```
/*+1============================================================================*/
/*   MODULE                          BOOL.C                                   */
/*============================================================================*/
/*   FUNCTION      Boolean expression compiler                                 */
/*                                                                             */
/*   SYSTEM        Standard(ANSI / ISO) C.                                     */
/*                 Tested on PC / MS DOS 5.0(MSC 600 A).                       */
/*                                                                             */
/*   SEE ALSO      Modules: GENERAL.H, STACK.H, ERROR.H/C, BOOL.H              */
/*                                                                             */
/*   PROGRAMMER    Allan Dystrup.                                              */
/*                                                                             */
/*   COPYRIGHT(c)  Allan Dystrup, FEB.1991                                     */
/*                                                                             */
/*   VERSION       $Header: d:/cwk/kf/bool/RCS/bool.c 1.1 92/10/25 16:52:50 */
/*                 Allan_Dystrup Exp Locker: Allan_Dystrup $                   */
/*                 ------------------------------------------------------------*/
/*                 $Log:   bool.c $                                            */
/*                 Revision 1.1  92/10/25  16:52:50  Allan_Dystrup             */
/*                 Initial revision                                            */
/*                                                                             */
/*======================COMPILER STRUCTURE ================================= */
/*                                                                             */
/* This module implements a simple compiler for evaluating boolean             */
/* expressions. The compiler is structured into two parts :                    */
/*  - a "front end" for transforming a boolean expression from infix form      */
/*    to postfix form (consisting of : SCANNER, PARSER, EMITTER);              */
/*  - a "back end" for interpreting the postfix boolean expression using       */
/*    an abstract/software stack machine (consisting of : INTERPRETER).        */
/* The compiler "front end" and "back end" communicates using a common         */
/* symbol table (cf BOOL.H). The overall structure of the module is thus :     */
/*                                                                             */
/*   infix string---* SCANNER---* PARSER---* EMITTER---* postfix string        */
/*                      |                       *              |             */
/*                      |                       |              |             */
/*                      +-------* SYMBOL--------+              |             */
/*                                *  |                         |             */
/*                                |  |                         |             */
/*                    (FIND)------+  |                         |             */
/*                                   *                         |             */
/*   evaluation result *-------INTERPRETER *--------------------+             */
/*                                                                             */
/*                                                                             */
/*                                                                             */
/*====================== COMPILER FRONT END ===================================*/
/*                                                                             */
/* The syntax of the input string is defined by the following rules.           */
/*                                                                             */
/* 1. Boolean operator precedence, arity and associativity :                   */
/*             operator(s)   precedence   arity and associativity              */
/*             ------------------------------------------------------          */
/*             NOT               1        unary                                */
/*             AND               2        binary                               */
/*             OR   XOR          3        binary, left associative             */
/*                                                                             */
/* 2. Context-free grammer (expressed in BNF:Bacus-Naur/Normal Form) :         */
/*    1. set of tokens(= terminal symbols) : { NOT, AND, OR, XOR, EOS, ID } */
/*       NOT,AND,OR,XOR  The 4 basic Boolean operators                         */
/*             EOS       End Of String                                       */
/*             ID        IDentifier                                          */
/*    2. set of nonterminals : { Z, E, T, F }                                  */
/*             Z       Destinguished start symbol                            */
/*             E       Expression                                            */
/*             T       Term                                                  */
/*             F       Factor                                                */
```

```
/*    3. set of productions (context-free: one nonterminal on left side)    */
/*             Z  ->   E EOS                                                */
/*             E  ->   E OR  T | E XOR T | T                                */
/*             T  ->   T AND F | F                                          */
/*             F  ->   ID | ( E ) | NOT F                                   */
/*                                                                          */
/*    We want to construct a recursive descent predictive parser, ie.       */
/*    a top-down parser without backtracking. Furthermore we will require   */
/*    the parser to work with 1 lookahead-character, ie. a LL(1) parser     */
/*    (scanning input Left->right, using Leftmost derivations, 1 lookahead).*/
/*    To make this possible, we have to transcribe the productions to get : */
/*     - no left recursion (thus eliminating infinite loops)                */
/*     - no conflict between two right sides for any lookahead symbol       */
/*       (ie. an unambigous FIRST-set for any production) :                 */
/*             Z  ->   E  EOS                                               */
/*             E  ->   T  E'                                                */
/*             E' ->   OR  T E' | XOR T E' | epsilon                        */
/*             T  ->   F  T'                                                */
/*             T' ->   AND F T' | epsilon                                   */
/*             F  ->   ( E )    | NOT F    | ID                            */
/*                                                                          */
/* 3. Syntax-directed translation :                                         */
/*    Combination of syntax analyzer and intermediate-code generator.       */
/*    The chosen intermediate code is postfix (= "reverse polish"),         */
/*    a notation in which the operators appear after their operands.        */
/*    Postfix notation is a compact way of representing a parse tree,       */
/*    with an unambigous decoding dictated only by position and arity       */
/*    (= number of operator arguments), - ie. parentheses are eliminated.   */
/*    The interpretation of postfix is analogous to a bottom-up traversal   */
/*    of the parse tree, - but may be performed by a simple stack machine.  */
/*    We use a simple syntax directed definition (with semantic rules       */
/*    defined by simple concatenation of translations) yielding a           */
/*    corresponding simple translation scheme that can execute the          */
/*    semantic actions (ie. emit intermediate code) during the parse.       */
/*    Thus it is not nessecary to explicitly build the parse tree (syntax   */
/*    analysys) before emitting the output (code ceneration); - hence       */
/*    the concept "syntax-directed translation" !                           */
/*                                                                          */
/*:-------------------------------------------------------------------------:*/
/*:       Simple syntax directed definition    :    Translation scheme     :*/
/*:--------------------------------------------:---------------------------:*/
/*: Productions def.  : Semantic rules for     : Semantic actions in       :*/
/*: gramm. constructs : postfix=.p translation : procedural notation       :*/
/*: (infix)           : (infix to postfix)     : (generate postfix)        :*/
/*:-------------------:------------------------:---------------------------:*/
/*  Z  -> E  EOS      : Z.p  := E.p  EOS       : Z -> E {terminate}         */
/*  E  -> T  E'       : E.p  := T.p, E'.p      : E -> T  E'                 */
/*  E' -> OR  T E'    : E'.p := T.p, OR,  E'   : E'-> OR  T {print(OR)}  E' */
/*     |  XOR T E'    :        | T.p, XOR, E'  :   |  XOR T {print(XOR)} E' */
/*     |  epsilon     :        | epsilon       :   |  epsilon               */
/*  T  -> F  T'       : T.p  := F.p, T'.p      : T -> F  T'                 */
/*  T' -> AND F T'    : T'.p := F.p, AND, T'p  : T'-> AND F {print(AND)} E' */
/*     |  epsilon     :        | epsilon       :   |  epsilon               */
/*  F  -> ( E )       : F.p  := E.p            : F -> ( E )                 */
/*     |  NOT F       :        | F.p, NOT      :   |  NOT F {print(NOT)}    */
/*     |  ID          :        | ID.p          :   |  ID {print(ID.slot)}   */
/*:-------------------------------------------------------------------------:*/
/*                                                                          */
/*                                                                          */
/*====================== COMPILER BACK END ==================================*/
/*                                                                          */
/*    The compiler back-end takes the front-end generated code (a postfix   */
/*    string) as input, and evaluates it on the target machine (a simple    */
/*    stack construction).                                                  */
/*                                                                          */
```

```
/*    To be specific, the type of syntax-directed translation used by the   */
/*    compiler front end is based on an S-attributed definition with only   */
/*    synthesized attributes (more complex grammars require context info    */
/*    passed down the syntax tree in the form of inherited attributes).     */
/*                                                                          */
/*    The "internal form" emitted by the front end is an abstract syntax    */
/*    tree casted as a postfix string; More precicely, the syntax tree is   */
/*    a Directed Acyclic Graph : a "DAG", because the value index for an    */
/*    operator that occurs multiple times in a boolean expression, is       */
/*    unique (a reference to the symboltable) in the postfix string.        */
/*                                                                          */
/*    These characteristics alow a simple depth-first evaluation of the     */
/*    syntax tree (no dependency graph transformation of the tree needed).  */
/*    Thus the evaluation function may be implemented by a stack machine    */
/*    interpreting the postfix-string from left to right.                   */
/*                                                                          */
/*-1========================================================================*/



/*--------------------------------------------------------------------------*/
/*                          INCLUDE HEADER-FILES                            */
/*--------------------------------------------------------------------------*/

/* Std.C header files */
#include <stdio.h>
#include <stddef.h>
#include <ctype.h>
#include <string.h>

/* Project header files */
#include "general.h"
#include "error.h"
#include "stack.h"

/* Module header file */
#define _BOOL_ALLOC
#include "bool.h"


#define COPYRIGH   "Copyright (c) 1990 Allan Dystrup / KMD IS"
#define VERSION    "v.1.0"



/*--------------------------------------------------------------------------*/
/*                          PRIVATE GLOBAL DATA                             */
/*--------------------------------------------------------------------------*/
/* Lexemes must by one char in range [0-127] */
/* Token       Lexeme             Semantics        */
#define OR     (BYTE) '/'      /* boolean OR       */
#define XOR    (BYTE) '%'      /* boolean XOR      */
#define AND    (BYTE) '&'      /* boolean AND      */
#define NOT    (BYTE) '^'      /* boolean NOT      */

#define LP     (BYTE) '('      /* left parenthesis */
#define RP     (BYTE) ')'      /* right parantesis */

#define QUOTE  (BYTE) ':'      /* quoted string    */
#define ESCAPE (BYTE) '\\'     /* escape next char */

#define ID     (BYTE) 'I'      /* identifier       */
#define EOI    (BYTE) '\0'     /* End Of Input     */
```

```
PRIVATE BYTE* yytext = (BYTE *) "";    /* Lexeme (not \0 terminated) */
PRIVATE int   yyleng = 0;              /* Lexeme length */
PRIVATE BYTE  aEmitStr[OUTMAX];        /* Output string from Emitter */



/*---------------------------------------------------------------------------*/
/*                        PRIVATE FUNCTION PROTOTYPES                        */
/*---------------------------------------------------------------------------*/

/* SCANNER */
PRIVATE BYTE
   bScan P((BYTE * pzStr));

/* PARSER */
PRIVATE void
   vAdvance    P((BYTE token));
PRIVATE void
   vExpr       P((void));
PRIVATE void
   vTerm       P((void));
PRIVATE void
   vFactor     P((void));

/* EMITTER */
PRIVATE void
   vEmit       P((BYTE token));

/* SYMBOL */
PRIVATE int
   iSymInsert  P((BYTE * pbStr, int len));




#ifdef MAIN
/*+2 MODULE BOOL.C ============================================================*/
/*   NAME   00                main                                            */
/*== SYNOPSIS ================================================================*/
/* DESCRIPTION                 Test driver for module bool.c
 *-2*/
int
main(argc, apzArgv)
   WORD    argc;
   BYTE*   apzArgv[];
{
    int         i;
    BYTE        bPeek;
    FLAG        fResult;

    BYTE        aInfix[256];   /* input string  : infix notation   */
    BYTE       *pzPostfix;     /* output string : postfix notation */


    /* 0: Echo input arguments */
    printf("\nargc=%d\n", argc);
    for (i = 0; apzArgv[i] != NULL; i++)
       printf("apzArgv[%d]=%s\n", i, apzArgv[i]);


    /* 1: Test SCANNER */
    printf("\nSCANNER ...\n");
    strcpy(aInfix, apzArgv[1]);        /* save org. input string (aInfix) */
```

```
    printf("Infix : %s\n", aInfix);
    bPeek = bScan(aInfix);
    while (bPeek != EOI) {
        printf("Token : [%c]   Lexeme : [yytext=%s yyleng=%d]\n", \
           bPeek, yytext, yyleng);
        bPeek = bScan("");
    }

    /* 2: Test PARSER */
    printf("\nPARSER ...\n");
    strcpy(aInfix, apzArgv[1]);        /* restore org. input string (aInfix) */
    pzPostfix = pzParse(aInfix);
    printf("INFIX.. : %s\n", aInfix);
    printf("POSTFIX : %s\n", pzPostfix);


    /* 3: Test INTERPRETER */
    printf("\nINTERPRETER ...\n");
    for (i = 1; i < SYMMAX; i++)       /* set up dymmy values for test */
        symtable[i].fValue = (i % 2 ? '\x01' : '\x00');
    fResult = fInterpret(pzPostfix);
    printf("Expression evaluates to : %s\n", (int) fResult ? "TRUE" : "FALSE");


    return 0;

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




/*+2 MODULE BOOL.C ============================================================*/
/*** NAME   01     ********** SCANNER ***************************************/
/*== SYNOPSIS ================================================================*/
/*
 * DESCRIPTION                Lexical analyzer
 * Linear analysis (= lexical analysis or scanning) of the input stream :
 * - The input string is read from left to right (using a "greedy algorithm")
 * - Characters are grouped into lexemes (sequences of characters with a
 *   collective meaning, according to the defined grammer)
 * - For each lexeme matched by a defined pattern in the grammer the scanner
 *   returns a unique type encoding (token), and assigns a pair of global
 *   attribute values :
 *    . "yytext" : a pointer to the lexeme (character string, not null term.)
 *    . "yyleng" : an integer giving the length of the lexeme (in chars.)
 *   The tokens returned by the scanner is used to direct the syntax analysis
 *   in the parser-function.
 * - If the token is ID (ie. boolean vExpression identifier), there are many
 *   possible lexemes for the same pattern, and the scanner then inserts the
 *   lexeme in a global symboltable "symtable[]" for easy reference by the
 *   later stages of the compilation process.
 * - The simple input string format does not warrant implementation of any
 *   error recovery/transformation or any special buffering scheme.
 *-2*/
BYTE
bScan(pzStr)
   BYTE* pzStr;
{
    static BYTE *index;        /* Index = current pointer into input string */
    BYTE        *p, *q, c;


    /* 1: Initialize & Skip leading whitespace */
    if (*pzStr)
```

```
   index = pzStr;

while (isspace(*index))
    ++index;


/* 2: Collect next token from pzStr; - break at valid token. */
/*    (Guarantied loop termination at EndOfString EOS = \0)  */
for (; TRUE; ++index) {
    yytext = index;        /* Point to first char in new token */
    yyleng = 1;

    switch (*index) {

       /* 2.1: One char operator */
       case EOI:              /* - terminate scan at EOI ! */
       case AND:
       case OR:
       case XOR:
       case NOT:
       case LP:
       case RP:
           return *index++;
           break;             /* Unreachable; - defensive programming */

      /* 2.2: Quoted string */
      case QUOTE:
           /* collect phrase lexeme */
           for (++index, ++yytext; *index != QUOTE; index++)
               if (*index == EOI)
                   vError(ELEX000, "Mangler slutmarkering af frase");

               yyleng = index - yytext;

               /* skip end-quote & return token */
               if (yyleng > 0) {
                   index++;
                   if (!iSymLookup(yytext, yyleng))
                       iSymInsert(yytext, yyleng);
                   return ID;
               }
               break;         /* Empty phrase : ignore & get next token */

       /* 2.3: Identifier lexeme */
       default:
           /* Collect identifier lexeme, incl. escaped characters;  */
           /* Break at whitespace or token, - unless escaped.       */
           /* (Guarantied loop termination at EndOfString EOS = \0) */
           for (; TRUE; index++)
               /* Terminate at reserved character */
               if (isspace(c = *index) ||
                   c == AND || c == OR || c == XOR   || c == NOT ||
                   c == LP  || c == RP || c == QUOTE || c == EOI)  {

                   /* unless previously ESCAPE'd (excluding EOI!) */
                   if (*(index - 1) == ESCAPE) {
                       for (p = index; p > yytext && *(p - 1) == ESCAPE; p--)
                           /* move p back over preceeding ESCAPEs */;
                       if ((index - p) % 2 != 0 && c != EOI)
                           continue;
                   }
                   break;
               }
               yyleng = index - yytext;
```

```
                    /* Eliminate ESCAPE-character(s) from lexeme */
                    for (p = yytext; p < index; p++)
                        if (*p == ESCAPE) {
                            for (q = p + 1; q < yytext + yyleng; q++)
                                *(q - 1) = *q;
                            *(q - 1) = ' ';
                            yyleng--;
                        }

                    /* Insert lexeme in symbol table */
                    if (!iSymLookup(yytext, yyleng))
                        iSymInsert(yytext, yyleng);

                    return ID;
                    break;              /* Unreachable; - defensive programming */

            }  /* end switch */

         }  /* end for */

       /* bScan() never returns implicitly by "falling out of scope" */
} /* END function bScan() */




/*+2 MODULE BOOL.C ============================================================*/
/*** NAME   02     ********** PARSER ***************************************/
/*== SYNOPSIS ================================================================*/
/*
 * DESCRIPTION                Syntax analyzer
 * Hierachial analysis (= syntax analysis or parsing) where tokens are
 * grouped hierachially into grammatical phrases (a parse tree).
 * We construct a "predictive parser" (relying on our context-free grammer
 * with no left-recursion and unambigous FIRST-set for all productions) :
 * For each lookahead symbol L, we :
 *  - Determine the grammatical production G, given by "L in FIRST(G)";
 *  - Advance the input stream to the next token (read next L);
 *  - Call the procedure P implementing G, where P :
 *     . for each nonterminal calls a sub-procedure (with possible "recursive
 *       descent" calls to lower nonterminal procedures)
 *     . may call on a code generator to vEmit code according to the semantic
 *       actions indicated in a translation scheme for the grammer.
 *       The code is emitted into the global string "aEmitStr" (declared
 *       static for this module), the address of which is returned to the
 *       calling function as a "handle".
 *-2*/

PRIVATE BYTE bLookahead;

PRIVATE void
vAdvance(bToken)
   BYTE bToken;
{
    if (bLookahead == bToken)              /* Match of expected token: */
       bLookahead = bScan((BYTE *) "");    /*    advance to next token */
    else                                   /* No match :               */
       vError(ESYN000, (char *) yytext);   /*    fatal syntax error!   */
}


/*---------- pzParse ----------*/
/*  Z -> E {terminate}         */
/*-----------------------------*/
BYTE*
```

```
pzParse(pzStr)
   BYTE* pzStr;
{
    /* Check for valid argument */
    if (pzStr == NULL)
       vError(EARG000, "pzParse(NULL pointer)");

    /* Initialize SCANNER (scan first lookahead symbol) */
    bLookahead = bScan(pzStr);

    /* Start the ball rolling : top-down recursive descent of EXPRESSION */
    D(printf("parse[%c]\n", (char) bLookahead));
    vExpr();

    /* Terminate OK or ERROR */
    if (bLookahead == EOI) {
       vEmit(EOI);
       return aEmitStr;
    } else
       vError(ESYN000, (char *) yytext);
}


/*---------- vExpr ------------*/
/*  E -> T  E'                 */
/*  E'-> OR  T {print(OR)}  E' */
/*    |  XOR T {print(XOR)} E' */
/*    |  epsilon               */
/*----------------------------*/
PRIVATE void
vExpr(void)
{
    BYTE        bOperator;

    D(printf("  vExpr[%c]\n", (char) bLookahead));
    vTerm();                    /* T */

    while (TRUE)                /* E' - perform recursively */
       switch (bLookahead) {    /* (replaced by iteration) */

           case OR:
           case XOR:
               bOperator = bLookahead;
               vAdvance(bLookahead);
               D(printf("  vExpr[%c]\n", (char) bLookahead));
               vTerm();
               vEmit(bOperator);
               continue;

           default:
               /* E' finished */
               return;
       } /* END switch(bLookahead) */
}


/*---------- vTerm ------------*/
/*  T -> F  T'                 */
/*  T'-> AND F {print(AND)} T' */
/*    |  epsilon               */
/*----------------------------*/
PRIVATE void
vTerm(void)
{
    D(printf("    term[%c]\n", (char) bLookahead));
```

```
    vFactor();                  /* F */

    while (TRUE)                /* T' - perform recursively */
       switch (bLookahead) {    /* (replaced by iteration) */

           case AND:
               vAdvance(bLookahead);
               D(printf("    term[%c]\n", (char) bLookahead));
               vFactor();
               vEmit(AND);
               continue;

           default:
               /* F' finished */
               return;

       } /* END switch(bLookahead) */
}


/*---------- vFactor ----------*/
/*  F -> ( E )                 */
/*    |  NOT E {print(NOT)}    */
/*    |  ID {print(ID.offset)} */
/*---------------------------*/
PRIVATE void
vFactor(void)
{
    D(printf("      vFactor[%c]\n", (char) bLookahead));
    switch (bLookahead) {
       case LP:
           vAdvance(LP);
           vExpr();
           vAdvance(RP);
           break;

       case NOT:
           vAdvance(bLookahead);
           vFactor();
           vEmit(NOT);
           break;

       case ID:
           vEmit(ID);
           vAdvance(bLookahead);
           break;

       default:
           vError(ESYN000, (char *) yytext);

    } /* END switch(bLookahead) */
}




/*+2 MODULE BOOL.C ============================================================*/
/*** NAME   03     ********** EMITTER ***************************************/
/*== SYNOPSIS ================================================================*/
/*
 * DESCRIPTION                Intermediate code generator
 * Generates an intermediate representation of the input string (aInfix)
 * in the form of a postfix string (="reverse polish notation") suitable
 * for subsequent interpretation by an abstract (ie. software) stack machine.
 * The input string syntax (a boolean expression) is extremely simple
```

```
 * consisting of only identifiers and boolean operands; - No variables or
 * control structures (apart from paranthetical nesting) are allowed, and
 * hence the postfix syntax does not have to take into account assignments
 * (l-values vs. r-values), control flow (jump instructions) et. al.
 * The syntax of the generated postfix string is as follows :
 *  - for boolean operators we store their token value with the high bit on;
 *    The token value for the boolean operators is identical to their lexeme
 *    representation (range [0...127]), so the range for stored operators
 *    is [128...255]. No operator attributes are stored in the symbol table.
 *  - for boolean identifiers we store their slot in the symbol table. Legal
 *    values for symbol table slots are [1...SYMMAX], where SYMMAX <= 127
 *    (to prevent collision with the value range for operators).
 *  - the postfix string is ended by a normal string terminator ('\0').
 *    It is allocated as a global static string in this module, but it's
 *    address is returned to the caller of parse as a handle.
 *-2*/
PRIVATE void
vEmit(bToken)
   BYTE bToken;
{
   int         i;
   static BYTE *index = aEmitStr;  /* Index = current pointer into output string  */

   if ((index - aEmitStr) > OUTMAX)
       vError(ETAB000, "OUTMAX");

   switch (bToken) {

       case AND:              /* Boolean operator */
       case OR:
       case XOR:
       case NOT:
           D(printf("EMIT OP : [%c]\n", bToken | '\x80'));
           *index++ = (bToken | (BYTE)'\x80');   /* raise high bit */
           break;

       case ID:               /* boolean identifier */
           i = iSymLookup(yytext, yyleng);
           D(printf("EMIT ID : [%c]-index[%d]->[%s]\n", (BYTE) i, i,
symtable[i].pzLexptr));
           *index++ = (BYTE) i;
           break;

       case EOI:
           D(printf("EMIT EOS: [NULL]\n"));
           *index++ = '\0';
           break;

       default:
           vError(ETOK000, "");       /* we should not end up here */
           break;
   }
}



/*+2 MODULE BOOL.C ============================================================*/
/*** NAME   04     ********** SYMBOL ***************************************/
/*== SYNOPSIS ================================================================*/
/*
 * DESCRIPTION               Symbol-table manager
 * Records input string identifiers in a symbol table, ie. a data structure
 * containing a record for each identifier, w. fields for various attributes
 * (in this case : a lexeme pointer and a boolean/flag value : "fValue").
 *  - The SCANNER enters the lexeme pointer into the symbol table for each
```

```
 *    new boolean vExpression identifier (the boolean attribute is undefined).
 *  - The PARSER calls the EMITTER to lookup identifiers and enter their
 *    symbol table slot into the intermediate code (ie. the postfix string).
 *-2*/
static int  lastentry = 0;     /* last used position in symboltable */

PRIVATE BYTE lexemes[STRMAX];  /* allocate lexeme array */
static int  lastchar = -1;     /* last used position in lexemes */

PRIVATE int
iSymInsert(pbLex, len)
   BYTE* pbLex;
   int len;
{
/* Insert lexeme s as last entry in the symbol table
 * Return slot for entry (or 0 if error).
 */

    if (lastentry + 2 >= SYMMAX)       /* room for array terminator  */
       vError(ETAB001, "SYMMAX");
    if (lastchar + len + 2 >= STRMAX)  /* room for string terminator */
       vError(ETAB002, "STRMAX");

    symtable[++lastentry].pzLexptr = &lexemes[++lastchar];
    strncpy((char *) symtable[lastentry].pzLexptr, (char *) pbLex, len);
    lastchar = lastchar + len;
    lexemes[lastchar++] = '\0';
    symtable[lastentry + 1].pzLexptr = NULL;

    D(printf("   INSERT :\tsymtable[%d] <- [%s]\n", \
       lastentry, symtable[lastentry].pzLexptr));
    return lastentry;
}


int
iSymLookup(pbLex, len)
   BYTE* pbLex;
   int len;
{
/* Determine whether there is an entry for lexeme s in the symbol table
 * If yes, return slot in symbol table [1...SYMMAX], else return 0
 */
    int         p;

    for (p = lastentry; p > 0; p--)
       if (((int)strlen((char *) symtable[p].pzLexptr) == len) &&
           (strncmp((char *) symtable[p].pzLexptr, (char *) pbLex, len) == 0))
           return p;

    return 0;                  /* no match */
}


void
vSymReset(void)
{
/* Reset boolean value for all lexemes in Symboltable to 'FALSE'
 */
    int         p;

    for (p = lastentry; p > 0; p--)
       symtable[p].fValue = FALSE;
}
```

```
/*+2 MODULE BOOL.C ============================================================*/
/*** NAME   05     ********* INTERPRETER ************************************/
/*== SYNOPSIS ================================================================*/
/*
 * DESCRIPTION               Stack machine
 * Interprets a postfix representation of a boolean vExpression to T or F.
 *
 * The syntactic structure of the boolean vExpression is captured (by the
 * compiler frontend) as an "abstract syntax tree" in the condensed form of
 * a postfix string (cf. format specified by the EMITTER function). This
 * "internal form" is passed to the interpreter function as parameter.
 *
 * The actual evaluation is performed by an abstract stack machine, which
 * reads the postfix string from left to right (corresponding to a "depth
 * first evaluation" of the abstract syntax tree), and :
 *  - pushes operand values onto the stack; An operand is represented in the
 *    postfix string as a "value number", ie. an index into the symbol table.
 *    The operand value (a boolean attribute "fValue") is retrieved from the
 *    symbol table and pushed on the stack
 *  - performing operator actions :
 *     . popping boolean operands as required (cf. operator arity)
 *     . executing the boolean operation (using C-language boolean functions)
 *     . pushing the boolean fResult
 * until the whole postfix string is evaluated (ie. EOI is encountered), and
 * the result resides as the top stack-element.
 *
 * The abstract stack machine is implemented as a collection of general macro
 * definitions (in the header file "stack.h"). This implementation is chosen
 * for maximum execution speed, but the extensive use of macro "side effects"
 * (in the form of stack pointer arithmetic) and the use of unchecked stack-
 * operations does require some programming care.
 *-2*/

stack_dcl(eval, int, SYMMAX);

FLAG
fInterpret(pzStr)
   BYTE* pzStr;
{
    BYTE        v;

    if (pzStr == NULL)
       vError(EARG001, "fInterpret(...NULL pointer...)");

    stack_clear(eval);

    for (; *pzStr != EOI; pzStr++)
       if (*pzStr & '\x80')           /* Boolean operator ? */

           switch (*pzStr & '\x7F') { /* Mask off high bit  */

               case AND:
                   D(printf("\tSTACK\t0->%c", stack_item(eval, 0) | 0x30));
                   D(printf("\t1->%c\n", stack_item(eval, 1) | 0x30));
                   v = (BYTE) (stack_item(eval, 0) & stack_item(eval, 1));
                   popn_(eval, 2);
                   push_(eval, v);
                   D(printf("\tAND"));
                   D(printf("\t0->%c\n", stack_item(eval, 0) | 0x30));
                   break;

               case OR:
```

```
                D(printf("\tSTACK\t0->%c", stack_item(eval, 0) | 0x30));
                D(printf("\t1->%c\n", stack_item(eval, 1) | 0x30));
                v = (BYTE) (stack_item(eval, 0) | stack_item(eval, 1));
                popn_(eval, 2);
                push_(eval, v);
                D(printf("\tOR"));
                D(printf("\t0->%c\n", stack_item(eval, 0) | 0x30));
                break;

            case XOR:
                D(printf("\tSTACK\t0->%c", stack_item(eval, 0) | 0x30));
                D(printf("\t1->%c\n", stack_item(eval, 1) | 0x30));
                v = (BYTE) (stack_item(eval, 0) ^ stack_item(eval, 1));
                popn_(eval, 2);
                push_(eval, v);
                D(printf("\tXOR"));
                D(printf("\t0->%c\n", stack_item(eval, 0) | 0x30));
                break;

            case NOT:
                D(printf("\tSTACK\t0->%c\n", stack_item(eval, 0) | 0x30));
                v = (BYTE) ((~pop_(eval)) & '\x01');
                push_(eval, v);
                D(printf("\tNOT"));
                D(printf("\t0->%c\n", stack_item(eval, 0) | 0x30));
                break;

            default:
                vError(ETOK001, "");    /* we should not end up here */
                break;
        }
        else {                  /* Boolean operand */
            D(printf("\tID\t%s\n", symtable[(int) *pzStr].pzLexptr));
            push_(eval, symtable[(int) *pzStr].fValue);
        }

        return ((pop_(eval)) == '\x00' ? FALSE : TRUE);

} /* END function fInterpret() */



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