpose: Writes concept to file in prolog-format. Should have the form:
 *           node(concept, element-type, loc, [cand1, cand2, ...], filename).
 *           in order to represent the node, the location and other candidates
 *  Note:    a bit tricky to achieve the correct layout form
 *  Returns: True if this element was successfully written to the file
*****************************************************************************************/
int ReportPrologFile(char fprolog[NAME_SIZE], char fprolog_non[NAME_SIZE],
                     char tagName[TAG_SIZE], char location[NAME_SIZE],
                     CVR *array, int boundary, char tempfile[NAME_SIZE],
                     char combfile[NAME_SIZE], char thisfilename[NAME_SIZE]){
  FILE *fp;
  char high[NAME_SIZE];
  char lowerstring[NAME_SIZE];
  char candidates[NAME_SIZE * 3];
  char noncandidates[NAME_SIZE *6];
  char concept[NAME_SIZE];
  int p=0; //argument of SelectConcept
  int i=0;
  int listsize=7;//number of candidates produced is listsize-1

  strcpy(high, ""); strcpy(candidates, ""); strcpy(lowerstring, ""); strcpy(noncandidates, "");

  if (boundary<=1) return;

    /* Else, go get the concept unless the combination tag+concept found before */
  do {
    SelectConcept(array, i++, concept, &p);  //highest value = concept
  }
  while ( IsCombination(concept, tagName, combfile) && (i<listsize)  );

    /* Next, get first candidate and add to string */
  do {
    SelectConcept(array, i++, high, &p); //get candidate no 1
  }
  while (0 == strcmp(high,"") && i<listsize);
  if (p<=0)
    return FALSE;//don't print if value is less than zero
  strcat(candidates, high);

    /* Get other candidates with high scores */
  while( i<listsize && boundary > 0){
    SelectConcept(array, i++, high,  &p);  //pick next candidate
    if (0 == strcmp(high,""))
      continue;//don't consider empty elements
    strcat(candidates, ", ");
    strcat(candidates, high);
  }//find additional candidates

    /* Write information to files */
  fp = fopen(fprolog, "a+");
  fprintf(fp, "knowledge_source(%s, %s, \"%s\", [%s], \"%s\").\n",
                               concept, tagName, location, candidates, thisfilename);
  fclose(fp);

  fp = fopen(tempfile, "w");//conceptfile is used later
  fputs(concept, fp); fputs("\n\n", fp);  //very impt with two \n due to stringlist requirement
  fclose(fp);
  return TRUE;
}//ReportPrologFile
```

```
/*****************************************************************************************
 *   Name: Main
 *  Purpose: Walk through the domain and get concepts, candidates and write
 *  information to files
 *  Plan:
 *    1. Initialise
 *      a:  Prepare files to be used
 *      b:  Read the weights of the rules (extend to "from file")
 *      c:  Get all links
 *    2. For each document, do (according to the Heuristics for finding concepts)
 *      H2: Get term frequency (uses stemming)
 *      H1: Check membership in DSL (implemented as opposite of a stoplist)
 *      H3: Identify emphasizers
 *      H4: Check for meta information and title
 *      H5a:Top level headings are important
 *      H5b:Remove href candidates from consideration
 *      H6: Check whether this document is linked to
 *      H7: If the same concept and type occur for two elements, then choose another concept
 *      (H9:Decide level of abstraction / analyse headings used - not implemented
 *      Write concept with highest value and ID to prolog file
 *   3. For each element within a document, do (if level of abstraction allows it to)
 *      H1: Check membership in DSL (anti-stoplist)
 *      H2: Get term frequency (may introduce stemming here)
 *      H3: Identify emphasizers
 *      H8: Remove occurences of the document concept selected
 *      (H10:Lists, tables and images (concept in image is "alt" or filename) - not implemented
 *      Write information on concepts, ID, tag and upper candidate list to prolog file
 *      Write additional information to report file
 *    Repeat loop
 *
 *  Notes: Heuristic H7 is implemented as a rule inside function ReportPrologFile
*****************************************************************************/

int main( void ) {

/* declaring filepointers */
  FILE *prologfile,  *domaindocs, *tempfp, *docfile;

/* declaring variables to hold filenames */
  char current[NAME_SIZE];            //the name of the current document
  char f1[NAME_SIZE], f2[NAME_SIZE], f3[NAME_SIZE], f4[NAME_SIZE],
       f5[NAME_SIZE], f6[NAME_SIZE], f7[NAME_SIZE];     //temporary results
  char dsl[NAME_SIZE];                //domain specific list
  char cf[NAME_SIZE];                 //file with concepts selected
  char freport[NAME_SIZE];            //reporting all info found
  char ft[NAME_SIZE];                 //temporary file of no use at all
  char fprolog[NAME_SIZE];            //holds concepts found in prolog format
  char fprolog_non[NAME_SIZE];        //holds lowerList candidates in prolog format
  char fprologdomain[NAME_SIZE];      //holds all filenames of the domain in prolog format
  char fdomain[NAME_SIZE];            //all filenames in the domain
  char fweight[NAME_SIZE];            //weights associated with the rules
  char fdomainhrefs[NAME_SIZE];       //all hyper references in a domain
  char fprologhrefs[NAME_SIZE];       //all href information in prolog format
  char combinations[NAME_SIZE];       //for the combination concept+tag
  char newfilename[NAME_SIZE];        //new filename that element is written to

/* declaring variables to hold strings */
  char elementid[NAME_SIZE];          //ID for an element
  char tagName[TAG_SIZE];
  char document[LINE_SIZE], element[LINE_SIZE]; //document and element in long lines

/* declaring variables to hold integers */
  int doc_used, elm_used, id_nr;   //global counters used in many routines
  int H1, H3, H4, H5a, H5b, H6, H8;//points associated with Heuristical rules
  int cvalue=0;                       //not used, but necessary as argument of SelectConcept

/* declaring other structures */
  CVR doc_conceptvalues[MAX_CANDIDATES];  //an array of records for concepts and values
  CVR elm_conceptvalues[MAX_CANDIDATES];  //the element array for concepts and values
  CVR *cand_pt, *elm_cand_pt;




/* Part 1: Initialising */
  cand_pt = &doc_conceptvalues[0];   //now cand_pt points to the first array-element
  elm_cand_pt = &elm_conceptvalues[0];  //same for the element pointer
  doc_used = elm_used = id_nr = 0;      //number of elements in array and elementcounter

/* Part 1a. Prepare files for use and init filenames */
  strcpy(dsl, "domain_specific_list.txt");
  strcpy(cf, "res/concepts_file.txt");          strcpy(ft, "res/t");
  strcpy(f1, "res/temp_bold.txt");              strcpy(f2, "res/temp_italic.txt");
  strcpy(f3, "res/temp_ahref.txt");             strcpy(f4, "res/temp_meta.txt");
  strcpy(f5, "res/temp_title.txt");             strcpy(f6, "res/temp_header.txt");
  strcpy(f7, "res/temp_linkedto.txt");          strcpy(combinations, "res/combinations.txt");
  strcpy(fdomain, "domaindocs.txt");            strcpy(fprolog, "res/prolog.pro");
```

```
  strcpy(fweight, "res/weights.txt");           strcpy(fdomainhrefs, "res/domainhrefs.txt");
  strcpy(fprologhrefs, "res/prologhrefs.pro");  strcpy(fprologdomain, "res/prologdomain.pro");

  tempfp = fopen(combinations, "w");
  fclose(tempfp);  //init this file to empty

  prologfile = fopen(fprolog, "a+");  //open prolog file

  (void) Stemmer(dsl); //all intermediate results are stemmed, so must the DSL be

/* Part 1b. Assign weights with the different Heuristics and initialise the Prologfiles */
  H1 = 100;  H3 = 10;  H4 = 30;  H5a = 20; H5b = -200; H6 = 200;  H8 = -400;
  InitPrologfiles(fprolog, fprologhrefs, H1, H3, H4, H5a, H5b, H6, H8);

/* Part 1c. Write all links in domain to file and confirm on screen */
  AllHrefsInDomain(fdomain, fdomainhrefs, fprologhrefs);
  printf("Hyper-refs written to the file domainhrefs.txt\n------\n");

/* Part 1d. Write all filenames from the domain to prolog understandable form */
  DomainToProlog(fdomain, fprologdomain);



/* Part 2: Collect the document concepts, walk through the domain */
  domaindocs = fopen(fdomain, "r");
  while ( fgets(current, NAME_SIZE, domaindocs) != NULL){
    strip_slash_n(current);  //must remove \n from filename.html\n
    FileToStr(current, document); //read document into one single string
    WriteInfoToFiles(document, f1, f2, f3, f4, f5, f6);

    InitArray(document, doc_conceptvalues, MAX_CANDIDATES, &doc_used);  // H2 - term frequency

    AssignPoints(doc_conceptvalues, dsl, H1,  doc_used);                // H1  - DSL
    AssignPoints(doc_conceptvalues, f1,  H3,  doc_used);                // H3  - bold
    AssignPoints(doc_conceptvalues, f2,  H3,  doc_used);                // H3  - italic
    AssignPoints(doc_conceptvalues, f3,  H5b, doc_used);                // H5b - punish links
    AssignPoints(doc_conceptvalues, f4,  H4,  doc_used);                // H4  - meta
    AssignPoints(doc_conceptvalues, f5,  H4,  doc_used);                // H4  - title
    AssignPoints(doc_conceptvalues, f6,  H5a, doc_used);                // H5a - Headings level 1
    LinkedTo(current, f7, fdomainhrefs);                                // is this linked to?
    AssignPoints(doc_conceptvalues, f7,  H6,  doc_used);                // H6  - linked to

    BubbleSort(doc_conceptvalues, doc_used);//conceptvalues == &conceptvalues[0]
    sprintf(freport, "res/report_%s.txt", current);                  // "report_file1.html"
    ReportDocCandidates(doc_conceptvalues, freport, current, doc_used);
    (void) ReportPrologFile(fprolog, fprolog_non, "html", current,
                            doc_conceptvalues, doc_used, cf, combinations, current);



/* Part 3: Walk through each section, search for element concepts, still in outer loop */
    docfile = fopen(current, "r");
    id_nr = 0;
    //start an inner loop
    while (GetNextElement(docfile, element, tagName) > 0 ) {  //tag found
       InitArray(element, elm_conceptvalues, MAX_CANDIDATES, &elm_used); // H2 - term frequency
       WriteInfoToFiles(element, f1, f2, f3, f4, f5, f6); // not all info will be used...
       AssignPoints(elm_conceptvalues, dsl, H1, elm_used);//H1  - DSL
       AssignPoints(elm_conceptvalues, f1,  H3, elm_used); //H3  - bold
       AssignPoints(elm_conceptvalues, f2,  H3, elm_used);//H3  - italic
       AssignPoints(elm_conceptvalues, cf,  H8, elm_used);//H8  - remove doc_concept

       BubbleSort(elm_conceptvalues, elm_used);
       Id(current, &id_nr, elementid);
       ReportElmCandidates(elm_conceptvalues, freport, elementid, tagName, elm_used);
       if (ReportPrologFile(fprolog, fprolog_non, tagName, elementid,
                            elm_conceptvalues, elm_used, ft, combinations, current) ) {
           strcpy(newfilename, "");
           sprintf(newfilename, "kwbase/%s", elementid);  //make string "kwbase/elementid"
           tempfp = fopen(newfilename, "w");
           fputs(element, tempfp);
           fclose(tempfp);
       }//if, store in knowledge base

    }//end inner while for the elements
    fclose(docfile);

    printf("The result is written to file %s\n", freport);
  } //while get domaindocs

  fclose(domaindocs);  //close files
  fclose(prologfile);

}//end of main program.
```

```
/*******************************   parseradt.h   *********************************

    Purpose: ADT for extracting elements and information from an HTML-document

*****************************************************************************/

/***************************   Public Constants   ***************************/
#include "constants.h"  //has all public constants
#include "ruleadt.h"

/***************************   Public Routines   ****************************/
externint  AllImages(char input[], char output[], int *counter);
extern  int  AntiStoplist(char inputfile[], char output[], char antiFilename[], int *counter);
extern  void FileToStr(char filename[NAME_SIZE], char output[]);
extern  int  GetFirstTag(char input[], char output[], char tag[]);
extern  int  GetNextElement(FILE *docfile, char output[], char tag[]);
extern  int Id(char docname[ID_SIZE], int *counter, char tagid[NAME_SIZE]);
extern  int  LexicalAnalysis(char input[], char output[], char FileOut[NAME_SIZE], int *counter);
extern int  SubElementTerms(char tag[TAG_SIZE], char input[], char output[], char tempfile[NAME_SIZE],
int *size);
extern  int  TagsFromFile(char filename[], char tag[TAG_SIZE], char pro[LINE_SIZE], char
ut2[LINE_SIZE]);
extern  int  TagsFromString(char tag[TAG_SIZE], char input[], char output[]);
extern  int  WordCount(char filename[], int *size);
extern  int  Stemmer(char filename[NAME_SIZE]);



/*******************************   ruleadt.h   *********************************/

/***************************   Public Constants   ***************************/
#include "constants.h" //has all public constants
#include "strlist.h"

/*****************************   Public Types   *****************************/

struct _HeuristicRecord {
    char heuristicId[HEUR_SIZE];/* Heuristic identifier */
    int  points;/* Points associated with the Heuristic */
            }  ;
typedef struct _HeuristicRecord HR;

struct _ConceptValueRecord {
    char candidate[NAME_SIZE];
    int  value;
    };
typedef struct _ConceptValueRecord CVR;

struct _ElementRecord {
    char id[ID_SIZE];/* ID for which this tag is retagged */
    char tagName[TAG_SIZE];/* the name of the element */
    char conceptName[NAME_SIZE];/* the concept name of the element */
    char content[LINE_SIZE];/* the original content of the element */
    char href[LINE_SIZE]; /* does this element contain a href? */
    char tempfile[NAME_SIZE];/* reference to a temporary file with words */
    int  stopcount;/* number of remaining words after lexical analysis */
    char stopwords[LINE_SIZE];/* all the remaining words after lexical analysis */
    int  subcount;/* number of words in identified sub-element */
    char subwords[LINE_SIZE];/* words in sub-element */
    int  anticount; /* number of words after anti-stoplist */
    char antiwords[LINE_SIZE];/* words after anti-stoplist */
    char maxwords[LINE_SIZE];/* words with more than two occurrences */
    int  maxcount;/* number of maximum occurrences of a word in the string */
            }  ;
typedef struct _ElementRecord ER;

/***************************   Public Routines   ****************************/
extern int  OlderParser(char docname[NAME_SIZE]);
extern void AllHrefsInDomain(char domainfilename[NAME_SIZE],
                             char linkfile[NAME_SIZE], char prologfilename[NAME_SIZE]);
extern void strip_slash_n(char temp[]);
extern void InitArray(char current[NAME_SIZE], CVR *conceptvalues, int arraysize, int *used);
extern int  MaxWords(char tempfile[], char output[], int *maximum ,
                  CVR *termfrequency, int *used);
extern void AssignPoints(CVR *candidatevalues, char incomingfile[], int points, int boundary);
extern void BubbleSort(CVR record[], int max);
extern void WriteInfoToFiles(char element[LINE_SIZE], char file1[NAME_SIZE],
                             char file2[NAME_SIZE], char file3[NAME_SIZE],
                             char file4[NAME_SIZE], char file5[NAME_SIZE],
                             char file6[NAME_SIZE] );
extern void SelectConcept(CVR *doc_conceptvalues, int location, char concept[], int *pt);
extern void LinkedTo(char current[NAME_SIZE], char fout[NAME_SIZE], char hrefs[NAME_SIZE]);
extern void DomainToProlog(char domainfile[NAME_SIZE], char output[NAME_SIZE]);
```

```
/*******************************   stop.h   *********************************
    Purpose: Stop list DFA generator and driver module header.
    Notes:   This module implements a fast finite state machine generator,
             and a driver, for implementing stop list filters.
*****************************************************************************/

#ifndef STOP_H
#define STOP_H

#include "strlist.h"              /* this code relies on the StrList package */

/*****************************   Public Types   ******************************/

typedef struct _DfaStruct *DFA;      /* Deterministic Finite Automaton object */

/*****************************   Public Routines   ***************************/

#ifdef __STDC__

extern DFA   BuildDFA( StrList words );
extern char *GetTerm( FILE *stream, DFA machine, int size, char *output );

#else

extern DFA   BuildDFA();
extern char *GetTerm();

#endif
#endif


/*******************************   strlist.h   *********************************

    Purpose: Simple string list abstract data type module header

    Notes:   This module implements a straightforward string ordered list
             abstract data type.  It is optimized for appending and deleting
             from the end of the list.  Since they are ordered lists, string
             lists may be sorted, and their members are addressed by ordinal
             position (starting from 0).
*****************************************************************************/

#ifndef STRLIST_H
#define STRLIST_H

/*****************************   Public Constants   **************************/

#define NULL_INDEX                    -1        /* invalid string index */

/*****************************   Public Types   ******************************/

typedef struct _StrListStruct *StrList;          /* the base string list type */

/*****************************   Public Routines   ***************************/

#ifdef __STDC__

extern void    StrListAppend( StrList list, char *string );
extern void    StrListAppendFile( StrList list, char *filename );
extern StrList StrListCreate( void );
extern void    StrListDestroy( StrList list );
extern int     StrListEqual( StrList list1, StrList list2 );
extern int     StrListElementEqual( int pos, StrList list1, StrList list2 );
extern int     StrListCount( int pos, StrList list1, StrList list2 );
extern char *  StrListPeek( StrList list, int index );
extern int     StrListSize( StrList list );
extern void    StrListSort( StrList list );
extern void    StrListUnique( StrList list );
extern int     StrListMember(char term[], StrList list);

#else

extern void    StrListAppend( /* list, string */ );
extern void    StrListAppendFile( /* list, filename */ );
extern StrList StrListCreate( /* void */ );
extern void    StrListDestroy( /* list */ );
extern int     StrListEqual( /* list1, list2 */ );
extern int     StrListElementEqual( /* pos list1, list2 */ );
extern int     StrListCount( /*pos, list1, list2 */ );
extern char *  StrListPeek( /* list, index */ );
extern int     StrListSize( /* list */ );
extern void    StrListSort( /* list */ );
extern void    StrListUnique( /* list */ );
extern int     StrListMember(/* char term[], StrList list*/ );

#endif
#endif
```

```
/*******************************   stem.h   *********************************

   Purpose: Header file for an implementation of the Porter stemming
            algorithm.

   Notes:   This module implemnts a fast stemming function whose results
            are about as good as any other.
*****************************************************************************/

#ifndef STEM_H
#define STEM_H


/*****************************************************************************/
/****************************   Public Routines   ****************************/

#ifdef __STDC__

extern int Stem( char *word );      /* returns 1 on success, 0 otherwise */

#else

extern int Stem();

#endif

#endif



/*******************************   constants.h   *********************************/


/****************************   Public Constants   ****************************/
#define HEUR_SIZE 5
#define EOS  '\0'
#define FALSE  0
#define TRUE   1
#define LINE_SIZE 50000// how many characters can an element hold
#define TAG_SIZE    20// how many characters in a tag-name
#define NAME_SIZE    50// number of characters in a file / concept name
#define DOC_SIZE 100000// size of a document
#define MAX_CANDIDATES 1000// maximum number of candidate concepts
#define ID_SIZENAME_SIZE+7// size of ID
#define TEMP_FILE_NAME "temp_output.txt"



/*********************************   ruleadt.c   *********************************/

#include <stdio.h>
#include <string.h>

#include "ruleadt.h"
#include "constants.h"

void strip_slash_n(char temp[]){
  int i = strlen(temp); //locate end of string
  temp[i-1] = EOS;//replace \n with \0
} //strip slash n



/*************************************************************************
 *  Name:    AllHrefsInDomain
 *  Purpose: searches all files in a domain and gets all the links
 *           The result is written to two files: one in prolog-format (prolog)
 *           and one for temporar use throughout the parsing (linkfile).
 *   Calls:   TagsFromFile() from parseradt.c
 *   Plan:
 *    1. Get next filename from domain-file
 *    2. Parse current file for hrefs
 *    3. Write result to file
*************************************************************************/

void AllHrefsInDomain(char domainfilename[NAME_SIZE], char linkfile[NAME_SIZE], char
prologfilename[NAME_SIZE]){
  char tagsearch[TAG_SIZE];
  char result[LINE_SIZE];
  char resultpro[LINE_SIZE];
  FILE *outputfile, *domainfile, *prologfile;
  char current[NAME_SIZE];

  domainfile = fopen(domainfilename, "r"); //prepare to parse domain
  outputfile = fopen(linkfile, "w");  //make sure it is empty
  fclose(outputfile);
  while ( fgets(current, 50, domainfile) != NULL){
```

```
    strip_slash_n(current);  //must remove \n from the string filename.html\n

    strcpy(result, "");strcpy(resultpro, "");
    strcpy(tagsearch, "A");

    outputfile = fopen(linkfile, "a+");prologfile = fopen(prologfilename, "a+");

    (void) TagsFromFile(current, tagsearch, resultpro, result);
    if (result) {
    fprintf(outputfile, "%s", result);fprintf(prologfile, "%s", resultpro);
    }
    fclose(outputfile);fclose(prologfile);
  }//while domain is parsed
  fclose(domainfile);
}//AllHrefsInDomain


/*************************************************************************
 *  Name:    BubbleSort
 *  Purpose: Sorts an array of records based on the value-field
 *           into ascending order
 *  Notes:   Ascending order, that is 1-2-3-...-n
*************************************************************************/

void BubbleSort(CVR record[], int max){
int x, y;
CVR temp;//temporary record
// bubble sort the array

  for (x=0; x < max-1; x++)
    for (y=0; y < max-x-1; y++)
        if (record[y+1].value > record[y].value){  //y, y+1 for ascending
            temp = record[y];
            record[y] = record[y+1];
            record[y+1] = temp;
        }//if
}//BubbleSort


/*************************************************************************
 *  Name:    DomainToProlog
 *  Purpose: Writes all filenames of a domain to an output file in
 *           prolog-format.
 *  Notes:   Receives the name of a file with one filename on each line.
*************************************************************************/

void DomainToProlog(char domainfile[NAME_SIZE], char output[NAME_SIZE]){
  FILE *fp;
  char tempstring[LINE_SIZE], current[NAME_SIZE];

  strcpy(tempstring, ""); strcpy(current, "");
  strcat(tempstring, "domain( [");

  fp = fopen(domainfile, "r");
  while ( fgets(current, NAME_SIZE, fp) != NULL){
     strip_slash_n(current);  //must remove \n from filename.html\n
     strcat(tempstring, "\"");
     strcat(tempstring, current);
     strcat(tempstring, "\", ");
  } //while
  strcat(tempstring, "]).");
  fclose(fp);
  fp = fopen(output, "w");
  fprintf(fp, tempstring);
  fclose(fp);
}//DomainToProlog

/*************************************************************************
 *  Name:    InitArray
 *  Purpose: Builds a list of all candidate concepts based on the input
 *           string and assigns values according to term frequencies
 *  Calls:   LexicalAnalysis from parseradt.c and MaxWords
 *  Plan:
 *   1. Init array with <candidate, value> tuples set to zero
 *   2. Run lexical analysis
 *   3. Stem the terms
 *   4. Count number of words. Put each <term, frequency> tuple in array
 *   5. Return information found
*************************************************************************/

void InitArray(char input[LINE_SIZE], CVR *conceptvalues, int arraysize, int *used){
  char stopwords[LINE_SIZE], maxstring[LINE_SIZE];
  CVR *temp;
  int p=0, i=0;
  int count=0;
  strcpy(stopwords, ""); strcpy(maxstring, "");
  *used = 0;  //how many fields in array is used
```

```
  temp = conceptvalues;
  for (i=0; i<arraysize; i++){
    strcpy(temp->candidate, "");  //blank'em all
    temp->value = 0;
    temp++;
  }
  (void) LexicalAnalysis(input, stopwords, "res/temp_lexical.txt", &p);  // 2
  (void) Stemmer("res/temp_lexical.txt");   // 3
  (void) MaxWords("res/temp_lexical.txt", maxstring, &count, conceptvalues, used); // 4
}//InitArray


/*************************************************************************
 *  Name:     MaxWords
 *  Purpose:  Identifies all terms in tempfile[] with more than two occurrences
 *             and writes them to the string output[], and overwrites tempfile
 *            with a list of unique words.
 *  Modified: New modified version also includes:
 *              Writes tuples <term, frequency> to array and returns how much
 *            of the array is used.
 *  Returns:  Maximum number of words
 *  Note:     Incoming file should have one word on each line, for instance
 *            by having been exposed to lexical analysis first.
 *  Plan:
 *   1. Make a string list based on the input file
 *   2. Count the number of each word and save result
 *   3. Build array of all candidates with tuples <term, frequency>
 *   4. Return the largest number
*************************************************************************/

int MaxWords(char tempfile[], char output[], int *maximum , CVR *termfrequency, int *used) {
   char *term;     // for the next term from the input line
   StrList words, taltOpp;   // string lists of all the words on a file
   int size, antall, i;// counters

    /* Part 1: Create a list of words from a file */
   words = StrListCreate();
   StrListAppendFile( words, tempfile); // tempfile opened/closed inside function
   size = StrListSize( words );

    /* Part 2: Count the number of words and save output */
   taltOpp = StrListCreate();//for the words already counted
   antall = 0;
   *maximum = 0;
   for (i=0; i<size; i++) {  //list traversal
    term = StrListPeek(words, i);
    if ( !StrListElementEqual(i, words, taltOpp) )  { //word not counted yet
    antall = StrListCount(i, words, words);  //function to count frequency of current word
    StrListAppend( taltOpp, term );  //marks current word as counted
        if (antall > 1) {   //save the information:
    sprintf(output, "%s%s - %d ", output, term, antall);  //concatenate through the first %s
    }
        if (antall > (*maximum)) {
            *maximum = antall;
        }

        /* Part 3: Array of <term, frequency> */
        strcpy((termfrequency->candidate), term);
        (termfrequency->value) = antall;
        termfrequency++;
        (*used)++;// update number of terms inserted into the array
    } //if count
   }//for
                /* Part 4: Return maximum */
   return *maximum;
}


/*************************************************************************
 *  Name:    AssignPoints
 *  Purpose: Assigns "points" to those candidates from "candidatevalues"-array
 *           found in "incomingfile". "boundary" is used for efficient traversal
 *           and holds the number of elements in the array with candidates.
 *   Note:    Incoming file should have one word on each line in order to convert
 *           correctly to stringlist. This method is rather brilliant.
 *           IR-functions all write their results to temporary files, and these have
 *            terms that must be subsets of all words surviving lexical analysis.
 *            Therefore the membership test makes sense: If one of the candidates have
 *           membership in the incoming file, it should be given extra points
 *  Plan:
 *   1. Make stringlist of incoming file
 *   2. Check every candidate concept for membership in the list
 *   3. Update values in array and save pointer to first array-element
*************************************************************************/

void AssignPoints(CVR *candidatevalues, char incomingfile[], int points, int boundary){
```

```
  CVR *temp;
  char term[NAME_SIZE];     // for the next term from the input line
  StrList list;   // string list of all the words from an anti-stoplist file
  int i=0;

    /* 1 */
  list = StrListCreate();
  StrListAppendFile( list, incomingfile); // file is opened inside function
  if (StrListSize(list) <= 1)
    ;      // do nothing
  else {
    /* 2 */
    temp = candidatevalues;
    while (i++<boundary){ // boundary has the size of the array
      strcpy(term, temp->candidate);
      if (StrListMember(term, list))
    temp->value += points;// 3
      temp++;
    }//searching for membership
  }//else
}//AssignPoints


/*************************************************************************
 *  Name:    IsCombination
 *  Purpose: Checks for the combination "concept-tagName"
*************************************************************************/

int IsCombination(char concept[NAME_SIZE], char tagName[TAG_SIZE], char file[NAME_SIZE]){
  FILE *fp;
  StrList list;
  char comb[TAG_SIZE+NAME_SIZE];

  strcpy(comb, ""); sprintf(comb, "%s_%s", concept, tagName);

  list = StrListCreate();
  StrListAppendFile(list, file); // file is opened inside function
  if (StrListMember(comb, list))
    return TRUE;
  else {  //did not find a combination
    fp = fopen(file, "a+");
    fputs(comb, fp);  //update file
    fputs("\n", fp);
    fclose(fp);
    return FALSE;
  }//else
} //IsCombination
/*************************************************************************
 *  Name:    WriteInfoToFiles
 *  Purpose: Collects information about an incoming element:
 *            . emphasizer elements like <B>, <I> etc
 *            . links
 *            . meta and title
 *           The result is placed in files
 *   Note:    Result from sub-element is written to a file, one term for each line
 *   Plan:
 *    1. Get terms from sub-element specified. Returns stemmed terms.
 *    2. Output is not used, but result is written to file
*************************************************************************/

void WriteInfoToFiles(char element[LINE_SIZE], char file1[NAME_SIZE],
                      char file2[NAME_SIZE], char file3[NAME_SIZE],
                      char file4[NAME_SIZE], char file5[NAME_SIZE],
                      char file6[NAME_SIZE]   ){
  char output[LINE_SIZE];
  int p=0;
  strcpy(output, "");

    /* Part 1: Write information to files */
  (void) SubElementTerms("B", element, output, file1, &p);  // get all bold tags
  (void) SubElementTerms("I", element, output, file2, &p); // italic tags from the element
  (void) SubElementTerms("A", element, output, file3, &p); // a-href tags from the element
  (void) SubElementTerms("META", element, output, file4, &p);// meta tags
  (void) SubElementTerms("H1", element, output, file6, &p);// heading1
}//WriteInfoToFiles


/*************************************************************************
 *  Name:    LinkedTo
 *  Purpose: Identifies all href-elements from the file hrefs that contains
 *           the string current. All such occurrences are documents that refers to
 *           the current one, and hence the terms are candidate concepts. The result
 *           is placed in file with name fout
 *   Note:    Result from L.A. is written to a file fout, one term for each line
 *   Plan:
 *    1. Read line by line from hrefs
```

```
 *    2. Add up all lines that contains the string "current"
 *    3. Send these lines to Lexical Analysis
 *    4. Stem result
*****************************************************************************/

void LinkedTo(char current[NAME_SIZE], char fout[NAME_SIZE], char hrefs[NAME_SIZE]){
  FILE *fp;
  char line[LINE_SIZE], temp[LINE_SIZE], notused[LINE_SIZE];
  int p=0;  //not used

  strcpy(line, ""); strcpy(temp, ""); strcpy(notused, "");
  fp = fopen(hrefs, "r");
  while (fgets(line, LINE_SIZE, fp) != NULL) {// 1.
    if (strstr(line, current) ) // 2.
    strcat(temp, line);
  }//while
  fclose(fp);
  LexicalAnalysis(temp, notused, fout, &p); // 3.
  (void) Stemmer(fout);   // 4

}//LinkedTo



/*****************************************************************************
 *  Name:    SelectConcept
 *  Purpose: Selects the concept with the highest score
 *   Plan:
 *    1. Pick concept with highest value
*****************************************************************************/

void SelectConcept(CVR *doc_conceptvalues, int location, char concept[], int *pt){

  //don't pick out if the field has an empty name
  if (0 == strcmp(doc_conceptvalues[location].candidate, "") )
    location++;

  strcpy(concept, doc_conceptvalues[location].candidate);
  (*pt) = doc_conceptvalues[location].value;
}//SelectConcept
```

```
/*********************************   parseradt.c   *********************************
     Written by:  Svend Andreas Horgen
     Date:        May 2002
     Purpose:     Functions to assist the parsing of an HTML-document
     Notes:       Does assume that the last text in a line is an end-tag
*************************************************************************************/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parseradt.h"
#include "constants.h"
#include "strlist.h"
#include "stop.h"
#include "stem.h"

/************************ Private Function Declarations   *******************/

/* Function to make sure every record field is correctly initialized */
int InitRecord(ER *pt){
     /* string fields are initialized */
  strcpy(pt->id, "");  strcpy(pt->tagName, "");
  strcpy(pt->conceptName, "");  strcpy(pt->content, "");
  strcpy(pt->tempfile, TEMP_FILE_NAME); strcpy(pt->href, "");
  strcpy(pt->stopwords, "");  strcpy(pt->subwords, "");
  strcpy(pt->antiwords, "");strcpy(pt->maxwords, "");
    /* integer-fields are set to 0 */
  pt->stopcount = pt->subcount = pt->anticount = pt->maxcount = 0;
  return TRUE;
} //InitRecord

/* Function to produce the string </TAG> */
char *MakeEndTag(char tagName[]){
  char endtag[TAG_SIZE];
  sprintf(endtag, "</%s>", tagName);
  return endtag;
}//EndTag

/* Function to produce the string "<TAG>" */
char *MakeStartTag(char tagName[]){
  char starttag[TAG_SIZE];
  sprintf(starttag, "<%s>", tagName);
  return starttag;
}//StartTag

/* Function to produce the string "<TAG " (accounts for <TAG STYLE="blabla">) */
char *MakeStartTagOpen(char tagName[]){
  char starttag[TAG_SIZE];
  sprintf(starttag, "<%s ", tagName);
  return starttag;
}//StartTag

/* Function to convert a string to lower case */
char *StrToLower(char temp[]){
  int c;
  int i=0;

  while( (c=temp[i]) != '\0') {
    if (isupper(c) )
      temp[i] = tolower(c);
    i++;
  }
  return temp;
}//StrToLower

/* Function to grab the next token from an input stream */
static char *GetNextTerm( FILE *stream, int size, char *term )  {
   char *ptr;  /* for scanning through the term buffer */
   int ch;     /* current character during input scan */

           /* Part 1: Return NULL immediately if there is no input */
   if ( EOF == (ch = getc(stream)) ) return( NULL );

                  /* Part 2: Initialize the local variables */
   *term = EOS;
   ptr = term;

        /* Part 3: Main Loop: Put the next word into the term buffer */
   do {
         /* scan past any leading non-alphabetic characters */
      while ( (EOF != ch ) && !isalpha(ch) ) ch = getc( stream );

         /* copy input to output while reading alphabetic characters */
      while ( (EOF != ch ) && isalpha(ch) ){
         if ( ptr == (term+size-1) ) ptr = term;
         *ptr++ = ch;
         ch = getc( stream );
```

```
        }

        /* terminate the output buffer */
      *ptr = EOS;
      }
   while ( (EOF != ch) && !*term );

                  /* Part 4: Return the output buffer */
   return( term );
} /* GetNextTerm */


/************************ Public Function Declarations   *********************/

/*****************************************************************************
 *  Name:    AllImages
 *  Purpose: Puts all image-tags from the input-string to output[]
 *  Returns: Number of images found
 *   Note:    To find all images in a document, simply convert the entire
 *           document to a single string first, using FileToStr()
 *           Then call AllImages with this string.
 *   Plan:
 *    1. Initializing
 *    2. Identify all image-elements
 *    3. Return
*****************************************************************************/

int AllImages(char input[], char output[], int *counter){

  char starttag[TAG_SIZE];
  char tempin[LINE_SIZE], tempout[LINE_SIZE];
  char *ptr, *start, *end;
  int j=0;

    /* Part 1: Initializing */
  strcpy(tempout, "");
  strcpy(tempin, input);
  (void) StrToLower(tempin);
  strcpy(starttag, MakeStartTagOpen("img"));  // <IMG SRC="imagename.jpg" ALT="optional">

    /* Part 2: Search for all sub-elements */
  ptr = tempin;
  while ( start = strstr(ptr, starttag)) {
       end = strstr(start, ">");//find matching ">"
       if (!end) //no endtag found
       break;
    /* Part 3: Catch the contents */
    end++;  //to move pointer behind ">"
       while (start != end)
       tempout[j++] = *start++ ; //shorthand notation that saves two more lines
       ptr = end;
       (*counter)++;  //else
  }//while

    /* Part 4: Update output string and return indication of success */
  tempout[j] = EOS; //if nothing found, then output string is empty since j=0
  strcpy(output, tempout);
  return *counter;
}//AllImages


/*****************************************************************************
 *  Name:    AntiStoplist
 *   Note:    This is the implementation of the DSL, domain specific list,
 *           which originally was called AntiStopList.
 *           The anti-stoplist itself must have just one \n after the last word.
 *  Purpose: Runs a file against an anti-stoplist given by antiFilename
 *           (An anti-stoplist contains domain specific keywords)
 *           Inputfile[] is a file that has gone through lexical
 *           analysis with one word for each line.
 *           Saves the words that occur both in the file and
 *           in the anti-stoplist to the string output[].
 *   Returns: Number of words found in anti-stoplist.
 *   Plan:
 *    1. Create string lists of the words from the anti-stoplist and inputfile
 *    2. Traverse the input-list and comparing with the anti-list
 *    3. Collect words found into the string pointed to by output[]
 *    4. Returning number of words
*****************************************************************************/

int AntiStoplist(char inputfile[], char output[], char antiFilename[], int *counter){
   char term[50];     /* for the next term from the input line */
   StrList words, anti;   /* string lists of all the words on a file */
                          /* and all the the words from an anti-stoplist file */
   int size, i;/* the number of words from the file and the anti-stoplist */

                     /* Part 1: Create a list of words read from a file */
```

```
   words = StrListCreate();
   anti = StrListCreate();
   StrListAppendFile( words, inputfile ); // file is opened in function, one word per line
   StrListAppendFile( anti, antiFilename); // correct anti-list file opened in function
   StrListUnique(words); //removing duplicates

                /* Part 2: Traverse the list while comparing with the anti-list */
   *counter = 0;
   size = StrListSize( words );
   for(i=0; i<size; i++) {
     strcpy(term, StrListPeek(words, i) );

                /* Part 3: Collect words found */
     if (StrListElementEqual(i, words, anti)) {
                //the term in position i in words is found in anti
    strcat(term, ", ");
    strcat(output, term);
    (*counter)++;
     } //if
   } //for

                /* Part 4: Returning number of words */
   return *counter;
} //AntiStoplist


/*************************************************************************
 *  Name:    FileToStr
 *  Purpose: Writes an entire file to a single string
 *   Plan:
 *    1. Open file and initialize output string
 *    2. Read line by line while adding to the string
 *    3. Close file. String is accessible through pointer
*************************************************************************/

void FileToStr(char filename[NAME_SIZE], char output[]){
  FILE *docfile;
  char line[LINE_SIZE];

    /* Part 1: Open file and initialize output string */
  strcpy(output, "");
  docfile = fopen(filename, "r");

    /* Part 2: Read line by line while adding to the string */
  while (fgets(line, LINE_SIZE, docfile) != NULL)
    strcat(output, line);

    /* Part 3: Close file. String is accessible through pointer */
  (void) fclose(docfile);
}//FileToStr


/*************************************************************************
 *  Name:     GetFirstTag
 *  Purspose: Searches for the first occurrence of a tag in the input string
 *            given, and places the result in output[]
 *   Returns:  indication of success, TRUE if a tag was found
 *   Plan:
 *    1. Initializing + avoid tag-errors
 *    2. Search for element
 *    3. Catch the contents of this tag
 *    4. Updating output-string and return
*************************************************************************/

int GetFirstTag(char input[], char output[], char tag[]){
  char *start, *end;
  int size, j, a;
  char temp[LINE_SIZE], temp2[LINE_SIZE]; //in order not to overwrite the contents...
  char starttag[TAG_SIZE], starttagopen[TAG_SIZE], endtag[TAG_SIZE];
                //need space for tags like </blockquote> etc

    /* Part 1: Initializing and dealing with possible tag-errors */
  (void) StrToLower(tag);
  strcpy(starttag, MakeStartTag(tag));
  strcpy(endtag, MakeEndTag(tag));
  strcpy(starttagopen, MakeStartTagOpen(tag));
                //to avoid <BR>, <B>, <B Style="blabla"> error
  size = j = 0;  strcpy(temp, "");
  strcpy(temp2, input);

    /* Part 2: Search for element */
  (void) StrToLower(temp2);
  if ( (start =  strstr(temp2, starttag)) || (start = strstr(temp2, starttagopen))   ) {
       end = strstr(temp2, endtag);
       if (!end)  //no endtag found
       return FALSE;
       for (a=0; a<strlen(endtag); a++) end++;
```

```
    /* Part 3: Catch the contents of this tag */
       while (start != end)
       temp[j++] = *start++ ; //shorthand notation that saves two more lines
       temp[j] = EOS; //first copy sub-element, then place a '\0' to mark end of string

        /* Part 4: Updating output-string and indicate success */
       strcpy(output, temp);
       return TRUE;
  }//if
  return FALSE; //if no tag found...
}//GetFirstTag


/*********************************************************************
 *  Name:    GetNextElement
 *  Purpose: Collects the text and tagname from the next element from a file stream.
 *  Returns: Indication of success.
 *  Note:    This function does case-insensitive tag-checking,
 *           so ul, Ul, uL and UL are all treated equally.
 *  Plan:
 *   1. Initializing
 *   2. Collect the text from the element found
 *   3. Return the text of the element
*********************************************************************/

int GetNextElement(FILE *docfile, char output[], char tag[]) {

  char line[LINE_SIZE], ut[LINE_SIZE]; //can hold LINE_SIZE characters
  int funnet=FALSE;
  char endtag[TAG_SIZE];
  char temp[LINE_SIZE];  //helping to make tags lowercase

    /* Part 1. Initializing the temporary variables  */
  strcpy(ut, ""); strcpy(line, "");

    /* Part 2. Read line by line until end of document or a tag is found */
  while ( fgets(line, LINE_SIZE, docfile) != NULL ) {
    strcpy(temp, line);
    (void) StrToLower(temp);
    if ( strstr(temp, "<h1") ) {  // accounts for tags with attributes
      funnet = TRUE;  // since the > is missing in the test
      strcpy(tag, "h1"); }
    else if ( strstr(temp, "<h2") ) {
      funnet = TRUE;
      strcpy(tag, "h2"); }
    else if ( strstr(temp, "<h3") ) {
      funnet = TRUE;
      strcpy(tag, "h3"); }
    else if ( strstr(temp, "<table") ) {
      funnet = TRUE;
      strcpy(tag, "table"); }
    else if ( strstr(temp, "<ul") ) {
      funnet = TRUE;
      strcpy(tag, "ul"); }
    else if ( strstr(temp, "<ol") ) {
      funnet = TRUE;
      strcpy(tag, "ol"); }
    else if ( strstr(temp, "<p") ) {
      funnet = TRUE;
      strcpy(tag, "p");}
    else if ( strstr(temp, "<blockquote") ) {
      funnet = TRUE;
      strcpy(tag, "blockquote"); }///etc...

    if (funnet) break; //jump out of while-loop
  } //while

    /* Part 2. Collect the text from the element found */
  if (funnet) {
    strcat(ut, line);//add the line to the content of the element (ut)
    strcpy(endtag, MakeEndTag(tag)); // making a string with "</tag>"
    while ( ! strstr(temp, endtag) ) {
    if (fgets(line, LINE_SIZE, docfile) != NULL){ //inserts an /0 by the end of line.
      strcat(ut, line);   //adds line to the string ut
      strcpy(temp, line);
        (void) StrToLower(temp);
    }//if
    else break; //jump out of while if no more lines in file (that is no endtag found)
    } //while

    /* Part 3. Update and return the element found */
    strcpy(output, ut);  //the text of the element
    return 1;  //return and exit function immediately
  }//funnet

  return 0;  //happens when no tags found.
}//GetNextElement
```

```
/*************************************************************************
 *  Name:    Id
 *  Purpose: Constructs an unique ID for a string. The ID takes the following
 *           form: ID = DocumentName + "#" + "sub" + counter.
 *           Example: testfile.html#sub3
 *   Returns: Indication of success
 *   Note:    the form testfile.html#sub3 is used because then it is easy to
 *           generate links in the adaptive document later.
 *   Plan:
 *    1. Compose the ID
 *    2. Update the counter and return
*************************************************************************/

int Id(char docname[ID_SIZE], int *counter, char elementid[NAME_SIZE]) {

  strcpy(elementid, "");

    /* Part 1: Compose the ID */
  sprintf(elementid, "%s#elm%d", docname, *counter);

    /* Part 2: Update the counter first, then return */
  return ++(*counter);
}//Id


/*************************************************************************
 *  Name:    Lexical Analysis
 *  Purpose: Removes stopwords and performs a lexical analysis on the element
 *           specified by the paramater input[]
 *           Output is a pointer to a string where the result is placed.
 *           The result is also written to the file FileOut
 *   Returns: number of words remaining after the analysis
 *   Uses:    parseradt.c
 *   Note:    The temporary file is written over when a new element is being
 *           analysed, but its filename is necessary as input for the function
 *           WordCount.   (char input[] is a pointer)
 *   Plan:
 *    1. Work against a temporary file
 *    2. Building DFA for filtering stopwords
 *    3. Remove stopwords from input file
 *    4. Close input file and return
*************************************************************************/

int LexicalAnalysis(char input[], char output[], char FileOut[NAME_SIZE], int *counter) {

  FILE *stream;/* temporary file */
  FILE *outputfil;/* file to hold remaining words after lexical analysis */
  char FileIn[20];/* logical file name */
  char term[128];/* for the next term found */
  StrList words;/* the stop list filtered */
  DFA machine;/* build DFA from the stop list */

  strcpy(FileIn, "res/temp_lex.txt"); // input string is written to this file

    /* Part 1. Write the content of the string input[] to tempfile */
  stream = fopen(FileIn, "w");
  fputs(input, stream);
  (void) fclose(stream);

    /* Part 2. Building a DFA for filtering stopwords */
  words = StrListCreate();
  StrListAppendFile( words, "stop.wrd" );
  machine = BuildDFA( words ); //create a DFA

    /* Part 3. Remove stopwords from input-file, and preserving information
       in both a file and the output-string */
  if ( !(stream = fopen(FileIn,"r")) ) exit(1); //open temporary file
  outputfil = fopen(FileOut, "w");
  while ( (NULL != GetTerm(stream,machine,128,term)) ){
      fputs(term, outputfil);
      fputs("\n", outputfil);
      strcat(term, ", ");
      strcat(output, term);
  }

    /* Part 4: Closing the files */
  (void)fclose( stream );
  (void)fclose( outputfil );

    /* Part 5: Counting number of words on file and return */
  (void) WordCount(FileOut, counter);  //file that is subject to word count
  return *counter;
} //LexicalAnalysis
```

```
/*************************************************************************
 *  Name:    Stemmer
 *  Purpose: Porter stemming function. Takes a single filename and writes
 *           the stemmed terms to the file result.
 *  Calls:   GetNextTerm
 *  Uses:    Stem (from stem.c)
 *  Plan:
 *   1. Open the input file
 *   2. Process each word in the file
 *   3. Close the input file
*************************************************************************/

int Stemmer(char filename[NAME_SIZE]){
   char term[64];   /* for the next term from the input line */
   char temp[LINE_SIZE];
   FILE *stream;    /* where to read characters from */
   strcpy(temp, "");

                       /* Part 1: Open the input file og outputfil */
   if ( !(stream = fopen(filename,"r")) ) exit(1);  //The file to be stemmed

                  /* Part 2: Process each word in the file */
   while( GetNextTerm(stream,64,term) ) {
      if ( Stem(term) ) {
          strcat(temp, term);
          strcat(temp, "\n");
      } //if
   } //while
                /* Part 3: Close the file */
   (void)fclose( stream );

                /* Part 4: Write stemmed result back to file */
   stream = fopen(filename, "w");
   fputs(temp, stream);  /* resulting term to output */
   (void)fclose( stream );
   return(0);
} //Stemmer


/*************************************************************************
 *  Name:    SubElementTerms
 *  Purpose: Locates all sub-elements specified by tag[] in the string input[].
 *           If any found, they are all run through lexical analysis. The resulting
 *           terms are put in the string output[] and in the file tempfile and the number
 *           of elements placed in the integer size
 *  Returns: Success
 *  Calls:   TagsFromString() and LexicalAnalysis()
 *  Note:    Lexical analysis writes the result to a file and the string output[].
 *           Here only the output is used further
 *  Plan:
 *   1. Perform lexical analysis on the sub-elements found
 *   2. Return number of survivors (0 if no remains or if no sub-elements found)
*************************************************************************/

int SubElementTerms(char tag[TAG_SIZE], char input[], char output[],
                    char filein[NAME_SIZE], int *size){

  FILE *fp;
  char result[LINE_SIZE];
  strcpy(result, "");
                 /* Part 1: Identify sub-element and run lexical analysis */
  *size = 0;
  fp = fopen(filein, "w"); fclose(fp);
  if (TagsFromString(tag, input, result) ) {
    (void) LexicalAnalysis(result, output, filein, size); //filein is used further
    (void) Stemmer(filein);     //stem
  }
                 /* Part 2: Return success */
  return *size;
}//SubElement
```

```
/*************************************************************************
 *  Name:    TagsFromFile
 *  Purpose: gets all elements specified by "tag" from the file and writes the result
 *  to the string ut[]
 *  Returns: True if some elements found
 *  Plan:
 *   1. Initialize the variables
 *   2. Read file into one line
 *   3. Get the next tag searching from position ptr
 *   4. Get link target
 *   5. Format link information for PROLOG
 *   6. Return success
*************************************************************************/

int TagsFromFile(char filename[], char tag[TAG_SIZE],
                 char pro[LINE_SIZE], char ut2[LINE_SIZE]){
  char line[LINE_SIZE], temp[LINE_SIZE], resultat[LINE_SIZE], resultat2[LINE_SIZE];
  char *start, *ptr, *a, *b;
  char c[NAME_SIZE];//to hold link target
  int i, j;

    /* Part 1: Initializing the variables */
  strcpy(pro, ""); strcpy(line, ""); strcpy(temp, ""); strcpy(resultat, ""); strcpy(c, "");
  strcpy(ut2, "");

    /* Part 2: Read file into one long line */
  (void) FileToStr(filename, temp);
  (void) StrToLower(temp);

    /* Part 3: Get the next tag searching from position ptr */
  ptr = temp;
  start = NULL;
  while( GetFirstTag(ptr, resultat, tag) ) {

    /* Part 4: Move ptr to the start of remaining string, get link */
    start = strstr(ptr, resultat);
    while (ptr != start)
    ptr++;
    for (i=0; i<strlen(resultat); i++) ++ptr;
    strcat(ut2, resultat);  strcat(ut2, "\n");  //get link

    /* Part 5: Formatting in PROLOG format */
    strcpy(c, "");j = 0;
    a = strchr(resultat, '\"');  a++;  b = strchr(a, '\"');
    while (a != b)
       c[j++] = *a++ ; //shorthand notation that saves two more lines
    c[j] = EOS; //after sub-element is copied, then place a '\0' to mark end of string
    sprintf(resultat2, "*href(\"%s\", \"%s\", %s).\n", filename, c, resultat);
    strcat(pro, resultat2);
  }//while

    /* Part 6: Return success */
  if (start) //an element was found
    return TRUE;
  else
    return FALSE;
} //TagsFromFile


/*************************************************************************
 *  Name:    TagsFromString
 *  Purpose: Searches for all tags requested for in the input string given, and places
 *           the result in the string output[]
 *  Returns: indication of success, TRUE if at least one tag was found
 *  Plan:
 *   1. Initializing + avoid tag-errors
 *   2. Search for all tags
 *   3. Add the contents to the output string
 *   4. Updating output-string and return
*************************************************************************/

int TagsFromString(char tag[TAG_SIZE], char input[], char output[]){
  char *start, *end, *ptr;
  int  size, j, a, success;
  char temp[LINE_SIZE], temp2[LINE_SIZE];
  char starttag[TAG_SIZE], starttagopen[TAG_SIZE], endtag[TAG_SIZE];
             //need space for tags like </blockquote> etc

    /* Part 1: Initializing */
  size = j = 0;   success = FALSE;   strcpy(temp, "");
  strcpy(temp2, input);
  (void) StrToLower(tag);
  (void) StrToLower(temp2);
  strcpy(starttag, MakeStartTag(tag));
  strcpy(endtag, MakeEndTag(tag));
  strcpy(starttagopen, MakeStartTagOpen(tag)); //to avoid <BR>, <B>, <B Style="blabla"> error
```

```
    /* Part 2: Search for all sub-elements */
  ptr = temp2;
  while ( (start = strstr(ptr, starttag)) || (start = strstr(ptr, starttagopen)) ) {
       end = strstr(ptr, endtag);
       if (!end) //no endtag found
       break;
       success = TRUE;

    /* Part 3: Catch the contents */
    for (a=0; a<strlen(endtag); a++) ++end;
       while (start != end)
       temp[j++] = *start++ ; //shorthand notation that saves two more lines
       ptr = end;
  }//while

    /* Part 4: Update output string and return indication of success */
  temp[j] = EOS; //if nothing found, then output string is empty since j=0
  strcpy(output, temp);
  return success; //1 if found, 0 if not
}//TagsFromString


/*************************************************************************
 *  Name:    WordCount
 *  Purpose: Counts the number of words (lines) in a file. The advantage with this
 *           version of word-count is that is removes duplicate terms.
 *  Returns: number of words
 *  Notes: Assumes that the file has one word only in each line. Uses the ADT StrList
 *  Plan:
 *   1. Create string list from file
 *   2. Computing the size
 *   3. Returning the size
*************************************************************************/

int WordCount(char filename[], int *size){

  StrList words;/* string lists of all the words on a file  */

    /* Part 1: Create a string list from the content of a file */
  words = StrListCreate();
  StrListAppendFile( words, filename);
            //read from the specified file and add terms to the list

    /* Part 2: Computing size */
  StrListUnique(words);  //removes duplicates in the list
  *size = StrListSize( words ); //return number of words now as duplicates are removed
  (*size)--;  //somehow one line is empty at the end...

    /* Part 3: Returning the size */
  return *size;
} //WordCount
```

```
/*********************************   stop.c   *******************************

    Purpose:     Stop list filter finite state machine generator and driver.

    Provenence:  Written by and unit tested by C. Fox, 1990.
                 Changed by C. Fox, July 1991.
                    - added ANSI C declarations
                    - branch tested to 95% coverage

    Notes:       This module implements a fast finite state machine
                 generator, and a driver, for implementing stop list
                 filters.  The strlist module is a simple string array
                 data type implementation.
**/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <malloc.h>

#include "stop.h"
#include "strlist.h"

/*****************************************************************************/

#define FALSE                       0
#define TRUE                        1
#define EOS                        '\0'

         /**************   Character Classification   ***************/
         /* Tokenizing requires that ASCII be broken into character */
         /* classes distinguished for tokenizing.  Delimiter chars  */
         /* separate tokens.  Digits and letters make up the body   */
         /* of search terms.                                        */

typedef enum {

               DELIM_CH,             /* whitespace, punctuation, etc. */
               DIGIT_CH,             /* the digits */
               LETTER_CH,            /* upper and lower case */

            } CharClassType;

static CharClassType char_class[128] = {
      /* ^@ */  DELIM_CH,    /* ^A */  DELIM_CH,    /* ^B */  DELIM_CH,
      /* ^C */  DELIM_CH,    /* ^D */  DELIM_CH,    /* ^E */  DELIM_CH,
      /* ^F */  DELIM_CH,    /* ^G */  DELIM_CH,    /* ^H */  DELIM_CH,
      /* ^I */  DELIM_CH,    /* ^J */  DELIM_CH,    /* ^K */  DELIM_CH,
      /* ^L */  DELIM_CH,    /* ^M */  DELIM_CH,    /* ^N */  DELIM_CH,
      /* ^O */  DELIM_CH,    /* ^P */  DELIM_CH,    /* ^Q */  DELIM_CH,
      /* ^R */  DELIM_CH,    /* ^S */  DELIM_CH,    /* ^T */  DELIM_CH,
      /* ^U */  DELIM_CH,    /* ^V */  DELIM_CH,    /* ^W */  DELIM_CH,
      /* ^X */  DELIM_CH,    /* ^Y */  DELIM_CH,    /* ^Z */  DELIM_CH,
      /* ^[ */  DELIM_CH,    /* ^\ */  DELIM_CH,    /* ^] */  DELIM_CH,
      /* ^^ */  DELIM_CH,    /* ^_ */  DELIM_CH,    /*    */  DELIM_CH,
      /*  ! */  DELIM_CH,    /*  " */  DELIM_CH,    /*  # */  DELIM_CH,
      /*  $ */  DELIM_CH,    /*  % */  DELIM_CH,    /*  & */  DELIM_CH,
      /*  ' */  DELIM_CH,    /*  ( */  DELIM_CH,    /*  ) */  DELIM_CH,
      /*  * */  DELIM_CH,    /*  + */  DELIM_CH,    /*  , */  DELIM_CH,
      /*  - */  DELIM_CH,    /*  . */  DELIM_CH,    /*  / */  DELIM_CH,
      /*  0 */  DIGIT_CH,    /*  1 */  DIGIT_CH,    /*  2 */  DIGIT_CH,
      /*  3 */  DIGIT_CH,    /*  4 */  DIGIT_CH,    /*  5 */  DIGIT_CH,
      /*  6 */  DIGIT_CH,    /*  7 */  DIGIT_CH,    /*  8 */  DIGIT_CH,
      /*  9 */  DIGIT_CH,    /*  : */  DELIM_CH,    /*  ; */  DELIM_CH,
      /*  < */  DELIM_CH,    /*  = */  DELIM_CH,    /*  > */  DELIM_CH,
      /*  ? */  DELIM_CH,    /*  @ */  DELIM_CH,    /*  A */  LETTER_CH,
      /*  B */  LETTER_CH,   /*  C */  LETTER_CH,   /*  D */  LETTER_CH,
      /*  E */  LETTER_CH,   /*  F */  LETTER_CH,   /*  G */  LETTER_CH,
      /*  H */  LETTER_CH,   /*  I */  LETTER_CH,   /*  J */  LETTER_CH,
      /*  K */  LETTER_CH,   /*  L */  LETTER_CH,   /*  M */  LETTER_CH,
      /*  N */  LETTER_CH,   /*  O */  LETTER_CH,   /*  P */  LETTER_CH,
      /*  Q */  LETTER_CH,   /*  R */  LETTER_CH,   /*  S */  LETTER_CH,
      /*  T */  LETTER_CH,   /*  U */  LETTER_CH,   /*  V */  LETTER_CH,
      /*  W */  LETTER_CH,   /*  X */  LETTER_CH,   /*  Y */  LETTER_CH,
      /*  Z */  LETTER_CH,   /*  [ */  DELIM_CH,    /*  \ */  DELIM_CH,
      /*  ] */  DELIM_CH,    /*  ^ */  DELIM_CH,    /*  _ */  DELIM_CH,
      /*  ` */  DELIM_CH,    /*  a */  LETTER_CH,   /*  b */  LETTER_CH,
      /*  c */  LETTER_CH,   /*  d */  LETTER_CH,   /*  e */  LETTER_CH,
      /*  f */  LETTER_CH,   /*  g */  LETTER_CH,   /*  h */  LETTER_CH,
      /*  i */  LETTER_CH,   /*  j */  LETTER_CH,   /*  k */  LETTER_CH,
      /*  l */  LETTER_CH,   /*  m */  LETTER_CH,   /*  n */  LETTER_CH,
      /*  o */  LETTER_CH,   /*  p */  LETTER_CH,   /*  q */  LETTER_CH,
      /*  r */  LETTER_CH,   /*  s */  LETTER_CH,   /*  t */  LETTER_CH,
      /*  u */  LETTER_CH,   /*  v */  LETTER_CH,   /*  w */  LETTER_CH,
      /*  x */  LETTER_CH,   /*  y */  LETTER_CH,   /*  z */  LETTER_CH,
      /*  { */  DELIM_CH,    /*  | */  DELIM_CH,    /*  } */  DELIM_CH,
      /*  ~ */  DELIM_CH,    /* ^? */  DELIM_CH,                          };
```

```
        /**************   Character Case Conversion   **************/
        /* Term text must be accumulated in a single case.  This   */
        /* array is used to convert letter case but otherwise      */
        /* preserve characters.                                    */

static char convert_case[128] = {
      /* ^@ */    0,    /* ^A */    0,    /* ^B */    0,    /* ^C */    0,
      /* ^D */    0,    /* ^E */    0,    /* ^F */    0,    /* ^G */    0,
      /* ^H */    0,    /* ^I */    0,    /* ^J */    0,    /* ^K */    0,
      /* ^L */    0,    /* ^M */    0,    /* ^N */    0,    /* ^O */    0,
      /* ^P */    0,    /* ^Q */    0,    /* ^R */    0,    /* ^S */    0,
      /* ^T */    0,    /* ^U */    0,    /* ^V */    0,    /* ^W */    0,
      /* ^X */    0,    /* ^Y */    0,    /* ^Z */    0,    /* ^[ */    0,
      /* ^\ */    0,    /* ^] */    0,    /* ^^ */    0,    /* ^_ */    0,
      /*    */  ' ',    /*  ! */  '!',    /*  " */  '"',    /*  # */  '#',
      /*  $ */  '$',    /*  % */  '%',    /*  & */  '&',    /*  ' */ '\'',
      /*  ( */  '(',    /*  ) */  ')',    /*  * */  '*',    /*  + */  '+',
      /*  , */  ',',    /*  - */  '-',    /*  . */  '.',    /*  / */  '/',
      /*  0 */  '0',    /*  1 */  '1',    /*  2 */  '2',    /*  3 */  '3',
      /*  4 */  '4',    /*  5 */  '5',    /*  6 */  '6',    /*  7 */  '7',
      /*  8 */  '8',    /*  9 */  '9',    /*  : */  ':',    /*  ; */  ';',
      /*  < */  '<',    /*  = */  '=',    /*  > */  '>',    /*  ? */  '?',
      /*  @ */  '@',    /*  A */  'a',    /*  B */  'b',    /*  C */  'c',
      /*  D */  'd',    /*  E */  'e',    /*  F */  'f',    /*  G */  'g',
      /*  H */  'h',    /*  I */  'i',    /*  J */  'j',    /*  K */  'k',
      /*  L */  'l',    /*  M */  'm',    /*  N */  'n',    /*  O */  'o',
      /*  P */  'p',    /*  Q */  'q',    /*  R */  'r',    /*  S */  's',
      /*  T */  't',    /*  U */  'u',    /*  V */  'v',    /*  W */  'w',
      /*  X */  'x',    /*  Y */  'y',    /*  Z */  'z',    /*  [ */  '[',
      /*  \ */   92,    /*  ] */  ']',    /*  ^ */  '^',    /*  _ */  '_',
      /*  ` */  '`',    /*  a */  'a',    /*  b */  'b',    /*  c */  'c',
      /*  d */  'd',    /*  e */  'e',    /*  f */  'f',    /*  g */  'g',
      /*  h */  'h',    /*  i */  'i',    /*  j */  'j',    /*  k */  'k',
      /*  l */  'l',    /*  m */  'm',    /*  n */  'n',    /*  o */  'o',
      /*  p */  'p',    /*  q */  'q',    /*  r */  'r',    /*  s */  's',
      /*  t */  't',    /*  u */  'u',    /*  v */  'v',    /*  w */  'w',
      /*  x */  'x',    /*  y */  'y',    /*  z */  'z',    /*  { */  '{',
      /*  | */  '|',    /*  } */  '}',    /*  ~ */  '~',    /* ^? */    0, };

#define DEAD_STATE                  -1   /* used to block a DFA */
#define TABLE_INCREMENT            256   /* used to grow tables */


       /*************************   Hashing   *************************/
       /* Sets of suffixes labeling states during the DFA construction */
       /* are hashed to speed searching.  The hashing function uses an */
       /* entire integer variable range as its hash table size;  in an */
       /* effort to get a good spread through this range, hash values  */
       /* start big, and are incremented by a lot with every new word  */
       /* in the list.  The collision rate is low using this method    */

#define HASH_START              5775863
#define HASH_INCREMENT         38873647


       /**************   State Label Binary Search Tree   **************/
       /* During DFA construction, all states must be searched by      */
       /* their labels to make sure that the minimum number of states  */
       /* are used.  This operation is sped up by hashing the labels   */
       /* to a signature value, then storing the signatures and labels */
       /* in a binary search tree.  The tree is destroyed once the DFA */
       /* is fully constructed.                                        */

typedef struct TreeNode {
       StrList label;            /* state label used as search key     */
       unsigned signature;       /* hashed label to speed searching    */
       int state;                /* whose label is representd by node  */
       struct TreeNode *left;    /* left binary search subtree         */
       struct TreeNode *right;   /* right binary search subtree        */
       } SearchTreeNode, *SearchTree;


       /*********************   DFA State Table   **********************/
       /* The state table is an array of structures holding a state    */
       /* label, a count of the arcs out of the state, a pointer into  */
       /* the arc table for these arcs, and a final state flag.  The   */
       /* label field is used only during machine construction.        */

typedef struct {
       StrList label;            /* for this state - used during build */
       int num_arcs;             /* for this state in the arc table    */
       int arc_offset;           /* for finding arcs in the arc table  */
       short is_final;           /* TRUE iff this is a final state     */
       } StateTableEntry, *StateTable;
```

```
       /**********************   DFA Arc Table   **********************/
       /* The arc table lists all transitions for all states in a DFA  */
       /* in compacted form.  Each state's transitions are offset from */
       /* the start of the table, then listed in arc label order.      */
       /* Transitions are found by a linear search of the sub-section  */
       /* of the table for a given state.                              */

typedef struct {
       char label;              /* character label on an out-arrow    */
       int target;              /* the target state for the out-arrow */
       } ArcTableEntry, *ArcTable;


       /**********************   DFA Structure   **********************/
       /* A DFA is represented as a pointer to a structure holding the */
       /* machine's state and transition tables, and bookkeepping      */
       /* counters.  The tables are arrays whose space is malloc'd,    */
       /* then realloc'd if more space is required.  Once a machine is */
       /* constructed, the table space is realloc'd one last time to   */
       /* fit the needs of the machine exactly.                        */

typedef struct _DfaStruct {
       int num_states;          /* in the DFA (and state table)       */
       int max_states;          /* now allocated in the state table   */
       int num_arcs;            /* in the arc table for this machine  */
       int max_arcs;            /* now allocated in the arc table     */
       StateTable state_table;  /* the compacted DFA state table      */
       ArcTable arc_table;      /* the compacted DFA transition table */
       SearchTree tree;         /* storing state labels used in build */
       } DFAStruct;

/*****************************************************************************/
/*************************   Function Declarations   *************************/

#ifdef __STDC__

static char *GetMemory( char *ptr, int num_bytes );
static void  DestroyTree( SearchTree tree );
static int   GetState( DFA machine, StrList label, unsigned signature );
static void  AddArc( DFA machine, int state, char arc_label,
                     StrList state_label, unsigned state_signature );

extern DFA   BuildDFA( StrList words );
extern char *GetTerm( FILE *stream, DFA machine, int size, char *output );

#else

static char *GetMemory();
static void  DestroyTree();
static int   GetState();
static void  AddArc();

extern DFA   BuildDFA();
extern char *GetTerm();

#endif

/*****************************************************************************/
/************************   Private Function Definitions   *******************/

/*FN**************************************************************************

        GetMemory( ptr, num_bytes )

   Returns: char * -- new/expanded block of memory

   Purpose: Rationalize memory allocation and handle errors

   Plan:    Part 1: Allocate memory with supplied allocation functions
            Part 2: Handle any errors
            Part 3: Return the allocated block of memory

   Notes:   None.
**/

static char *
GetMemory( ptr, num_bytes )
   char *ptr;      /* in: expanded block; NULL if nonesuch */
   int num_bytes;  /* in: number of bytes to allocate */
   {
   char *memory;   /* temporary for holding results */

        /* Part 1: Allocate memory with supplied allocation functions */
   if ( NULL == ptr )
      memory = malloc( (unsigned)num_bytes );
   else
      memory = realloc( ptr, (unsigned)num_bytes );
```

```
                    /* Part 2: Handle any errors */
   if ( NULL == memory )
      {
      (void)fprintf( stderr, "malloc failure--aborting\n" );
      exit(1);
      }

             /* Part 3: Return the allocated block of memory */
   return( memory );

   } /* GetMemory */

/*FN*************************************************************************

        DestroyTree( tree )

   Returns: void

   Purpose: Destroy a binary search tree created during machine construction

   Plan:    Part 1: Return right away of there is no tree
            Part 2: Deallocate the subtrees
            Part 3: Deallocate the root

   Notes:   None.
**/

static void
DestroyTree( tree )
   SearchTree tree;   /* in: search tree destroyed */
   {
              /* Part 1: Return right away of there is no tree */
   if ( NULL == tree ) return;

                     /* Part 2: Deallocate the subtrees */
   if ( NULL != tree->left )  DestroyTree( tree->left );
   if ( NULL != tree->right ) DestroyTree( tree->right );

                      /* Part 3: Deallocate the root */
   tree->left = tree->right = NULL;
   (void)free( (char *)tree );

   } /* DestroyTree */

/*FN*************************************************************************

        GetState( machine, label, signature )

   Returns: int -- state with the given label

   Purpose: Search a machine and return the state with a given state label

   Plan:    Part 1: Search the tree for the requested state
            Part 2: If not found, add the label to the tree
            Part 3: Return the state number

   Notes:   This machine always returns a state with the given label
            because if the machine does not have a state with the given
            label, then one is created.
**/

static int
GetState( machine, label, signature )
   DFA machine;        /* in: DFA whose state labels are searched;*/
   StrList label;      /* in: state label searched for */
   unsigned signature; /* in: signature of the label requested */
   {
   SearchTree *ptr;       /* pointer to a search tree link field */
   SearchTree new_node;   /* for a newly added search tree node */

            /* Part 1: Search the tree for the requested state */
   ptr = &(machine->tree);
   while ( (NULL != *ptr) && (   (signature != (*ptr)->signature)
                              || !StrListEqual(label,(*ptr)->label)) )
      ptr = (signature <= (*ptr)->signature) ? &(*ptr)->left : &(*ptr)->right;

            /* Part 2: If not found, add the label to the tree */
   if ( NULL == *ptr )
      {
         /* create a new node and fill in its fields */
      new_node = (SearchTree)GetMemory( NULL, sizeof(SearchTreeNode) );
      new_node->signature = signature;
      new_node->label = (StrList)label;
      new_node->state = machine->num_states;
      new_node->left = new_node->right = NULL;
```

```
         /* allocate more states if needed, set up the new state */
      if ( machine->num_states == machine->max_states )
         {
         machine->max_states += TABLE_INCREMENT;
         machine->state_table =
            (StateTable)GetMemory( (char*)(machine->state_table), machine-
>max_states*sizeof(StateTableEntry));
          }
      machine->state_table[machine->num_states].label = (StrList)label;
      machine->num_states++;

         /* hook the new node into the binary search tree */
      *ptr = new_node;
      }
   else
      StrListDestroy( label );

                  /* Part 3: Return the state number */
   return( (*ptr)->state );

   } /* GetState */

/*FN*************************************************************************

        AddArc( machine, state, arc_label, state_label, state_signature )

   Returns: void

   Purpose: Add an arc between two states in a DFA

   Plan:    Part 1: Search for the target state among existing states
            Part 2: Make sure the arc table is big enough
            Part 3: Add the new arc

   Notes:   None.
**/

static void
AddArc( DFA machine, int state, char arc_label,
                     StrList state_label, unsigned state_signature )
//   DFA machine;              /* in/out: machine with an arc added */
//   int state;                /* in: with an out arc added */
//   char arc_label;           /* in: label on the new arc */
//   StrList state_label;      /* in: label on the target state */
//   unsigned state_signature; /* in: label hash signature to speed searching */
   {
   register int target;   /* destination state for the new arc */

        /* Part 1: Search for the target state among existing states */
   StrListSort( state_label );
   target = GetState( machine, state_label, state_signature );

             /* Part 2: Make sure the arc table is big enough */
   if ( machine->num_arcs == machine->max_arcs )
      {
      machine->max_arcs += TABLE_INCREMENT;

        machine->arc_table =
        machine->arc_table = (ArcTable) GetMemory( (char*) machine->arc_table,
         machine->max_arcs * sizeof(ArcTableEntry) );
      }

                       /* Part 3: Add the new arc */
   machine->arc_table[machine->num_arcs].label = arc_label;
   machine->arc_table[machine->num_arcs].target = target;
   machine->num_arcs++;
   machine->state_table[state].num_arcs++;

   } /* AddArc */

/*FN*************************************************************************

        BuildDFA( words )

   Returns: DFA -- newly created finite state machine

   Purpose: Build a DFA to recognize a list of words

   Plan:    Part 1: Allocate space and initialize variables
            Part 2: Make and label the DFA start state
            Part 3: Main loop - build the state and arc tables
            Part 4: Deallocate the binary search tree and the state labels
            Part 5: Reallocate the tables to squish them down
            Part 6: Return the newly constructed DFA

   Notes:   None.
**/
```

```
DFA
BuildDFA( words )
   StrList words;  /* in: that the machine is built to recognize */
   {
   DFA machine;               /* local for easier access to machine */
   register int state;        /* current state's state number */
   char arc_label;            /* for the current arc when adding arcs */
   char *string;              /* element in a set of state labels */
   char ch;                   /* the first character in a new string */
   StrList current_label;     /* set of strings labeling a state */
   StrList target_label;      /* labeling the arc target state */
   unsigned target_signature; /* hashed label for binary search tree */
   register int i;            /* for looping through strings */

             /* Part 1: Allocate space and initialize variables */
   machine = (DFA)GetMemory( NULL, sizeof(DFAStruct) );

   machine->max_states = TABLE_INCREMENT;
   machine->state_table =
      (StateTable)GetMemory(NULL, machine->max_states*sizeof(StateTableEntry));
   machine->num_states = 0;

   machine->max_arcs = TABLE_INCREMENT;
   machine->arc_table =
      (ArcTable)GetMemory( NULL, machine->max_arcs * sizeof(ArcTableEntry) );
   machine->num_arcs = 0;

   machine->tree = NULL;

               /* Part 2: Make and label the DFA start state */
   StrListUnique( words );              /* sort and unique the list */
   machine->state_table[0].label = words;
   machine->num_states = 1;

            /* Part 3: Main loop - build the state and arc tables */
   for ( state = 0; state < machine->num_states; state++ )
      {
              /* The current state has nothing but a label, so */
              /* the first order of business is to set up some */
              /* of its other major fields                     */
      machine->state_table[state].is_final = FALSE;
      machine->state_table[state].arc_offset = machine->num_arcs;
      machine->state_table[state].num_arcs = 0;

             /* Add arcs to the arc table for the current state */
             /* based on the state's derived set.  Also set the */
             /* state's final flag if the empty string is found */
             /* in the suffix list                              */
      current_label = machine->state_table[state].label;
      target_label = StrListCreate();
      target_signature = HASH_START;
      arc_label = EOS;
      for ( i = 0; i < StrListSize(current_label); i++ )
         {
            /* get the next string in the label and lop it */
         string = StrListPeek( current_label, i );
         ch = *string++;

            /* the empty string means mark this state as final */
         if ( EOS == ch )
            { machine->state_table[state].is_final = TRUE; continue; }

            /* make sure we have a legitimate arc_label */
         if ( EOS == arc_label ) arc_label = ch;

            /* if the first character is new, then we must */
            /* add an arc for the previous first character */
         if ( ch != arc_label )
            {
            AddArc(machine, state, arc_label, target_label, target_signature);
            target_label = StrListCreate();
            target_signature = HASH_START;
            arc_label = ch;
            }

            /* add the current suffix to the target state label */
         StrListAppend( target_label, string );
         target_signature += (*string + 1) * HASH_INCREMENT;
         while ( *string ) target_signature += *string++;
         }

              /* On loop exit we have not added an arc for the */
              /* last bunch of suffixes, so we must do so, as  */
              /* long as the last set of suffixes is not empty */
              /* (which happens when the current state label   */
              /* is the singleton set of the empty string).    */
```

```
      if ( 0 < StrListSize(target_label) )
         AddArc( machine, state, arc_label, target_label, target_signature );
      }

      /* Part 4: Deallocate the binary search tree and the state labels */
   DestroyTree( machine->tree );  machine->tree = NULL;
   for ( i = 0; i < machine->num_states; i++ )
      {
      StrListDestroy( machine->state_table[i].label );
      machine->state_table[i].label = NULL;
      }

            /* Part 5: Reallocate the tables to squish them down */
   machine->state_table = (StateTable)GetMemory( (char *) machine->state_table,
                              machine->num_states * sizeof(StateTableEntry) );
   machine->arc_table = (ArcTable)GetMemory( (char *) machine->arc_table,
                              (machine->num_arcs * sizeof(ArcTableEntry)) );

               /* Part 6: Return the newly constructed DFA */
   return( machine );

   } /* BuildDFA */

/*FN*************************************************************************

        GetTerm( stream, machine, size, output )

   Returns: char * -- NULL if stream is exhausted, otherwise output buffer

   Purpose: Get the next token from an input stream, filtering stop words

   Plan:    Part 1: Return NULL immediately if there is no input
            Part 2: Initialize the local variables
            Part 3: Main Loop: Put an unfiltered word into the output buffer
            Part 4: Return the output buffer

   Notes:   This routine runs the DFA provided as the machine parameter,
            and collects the text of any term in the output buffer.  If
            a stop word is recognized in this process, it is skipped.
            Care is also taken to be sure not to overrun the output buffer.
**/

char *
GetTerm( stream, machine, size, output )
   FILE *stream;    /* in: source of input characters */
   DFA machine;     /* in: finite state machine driving process */
   int size;        /* in: bytes in the output buffer */
   char *output;    /* in/out: where the next token in placed */
   {
   char *outptr;        /* for scanning through the output buffer */
   int ch;              /* current character during input scan */
   register int state;  /* current state during DFA execution */

          /* Part 1: Return NULL immediately if there is no input */
   if ( EOF == (ch = getc(stream)) ) return( NULL );

                /* Part 2: Initialize the local variables */
   outptr = output;

     /* Part 3: Main Loop: Put an unfiltered word into the output buffer */
   do
      {
         /* scan past any leading delimiters */
      while ( (EOF != ch ) &&
              ((DELIM_CH == char_class[ch]) ||
               (DIGIT_CH == char_class[ch])) ) ch = getc( stream );

         /* start the machine in its start state */
      state = 0;

         /* copy input to output until reaching a delimiter, and also */
         /* run the DFA on the input to watch for filtered words      */
      while ( (EOF != ch) && (DELIM_CH != char_class[ch]) )
         {
         if ( outptr == (output+size-1) ) { outptr = output; state = 0; }
         *outptr++ = convert_case[ch];

         if ( DEAD_STATE != state )
            {
            register int i;   /* for scanning through arc labels */
            int arc_start;    /* where the arc label list starts */
            int arc_end;      /* where the arc label list ends */

            arc_start = machine->state_table[state].arc_offset;
            arc_end = arc_start + machine->state_table[state].num_arcs;

            for ( i = arc_start; i < arc_end; i++ )
```

```
            if ( convert_case[ch] == machine->arc_table[i].label )
               { state = machine->arc_table[i].target; break; }

         if ( i == arc_end ) state = DEAD_STATE;
         }

      ch = getc( stream );
      }

      /* start from scratch if a stop word is recognized */
   if ( (DEAD_STATE != state) && machine->state_table[state].is_final )
      outptr = output;

      /* terminate the output buffer */
   *outptr = EOS;
   }
while ( (EOF != ch) && !*output );

                /* Part 4: Return the output buffer */
return( output );

} /* GetTerm */
```

```
/*******************************   strlist.c   *********************************

    Purpose: String list abstract data type implementation.

    Notes:   This module implements a straightforward string ordered list
             abstract data type.  It is optimized for appending and deleting
             from the end of the list.  Since they are ordered lists, string
             lists may be sorted, and their members are addressed by ordinal
             position (starting from 0).
**/

#include <stdio.h>
#include <malloc.h>
#include <string.h>

#include "strlist.h"

/*****************************************************************************/
/******************   Private Defines and Data Structures   ******************/

#define FALSE                          0
#define TRUE                           1
#define EOS                           '\0'

#define INCREMENT                     32    /* increase size by this much */
#define MAX_LINE                     128    /* when reading text files */

typedef struct _StrListStruct {

               short size;             /* current length of the list */
               short max_size;         /* room for this many strings */
               char **string;          /* the string array */

               } StrListStruct;

            /*********   GetMemory and FreeMemory Macros   ********/

#define GetMemory(b,s)                ( (b) ? realloc(b,s) : malloc(s) )
#define FreeMemory(b)                 ( (void)free( b ) )

/*****************************************************************************/
/**********************   Private Routine Declarations   **********************/

#ifdef __STDC__

static int  ExpandArray( StrList list );
static void ISort( char **string, int lb, int ub );
static void QSort( char **string, int lb, int ub );

#else

static int  ExpandArray( /* list */ );
static void ISort( /* string, lb, ub */ );
static void QSort( /* string, lb, ub */ );

#endif

/*FN*************************************************************************

         ExpandArray( list )

   Returns: int -- TRUE (1) on success, FALSE (0) otherwise

   Purpose: Increase the string array to hold more data

   Plan:    Part 1: Increase the maximum list size to its new value
            Part 2: Allocate a new chunk of memory
            Part 3: Return an indication of success

   Notes:   None
**/

static int
ExpandArray( list )
   StrList list;   /* in: string list whose string array is enlarged */
   {

         /* Part 1: Increase the maximum list size to its new value */
   list->max_size += INCREMENT;

                 /* Part 2: Allocate a new chunk of memory */
   list->string = (char **)GetMemory( (char *)list->string,
                                      (list->max_size*sizeof(char *)) );

               /* Part 3: Return an indication of success */
   return( (list->string) ? TRUE : FALSE );
```

```
   } /* ExpandArray */

/*FN*************************************************************************

         ISort( string, lb, ub )

   Returns: void

   Purpose: Insertion sort a string array forward using strcmp ordering

   Plan:    Part 1: Put smallest in place as a sentinal
            Part 2: Insert as necessary

   Notes:   None
**/

static void
ISort( string, lb, ub )
   char **string;   /* in/out: string array sorted */
   int lb,ub;       /* in: array bounds for sort */
   {
   register int i,j;  /* for scanning through the list */
   char *tmp;         /* for swaps */

               /* Part 1: Put smallest in place as a sentinal */
   for ( j = lb, i = lb+1; i <= ub; i++ )
      if ( 0 < strcmp(string[j],string[i]) ) j = i;
   tmp = string[lb]; string[lb] = string[j]; string[j] = tmp;

                      /* Part 2: Insert as necessary */
   for ( i = lb+2; i <= ub; i++ )
      {
      tmp = string[i];
      for ( j = i; 0 < strcmp(string[j-1],tmp); j-- ) string[j] = string[j-1];
      string[j] = tmp;
      }

   } /* ISort*/

/*FN*************************************************************************

         QSort( string, lb, ub )

   Returns: void

   Purpose: Quicksort an array of strings forward using strcmp ordering

   Plan:    Part 1: Use insertion sort of the list is short
            Part 2: Do median of three pivot value selection
            Part 3: Put the pivot out of the way at the top
            Part 5: Swap the pivot back into the mid of the list
            Part 6: Recursively sort the sublists

   Notes:   Standard quicksort function with the two main enhancements:
            median of three partitioning to find a good pivot value,
            and sorting small arrays with insertion sort.
**/

static void
QSort(char **string, int lb, int ub )
   //char **string;  /* in/out: string array sorted */
   //int lb,ub;      /* in: array bounds for sort */
   {
   register int lft;  /* list pointer that closes from the left */
   register int rgt;  /* list pointer that closes from the right */
   register int mid;  /* index of the median of three value */
   char *tmp;         /* for string pointer swaps */
   char *pivot;       /* the pivot value string */

              /* Part 1: Use insertion sort of the list is short */
   if ( ub-lb < 12 ) { ISort( string, lb, ub ); return; }

             /* Part 2: Do median of three pivot value selection */
   mid = (lb+ub)/2;
   if ( strcmp(string[mid],string[lb]) < 0 )
      { tmp = string[mid]; string[mid] = string[lb]; string[lb] = tmp; }
   if ( strcmp(string[ub],string[mid]) < 0 )
      { tmp = string[mid]; string[mid] = string[ub]; string[ub] = tmp; }
   if ( strcmp(string[mid],string[lb]) < 0 )
      { tmp = string[mid]; string[mid] = string[lb]; string[lb] = tmp; }

             /* Part 3: Put the pivot out of the way at the top */
   tmp = string[mid]; string[mid] = string[ub-1]; string[ub-1] = tmp;

                /* Part 4: Partition around the pivot value */
   lft = lb;
   rgt = ub-1;
```

```
   pivot = string[ub-1];
   do
      {
      do lft++; while ( strcmp(string[lft],pivot) < 0 );
      do rgt--; while ( strcmp(pivot,string[rgt]) < 0 );
      tmp = string[lft]; string[lft] = string[rgt]; string[rgt] = tmp;
      }
   while ( lft < rgt );

         /* Part 5: Swap the pivot back into the mid of the list */
   string[rgt] = string[lft]; string[lft] = string[ub-1]; string[ub-1] = tmp;

                  /* Part 6: Recursively sort the sublists */
   QSort( string, lb, lft-1 );
   QSort( string, rgt+1, ub );

   } /* QSort */

/*****************************************************************************/
/**********************   Public Routine Declarations   **********************/

/*FN*************************************************************************

         StrListAppend( list, string )

   Returns: void

   Purpose: Place a string on the end of a string list

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Expand the list as necessary
            Part 3: Append the new string to the tail

   Notes:   None
**/

void
StrListAppend( list, string )
   StrList list;   /* in/out: list appended to */
   char *string;   /* in: the appended string */
   {
   int length;  /* of the added string and its terminator */

                /* Part 1: Standard parameter sanity check */
   if ( !list || !string ) return;

                  /* Part 2: Expand the list as necessary */
   if ( (list->size == list->max_size) && !ExpandArray(list) ) return;

                /* Part 3: Append the new string to the tail */
   length = strlen( string ) + 1;
   list->string[list->size] = GetMemory( NULL, length );
   (void)memcpy( list->string[list->size], string, length );
   list->size++;

   } /* StrListAppend */

/*FN*************************************************************************

         StrListAppendFile( list, filename )

   Returns: void

   Purpose: Place all lines from a file on the end of a string list

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Expand the list as necessary
            Part 3: Append the new string to the tail

   Notes:   None
**/

void
StrListAppendFile( list, filename )
   StrList list;     /* in/out: list appended to */
   char *filename;   /* in: the appended file */
   {
   FILE *file;            /* file handle for the text input file */
   char buffer[MAX_LINE]; /* for storing text input file lines */
   int length;            /* of the added string and its terminator */
   register int i;        /* for looping through the TextBlock lines */

                /* Part 1: Standard parameter sanity check */
   if ( !list || !filename ) return;

             /* Part 2: Open the text input file; check for error */
   if ( NULL == (file = fopen(filename,"r")) ) return;
```

```
            /* Part 3: Append to the list, checking for errors */
   while ( NULL != fgets(buffer,MAX_LINE,file) )
      {
      if ( (list->size == list->max_size) && !ExpandArray(list) ) return;

      i = list->size;
      length = strlen( buffer );
      list->string[i] = GetMemory( NULL, (unsigned)length );
      if ( NULL == list->string[i] ) return;
      (void)memcpy( list->string[i], buffer, length );
      list->string[i][length-1] = EOS;
      list->size++;
      }

                  /* Part 4: Close the text input file */
   (void)fclose( file );

   } /* StrListAppendFile */

/*FN*************************************************************************

         StrListCreate()

   Returns: StrList -- a new structure, or NULL on failure

   Purpose: Allocate and initialize a new string list structure

   Plan:    Part 1: Allocate space for the string list object
            Part 2: Initialize the structure fields
            Part 3: Return the new string list

   Notes:   None
**/

StrList
StrListCreate()
   {
   StrList list;  /* the new list returned */

            /* Part 1: Allocate space for the string list object */
   if ( !(list = (StrList)GetMemory(NULL,sizeof(StrListStruct))) )
      return( NULL );

               /* Part 2: Initialize the structure fields */
   list->string = NULL;
   list->size = list->max_size = 0;
   if ( !ExpandArray(list) )
      { FreeMemory( (char *)list ); return( NULL ); }

                  /* Part 3: Return the new string list */
   return( list );

   } /* StrListCreate */

/*FN*************************************************************************

         StrListDestroy( list )

   Returns: void

   Purpose: Deallocate the space used for a string list

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Free all the space

   Notes:   None
**/

void
StrListDestroy( list )
   StrList list;    /* in: the list destroyed */
   {
   register int i;  /* for scanning through the list */

               /* Part 1: Standard parameter sanity check */
   if ( !list ) return;

                        /* Part 2: Free all the space */
   for ( i = 0; i < list->size; i++ ) FreeMemory( (char *)(list->string[i]) );
   FreeMemory( (char *)list );

   } /* StrListDestroy */



/*FN*************************************************************************
```

```
         StrListElementEqual( list1, list2 )

   Returns: int -- TRUE if the term in position "pos" from list1 is found in list2

   Purpose: See if the term in position "pos" from list1 also exists in list2

   Plan:    Part 1: Say not equal if the parameters are bad
            Part 2: Compare the element in the first list with every element in list2
            Part 3: Say false if nothing is found

   Notes:   None
**/

int
StrListElementEqual( pos, list1, list2 )
   StrList list1,list2;   /* in: lists compared */
   int pos;
   {
   register int i;  /* for scanning through the anti-stoplist */
   int success;

             /* Part 1: Say not equal if the parameters are bad */
   if ( !list1 || !list2 ) return( FALSE );

              /* Part 3: Compare the lists element by element */
   success = 0;
   for ( i = 0; i < list2->size; i++ ){
      if ( strcmp(list1->string[pos],list2->string[i]) ==0 )  // compares two strings
         success = 1;
   }
   if (success) return( TRUE );
   else return (FALSE);

   } /* StrListElementEqual */


/*FN*************************************************************************

         StrListCount( pos, list1, list2, start2 )

**/

int
StrListCount( pos, list1, list2, start2 )
   StrList list1,list2;   /* in: lists compared */
   int pos, start2;
   {
   register int i;  /* for scanning through the anti-stoplist */
   int antall;

             /* Part 1: Say not equal if the parameters are bad */
   if ( !list1 || !list2 ) return( FALSE );

              /* Part 3: Compare the lists element by element */
   antall = 0;
   for ( i = 0; i < (list2->size); i++ ){
//   while ( i < list2->size){
      if ( strcmp(list1->string[pos],list2->string[i]) ==0 )  // compares two strings
         antall++;
   }
   return antall;

   } /* StrListCount */

/*FN*************************************************************************

         StrListEqual( list1, list2 )

   Returns: int -- TRUE if the lists are equivalent, FALSE otherwise

   Purpose: See if two lists have identical elements

   Plan:    Part 1: Say not equal if the parameters are bad
            Part 2: Say not equal if sizes are different
            Part 3: Compare lists element by element
            Part 4: Say equal if everything checks out

   Notes:   None
**/

int
StrListEqual( list1, list2 )
   StrList list1,list2;   /* in: lists compared */
   {
   register int i;  /* for scanning through the lists */
```

```
            /* Part 1: Say not equal if the parameters are bad */
   if ( !list1 || !list2 ) return( FALSE );

              /* Part 2: Say not equal if sizes are different */
   if ( list1->size != list2->size ) return( FALSE );

             /* Part 3: Compare the lists element by element */
   for ( i = 0; i < list1->size; i++ )
      if ( *(list1->string[i]) != *(list2->string[i]) )
         return( FALSE );
      else if ( 0 != strcmp(list1->string[i],list2->string[i]) )
         return( FALSE );

              /* Part 5: Say equal if everything checks out */
   return( TRUE );

   } /* StrListEqual */


/*FN*************************************************************************
SAH july 2002: Check if term is included in list. Return false otherwise
**/

int
StrListMember(char term[], StrList list){
register int i;

  for ( i = 0; i < list->size; i++ ) {
    if ( 0 == strcmp(term,list->string[i]) )  //strcmp returns zero if strings are identical
       return TRUE;  //found
  }
  return FALSE; //if word not found


} //StrListMember

/*FN*************************************************************************

         StrListPeek( list, index )

   Returns: char * -- pointer to the requested string; NULL on error

   Purpose: Peek a string by its list index. (Get an element from the list)

   Plan:    Part 1: Standard parameter sanity check
            Part 2: Return the requested string

   Notes:   Note that this function is a hole in the data type encapsulation:
            it should return a copy, but this would force the consumer to
            deallocate the string.  Design call.
**/

char *
StrListPeek( list, index )
   StrList list;   /* in: list retrieved from */
   int index;      /* in: which string to fetch */
   {
                /* Part 1: Standard parameter sanity check */
   if ( !list || (index < 0) || (list->size <= index) ) return( NULL );

                  /* Part 2: Return the requested string */
   return( list->string[index] );

   } /* StrListPeek */

/*FN*************************************************************************

         StrListSize( list )

   Returns: int -- the size of the list, 0 on error

   Purpose: Grab the list size

   Plan:    Return the list size field

   Notes:   None
**/

int
StrListSize( list )
   StrList list;   /* in: list queried */
   {

   if ( !list ) return( 0 ); else return( list->size );

   } /* StrListSize */
```

```
/*FN*************************************************************************

         StrListSort( list )

   Returns: void

   Purpose: Sort a single string list using strcmp ordering

   Plan:    Part 1: Do parameter sanity checks, then sort

   Notes:   None
**/

void
StrListSort( list )
   StrList list;   /* in/out: list sorted */
   {
            /* Part 1: Do parameter sanity checks, then sort */
   if ( !list ) return;
   QSort( list->string, 0, list->size-1 );

   } /* StrListSort */

/*FN*************************************************************************

         StrListUnique( list )

   Returns: void

   Purpose: Sort a single string list using strcmp ordering, then remove
            duplicates.

   Plan:    Part 1: Do parameters sanity checks
            Part 2: Sort the list
            Part 3: Remove duplicate strings

   Notes:   None
**/

void
StrListUnique( list )
   StrList list;   /* in/out: list sorted and uniqued */
   {
   register i,j;   /* counters for copying down over duplicates */

                    /* Part 1: Do parameter sanity checks */
   if ( !list ) return;

                        /* Part 2: Sort the list */
   QSort( list->string, 0, list->size-1 );

                   /* Part 3: Remove duplicate strings */
   if ( 1 < list->size )
      {
      for ( j = 0, i = 1; i < list->size; i++ )
         {
         if ( 0 == strcmp(list->string[i],list->string[j]) )
            (void)free( list->string[j] );
         else
            j++;
         if ( j < i ) list->string[j] = list->string[i];
         }
      list->size = j + 1;
      }

   } /* StrListUnique */
```

```
/*******************************   stem.c   **********************************

   Purpose:    Implementation of the Porter stemming algorithm documented
               in: Porter, M.F., "An Algorithm For Suffix Stripping,"
               Program 14 (3), July 1980, pp. 130-137.

   Provenance: Written by B. Frakes and C. Cox, 1986.
               Changed by C. Fox, 1990.
                  - made measure function a DFA
                  - restructured structs
                  - renamed functions and variables
                  - restricted function and variable scopes
               Changed by C. Fox, July, 1991.
                  - added ANSI C declarations
                  - branch tested to 90% coverage

   Notes:      This code will make little sense without the the Porter
               article.  The stemming function converts its input to
               lower case.
**/

/************************   Standard Include Files   *************************/

#include <stdio.h>
#include <string.h>
#include <ctype.h>

/*****************************************************************************/
/******************   Private Defines and Data Structures   ******************/

#define FALSE                          0
#define TRUE                           1
#define EOS                           '\0'

#define IsVowel(c)        ('a'==(c)||'e'==(c)||'i'==(c)||'o'==(c)||'u'==(c))

typedef struct {
           int id;                 /* returned if rule fired */
           char *old_end;          /* suffix replaced */
           char *new_end;          /* suffix replacement */
           int old_offset;         /* from end of word to start of suffix */
           int new_offset;         /* from beginning to end of new suffix */
           int min_root_size;      /* min root word size for replacement */
           int (*condition)();     /* the replacement test function */
           } RuleList;

static char LAMBDA[1] = "";        /* the constant empty string */
static char *end;                  /* pointer to the end of the word */

/*****************************************************************************/
/********************   Private Function Declarations   **********************/

#ifdef __STDC__

static int WordSize( char *word );
static int ContainsVowel( char *word );
static int EndsWithCVC( char *word );
static int AddAnE( char *word );
static int RemoveAnE( char *word );
static int ReplaceEnd( char *word, RuleList *rule );
//her var det ikke samsvar mellom prototype og funksjonens typer.
//Stod slik før (har nå modifisert til at pekeren er med) :
//static int ReplaceEnd( char *word, RuleList rule );  //rule skal være *rule

#else

static int WordSize( /* word */ );
static int ContainsVowel( /* word */ );
static int EndsWithCVC( /* word */ );
static int AddAnE( /* word */ );
static int RemoveAnE( /* word */ );
static int ReplaceEnd( /* word, rule */ );

#endif

/*****************************************************************************/
/******************   Initialized Private Data Structures   ******************/

static RuleList step1a_rules[] =
           {
             101,  "sses",      "ss",    3,  1, -1,  NULL,
             102,  "ies",       "i",     2,  0, -1,  NULL,
             103,  "ss",        "ss",    1,  1, -1,  NULL,
             104,  "s",         LAMBDA,  0, -1, -1,  NULL,
             000,  NULL,        NULL,    0,  0,  0,  NULL,
           };
```

```
static RuleList step1b_rules[] =
          {
            105,  "eed",        "ee",     2,  1,  0,  NULL,
            106,  "ed",         LAMBDA,   1, -1, -1,  ContainsVowel,
            107,  "ing",        LAMBDA,   2, -1, -1,  ContainsVowel,
            000,  NULL,         NULL,     0,  0,  0,  NULL,
          };

static RuleList step1b1_rules[] =
          {
            108,  "at",         "ate",    1,  2, -1,  NULL,
            109,  "bl",         "ble",    1,  2, -1,  NULL,
            110,  "iz",         "ize",    1,  2, -1,  NULL,
            111,  "bb",         "b",      1,  0, -1,  NULL,
            112,  "dd",         "d",      1,  0, -1,  NULL,
            113,  "ff",         "f",      1,  0, -1,  NULL,
            114,  "gg",         "g",      1,  0, -1,  NULL,
            115,  "mm",         "m",      1,  0, -1,  NULL,
            116,  "nn",         "n",      1,  0, -1,  NULL,
            117,  "pp",         "p",      1,  0, -1,  NULL,
            118,  "rr",         "r",      1,  0, -1,  NULL,
            119,  "tt",         "t",      1,  0, -1,  NULL,
            120,  "ww",         "w",      1,  0, -1,  NULL,
            121,  "xx",         "x",      1,  0, -1,  NULL,
            122,  LAMBDA,       "e",     -1,  0, -1,  AddAnE,
            000,  NULL,         NULL,     0,  0,  0,  NULL,
            };

static RuleList step1c_rules[] =
          {
            123,  "y",          "i",       0,  0, -1,  ContainsVowel,
            000,  NULL,         NULL,     0,  0,  0,  NULL,
          };

static RuleList step2_rules[] =
          {
            203,  "ational",    "ate",    6,  2,  0,  NULL,
            204,  "tional",     "tion",   5,  3,  0,  NULL,
            205,  "enci",       "ence",   3,  3,  0,  NULL,
            206,  "anci",       "ance",   3,  3,  0,  NULL,
            207,  "izer",       "ize",    3,  2,  0,  NULL,
            208,  "abli",       "able",   3,  3,  0,  NULL,
            209,  "alli",       "al",     3,  1,  0,  NULL,
            210,  "entli",      "ent",    4,  2,  0,  NULL,
            211,  "eli",        "e",      2,  0,  0,  NULL,
            213,  "ousli",      "ous",    4,  2,  0,  NULL,
            214,  "ization",    "ize",    6,  2,  0,  NULL,
            215,  "ation",      "ate",    4,  2,  0,  NULL,
            216,  "ator",       "ate",    3,  2,  0,  NULL,
            217,  "alism",      "al",     4,  1,  0,  NULL,
            218,  "iveness",    "ive",    6,  2,  0,  NULL,
            219,  "fulnes",     "ful",    5,  2,  0,  NULL,
            220,  "ousness",    "ous",    6,  2,  0,  NULL,
            221,  "aliti",      "al",     4,  1,  0,  NULL,
            222,  "iviti",      "ive",    4,  2,  0,  NULL,
            223,  "biliti",     "ble",    5,  2,  0,  NULL,
            000,  NULL,         NULL,     0,  0,  0,  NULL,
          };

static RuleList step3_rules[] =
          {
            301,  "icate",      "ic",     4,  1,  0,  NULL,
            302,  "ative",      LAMBDA,   4, -1,  0,  NULL,
            303,  "alize",      "al",     4,  1,  0,  NULL,
            304,  "iciti",      "ic",     4,  1,  0,  NULL,
            305,  "ical",       "ic",     3,  1,  0,  NULL,
            308,  "ful",        LAMBDA,   2, -1,  0,  NULL,
            309,  "ness",       LAMBDA,   3, -1,  0,  NULL,
            000,  NULL,         NULL,     0,  0,  0,  NULL,
          };

static RuleList step4_rules[] =
          {
            401,  "al",         LAMBDA,   1, -1,  1,  NULL,
            402,  "ance",       LAMBDA,   3, -1,  1,  NULL,
            403,  "ence",       LAMBDA,   3, -1,  1,  NULL,
            405,  "er",         LAMBDA,   1, -1,  1,  NULL,
            406,  "ic",         LAMBDA,   1, -1,  1,  NULL,
            407,  "able",       LAMBDA,   3, -1,  1,  NULL,
            408,  "ible",       LAMBDA,   3, -1,  1,  NULL,
            409,  "ant",        LAMBDA,   2, -1,  1,  NULL,
            410,  "ement",      LAMBDA,   4, -1,  1,  NULL,
            411,  "ment",       LAMBDA,   3, -1,  1,  NULL,
            412,  "ent",        LAMBDA,   2, -1,  1,  NULL,
            423,  "sion",       "s",      3,  0,  1,  NULL,
            424,  "tion",       "t",      3,  0,  1,  NULL,
            415,  "ou",         LAMBDA,   1, -1,  1,  NULL,
```

```
            416,  "ism",        LAMBDA,  2, -1,  1,  NULL,
            417,  "ate",        LAMBDA,  2, -1,  1,  NULL,
            418,  "iti",        LAMBDA,  2, -1,  1,  NULL,
            419,  "ous",        LAMBDA,  2, -1,  1,  NULL,
            420,  "ive",        LAMBDA,  2, -1,  1,  NULL,
            421,  "ize",        LAMBDA,  2, -1,  1,  NULL,
            000,  NULL,         NULL,    0,  0,  0,  NULL,
          };

static RuleList step5a_rules[] =
          {
            501,  "e",          LAMBDA,  0, -1,  1,  NULL,
            502,  "e",          LAMBDA,  0, -1, -1,  RemoveAnE,
            000,  NULL,         NULL,    0,  0,  0,  NULL,
          };

static RuleList step5b_rules[] =
          {
            503,  "ll",         "l",     1,  0,  1,  NULL,
            000,  NULL,         NULL,    0,  0,  0,  NULL,
          };

/*****************************************************************************/
/*******************   Private Function Declarations   **********************/

/*FN*************************************************************************

       WordSize( word )

   Returns: int -- a weird count of word size in adjusted syllables

   Purpose: Count syllables in a special way:  count the number
            vowel-consonant pairs in a word, disregarding initial
            consonants and final vowels.  The letter "y" counts as a
            consonant at the beginning of a word and when it has a vowel
            in front of it; otherwise (when it follows a consonant) it
            is treated as a vowel.  For example, the WordSize of "cat"
            is 1, of "any" is 1, of "amount" is 2, of "anything" is 3.

   Plan:    Run a DFA to compute the word size

   Notes:   The easiest and fastest way to compute this funny measure is
            with a finite state machine.  The initial state 0 checks
            the first letter.  If it is a vowel, then the machine changes
            to state 1, which is the "last letter was a vowel" state.
            If the first letter is a consonant or y, then it changes
            to state 2, the "last letter was a consonant state".  In
            state 1, a y is treated as a consonant (since it follows
            a vowel), but in state 2, y is treated as a vowel (since
            it follows a consonant.  The result counter is incremented
            on the transition from state 1 to state 2, since this
            transition only occurs after a vowel-consonant pair, which
            is what we are counting.
**/

static int
WordSize( word )
   char *word;   /* in: word having its WordSize taken */
   {
   register int result;   /* WordSize of the word */
   register int state;    /* current state in machine */

   result = 0;
   state = 0;

                /* Run a DFA to compute the word size */
   while ( EOS != *word )
      {
      switch ( state )
         {
         case 0: state = (IsVowel(*word)) ? 1 : 2;
                 break;
         case 1: state = (IsVowel(*word)) ? 1 : 2;
                 if ( 2 == state ) result++;
                 break;
         case 2: state = (IsVowel(*word) || ('y' == *word)) ? 1 : 2;
                 break;
         }
      word++;
      }

   return( result );

   } /* WordSize */

/*FN*************************************************************************
```

```
       ContainsVowel( word )

   Returns: int -- TRUE (1) if the word parameter contains a vowel,
            FALSE (0) otherwise.

   Purpose: Some of the rewrite rules apply only to a root containing
            a vowel, where a vowel is one of "aeiou" or y with a
            consonant in front of it.

   Plan:    Obviously, under the definition of a vowel, a word contains
            a vowel iff either its first letter is one of "aeiou", or
            any of its other letters are "aeiouy".  The plan is to
            test this condition.

   Notes:   None
**/

static int
ContainsVowel( word )
   char *word;   /* in: buffer with word checked */
   {

   if ( EOS == *word )
      return( FALSE );
   else
      return( IsVowel(*word) || (NULL != strpbrk(word+1,"aeiouy")) );


   } /* ContainsVowel */

/*FN*************************************************************************

       EndsWithCVC( word )

   Returns: int -- TRUE (1) if the current word ends with a
            consonant-vowel-consonant combination, and the second
            consonant is not w, x, or y, FALSE (0) otherwise.

   Purpose: Some of the rewrite rules apply only to a root with
            this characteristic.

   Plan:    Look at the last three characters.

   Notes:   None
**/

static int
EndsWithCVC( word )
   char *word;   /* in: buffer with the word checked */
   {
   int length;         /* for finding the last three characters */

   if ( (length = strlen(word)) < 2 )
      return( FALSE );
   else
      {
      end = word + length - 1;
      return(    (NULL == strchr("aeiouwxy",*end--))      /* consonant */
              && (NULL != strchr("aeiouy",  *end--))        /* vowel */
              && (NULL == strchr("aeiou",   *end  )) );   /* consonant */
      }

   } /* EndsWithCVC */

/*FN*************************************************************************

       AddAnE( word )

   Returns: int -- TRUE (1) if the current word meets special conditions
            for adding an e.

   Purpose: Rule 122 applies only to a root with this characteristic.

   Plan:    Check for size of 1 and a consonant-vowel-consonant ending.

   Notes:   None
**/

static int
AddAnE( word )
   char *word;
   {

   return( (1 == WordSize(word)) && EndsWithCVC(word) );

   } /* AddAnE */
```

```
/*FN*************************************************************************

       RemoveAnE( word )

   Returns: int -- TRUE (1) if the current word meets special conditions
            for removing an e.

   Purpose: Rule 502 applies only to a root with this characteristic.

   Plan:    Check for size of 1 and no consonant-vowel-consonant ending.

   Notes:   None
**/

static int
RemoveAnE( word )
   char *word;
   {

   return( (1 == WordSize(word)) && !EndsWithCVC(word) );

   } /* RemoveAnE */

/*FN*************************************************************************

       ReplaceEnd( word, rule )

   Returns: int -- the id for the rule fired, 0 is none is fired

   Purpose: Apply a set of rules to replace the suffix of a word

   Plan:    Loop through the rule set until a match meeting all conditions
            is found.  If a rule fires, return its id, otherwise return 0.
            Connditions on the length of the root are checked as part of this
            function's processing because this check is so often made.

   Notes:   This is the main routine driving the stemmer.  It goes through
            a set of suffix replacement rules looking for a match on the
            current suffix.  When it finds one, if the root of the word
            is long enough, and it meets whatever other conditions are
            required, then the suffix is replaced, and the function returns.
**/

static int
ReplaceEnd( word, rule )
   char *word;        /* in/out: buffer with the stemmed word */
   RuleList *rule;    /* in: data structure with replacement rules
     rule er en peker til en RuleList struct... */
   {
   register char *ending;   /* set to start of possible stemmed suffix */
   char tmp_ch;             /* save replaced character when testing */

   while ( 0 != rule->id )
      {
      ending = end - rule->old_offset;
      if ( word <= ending )
         if ( 0 == strcmp(ending,rule->old_end) )
            {
            tmp_ch = *ending;
            *ending = EOS;
            if ( rule->min_root_size < WordSize(word) )
               if ( !rule->condition || (*rule->condition)(word) )
                  {
                  (void)strcat( word, rule->new_end );
                  end = ending + rule->new_offset;
                  break;
                  }
            *ending = tmp_ch;
            }
      rule++;
      }

   return( rule->id );

   } /* ReplaceEnd */
```

```
/*****************************************************************************/
/*********************   Public Function Declarations   **********************/

/*FN*************************************************************************

       Stem( word )

   Returns: int -- FALSE (0) if the word contains non-alphabetic characters
            and hence is not stemmed, TRUE (1) otherwise

   Purpose: Stem a word

   Plan:    Part 1: Check to ensure the word is all alphabetic
            Part 2: Run through the Porter algorithm
            Part 3: Return an indication of successful stemming

   Notes:   This function implements the Porter stemming algorithm, with
            a few additions here and there.  See:

               Porter, M.F., "An Algorithm For Suffix Stripping,"
               Program 14 (3), July 1980, pp. 130-137.

            Porter's algorithm is an ad hoc set of rewrite rules with
            various conditions on rule firing.  The terminology of
            "step 1a" and so on, is taken directly from Porter's
            article, which unfortunately gives almost no justification
            for the various steps.  Thus this function more or less
            faithfully refects the opaque presentation in the article.
            Changes from the article amount to a few additions to the
            rewrite rules;  these are marked in the RuleList data
            structures with comments.
**/

int
Stem( word )
   char *word;  /* in/out: the word stemmed */
   {
   int rule;    /* which rule is fired in replacing an end */

            /* Part 1: Check to ensure the word is all alphabetic */
   for ( end = word; *end != EOS; end++ )
      if ( !isalpha(*end) ) return( FALSE );
      else *end = tolower( *end );
   end--;

               /*  Part 2: Run through the Porter algorithm */
   (void)ReplaceEnd( word, step1a_rules );  //dette er vel pekeren til arrayet?
   rule = ReplaceEnd( word, step1b_rules );
   if ( (106 == rule) || (107 == rule) )
      (void)ReplaceEnd( word, step1b1_rules );
   (void)ReplaceEnd( word, step1c_rules );

   (void)ReplaceEnd( word, step2_rules );

   (void)ReplaceEnd( word, step3_rules );

   (void)ReplaceEnd( word, step4_rules );

   (void)ReplaceEnd( word, step5a_rules );
   (void)ReplaceEnd( word, step5b_rules );

   /* Prøver med dette å sende inn verdiene til pekerne. Det blir selvsagt feil...
      Riktig slik det var med å sende inn pekeren. Må rette opp i funksjonen ReplaceEnd

      Før stod det (void)ReplaceEnd( word, step5b_rules );
      Dette har jeg endret til  (void)ReplaceEnd( word, *step5b_rules );
      for alle reglene. Funksjonen forventer nemlig en peker inn, mens
      det som opprinnelig stod i koden var et vanlig arraynavn. */

           /* Part 3: Return an indication of successful stemming */
   return( TRUE );

   } /* Stem */
```