```
/*+1============================================================================*/
/* MODULE                              AC.C                                     */
/*============================================================================*/
/* FUNCTION    Aho & Corasic algorithm for searching a text for several fixed
 *             substrings. This module implements a simple "Finite State
 *             Automaton" (FSA) to locate all occurences of any of a number
 *             of keywords in a string of text. The algorithm :
 *             - constructs a Deterministic Finite State Automaton (DFSA)
 *               for pattern matching from the keywords; Construction of the
 *               DFSA takes time proportional to the sum of the lengths of
 *               the keywords, ie. time complexity O(n).
 *             - then uses the DFSA to process the text in one pass;
 *               The number of state transitions made by the DFSA in pro-
 *               cessing the text is independent of the number of keywords.
 *
 * SYSTEM      Standard Ansi C.
 *             Tested on UNIX V.3 and PC/MS DOS V.3.3.
 *
 * SEE ALSO    Modules : general.h, bool.h/c, error.h/c
 *
 * PROGRAMMER  Allan Dystrup
 *
 * COPYRIGHT   (c) Allan Dystrup, SEP. 1991
 *
 * VERSION     $Header:$
 *             $Log:$
 *
 * REFERENCES  Algorithm : "Efficient String Matching: An Aid to Bibliographic
 *             Search", Aho & Corasick, CACM, Vol. 18 No. 6 (June '75)
 *             Implementation : Inspired by the UNIX "fgrep" utility as
 *             designed by Ian Ashdown, byHeart Software.
 *
 * USAGE       ac [<options>] <boolexpr> <file>, where :
 *                <options>  -N to force a NFSA search (default DFSA)
 *                           -U to force a case-INsensitive search
 *                <boolexpr> is a boolean expression of search phrases
 *                           example: This&^(That/Those)
 *                           (cf. module: bool.c)
 *                <file>     is the file to search for boolexpr, line by line.
 *
 * BUGS        A simple-minded test-driver :
 *             - Text is searched on a line-by-line basis
 *             - Lines are limited to 256 characters.
 *
 *
 *
 *======================== BIBLIOGRAPHIC SEARCH ==============================
 *
 * Searching for a simple pattern such as a single word or phrase in a
 * set of data is adequately solved by algorithms such as "Boyer-Moore" or
 * "Knuth-Morris-Pratt" (KMP). These algorithms however are not appropriate
 * in a search for more than one pattern at a time.
 *
 * Bibliographic search often requires locating a finite set of keywords
 * in an arbitrary string of text. The keywords can be represented by a
 * restricted class of regular expressions (excluding character clases,
 * closure et. al.). From these simple regular expressions we can construct
 * an efficient finite state pattern matching machine, which when applied
 * to an input text will signal whenever it finds a match for a keyword.
 * The search is performed for all keywords "in parallel" (ie. without back-
 * tracking), and the execution speed is thus independent of the number of
 * patterns to be matched. The algorithm also identifies overlapping strings.
 *
 * The pattern matching machine may be combined with a simple stack machine
 * to evaluate a search criterion expressed as a Boolean function of keywords
 * and phrases. (This extension of the basic search algorithm is implemented
 * by the module bool.c, which is called from this module).
 *
```

```
*====================== PATTERN MATCHING MACHINE ==========================
*
* Let K = {k1, k2, ... ,kn} be the finite set of keywords. By a pattern
* matching machine for K we mean a program which given the input text
* string X produces as output the locations in X at which keywords in K
* appear as substrings.
*
* The pattern matching machine is constructed as a set of states represen-
* ting matched characters in K. The machine successively reads the symbols
* in X, making state transitions and emitting output whenever a keyword is
* located.
*
* A pattern matching machine also called a "Finite State Automaton" (FSA)
* may be of type nondeterministic or deterministic.
* A "Nondeterministic FSA" (NFSA) uses intermediate failure transitions
* between keyword states and may thus require more than one state transition
* per input symbol.
* A "Deterministic FSA" (DFSA) encodes all possible combinations of keyword
* characters, but thereby eliminates failure transitions making only one
* transition per input symbol.
* The desicion to use a NFSA or a DFSA is a classic choice of memory usage
* (minimized by NFSA) vs. execution speed (minimized by DFSA). This imple-
* mentation gives you both choices.
*
```

```
*============================== NFSA ===================================
 *
 * The behavior of the NFSA is described by 3 functions :
 *  - a goto function G, mapping a pair of (state s, inputSymbol x) into
 *    a state s' or into the message "fail".
 *  - a failure function F, mapping a state s into a state s'.
 *    F is consulted whenever G reports "fail".
 *  - an output function O, associated with output states, ie. terminal
 *    states indicating match of a keyword (for all other states, O is empty).
 *
 * The operating cycle of the NFSA is defined by :
 *  1 if transition G(s,x) == s' then
 *       enter state s' and advance input
 *       if O(s') != empty then emit O(s')
 *  2 if transition G(s,x) == "fail" then
 *       s' = F(s)     // repeat cycle with G(s',x), ie. don't advance input
 *
 * The NFSA algorithm is thus :
 *    BEGIN
 *      s = 0                  // initialize current state to start state
 *      FOR i = 1 UNTIL m DO   // proces input text string X = x1x1 ... xm
 *      BEGIN                  // next operating cycle (each new input char)
 *        WHILE (G(s, xi) == fail)
 *           s = F(s)
 *        s = G(s, xi)
 *        IF O(s) != empty THEN
 *          PRINT (i, O(s))    // located keyword O(s) at position i in X
 *      END
 *    END
 *
 *============================== DFSA ===================================
 *
 * The behavior of the DFSA is described by 2 functions :
 *  - a next move function M, mapping a pair of (state s, inputSymbol x) into
 *    a state s'. M replaces the two NFSA functions : G and F.
 *  - an output function O, identical to the NFSA O-function.
 *
 * The operating cycle of the DFSA is characterized by exactly one transition
 * per input character (ie. failure transitions of the NFSA are eliminated).
 *
 * The DFSA algorithm is :
 *    BEGIN
 *      s = 0                  // initialize current state to start state
 *      FOR i = 1 UNTIL m DO   // proces input text string X = x1x1 ... xm
 *      BEGIN                  // next operating cycle (each new input char)
 *        s = M(s, xi)         // failure transitions eliminated!
 *        IF O(s) != empty THEN
 *          PRINT (i, O(s))    // located keyword O(s) at position i in X
 *      END
 *    END
 *
 * DIAGNOSTICS Exit status is 0 if any matches are found, 1 if none, 2 for
 *             error condition.
 *
 *-1==========================================================================*/
```

```
/*====================================================================*/
/*                        Includes                                    */
/*====================================================================*/
#include <string.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <limits.h>

#include "general.h"
#include "bool.h"
#include "error.h"
#define  _AC_ALLOC
#include "ac.h"

/*====================================================================*/
/*                        Definitions                                 */
/*====================================================================*/
#define NFSA        1
#define DFSA        2


/*====================================================================*/
/*                        Typedefs                                    */
/*====================================================================*/

typedef struct transition {    /* FSA transition element -----------  */
    BYTE   cTrans;             /* Transition symbol                   */
    struct state *psState;     /* Ptr to transition state             */
    struct transition *psTrans;/* Ptr to next transition in list      */
} sTRANS;

typedef struct state {          /* FSA state element ----------------- */
    sTRANS   *psGoList;        /* Ptr to head of "go" list for NFSA   */
    sTRANS   *psMvList;        /* Ptr to head of "move" list for DFSA */
    struct state *psFailSt;    /* Ptr to failure state for NFSA       */
    int      *index;            /* Ptr to list of keyword indices      */
} sSTATE;

typedef struct QElement {      /* Queue element --------------------- */
    struct state *psState;     /* Ptr to state put in queue           */
    struct QElement *psQNext;  /* Ptr to next queue element           */
} sQELEM;
```

```
/*=========================================================================*/
/*                        Global Variables                                 */
/*=========================================================================*/
/* Define a separate data structure for State 0 of the sSTATE to
 * speed processing of the input while the sSTATE is in that state.
 * Since the Aho-Corasick algorithm only defines "go" transitions
 * for this state (one for each valid input character) and no
 * "failure" transitions or output messages, only an array of
 * "go" transition state numbers is needed. The array is accessed
 * directly, using the input character as the index.
 */
PRIVATE sSTATE   *aState0[UCHAR_MAX + 1];
PRIVATE sSTATE   sFail;                         /* Dummy "failure" state */
static int       depth;
sQELEM           *f = NULL, *l = NULL;



/*=========================================================================*/
/*                        Function Prototypes                              */
/*=========================================================================*/

/* Build the state machine */
PRIVATE void     vBuildGoGraph(void);
PRIVATE void     vBuildKeyword(BYTE* str);
PRIVATE void     vBuildFailMoveTrans(int type);

/* Run the state machine */
PRIVATE sSTATE*  psRunTrans(int type, sSTATE* psState, register BYTE cText);

/* Dynamic memory handling */
PRIVATE sSTATE*  psAllocState(void);
PRIVATE sTRANS*  psAllocTrans(sSTATE* psState, BYTE cText);
PRIVATE void     vDelNode(sSTATE *psS);
PRIVATE void     vAllocQElem(sQELEM** head_ptr, sQELEM** tail_ptr, sSTATE* psState);
PRIVATE FLAG     fCheckQElem(sQELEM* head_ptr, sSTATE* psState);
PRIVATE void     vDelQElem(sQELEM** head_ptr);
PRIVATE BYTE*    pzToupperStr(BYTE* str);

/* Debugging */
PRIVATE void     vDumpFsa(int type);
PRIVATE void     vDumpNode(int type, sSTATE * s);
```

```
#ifdef MAIN
#define MAX_LINE   257
/*+2 MODULE AC.C ============================================================*/
/*   NAME   00            Main (Module Testdriver)                            */
/*== SYNOPSIS ================================================================*/
void
main(
   int argc,
   BYTE** argv)
{
/* DESCRIPTION (USAGE)
 * ac [<options>] <boolexpr> <file>, where :
 *    <options>  -N to force a NFSA search (default DFSA)
 *               -U to force a case-INsensitive search
 *    <boolexpr> is a boolean expression of search phrases
 *               example: This&^(That/Those)
 *               The boolexpr is parsed by function pzParse (in bool.c) to :
 *               - a postfix representation of the Boolean logic tree (pzPostfix)
 *               - a set of keywords K= {k1,k2,...,kn} to search for (symtable[])
 *    <file>     is the file to search for boolexpr, line by line.
 *
 * Search <file> line-by-line for boolean expression <boolexpr> :
 *    1: Parse command line <options>, if any.
 *    2: Parse Bool search expression <boolexpr> to postfix string and keyword list
 *    3: Perform search on <file> ...
 *       3.1: Initialize AC-search
 *       3.2: Run AC-search machine (NFSA or DFSA) on each line of input-file <file>
 *       3.3: Terminate search & Ret status to parent proces: 0 if match, 1 otherwise
 *-2*/
    BYTE buffer[MAX_LINE], *nl;        /* Buffer for Input string (file line)      */
    BYTE *pzPostfix = NULL;            /* Postfix string for parsed boolean logic  */
    FLAG fMatch = FALSE;               /* TRUE if match of <boolexpr> vs file line */
    FILE *in_fd;                       /* File Handle for inputfile */
    char *temp;                        /* Pointer for walking argv  */

                                       /* Optional parameters to AC */
    int  type  = DFSA;                 /* Type of FSA, default DFSA */
    FLAG UCASE = FALSE;                /* Case INsensitive search ? */


    /* 1: Parse command line options */
    while (--argc && (*++argv)[0] == '-')
            for (temp = argv[0] + 1; *temp != '\0'; temp++)
                switch (toupper(*temp)) {
               case 'N':
                            type  = NFSA;
                            break;
               case 'U':
                   UCASE = TRUE;
                   break;
                        default:
                            exit(1);
                }


    /* 2: Parse Bool search expression to postfix string and keyword list */
    D(printf("\n\n=============== BUILDING NEW FSA ================\n\n"));
    D(puts("PARSE INPUT STRING ..."));
    pzPostfix = pzParse(UCASE ? pzToupperStr(*argv++) : *argv++);
    D(printf("\t%s\n", *argv));
    argc--;
```

```
/* 3.1: Initialize AC-search */
    vBuildFsa(type);


    /* 3.2: Run AC-search on each line of input-file */
    in_fd = ( argc ? fopen(*argv, "r") : stdin);
    while (fgets(buffer, MAX_LINE, in_fd)) {

       if (nl = strchr(buffer, '\n'))           /* Remove newline */
              *nl = '\0';

       if (fMatch = fRunFsa(type,
                           (UCASE ? pzToupperStr(buffer) : buffer),
                           pzPostfix)) {
          D(printf("\nOUTPUT ...\n>\t"));
               puts(buffer);
            }
    }


    /* 3.3: Terminate search & Ret status to the parent proces */
    /*      0 if match(es), 1 if none                          */
    vDelFsa();
    exit (fMatch ? 0 : 1);

} /* END function main() */
#endif /*MAIN*/
```

```
/****************************** 1 ************************************/
/********************** BUILDING THE STATE MACHINE **********************/
/*****************************************************************************/


/*+2 MODULE AC.C=============================================================*/
/*   NAME   01                 vBuildFsa                                     */
/*== SYNOPSIS ===============================================================*/
void
vBuildFsa(int type)
{
/* DESCRIPTION
 * Construct a "Finite State Automaton" (FSA) from the keywords in symtable[]
 *    1: Build the "go" graph
 *    2: Build the "failure" and optionally "move" transitions
 *    3: Dump of FSA for debygging (optional)
 *
 *-2*/
    /* 1: Build the "go" graph (G) */
    vBuildGoGraph();


    /* 2: Build the "failure" (F) and optionally "move" (M) transitions */
    vBuildFailMoveTrans(type);


    /* 3: Dump FSA for debygging (optional) */
    D(vDumpFsa(NFSA));
    if (type == DFSA)
            D(vDumpFsa(DFSA));

} /* END function vBuildFsa() */



/*+3 MODULE AC.C -----------------------------------------------------------*/
/*   NAME   01.01              vBuildGoGraph                                 */
/*-- SYNOPSIS -----------------------------------------------------------*/
void
vBuildGoGraph(void)
{
/* DESCRIPTION
 * Construction of goto graph data structure for the goto function G of FSA.
 *
 * The graph is started by one vertex/node : the array representing state 0.
 *
 * We now enter each keyword ki into the graph beginning at the start state:
 * new vertices (states between chars) and edges (transitions on chars) are
 * added to the graph thus building a complete path that spells out ki.
 * The output function is defined for the state at which the path terminates.
 *
 * Up to this point the graph is a rooted tree. To complete the construction
 * of the goto function we add a loop from state 0 to state 0 on all input
 * chars NOT starting a keyword path.
 *
 * In summary the algorithm for building the goto graph is  :
 *    BEGIN
 *      (1) initialize all transitions out of state-0 to FAIL
 *      (2) enter all entries in keyword-list symtable[] into the goto graph
 *      (3) reset all FAIL-transitions out of state-0 to state-0
 *    END
 *-3*/
    register BYTE cText;       /* character in the input alfabet */
    int       i;               /* integer counter */

    /* 1: Initialize FSA State 0 go transition array to : G(0,x) = FAIL */
    for (cText = 1; (0 < cText && cText <= UCHAR_MAX); cText++)
            aState0[cText] = &sFail;
```

```
    /* 2: Put all keywords into the goto graph by calling vBuildKeyword */
    for (i = 1; symtable[i].pzLexptr != NULL; i++)
            vBuildKeyword(symtable[i].pzLexptr);

    /* 3: For all x such that G(0,x) == FAIL, set G(0,x) = 0 */
    for (cText = 1; (0 < cText && cText <= UCHAR_MAX); cText++)
            if (aState0[cText] == &sFail)
                aState0[cText] = NULL;

} /* END function vBuildGoGraph() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   01.02              vBuildKeyword                                 */
/*-- SYNOPSIS ------------------------------------------------------------*/
void
vBuildKeyword(BYTE* pzWord)
{
/* DESCRIPTION
 * Enter a keyword (text string) into the goto graph;  Note that '\0' can
 * never be a valid character (Used as C string terminator).
 *
 * First run each character of the keyword in turn through the current
 * partially-built goto graph.
 *
 * When a failure occurs, add the remainder of the keyword to the graph
 * as one new transition and state per char.
 *
 * Finally define the output function for the keyword's terminal state.
 *
 * The complete algorithm for entering a keyword K into the goto graph is :
 *    BEGIN
 *       (1) // Run keyword through partially built goto graph until FAIL
 *           FOR (i = 0, S = 0; G(S, K[i]) != FAIL; i++)
 *              S = G(S, K[i])
 *       (2) // Enter remainder of keyword into goto graph (build new states)
 *           FOR( ; (K[i]; i++)
 *              BEGIN
 *                 G(S, K[i]) = newS
 *                 S = newS
 *              END
 *       (3) // Define output function for the keyword's terminal state
 *           O(S) = function(keyword)
 *    END
 *-3*/
    register BYTE   *pzIndex;          /* Index into pzWord               */
    register sSTATE *psState;          /* Current state; start in state 0 */
    register sSTATE *psNextState;      /* Next state in goto graph        */
    sSTATE   *s;                       /* Temp variable for state         */
    sTRANS   *t;                       /* Temp variable for transition    */


    /* 1: Run chars in turn through partially-built goto graph until fail */
    for (pzIndex = pzWord, psState = NULL;
             ((s = psRunTrans(NFSA, psState, *pzIndex)) != &sFail);
             pzIndex++)
            psState = s;


    /* 2: Enter the remainder of the keyword string into the goto graph */
    while (*pzIndex) {
            if (!psState)
                /* State 0 : add new state from state 0 */
                psNextState = aState0[*pzIndex++] = psAllocState();
            else if (!(t = psState->psGoList)) {
                /* No goList : add trans and state as first in goList */
```

```
                psNextState = psAllocState();
                psState->psGoList = psAllocTrans(psNextState, *pzIndex++);
            }
            else {
                /* goList exists : add trans and state as last in goList */
                while (t->psTrans)
                        t = t->psTrans;
                psNextState = psAllocState();
                t->psTrans = psAllocTrans(psNextState, *pzIndex++);
            }
            psState = psNextState;
    }

    /* 3: Define keyword's index in symtable as terminal state O-function */
    if (!(psState->index = (int *) malloc(sizeof(int) * (strlen(pzWord) + 1))))
            vError(EMEM000, pzWord);
    *(psState->index) = iSymLookup(pzWord, strlen(pzWord));
    *(psState->index + 1) = '\0';
    D(printf("\nENTER ...\n\tTerminalState[%d], Index[%d]-->[%s]\n",
                psState, *(psState->index), pzWord));

} /* END function vBuildKeyword() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   01.03       vBuildFailMoveTrans                                  */
/*-- SYNOPSIS ----------------------------------------------------------------*/
void
vBuildFailMoveTrans(int type)
{
/* DESCRIPTION
 * Build the "failure" and optionally "move" transitions from the defined
 * go graph.
 *
 * ----------------- Build the failure function F for NFSA ------------------
 *
 *   For all states s1 of depth 1 : F(s1) = 0  // depth = 1, loop to state 0
 *   For all states s2 of depth d > 1          // depth > 1
 *      For each state s1 of depth d-1 :
 *         For each valid inputsymbol x
 *            If defined go-trans : s1 -(x)-> s2
 *               sf = F(s1)                    // G(0,x) != FAIL
 *               Execute sf = F(sf) until G(sf, x ) != FAIL
 *               F(s2) = G(sf, x)
 *
 *   Algorithm using queue Q to hold the states :
 *   BEGIN
 *   (1) initialize Q to empty
 *   (2) FOR each valid inputsymbol x              // depth = 1
 *          IF ( (s1 = G(0, x)) != 0 )
 *          BEGIN
 *             add s1 to Q-tail
 *             F(s1) = 0
 *          END
 *   (3) WHILE ( (s1 = remove from Q-head) != 0 )  // depth > 1
 *          FOR each valid inputsymbol x
 *            IF ( (s2 = G(s1, x)) != FAIL )
 *            BEGIN
 *               add s2 to Q-tail
 *               FOR (sf = F(s1); G(sf,x) == FAIL; sf = F(sf))
 *                  ;
 *               F(s2) = G(sf, x)
 *               O(s2) = O(sf) + O(F(sf))
 *            END
 *   END
 *
 * ------------- Optionally build the move function M for DFSA --------------
```

```
*
*   For all states s1 of depth 1 : M(0, x) = G(0, x)  // depth = 1, use G
*   For all states s2 of depth d > 1                  // depth > 1
*      For each state s1 of depth d-1 :
*         For each valid inputsymbol x
*            If defined go-trans : s1 -(x)-> s2
*               use this as move-trans
*            else
*               set move-trans s1 -(x)-> = move-trans(F(s1), x)
*
*   Algorithm using queue Q to hold the states :
*   BEGIN
*   (1) initialize Q to empty
*   (2) FOR each valid inputsymbol x              // depth = 1
*          M(0, x) = G(0, x)
*          IF ( (s1 = G(0, x)) != 0 )
*             add s1 to Q-tail
*   (3) WHILE ( (s1 = remove from Q-head) != 0 )  // depth > 1
*          FOR each valid inputsymbol x
*             IF ( (s2 = G(s1, x)) != FAIL )
*             BEGIN
*                add s2 to Q-tail
*                M(s1, x ) = s2
*             END ELSE
*                M(s1, x) = M(F(s1), x)
*   END
*
* -------------- Combined algorithm for fail and move functions ------------
*
*   Merged algorithm using queue Q to hold the states :
*   BEGIN
*   (1) initialize Q to empty
*   (2) FOR each valid inputsymbol x              // depth = 1
*          M(0, x) = G(0, x)
*          IF ( (s1 = G(0, x)) != 0 )
*          BEGIN
*             add s1 to Q-tail
*             F(s1) = 0
*          END
*   (3) WHILE ( (s1 = remove from Q-head) != 0 )  // depth > 1
*          FOR each valid inputsymbol x
*             IF ( (s2 = G(s1, x)) != FAIL )
*             BEGIN
*                add s2 to Q-tail
*                FOR (sf = F(s1); G(sf,x) == FAIL; sf = F(sf))
*                   ;
*                F(s2) = G(sf, x)
*                O(s2) = O(sf) + O(F(sf))
*                M(s1, x ) = s2
*             END ELSE
*                M(s1, x) = M(F(s1), x)
*   END
*-3*/
   register BYTE cText;
   register sSTATE *s1, *s2, *sf;
   sTRANS   *t;
   sQELEM   *first, *last;            /* Pointer to head & tail of queue */
   int      *i, *j;

   /* 1: Initialize Q (empty) */
   last = first = NULL;

   /* 2: We use a common array for transitions out of state 0 : aState0[] */
   /* In vBuildGoGraph we initialize aState0[] to &sFail for all input */
   /* We also use aState0[] for the DFSA, so for all x M(0,x) = G(0,x) */
   /* Now we only have to fill Q with go-transitions out of state 0.   */
   for (cText = 1; (0 < cText && cText <= UCHAR_MAX); cText++)
           if (s1 = psRunTrans(NFSA, NULL, cText))
```

```
            vAllocQElem(&first, &last, s1);

    /* 3: While Q not empty ... */
    while (first) {

            /* Get state "s1" at head of Q */
            s1 = first->psState;
            vDelQElem(&first);

            /* For every character "cText" in the input alfabet : */
            for (cText = 1; (0 < cText && cText <= UCHAR_MAX); cText++) {

                /* If a transition from state s1 to state s2 on cText */
                if ((s2 = psRunTrans(NFSA, s1, cText)) != &sFail) {

                        /* Add s2 to end of Q */
                        vAllocQElem(&first, &last, s2);

                        /* Define failure transition for s2 : F(s2) */
                        for (sf = s1->psFailSt; psRunTrans(NFSA, sf, cText) == &sFail;)
                            sf = sf->psFailSt;
                        s2->psFailSt = psRunTrans(NFSA, sf, cText);

                        /* Define complete output function O for s2 */
                        /* by concatenating O(s2) with O(F(s2))     */
                        i = s2->index;
                        j = s2->psFailSt->index;
                        if (j) {
                            if (!i) {
                                    if (!(i = s2->index = (int *) malloc( sizeof(int) *

            (strlen(symtable[*j].pzLexptr) + 1))))
                                            vError(EMEM001, symtable[*j].pzLexptr);
                            }
                            else
                                    for (; *i; i++)
                               ;
                                for (; *i = *j; i++, j++)
                          ;
                                *i = '\0';
                            }
                }
                else {
                        /* No transition from s1 to s2 on input cText :  */
                        /* Set s2 to the precalculated move transition   */
                        /* from s1's failure state on input "cText".     */
                        if (type == DFSA)
                            s2 = psRunTrans(DFSA, s1->psFailSt, cText);
                }

                /* Add move transition from s1 to s2 on input "cText" */
                if (type == DFSA && s2)
                        if (!s1->psMvList)      /* First instance of the list? */
                            t = s1->psMvList = psAllocTrans(s2, cText);
                        else                /* No, just another one ... */
                            t = t->psTrans = psAllocTrans(s2, cText);
            }
    }

} /* END function vBuildFailMoveTrans() */
```

```
/****************************** 2 ***************************************/
/*********************** RUNNING THE STATE MACHINE ***********************/
/*****************************************************************************/


/*+2 MODULE AC.C=============================================================*/
/*   NAME   02                fRunFsa                                        */
/*== SYNOPSIS ================================================================*/
FLAG
fRunFsa(
   int            type,               /* Search type : NFSA or DFSA    */
   register BYTE* str,                /* String to search for keywords */
   BYTE*          pzPostfix)          /* Parsed Boolean expression     */
{
/* DESCRIPTION
 * Run the finite state automaton with string "str" as input ...
 *    1: Reset state of symbol-table and go-graph
 *    2: Run FSA on input-line <str>, and note "hits" on keywords
 *    3: Evaluate boolean expression, using keyword "hits" from FSA-run
 *       Return TRUE if match, FALSE otherwise.
 *-2*/
    register sSTATE *psState;
    FLAG      fMatch = FALSE;
    int       i, j;

    /* 1: Reset state of symbol-table and go-graph */
    vSymReset();               /* Initialize symbol table */
    psState = NULL;             /* Initialize go-graph start state */


    /* 2: Run FSA on input-line <str> */
    D(puts("\n-----------------------------------------------------------"));
    D(printf("INPUT ...\n<\t[%s]\n\n", str));
    D(puts("RUN FSA ON INPUT ...\n"));

    if (type == NFSA)
       /*-2.1---------------------------------------------------------------*/
            /* NFSA : Nondeterministic Finite State Automaton, use go-trans      */
       /*-------------------------------------------------------------------*/
            while (*str) {

                while (psRunTrans(NFSA, psState, *str) == &sFail) {
                        D(printf("\tFAIL  [%d] --(%c)--> ", psState, *str));
                        psState = psState->psFailSt;
                        D(printf("[%d]\n", psState));
                }

                D(printf("\tGO  [%d] --(%c)--> ", psState, *str));
                psState = psRunTrans(NFSA, psState, *str);
                D(printf("[%d]\n", psState));

                /* Print terminal state message(s) if any */
                for (i = 0; psState && (j = *(psState->index + i)); i++) {
                        fMatch = TRUE;
                        symtable[j].fValue = TRUE;
                        D(printf("\t\tHIT lexeme [%s]\n", symtable[j].pzLexptr));
                }
                str++;
            }

    else /* type == DFSA */
       /*-2.2---------------------------------------------------------------*/
            /* DFSA : Deterministic Finite State Automaton, use move-trans       */
       /*-------------------------------------------------------------------*/
            while (*str) {

                D(printf("\tMOVE  [%d] --(%c)--> ", psState, *str));
                psState = psRunTrans(DFSA, psState, *str);
```

```
                D(printf("[%d]\n", psState));

                /* Print terminal state message(s) if any */
                for (i = 0; psState && (j = *(psState->index + i)); i++) {
                        fMatch = TRUE;
                        symtable[j].fValue = TRUE;
                        D(printf("\t\tHIT lexeme [%s]\n", symtable[j].pzLexptr));
                }
                str++;
            }


    /* 3: Evaluate boolean expression, using keyword "hits" from FSA-run */
    D(puts("\nINTERPRET ..."));
    D(puts("Symboltable Boolean values :"));
    D(for (i = 1; symtable[i].pzLexptr; i++)
       printf("\tsymtable[%d] : %s %s\n",
           i, symtable[i].pzLexptr, symtable[i].fValue ? "TRUE" : "FALSE")
                );

    D(puts("Postfix Boolean evaluation :"));
    return (fInterpret(pzPostfix) == TRUE);

} /* END function fRunFsa() */


/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   02.01             psRunTrans                                     */
/*-- SYNOPSIS ------------------------------------------------------------*/
sSTATE   *
psRunTrans(
   int           type,
   sSTATE*       psState,
   register BYTE cText )
{
/* DESCRIPTION
 * Perform one transition from psState via character cText to the next state.
 * Return a pointer to next state, or - if cText is not on the transition
 * list - return pointer to failure-state (type NFSA) or NULL (type DFSA).
 *-3*/
    register sTRANS *t;

    /* If state 0, access state 0 array of state pointers directly */
    if (!psState)
            return aState0[cText];
    else {
            /* Point to the head of the linked list of transitions */
            t = (type == NFSA ? psState->psGoList : psState->psMvList);

            /* Traverse the list looking for a match to the input character. */
            for (; t && (t->cTrans != cText); t = t->psTrans);

            /* Return pointer to new state, or failure (NFSA) resp. NULL (DFSA) */
            return (t ? t->psState : (type == NFSA ? &sFail : NULL));
    }

} /* END function psRunTrans() */
```

```
/****************************** 3 ******************************************/
/******************** DYNAMIC MEMORY ADMINISTRATION ********************/
/*****************************************************************************/


/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.01         psAllocState                                        */
/*-- SYNOPSIS ------------------------------------------------------------*/
sSTATE*
psAllocState(void)
{
/* DESCRIPTION
 * Create a new state and return a pointer to it.
 *-3*/
    sSTATE   *psS;

    if (!(psS = (sSTATE *) malloc(sizeof(sSTATE))))
            vError(EMEM002, "sSTATE");

    psS->psGoList = psS->psMvList = NULL;
    psS->psFailSt = NULL;
    psS->index = NULL;

    return psS;

} /* END function psAllocState() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.02         psAllocTrans                                        */
/*-- SYNOPSIS ------------------------------------------------------------*/
sTRANS*
psAllocTrans(sSTATE * psState, BYTE cText)
{
/* DESCRIPTION
 * Create a new transition and return a pointer to it
 *-3*/
    sTRANS   *psT;

    if (!(psT = (sTRANS *) malloc(sizeof(sTRANS))))
            vError(EMEM003, "sTRANS");

    psT->cTrans = cText;       /* transition : --(cText)-->psState */
    psT->psState = psState;
    psT->psTrans = NULL;

    return psT;

} /* END function psAllocTrans() */



/*+2 MODULE AC.C ============================================================*/
/*   NAME   03.03           vDelFsa                                           */
/*== SYNOPSIS ============================================================*/
void
vDelFsa(void)
{
/* DESCRIPTION
 * Deallocate whole FSA structure (free all node-lists from aState0[])
 *-3*/
    BYTE     cText;
    sSTATE   *s;

    D(puts("\nDELETION OF STATE MACHINE ...\n"));

    for (cText = 1; (0 < cText && cText <= UCHAR_MAX); cText++)
```

```
            if( (s = aState0[cText]) != NULL ) {
               D(printf("\n[%d] --(%c)--> [%d]\n", 0, cText, s));
            vDelNode(s);
         }

} /* END function vDelFsa() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.04           vDelNode                                          */
/*-- SYNOPSIS ------------------------------------------------------------*/
void
vDelNode(sSTATE *psS)
{
/* DESCRIPTION
 * Deallocate one list of nodes in FSA structure ...
 *    1: Remove move-list (for DFSA). NB: states removed via go-list
 *    2: Remove go-list (for NFSA)
 *    3: Finally remove state and associated O-function
 *-3*/
    sTRANS   *psT;
    BYTE     *templ = "                                                  ";
    BYTE     indent[80];

    ++depth;
    strncpy(indent, templ, depth * 3);
    indent[depth * 3] = '\0';

    /* 1: Remove move-list (for DFSA). NB: states removed via go-list */
    for (psT = psS->psMvList; psT != NULL; psT = psS->psMvList) {
       psS->psMvList = psT->psTrans;   /* Scan through move trans-list */
            D(printf("%sFREE MV-sTRANS: [%d] --(%c)--> [%d]\n",
               indent, psS, psT->cTrans, psT->psState));
       free(psT);                      /* Release sTRANS */
    }


    /* 2: Remove go-list (for NFSA) */
    for (psT = psS->psGoList; psT != NULL; psT = psS->psGoList) {
       psS->psGoList = psT->psTrans;   /* Scan through go trans-list */
       vDelNode(psT->psState);         /* Recursively delete next node */
            D(printf("%sFREE GO-sTRANS: [%d] --(%c)--> [%d]\n",
               indent, psS, psT->cTrans, psT->psState));
       free(psT);                      /* Then release sTRANS */
    }


    /* 3: Finally remove state and associated O-function */
    D(printf("%sFREE sSTATE: [%d]\n", indent, psS));
    if (psS->index)                    /* If O-function defined */
       free(psS->index);               /* Release O-list */
    free(psS);                         /* Release sSTATE */

    depth--;

} /* END function vDelNode() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.05           vAllocQElem                                       */
/*-- SYNOPSIS ------------------------------------------------------------*/
void
vAllocQElem(sQELEM ** head_ptr, sQELEM ** tail_ptr, sSTATE * psState)
{
/* DESCRIPTION
 * Add an instance to the tail of a queue
```

```
 *-3*/
    sQELEM*   pq;

    if (!(pq = (sQELEM *) malloc(sizeof(sQELEM))))
            vError(EMEM004, "sQELEM");

    pq->psState = psState;
    pq->psQNext = NULL;

    if (!*head_ptr)            /* First instance of the queue? */
            *tail_ptr = *head_ptr = pq;
    else                       /* No, just another one ... */
            *tail_ptr = (*tail_ptr)->psQNext = pq;

} /* END function vAllocQElem() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.06           fCheckQElem                                      */
/*-- SYNOPSIS ------------------------------------------------------------*/
FLAG
fCheckQElem(sQELEM * head_ptr, sSTATE * psState)
{
/* DESCRIPTION
 * Check for a specified FSA state in the queue
 *-3*/

    for (; head_ptr != NULL; head_ptr = head_ptr->psQNext)
            if (head_ptr->psState == psState)
                return (TRUE);

    return (FALSE);

} /* END function fCheckQElem() */



/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.07           vDelQElem                                        */
/*-- SYNOPSIS ------------------------------------------------------------*/
void
vDelQElem(sQELEM ** head_ptr)
{
/* DESCRIPTION
 * Delete an instance from the head of queue
 *-3*/
    sQELEM   *pQElem;

    pQElem = *head_ptr;
    *head_ptr = (*head_ptr)->psQNext;

    free(pQElem);              /* Deallocate storage pointed to by pQElem */

} /* END function vDelQElem() */
```

```
/*+3 MODULE AC.C ------------------------------------------------------------*/
/*   NAME   03.08           pzToupperStr                                      */
/*-- SYNOPSIS ------------------------------------------------------------*/
BYTE*
pzToupperStr(register BYTE * str)
/* DESCRIPTION
 * Map entire string pointed to by "str" to upper case, - incl. danish `›†
 *-3*/
{
    static BYTE pcMap[] = {
/*                       0      1       2       3       4       5       6       7  */
/*                       8      9       A       B       C       D       E       F  */
            /*---------------------------cntrl chars-- -- -------------------------*/
             /* 00 */ '\000', '\001', '\002', '\003', '\004', '\005', '\006', '\007',
             /* 08 */ '\010', '\011', '\012', '\013', '\014', '\015', '\016', '\017',
             /* 10 */ '\020', '\021', '\022', '\023', '\024', '\025', '\026', '\027',
             /* 18 */ '\030', '\031', '\032', '\033', '\034', '\035', '\036', '\037',
            /*-------------------------- spec & numeric --------------------------*/
             /* 20 */ '\040', '\041', '\042', '\043', '\044', '\045', '\046', '\047',
             /* 28 */ '\050', '\051', '\052', '\053', '\054', '\055', '\056', '\057',
             /* 30 */ '\060', '\061', '\062', '\063', '\064', '\065', '\066', '\067',
             /* 38 */ '\070', '\071', '\072', '\073', '\074', '\075', '\076', '\077',
            /*-------------------------- upper letters ---------------------------*/
             /* 40 */ '\100', '\101', '\102', '\103', '\104', '\105', '\106', '\107',
             /* 48 */ '\110', '\111', '\112', '\113', '\114', '\115', '\116', '\117',
             /* 50 */ '\120', '\121', '\122', '\123', '\124', '\125', '\126', '\127',
             /* 58 */ '\130', '\131', '\132', '\133', '\134', '\135', '\136', '\137',
            /*-------------------------- lower letters ---------------------------*/
             /* 60 */ '\140',    'A',    'B',    'C',    'D',    'E',    'F',    'G',
             /* 68 */    'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',
             /* 70 */    'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',
             /* 78 */    'X',    'Y',    'Z', '\173', '\174', '\175', '\176', '\177',
            /*---------------------- international letters ----------------------*/
             /* 80 */ '\200', '\201', '\202', '\203', '\204', '\205',    '□', '\207',
             /* 88 */ '\210', '\211', '\212', '\213', '\214', '\215', '\216', '\217',
             /* 90 */ '\220',    '’', '\222', '\223', '\224', '\225', '\226', '\227',
             /* 98 */ '\230', '\231', '\232',    '□', '\234', '\235', '\236', '\237',
            /*---------------------------- graphics 1 ----------------------------*/
             /* A0 */ '\240', '\241', '\242', '\243', '\244', '\245', '\246', '\247',
             /* A8 */ '\250', '\251', '\252', '\253', '\254', '\255', '\256', '\257',
             /* B0 */ '\260', '\261', '\262', '\263', '\264', '\265', '\266', '\267',
             /* B8 */ '\270', '\271', '\272', '\273', '\274', '\275', '\276', '\277',
            /*---------------------------- graphics 2 ----------------------------*/
             /* C0 */ '\300', '\301', '\302', '\303', '\304', '\305', '\306', '\307',
             /* C8 */ '\310', '\311', '\312', '\313', '\314', '\315', '\316', '\317',
             /* D0 */ '\320', '\321', '\322', '\323', '\324', '\325', '\326', '\327',
             /* D8 */ '\330', '\331', '\332', '\333', '\334', '\335', '\336', '\337',
            /*------------------------ greek/mathematics -------------------------*/
             /* E0 */ '\340', '\341', '\342', '\343', '\344', '\345', '\346', '\347',
             /* E8 */ '\350', '\351', '\352', '\353', '\354', '\355', '\356', '\357',
             /* F0 */ '\360', '\361', '\362', '\363', '\364', '\365', '\366', '\367',
             /* F8 */ '\370', '\371', '\372', '\373', '\374', '\375', '\376', '\377'
            /*---------------------------------------------------------------------*/
    };

    register BYTE *temp;


    for (temp = str; *temp; temp++)
            *temp = pcMap[*temp];

    return str;

} /* END function pzToupperStr() */
```

```
/******************************** 4 ***************************************/
/************************ DEBUGGING ROUTINES ******************************/
/*************************************************************************/


/*+3 MODULE AC.C ----------------------------------------------------------*/
/*   NAME   04.01           vDumpFsa                                      */
/*-- SYNOPSIS ----------------------------------------------------------*/
void
vDumpFsa(int type)
{
/*-3*/
    BYTE      cText;
    sSTATE   *s;

    printf("\nDUMP OF STATE MACHINE TYPE %s ...\n",
              type == NFSA ? "NFSA" : "DFSA");

    if (type == DFSA)
            vAllocQElem(&f, &l, NULL);
    depth = 0;

    for (cText = 1; (0 < cText && cText <= UCHAR_MAX); cText++)
            if (s = psRunTrans(NFSA, NULL, cText)) {
                printf("\nState: 0[%c]\n%c -> %d\n", cText, cText, s);
                if (type == NFSA)
                        vDumpNode(type, s);
                else if (!fCheckQElem(f, s)) {  /* type == DFSA */
                        vAllocQElem(&f, &l, s);
                        vDumpNode(type, s);
                }
            }

   /* Clean up dumpQ */
   while (f)
       vDelQElem(&f);

} /* END function vDumpFsa() */



/*+3 MODULE AC.C ----------------------------------------------------------*/
/*   NAME   04.02           vDumpNode                                     */
/*-- SYNOPSIS ----------------------------------------------------------*/
void
vDumpNode(int type, sSTATE * s)
{
/*-3*/
    sTRANS   *t;
    BYTE     *templ = "                                                  ";
    BYTE      indent[80];
    int       i, j;

    ++depth;
    strncpy(indent, templ, depth * 3);
    indent[depth * 3] = '\0';

    /* dump state */
    printf("%sState: %d [ ", indent, s);
    for (i = 0; j = *(s->index + i); i++)
            printf("%s ", symtable[j].pzLexptr);
    printf("]\n");
    printf(type == NFSA ? "%sFAIL\t-> %d\n" : "", indent, s->psFailSt);

    /* dump trans list */
    for (t = (type == NFSA ? s->psGoList : s->psMvList); t; t = t->psTrans) {
            printf("%s%c\t-> %d\n", indent, t->cTrans, t->psState);
            if (type == NFSA)
```

```
                vDumpNode(type, t->psState);
            else if (!fCheckQElem(f, t->psState)) {          /* type == DFSA */
                vAllocQElem(&f, &l, t->psState);
                vDumpNode(type, t->psState);
            }
    }

    depth--;

} /* END function vDumpNode() */



/* END module AC.C                                                          */
/*==========================================================================*/
```