```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/* -----------------------------------------------*/

#define MAXARGUMENTLENGTH 100
#define MAXPROGRAMLENGTH 1500
#define MAXFILENAMELENGTH 500
#define MAXFILELINELENGTH 1000
#define MAXNESTING 20
#define FALSE 0
#define TRUE 1
#define ENDOFSTREAM 0
#define MAXTAPELENGTH 1000
#define GROWFACTOR 50
#define INITIALSTACKSIZE 50
#define INITIALTAPEELEMENTSIZE 50
#define TEMPSTRINGSIZE 10000
#define TEXTBUFFERSIZE 1000

enum commandtypes
 { ADD=1,  /* adds a given text to the workbuffer */
   CLEAR,  /* clears the workspace */
   PRINT,  /* prints the workspace to stdout */
   STATE,  /* prints the current state of the machine */
   REPLACE,
   INDENT, /* indents each line of the workspace */
   CLIP,   /* removes the last character of the workspace */
   CLOP,   /* removes the first character of the workspace */
   NEWLINE,/* adds a newline to the  workspace */
   PUSH,   /* pushes part/all of the workspace to the stack */
   POP,    /* pops the stack to the workspace */
   PUT,
   GET,
   INCREMENT,
   DECREMENT,
   READ,   /* read a character from the input stream */
   UNTIL,
   WHILE,
   WHILENOT,
   TESTIS,
   TESTBEGINS,
   TESTCLASS,
   TESTENDS,
   TESTLIST,
   TESTEOF,
   TESTTAPE,
   COUNT,
   INCC,   /* increment the accumulator or counter */
   DECC,   /* decrement the accumulator or counter */
   CRASH,  /* exits the script immediately */
   OPENBRACE,
   CLOSEBRACE,
   ZERO,    /* sets the accumulator to zero */
   JUMP,    /* an unconditional jump */
   CHECK,   /* a shift reduce jump */
   LABEL,   /* */
   NOP,     /* no operation */
   UNDEFINED, /* the default */
   UNKNOWN
 };

enum debugModes
 {
   ANALYSE,
   FULLTRACE,
   STACKTRACE,
   TAPETRACE,
   NONE
 };



/* -----------------------------------------------*/
/* represents a compiled instruction */
typedef struct
{
  enum commandtypes command;
  char argument1[MAXARGUMENTLENGTH];
  char argument2[MAXARGUMENTLENGTH];
  int trueJump;
  int falseJump;
  int isNegated;
} Instruction;

/* -----------------------------------------------*/
void fnPrintClasses()
{
  printf("Character classes for [] tests and the 'while' command \n");
  printf("-a: is an alphanumeric character \n");
  printf("--: is the '-' character \n");
  printf("-n: is a newline character \n");
  printf("-r: is a carriage return character \n");
  printf("-t: is a tab character \n");
  printf("-s: is any space character except a newline \n");
  printf("- : is any space character \n");
  printf("-:: is a punctuation character \n");
  printf("-1: is a digit \n");
}


/* -----------------------------------------------*/
/*  */
int fnIsInClass(char * sClass, char cCharacter)
{
   int ii = 0;

   while (ii < strlen(sClass))
   {
     if (sClass[ii] == '-')
     {

        ii++;
        if (ii == strlen(sClass))
         { return FALSE; }

        /* chars appearing after a '-' in the class string
           are special characters */
        switch(sClass[ii])
        {
           case 'a':
              if (isalpha(cCharacter))
               { return TRUE; }
              break;
           case '-':
              if (cCharacter == '-')
              { return TRUE; }
              break;
           case 'n':
              if (cCharacter == '\n')
              { return TRUE; }
              break;
           case 'r':
              if (cCharacter == '\r')
              { return TRUE; }
              break;
            case 't':
             if (cCharacter == '\t')
              { return TRUE; }
              break;
            case 's':
              if (cCharacter == '\r')
              { return TRUE; }
              if (cCharacter == '\t')
              { return TRUE; }
              if (cCharacter == ' ')
              { return TRUE; }
              break;
            case ' ':
              if (isspace(cCharacter))
               { return TRUE; }
              break;
            case ':':
              if (ispunct(cCharacter))
               { return TRUE; }
              break;
            case '1':
              if (isdigit(cCharacter))
               { return TRUE; }
              break;
            default:
              break;
        } /* switch */
      }
      else
      {
        if (cCharacter == sClass[ii])
          { return TRUE; }

      } /* if */

      ii++;
   } /* for */
   return FALSE;

}

/* ------------------------------------------*/
int fnStringEndsWith(char * sText, char * sSuffix)
{

  /*
  printf("sText=%d \n", sText);
  printf("strlen sText=%d \n", strlen(sText));
  printf("strlen sSuffix=%d \n", strlen(sSuffix));
  printf("strstr %s %s =%d \n", sText, sSuffix, strstr(sText, sSuffix));
  */

  char * pSuffix;
  pSuffix = sText + strlen(sText) - strlen(sSuffix);

  if (strcmp(pSuffix, sSuffix) == 0)
   { return TRUE;  }

  return FALSE;
}

/* ------------------------------------------*/
int fnStringBeginsWith(char * sText, char * sPrefix)
{

  if (strstr(sText, sPrefix) == sText)
   { return TRUE; }

  return FALSE;
}

/* ------------------------------------------*/
int fnStringReplace(char *sText, char * sOld, char * sNew)
{

  if (!strstr(sText, sOld))
    return FALSE;


  return TRUE;
}
/* ------------------------------------------*/
char * fnStringTrim(char * sText)
{

  if (strlen(sText) == 0) return sText;
  while ((sText[strlen(sText) - 1] == '\n') ||
         (sText[strlen(sText) - 1] == '\r') ||
         (sText[strlen(sText) - 1] == '\t') ||
         (sText[strlen(sText) - 1] == ' '))
  {
     sText[strlen(sText) - 1] = '\0';
     if (strlen(sText) == 0) return sText;
  }

  return sText;
}

/* ------------------------------------------*/
char * fnStringIndent(char * sText, int iIndentation)
{
  int ii;
  char sTemp[MAXARGUMENTLENGTH];
  strcpy(sTemp, "  ");
  printf("%s%c  ", sTemp, sText);
  for (ii = 0; ii < strlen(sText); ii++)
  {
    sprintf(sTemp, "%s%c", sTemp, sText[ii]);
    if (sText[ii] == '\n')
    {
      sprintf(sTemp, "%s  ", sTemp);
    }

  } //-- for


  strcpy(sText, sTemp);
  return sText;
}

/* ------------------------------------------*/
char * fnStringClip(char *sText)
{
  sText[strlen(sText) - 2] = '\0';
  return sText;
  // sText.sText + strlen(sText) - 1 =  '\0';
}

/* ------------------------------------------*/
int fnCommandFromString(char * sCommand)
{
  if (strcmp(sCommand, "add") == 0) { return ADD; }
  else if (strcmp(sCommand, "clear") == 0) { return CLEAR; }
  else if (strcmp(sCommand, "crash") == 0) { return CRASH; }
  else if (strcmp(sCommand, "print") == 0) { return PRINT; }
  else if (strcmp(sCommand, "state") == 0) { return STATE; }
  else if (strcmp(sCommand, "replace") == 0) { return REPLACE; }
  else if (strcmp(sCommand, "indent") == 0) { return INDENT; }
  else if (strcmp(sCommand, "clip") == 0) { return CLIP; }
  else if (strcmp(sCommand, "clop") == 0) { return CLOP; }
  else if (strcmp(sCommand, "newline") == 0) { return NEWLINE; }
  else if (strcmp(sCommand, "push") == 0) { return PUSH; }
  else if (strcmp(sCommand, "pop") == 0) { return POP; }
  else if (strcmp(sCommand, "put") == 0) { return PUT; }
  else if (strcmp(sCommand, "get") == 0) { return GET; }
  else if (strcmp(sCommand, "++") == 0) { return INCREMENT; }
  else if (strcmp(sCommand, "--") == 0) { return DECREMENT; }
  else if (strcmp(sCommand, "read") == 0) { return READ; }
  else if (strcmp(sCommand, "until") == 0) { return UNTIL; }
  else if (strcmp(sCommand, "while") == 0) { return WHILE; }
  else if (strcmp(sCommand, "whilenot") == 0) { return WHILENOT; }
  else if (strcmp(sCommand, "count") == 0) { return COUNT; }
  else if (strcmp(sCommand, "plus") == 0) { return INCC; }
  else if (strcmp(sCommand, "minus") == 0) { return DECC; }
  else if (strcmp(sCommand, "jump") == 0) { return JUMP; }
  else if (strcmp(sCommand, "check") == 0) { return CHECK; }
  else if (strcmp(sCommand, "@@@") == 0) { return LABEL; }
  else if (strcmp(sCommand, "zero") == 0) { return ZERO; }
  else if (strcmp(sCommand, "nop") == 0) { return NOP; }
  else { return UNKNOWN; }
}

/* ------------------------------------------*/
char * fnCommandToString(char * sReturn, int iCommand)
{
```

```
  switch (iCommand)
  {
    case ADD:
      strcpy(sReturn, "add");
      break;
    case CLEAR:
      strcpy(sReturn, "clear");
      break;
    case PRINT:
      strcpy(sReturn, "print");
      break;
    case STATE:
      strcpy(sReturn, "state");
      break;
    case REPLACE:
      strcpy(sReturn, "replace");
      break;
    case INDENT:
      strcpy(sReturn, "indent");
      break;
    case CLIP:
      strcpy(sReturn, "clip");
      break;
    case CLOP:
      strcpy(sReturn, "clop");
      break;
    case NEWLINE:
      strcpy(sReturn, "newline");
      break;
    case PUSH:
      strcpy(sReturn, "push");
      break;
    case POP:
      strcpy(sReturn, "pop");
      break;
    case PUT:
      strcpy(sReturn, "put");
      break;
    case GET:
      strcpy(sReturn, "get");
      break;
    case COUNT:
      strcpy(sReturn, "count");
      break;
    case INCREMENT:
      strcpy(sReturn, "++");
      break;
    case DECREMENT:
      strcpy(sReturn, "--");
      break;
    case READ:
      strcpy(sReturn, "read");
      break;
    case UNTIL:
      strcpy(sReturn, "until");
      break;
    case WHILE:
      strcpy(sReturn, "while");
      break;
    case WHILENOT:
      strcpy(sReturn, "while-not");
      break;
    case TESTIS:
      strcpy(sReturn, "testis");
      break;
    case TESTBEGINS:
      strcpy(sReturn, "testbeginswith");
      break;
    case TESTENDS:
      strcpy(sReturn, "testendswith");
      break;
    case TESTCLASS:
      strcpy(sReturn, "testclass");
      break;
    case TESTLIST:
      strcpy(sReturn, "testlist");
      break;
    case TESTEOF:
      strcpy(sReturn, "testeof");
      break;
    case TESTTAPE:
      strcpy(sReturn, "testtape");
      break;
    case INCC:
      strcpy(sReturn, "plus");
      break;
    case DECC:
      strcpy(sReturn, "minus");
      break;
    case CRASH:
      strcpy(sReturn, "crash");
      break;
    case UNDEFINED: /* the default */
      strcpy(sReturn, "undefined");
      break;
    case JUMP:
      strcpy(sReturn, "jump");
      break;
    case CHECK:
      strcpy(sReturn, "check");
      break;
    case LABEL:
      strcpy(sReturn, "label");
      break;
    case NOP:     /* no operation */
      strcpy(sReturn, "nop");
      break;
    case ZERO:     /*  */
      strcpy(sReturn, "zero");
      break;
    case OPENBRACE:
      strcpy(sReturn, "open-brace");
      break;
    case CLOSEBRACE:
      strcpy(sReturn, "close-brace");
      break;
    default:
      strcpy(sReturn, "unknown command");
      break;

  } /* switch */
  return sReturn;
}


/* ------------------------------------------*/
char * fnCommandToDisplayString(char * sReturn, int iCommand)
{
  switch (iCommand)
  {
    case ADD:
      strcpy(sReturn, "add");
      break;
    case CLEAR:
      strcpy(sReturn, "clear");
      break;
    case PRINT:
      strcpy(sReturn, "print");
      break;
    case STATE:
      strcpy(sReturn, "state");
      break;
    case REPLACE:
      //--unimplemented command
      //strcpy(sReturn, "replace");
      strcpy(sReturn, "");
      break;
    case INDENT:
      strcpy(sReturn, "indent");
      break;
    case CLIP:
      strcpy(sReturn, "clip");
      break;
    case CLOP:
      strcpy(sReturn, "clop");
      break;
    case NEWLINE:
      strcpy(sReturn, "newline");
      break;
    case PUSH:
      strcpy(sReturn, "push");
      break;
    case POP:
      strcpy(sReturn, "pop");
      break;
    case PUT:
      strcpy(sReturn, "put");
      break;
    case GET:
      strcpy(sReturn, "get");
      break;
    case COUNT:
      strcpy(sReturn, "count");
      break;
    case INCREMENT:
      strcpy(sReturn, "++");
      break;
    case DECREMENT:
      strcpy(sReturn, "--");
      break;
    case READ:
      strcpy(sReturn, "read");
```

```c
      break;
    case UNTIL:
      strcpy(sReturn, "until");
      break;
    case WHILE:
      strcpy(sReturn, "while");
      break;
    case WHILENOT:
      //unimplemented command
      //strcpy(sReturn, "whilenot");
      strcpy(sReturn, "");
      break;
    case TESTIS:
      strcpy(sReturn, "/text/");
      break;
    case TESTBEGINS:
      strcpy(sReturn, "<text>");
      break;
    case TESTENDS:
      strcpy(sReturn, "(text)");
      break;
    case TESTCLASS:
      strcpy(sReturn, "[text]");
      break;
    case TESTLIST:
      strcpy(sReturn, "=text=");
      break;
    case TESTEOF:
      strcpy(sReturn, "<>");
      break;
    case TESTTAPE:
      strcpy(sReturn, "==");
      break;
    case INCC:
      strcpy(sReturn, "plus");
      break;
    case DECC:
      strcpy(sReturn, "minus");
      break;
    case CRASH:
      strcpy(sReturn, "crash");
      break;
    case JUMP:
      strcpy(sReturn, "jump");
      break;
    case CHECK:
      strcpy(sReturn, "check");
      break;
    case LABEL:
      strcpy(sReturn, "@@@");
      break;
    case NOP:     /* no operation */
      strcpy(sReturn, "nop");
      break;
    case ZERO:     /*  */
      strcpy(sReturn, "zero");
      break;
    case OPENBRACE:
      strcpy(sReturn, "{");
      break;
    case CLOSEBRACE:
      strcpy(sReturn, "}");
      break;
    default:
      strcpy(sReturn, "unknown command");
      break;

  } /* switch */
  return sReturn;

}

/* ------------------------------------------*/
void fnPrintCommands()
{
  char sCommand[100];
  int iCommand;
  strcpy(sCommand, "");

  printf("legal commands: \n   ");

  for (iCommand = 1; iCommand < UNDEFINED; iCommand++)
  {
    fnCommandToDisplayString(sCommand, iCommand);
    printf("%s ", sCommand);
    if (iCommand % 6 == 0)
     { printf("\n   "); }
  }

  printf("\n");
  //printf("* All commands except tests and braces must end with ';' \n"
);
  //printf("* All statement blocks must be enclosed in {} \n");
}

/* ------------------------------------------*/
void fnPrintInstruction(Instruction instruction)
{
  char sCommandName[50] = "";
  fnCommandToString(sCommandName, instruction.command);
  printf("%s '%s' '%s' (True=%d, False=%d, NOT=%d)",
          sCommandName, instruction.argument1, instruction.argument2,
          instruction.trueJump, instruction.falseJump, instruction.isNeg
ated);

}

/* ------------------------------------------*/
void fnPrintScriptInstruction(Instruction instruction)
{
  char sCommandName[50] = "";
  char sDisplay[3 * MAXARGUMENTLENGTH];
  strcpy(sDisplay, "");

  fnCommandToDisplayString(sCommandName, instruction.command);
  switch (instruction.command)
  {
    case ADD:
    case CLEAR:
    case PRINT:
    case STATE:
    case INDENT:
```

```
    case CLIP:
    case CLOP:
    case NEWLINE:
    case PUSH:
    case POP:
    case PUT:
    case GET:
    case COUNT:
    case INCREMENT:
    case DECREMENT:
    case INCC:
    case DECC:
    case CRASH:
    case UNDEFINED: /* the default */
    case CHECK:
    case LABEL:
    case NOP:     /* no operation */
    case ZERO:     /*  */
    case READ:
      strcpy(sDisplay, sCommandName);
      strcpy(sDisplay, ";");
      break;
    case JUMP:
      if (instruction.trueJump != 0)
      {

      }
      break;

    case OPENBRACE:
    case CLOSEBRACE:
      strcpy(sDisplay, sCommandName);
      break;
    case REPLACE:
      break;
    case UNTIL:
      sprintf(sDisplay, "%s '%s' '%s';",
         sCommandName, instruction.argument1, instruction.argument2);

      break;
    case WHILENOT:
      break;
    case WHILE:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "%s !'%s';",
           sCommandName, instruction.argument1);
      }
      else
      {
        sprintf(sDisplay, "%s '%s';",
           sCommandName, instruction.argument1);
      }

     break;
    case TESTIS:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "!/%s/ {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }
      else
      {
         sprintf(sDisplay, "/%s/ {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }

      break;

    case TESTBEGINS:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "!<%s> {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }
      else
      {
         sprintf(sDisplay, "<%s> {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }

      break;
    case TESTENDS:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "!(%s) {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }
      else
      {
         sprintf(sDisplay, "(%s) {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }

      break;

    case TESTCLASS:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "![%s] {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }
      else
      {
         sprintf(sDisplay, "[%s] {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }

      break;

    case TESTLIST:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "!=%s= {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }
      else
      {
         sprintf(sDisplay, "=%s= {=%d }=%d",
           instruction.argument1, instruction.trueJump, instruction.fals
eJump);
      }

      break;

    case TESTEOF:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "!<> {=%d }=%d",
           instruction.trueJump, instruction.falseJump);
      }
      else
      {
         sprintf(sDisplay, "/%s/ {=%d }=%d",
           instruction.trueJump, instruction.falseJump);
      }

      break;
    case TESTTAPE:
      if (instruction.isNegated)
      {
         sprintf(sDisplay, "!== {=%d }=%d",
           instruction.trueJump, instruction.falseJump);
      }
      else
      {
         sprintf(sDisplay, "== {=%d }=%d",
           instruction.trueJump, instruction.falseJump);
      }

      break;
    default:
      break;

  } //-- switch

  printf("%s", sDisplay);

}


/* ------------------------------------------*/
void fnInitializeInstruction(Instruction * instruction)
{

  instruction->command = UNDEFINED;
  strcpy(instruction->argument1, "");
  strcpy(instruction->argument2, "");
  instruction->trueJump = -1;
  instruction->falseJump = -1;
  instruction->isNegated = FALSE;

}
```

```
/* ------------------------------------------*/
typedef struct
{
  Instruction instructionSet[MAXPROGRAMLENGTH + 1];
  int size;
  int braceStack[MAXNESTING + 1];
  int instructionPointer;
  int compileTime;
  int executionTime;
  char listFile[MAXARGUMENTLENGTH];
  int fileError;
} Program;



/* ------------------------------------------*/
void fnInitializeProgram(Program * program)
{
  int ii;
  program->size = 0;
  program->instructionPointer = 0;
  program->compileTime = -1;
  program->executionTime = -1;
  strcpy(program->listFile, "");
  program->fileError = FALSE;

  for (ii = 0; ii < MAXPROGRAMLENGTH; ii++)
  {
    fnInitializeInstruction(&program->instructionSet[ii]);
  }

  for (ii = 0; ii < MAXNESTING; ii++)
  {
    program->braceStack[ii] = -1;
  }
}

/* ------------------------------------------*/

void fnPrintProgram(Program * program)
{

  int ii;

  printf("IP=%d: Size=%d \n", program->instructionPointer, program->size
);
  printf("Maximum program length  =%d \n", MAXPROGRAMLENGTH);
  printf("Maximum nesting of '{'  =%d \n", MAXNESTING);
  printf("Maximum argument length =%d \n", MAXARGUMENTLENGTH);
  printf("Maximum tape length     =%d \n", MAXTAPELENGTH);
  printf("Compilation time (msec) =%d \n", program->compileTime);
  printf("Execution time          =%d \n", program->executionTime);
  printf("List file name          =%s \n", program->listFile);
  printf("List file error         =%d \n", program->fileError);
  printf("Brace stack=(");
  for (ii = 0; ii < MAXNESTING - 1; ii++)
  {
    printf("%d, ", program->braceStack[ii]);
  }
  printf("%d)\n", program->braceStack[ii]);
```

```c
  for (ii = 0; ii < program->size; ii++)
  {

    if (ii == program->instructionPointer)
      { printf("> "); }
    else
      { printf("  "); }

    printf("%d:", ii);
    fnPrintInstruction(program->instructionSet[ii]);
    printf("\n");
  }

}


/* ------------------------------------------*/
typedef struct
{
  char * text;
  int size;
} Element;

/* ------------------------------------------*/
typedef struct
{
  int peep; /* may contain EOF */
  Element tape[MAXTAPELENGTH + 1];
  Element * tapepointer;
  char * stack;
  char * workspace;
  int counter;
  int stacklength;
  enum commandtypes lastoperation;
  int stacksize;
} Machine;

/* ------------------------------------------*/
Machine * fnInitializeMachine(Machine * machine)
{
  machine->peep = '\0';
  machine->stack = (char *) malloc(sizeof(char) * INITIALSTACKSIZE);
  machine->workspace = machine->stack;
  strcpy(machine->stack, "");
  machine->counter = 0;
  machine->lastoperation = UNDEFINED;
  machine->stacksize = 0;
  machine->stacklength = INITIALSTACKSIZE;
  int ii = 0;


  machine->tapepointer = &machine->tape[0];
  Element * pElement = &machine->tape[0];
  for (ii = 0; ii < MAXTAPELENGTH + 1; ii++)
  {
    pElement->text = (char *) malloc(sizeof(char) * GROWFACTOR);
    strcpy(pElement->text, "");
    pElement->size = GROWFACTOR;
    pElement++;
  }
} //--

/* ------------------------------------------*/
void fnDestroyMachine(Machine * machine)
{
  free(machine->stack);
  Element * pElement;
  int ii = 0;
  for (ii = 0; ii < MAXTAPELENGTH; ii++)
  {
    pElement = &machine->tape[ii];
    free(pElement->text);
  }

} //-- fnDestroyMachine

/* ------------------------------------------*/
Machine * appendToWorkspace(Machine * machine, char * sText)
{
   int iDifference;
   if ((strlen(machine->stack) + strlen(sText)) >= machine->stacklength)
   {
     iDifference = machine->workspace - machine->stack;
     machine->stacklength = machine->stacklength + strlen(sText) + GROWF
ACTOR;
     machine->stack = (char *) realloc(machine->stack, machine->stacklen
gth * sizeof(char));
     machine->workspace = machine->stack + iDifference;
   }

   if (machine->stack == NULL)
   {
     printf ("\nError reallocating memory for the stack/workspace \n");
     exit (1);
   }

   strcat(machine->workspace, sText);
   return machine;
}

/* ------------------------------------------*/
Machine * indentWorkspace(Machine * machine, int iIndentation)
{

}
/* ------------------------------------------*/
void fnPrintStackTape(Machine * machine)
{
  char sText[30] = "";
  char * pp;
  pp = machine->stack;

  /* ---
  fnCommandToString(sText, machine->lastoperation);
  printf("  -last       :%s\n", sText);
  printf("  -counter    :[%d]\n", machine->counter);
  if (machine->peep == 0)
   { printf("  -peep       :[\\0] <ascii:0> \n", machine->peep, machine-
>peep); }
  else
   { printf("  -peep       :[%c] <ascii:%d> \n", machine->peep, machine-
>peep); }
```

```
  printf("  -stack      :[");
  while (pp != machine->workspace)
  {
    putchar(*pp);
    pp++;
  }
  printf("]\n");
  */

  printf("  -workspace  :[%s]\n", machine->workspace);
  printf("  -stack size :%d\n", machine->stacksize);

  Element * ee;
  int iCount = 0;
  ee = &machine->tape[0];
  printf("\n Tape \n");
  while ((iCount < MAXTAPELENGTH) && (iCount < (machine->stacksize + 4))
)
  {
    printf("<");
    while ((*pp != '*') && (pp < machine->workspace))
    {
      putchar(*pp);
      pp++;
    }

    if (*pp == '*')
     { putchar(*pp); pp++; printf(">"); putchar('\n'); }



    if (ee == machine->tapepointer)
      { printf(">%d:", iCount); }
    else
      { printf(" %d:", iCount); }

    printf("%s\n", ee->text);
    ee++;
    iCount++;
  } //--while

  if (iCount == MAXTAPELENGTH)
  {
    printf("maximum tape length = %d \n", MAXTAPELENGTH);
  }


} //-- fnPrintMachineState


/* -------------------------------------------*/
void fnPrintMachineState(Machine * machine)
{
  char sText[30] = "";
  fnCommandToString(sText, machine->lastoperation);
  printf("  -last       :%s\n", sText);
  printf("  -counter    :[%d]\n", machine->counter);
  if (machine->peep == 0)
   { printf("  -peep       :[\\0] <ascii:0> \n", machine->peep, machine-
>peep); }
  else
   { printf("  -peep       :[%c] <ascii:%d> \n", machine->peep, machine-
>peep); }

  printf("  -workspace  :[%s]\n", machine->workspace);
  char * pp;
  printf("  -stack      :[");
  pp = machine->stack;
  while (pp != machine->workspace)
  {
    putchar(*pp);
    pp++;
  }
  printf("]\n");
  printf("  -stack size :%d\n", machine->stacksize);
  // printf("strlen(machine->stack)     :[%d] \n", strlen(machine->stack
));
  // printf("strlen(machine->workspace) :[%d] \n", strlen(machine->works
pace));

  Element * ee;
  int iCount = 0;
  ee = &machine->tape[0];
  printf("\n Tape \n");
  while ((iCount < MAXTAPELENGTH) && (iCount < (machine->stacksize + 4))
)
  {
    if (ee == machine->tapepointer)
      { printf(">%d:", iCount); }
    else
      { printf(" %d:", iCount); }

    printf("%s\n", ee->text);
    ee++;
    iCount++;
  } //--while

  if (iCount == MAXTAPELENGTH)
  {
    printf("maximum tape length = %d \n", MAXTAPELENGTH);
  }


} //-- fnPrintMachineState


/* -------------------------------------------*/
void fnPrintMachine(Machine * machine)
{
  char sText[30] = "";
  fnCommandToString(sText, machine->lastoperation);
  printf("  -last       :%s\n", sText);
  printf("  -counter    :[%d]\n", machine->counter);
  if (machine->peep == 0)
  {
    printf("  -peep       :[\\0] <ascii:0> \n", machine->peep, machine-
>peep);
  }
  else
  {
    printf("  -peep       :[%c] <ascii:%d> \n", machine->peep, machine-
>peep);
  }

  printf("  -workspace   :[%s]\n", machine->workspace);
  char * pp;
  printf("  -stack <%d>  :[", machine->stacklength);
  pp = machine->stack;
  while (pp != machine->workspace)
  {
    putchar(*pp);
    pp++;
  }
  printf("]\n");
  printf("  -stack size:%d\n", machine->stacksize);
  // printf("strlen(machine->stack)     :[%d] \n", strlen(machine->stack
));
  // printf("strlen(machine->workspace) :[%d] \n", strlen(machine->works
pace));

  Element * ee;
  int iCount = 0;
  ee = &machine->tape[0];
  printf("\n Tape \n");
  while ((iCount < MAXTAPELENGTH) && (iCount < (machine->stacksize + 4))
)
  {
    if (ee == machine->tapepointer)
     { printf(">%d <%d>:", iCount, ee->size); }
    else
     { printf(" %d <%d>:", iCount, ee->size); }

    printf("%s \n", ee->text);
    ee++;
    iCount++;
  } //--while

  if (iCount == MAXTAPELENGTH)
  {
    printf("maximum tape length = %d \n", MAXTAPELENGTH);
  }


} //-- fnPrintMachine


/* ------------------------------------------*/
void fnCompile(Program * program, FILE * inputstream)
{
   int iCharacter;
   int iLabelLine = -1;
   int iLineMark = 1;
   int iLineCharacterMark = 1;
   int iLineCount = 1;
   int iCharacterCount = 1;
   int iLineCharacterCount = 1;
   int iOpenBraceCount = 0;
   int iCloseBraceCount = 0;
   int iTextLength = 0;
   int iTestPointer = 0;
   int iCommand = 0;
   clock_t tBeginCompile;
   clock_t tEndCompile;
   /* pointer into the program.braceStack array */
   int * pBraceStackPointer;

   char sText[TEXTBUFFERSIZE];

   char sCommandName[20] = "";

   //FILE * inputstream = stdin;
   //Program program;
   //fnInitializeProgram(&program);

   tBeginCompile = clock();
   pBraceStackPointer = &program->braceStack[0];
   Instruction * instruction = &program->instructionSet[0];


   iCharacter = getc(inputstream);

   while (iCharacter != EOF)
   {

      if (program->size >= MAXPROGRAMLENGTH - 1)
      {

         fprintf(stderr, "line %d: the maximum number of script statemen
ts \n",
                iLineCount);
         fprintf(stderr, "(%d) is exceeded. This may be remedied \n",
                MAXPROGRAMLENGTH);
         fprintf(stderr, "by changing the MAXPROGRAMLENGTH constant \n")
;
         fprintf(stderr, "in the file 'library.c' and recompiling. \n");
         exit(2);
      }

     switch (iCharacter)
     {
       /*----------------------------------------*/
       case '"':
         iLineMark = iLineCount;
         switch (instruction->command)
         {
           case UNDEFINED:
             fprintf(stderr, "misplaced quote (\"): line %d, char %d",
               iLineCount, iLineCharacterCount);
             exit(2);
           case CLEAR:
           case CLIP:
           case CLOP:
           case CRASH:
           case POP:
           case PUSH:
           case PUT:
           case GET:
           case INDENT:
           case INCREMENT:
           case DECREMENT:
           case INCC:
           case DECC:
           case NEWLINE:
```

```
          case READ:
          case TESTIS:
          case TESTBEGINS:
          case TESTCLASS:
          case TESTLIST:
          case TESTEOF:
          case STATE:
          case NOP:
          case JUMP:
          case ZERO:
            fnCommandToString(sCommandName, instruction->command);
            fprintf(stderr,
             "\n Syntax error: Command %s cannot take an argument: line
 %d, char %d",
             sCommandName, iLineCount, iLineCharacterCount);
            exit(2);
        } //-- switch

        strcpy(sText, "");
        iTextLength = 0;
        iCharacter = getc(inputstream);
        iCharacterCount++;
        if (iCharacter == EOF)
        {
          fprintf(stderr,
            "stray quote (\") at line %d, char %d \n", iLineCount, iLin
eCharacterCount);
          exit(2);
        }

        if (iCharacter == '"')
        {
          fprintf(stderr,
           "\n Script syntax error: empty quotes (\"\") at line %d, cha
r %d \n",
           iLineCount, iLineCharacterCount);
          exit(2);
        }

        while ((iCharacter != EOF) && (iCharacter != '"') &&
               (iTextLength < MAXARGUMENTLENGTH))
        {
          sprintf(sText, "%s%c", sText, iCharacter);
          iTextLength++;
          iCharacter = getc(inputstream);
          if (iCharacter == '\n')
          { iLineCount++; iLineCharacterCount = 1; }
          iCharacterCount++;
        }

        if (iCharacter == EOF)
        {
          fprintf(stderr, "unterminated quote (\") starting at line %d,
 char %d \n",
            iLineMark, iLineCharacterMark);
          exit(2);
        }

        if (iTextLength >= MAXARGUMENTLENGTH)
        {
          fprintf(stderr,
            "\n Script error: the argument (text between quotes) at lin
e %d, char %d \n",
            iLineMark, iLineCharacterMark);
          fprintf(stderr, "is too long. The maximum is %d characters \n
",
            MAXARGUMENTLENGTH);
          exit(2);
        }

        if (iCharacter == '"')
        {
          if (strlen(instruction->argument1) == 0)
            { strcpy(instruction->argument1, sText); }
          else if (strlen(instruction->argument2) == 0)
            { strcpy(instruction->argument2, sText); }
          else
          {
            fprintf(stderr, "The instruction at line %d has too many ar
guments \n");
            fprintf(stderr, "The maximum permitted is 2. \n");
            exit(2);
          }
        }
        else
        {
          fprintf(stderr, "error parsing quoted text at line %d. \n",
            iLineCount, iLineCharacterCount);
          fprintf(stderr, "this error indicates a bug in the code 'libr
ary.c' \n");
          exit(2);
        }
        break;
      /*----------------------------------------*/
      case '\'':
        iLineMark = iLineCount;
        iLineCharacterMark = iLineCharacterCount;
        switch (instruction->command)
        {
          case UNDEFINED:
            fprintf(stderr,
              "\n script syntax error: misplaced quote ('): line %d, ch
ar %d",
              iLineCount, iLineCharacterCount);
            exit(2);
          case CLEAR:
          case CLIP:
          case CLOP:
          case CRASH:
          case POP:
          case PUSH:
          case PUT:
          case GET:
          case INDENT:
          case INCREMENT:
          case DECREMENT:
          case INCC:
          case DECC:
          case NEWLINE:
          case READ:
          case TESTIS:
          case TESTBEGINS:
```

```
          case TESTENDS:
          case TESTCLASS:
          case TESTLIST:
          case TESTEOF:
          case STATE:
          case NOP:
          case JUMP:
          case CHECK:
          case ZERO:
            fnCommandToString(sCommandName, instruction->command);
            fprintf(stderr, "syntax error: command %s cannot take an ar
gument: line %d",
              sCommandName, iLineCount, iLineCharacterCount);
            exit(2);
        }
        strcpy(sText, "");
        iTextLength = 0;
        iCharacter = getc(inputstream);
        iCharacterCount++;
        if (iCharacter == EOF)
        {
          fprintf(stderr,
            "\n Script syntax error: stray quote (’) at line %d, char %
d \n",
            iLineCount, iLineCharacterCount);
          exit(2);
        }

        if (iCharacter == ’\’’)
        {
          fprintf(stderr,
             "\n Script syntax error: empty quotes (’’) at line %d, cha
r %d \n",
             iLineCount, iLineCharacterCount);
          exit(2);
        }

        while ((iCharacter != EOF) &&
               (iCharacter != ’\’’) &&
               (iTextLength < MAXARGUMENTLENGTH))
        {
          sprintf(sText, "%s%c", sText, iCharacter);
          iTextLength++;
          iCharacter = getc(inputstream);
          if (iCharacter == ’\n’)
          {
            iLineCount++;
            iLineCharacterCount = 1;
          }
          iCharacterCount++;
        }

        if (iCharacter == EOF)
        {
          fprintf(stderr,
            "\n Script syntax error: unterminated quote (’) at line %d,
 char %d \n",
            iLineCount, iLineCharacterCount);
          exit(2);
        }

        if (iTextLength >= MAXARGUMENTLENGTH)
        {
          fprintf(stderr,
            "\n Script error: the argument (text between quotes) at lin
e %d, char %d \n",
            iLineCount, iLineCharacterCount);
          fprintf(stderr, "is too long. The maximum is %d characters \n
",
            MAXARGUMENTLENGTH);
          exit(2);
        }

        if (iCharacter != ’\’’)
        {
          fprintf(stderr,
            "error parsing quoted text at line %d. \n",
            iLineCount, iLineCharacterCount);
          fprintf(stderr,
            "this error indicates a bug in the code ’library.c’ near li
ne 600 \n");
          exit(2);
        }

        if (iCharacter == ’\’’)
        {
          if (strlen(instruction->argument1) == 0)
            { strcpy(instruction->argument1, sText); }
          else if (strlen(instruction->argument2) == 0)
            { strcpy(instruction->argument2, sText); }
          else
          {
            fprintf(stderr,
              "\n Script syntax error: The instruction at line %d, char
 %d has too many arguments (2 is the maximum number)\n",
              iLineCount, iLineCharacterCount);
            fprintf(stderr, "The maximum permitted is 2. \n");
            exit(2);
          }
        } //-- if

        break;

      /*------------------------------------------*/
      //-- ignore comments in the script

      case ’#’:
        iLineMark = iLineCount;
        iLineCharacterMark = iLineCharacterCount;
        strcpy(sText, "");
        iTextLength = 0;
        iCharacter = getc(inputstream);
        iCharacterCount++;
        if (iCharacter == EOF)
        {
          fprintf(stderr,
            "syntax error: unterminated comment ’#...#’  at at end of s
cript, line %d, char %d \n",
            iLineCount, iLineCharacterCount);
          exit(2);
        }
```

```
         if (iCharacter == '#')
         {
           break;
         }

         while ((iCharacter != EOF) && (iCharacter != '#'))
         {
           iCharacter = getc(inputstream);
           if (iCharacter == '\n')
           {
             iLineCount++;
             iLineCharacterCount = 1;
           }
           iCharacterCount++;
         }

         if (iCharacter == EOF)
         {
           fprintf(stderr,
              "script error: unterminated comment (#..#) starting at lin
e %d, char %d \n",
              iLineMark, iLineCharacterMark);
           exit(2);
         }

         if (iCharacter != '#')
         {
           fprintf(stderr, "error parsing comment at line %d, char %d. \
n", iLineMark, iLineCharacterMark);
           fprintf(stderr, "this error indicates a bug in the code 'libr
ary.c' near line 700 \n");
           exit(2);
         }

         break;


       /*-----------------------------------------*/
       // ignore whitespace
       case '\r':
       case '\t':
       case ' ': break;
       /*-----------------------------------------*/
       // parse 'begin tests'  <...>
       case '<':
         switch(instruction->command)
         {
           case UNDEFINED:
             break;
           default:
             fprintf(stderr,
               "Line %d, char %d: script error before '<' character. \n"
,
               iLineCount, iLineCharacterCount);
             fprintf(stderr, "(missing semi-colon?)\n");
             exit(2);
         }
         iLineMark = iLineCount;
         iLineCharacterMark = iLineCharacterCount;
         strcpy(sText, "");
         iTextLength = 0;
         iCharacter = getc(inputstream);
         iCharacterCount++;
         if (iCharacter == EOF)
         {
           fprintf(stderr,
             "script ends badly, unterminated test '<...>' starting at l
ine %d, char %d \n",
             iLineMark, iLineCharacterMark);
           exit(2);
         }

         //-- End of file test can be written '<>'
         if (iCharacter == '>')
         {
           if (strlen(instruction->argument1) != 0)
           {
             fprintf(stderr,
               "syntax error: The eof test '<>' at line %d, char %d alre
ady has an argument \n",
               iLineMark, iLineCharacterMark);
             exit(2);
           }

           instruction->command = TESTEOF;
           program->size++;
           instruction++;
           break;
         }

         while ((iCharacter != EOF) && (iCharacter != '>') && (iTextLeng
th < MAXARGUMENTLENGTH))
         {
           /* handle the escape sequence */
           if (iCharacter == '\\')
           {
             iCharacter = getc(inputstream);
             if (iCharacter == EOF)
             {
               fprintf(stderr,
                 "script ends badly: unterminated test '<...>', and back
slash starting at line %d, char %d \n",
                 iLineMark, iLineCharacterMark);
               exit(2);
             }
           }

           sprintf(sText, "%s%c", sText, iCharacter);
           iTextLength++;
           iCharacter = getc(inputstream);
           if (iCharacter == '\n')
           {
             iLineCount++;
             iLineCharacterCount = 1;
           }
           iCharacterCount++;
         }

         if (iCharacter == EOF)
         {
           fprintf(stderr,
             "unterminated test '<...>' starting at line %d, char %d \n"
,
             iLineMark, iLineCharacterMark);
           exit(2);
         }

         if (iTextLength >= MAXARGUMENTLENGTH)
         {
           fprintf(stderr, "the test '<...>' starting at line %d, char %
d \n", iLineMark, iLineCharacterMark);
           fprintf(stderr, "is too long. The maximum is %d characters \n
", MAXARGUMENTLENGTH);
           exit(2);
         }

         if (iCharacter != '>')
         {
           fprintf(stderr, "error parsing test at line %d, char %d. \n",
              iLineCount, iLineCharacterCount);
           fprintf(stderr, "code bug near line 740 of library.c \n");
           exit(2);
         }

         if (iCharacter == '>')
         {
           if (strlen(instruction->argument1) == 0)
           {
             instruction->command = TESTBEGINS;
             strcpy(instruction->argument1, sText);

             program->size++;
             instruction++;
           }
           else
           {
             fprintf(stderr, "The test '<...>' at line %d, char %d alrea
dy has an argument \n",
               iLineMark, iLineCharacterMark);
             fprintf(stderr, "code bug near line 740 of library.c \n");
             exit(2);
           }
         }
         break;
       /*-------------------------------------------*/
       // parse 'ends tests'  (...)
       case '(':
         switch(instruction->command)
         {
           case UNDEFINED:
             break;
           default:
             fprintf(stderr,
               "Line %d, char %d: script error before '(' character. \n"
,
               iLineCount, iLineCharacterCount);
             fprintf(stderr, "(missing semi-colon?)\n");
             exit(2);
         }
         iLineMark = iLineCount;
         iLineCharacterMark = iLineCharacterCount;
         strcpy(sText, "");
         iTextLength = 0;
         iCharacter = getc(inputstream);
         iCharacterCount++;
         if (iCharacter == EOF)
         {
           fprintf(stderr,
             "script ends badly, unterminated test '(...)' starting at l
ine %d, char %d \n",
             iLineMark, iLineCharacterMark);
           exit(2);
         }

         //-- some test can be written '()'
         if (iCharacter == ')')
         {
           fprintf(stderr,
             "empty test '()' at line %d, char %d \n", iLineMark, iLineC
haracterMark);
           exit(2);
         }

         while ((iCharacter != EOF) && (iCharacter != ')') && (iTextLeng
th < MAXARGUMENTLENGTH))
         {
           /* handle the escape sequence */
           if (iCharacter == '\\')
           {
             iCharacter = getc(inputstream);
             if (iCharacter == EOF)
             {
               fprintf(stderr,
                 "script ends badly: unterminated test '(...)', and back
slash starting at line %d, char %d \n",
                 iLineMark, iLineCharacterMark);
               exit(2);
             }
           }

           sprintf(sText, "%s%c", sText, iCharacter);
           iTextLength++;
           iCharacter = getc(inputstream);
           if (iCharacter == '\n')
           {
             iLineCount++;
             iLineCharacterCount = 1;
           }
           iCharacterCount++;
         }

         if (iCharacter == EOF)
         {
           fprintf(stderr,
             "unterminated test '(...)' starting at line %d, char %d \n"
,
             iLineMark, iLineCharacterMark);
           exit(2);
         }

         if (iTextLength >= MAXARGUMENTLENGTH)
         {
           fprintf(stderr, "the test '(...)' starting at line %d, char %
d \n", iLineMark, iLineCharacterMark);
```

```
          fprintf(stderr, "is too long. The maximum is %d characters \n
", MAXARGUMENTLENGTH);
          exit(2);
        }

        if (iCharacter != ')')
        {
          fprintf(stderr, "error parsing test at line %d, char %d. \n",
             iLineCount, iLineCharacterCount);
          fprintf(stderr, "code bug near line 740 of library.c \n");
          exit(2);
        }

        if (iCharacter == ')')
        {
          if (strlen(instruction->argument1) == 0)
          {
            instruction->command = TESTENDS;
            strcpy(instruction->argument1, sText);

            program->size++;
            instruction++;
          }
          else
          {
            fprintf(stderr, "The test '(...)' at line %d, char %d alrea
dy has an argument \n",
               iLineMark, iLineCharacterMark);
            fprintf(stderr, "code bug near line 1312 of library.c \n");
            exit(2);
          }
        }
        break;
      /*----------------------------------------*/
      //-- parse 'class tests'
      case '[':
        switch(instruction->command)
        {
          case UNDEFINED:
            break;
          default:
            fprintf(stderr,
               "Line %d, char %d: syntax error before '[' character. \n
",
               iLineCount, iLineCharacterCount);
            fprintf(stderr, "(missing semi-colon?)\n");
            exit(2);
        }
        iLineMark = iLineCount;
        iLineCharacterMark = iLineCharacterCount;
        strcpy(sText, "");
        iTextLength = 0;
        iCharacter = getc(inputstream);
        iCharacterCount++;
        if (iCharacter == EOF)
        {
          fprintf(stderr, "script ends badly, unterminated test \n");
          exit(2);
        }

        if (iCharacter == ']')
        {
          fprintf(stderr,
            "empty test '[]' at line %d, char %d \n", iLineMark, iLineC
haracterMark);
          exit(2);
        }

        while ((iCharacter != EOF) && (iCharacter != ']') && (iTextLeng
th < MAXARGUMENTLENGTH))
        {
          /* handle the escape sequence */
          if (iCharacter == '\\')
          {
            iCharacter = getc(inputstream);
            if (iCharacter == EOF)
            {
              fprintf(stderr,
               "script ends badly: unterminated test, and backslash sta
rting at line %d, char %d",
               iLineMark, iLineCharacterMark);
              exit(2);
            }
          }

          sprintf(sText, "%s%c", sText, iCharacter);
          iTextLength++;
          iCharacter = getc(inputstream);
          if (iCharacter == '\n')
          {
             iLineCount++;
             iLineCharacterCount = 1;
          }
          iCharacterCount++;
        }

        if (iCharacter == EOF)
        {
          fprintf(stderr, "unterminated test '[]' starting at line %d,
char \n", iLineMark, iLineCharacterCount);
          exit(2);
        }

        if (iTextLength >= MAXARGUMENTLENGTH)
        {
          fprintf(stderr,
            "script error: the class test '[...]' starting at line %d,
char %d \n",
            iLineMark, iLineCharacterMark);
          fprintf(stderr, "is too long. The maximum is %d characters \n
", MAXARGUMENTLENGTH);
          fprintf(stderr, "This limit can be changed by editing the val
ue ");
          fprintf(stderr, "of MAXARGUMENTLENGTH in library.c and recomp
iling  \n");
          exit(2);
        }

        if (iCharacter == ']')
        {
          if (strlen(instruction->argument1) == 0)
          {
            instruction->command = TESTCLASS;
            strcpy(instruction->argument1, sText);
            program->size++;
            instruction++;
          }
          else
          {
            fprintf(stderr,
              "The test '[...]' starting at line %d, char %d already ha
s an argument \n",
              iLineMark, iLineCharacterMark);
            fprintf(stderr, "This indicates a code bug near line 820 of
 library.c \n");
            exit(2);
          }
        }
        else
        {
          fprintf(stderr, "error parsing test at line %d. \n", iLineCou
nt, iLineCharacterCount);
          fprintf(stderr, "code bug near line 820 of library.c \n");
          exit(2);
        }
        break;
      /*-----------------------------------------*/
      case '=':
        switch(instruction->command)
        {
          case UNDEFINED:
            break;
          default:
            fprintf(stderr, "Line %d, char %d: syntax error before '='
character. \n",
              iLineCount, iLineCharacterCount);
            fprintf(stderr, "(missing semi-colon?)\n");
            exit(2);
        }

        iLineMark = iLineCount;
        iLineCharacterMark = iLineCharacterCount;

        strcpy(sText, "");
        iTextLength = 0;
        iCharacter = getc(inputstream);
        iCharacterCount++;
        if (iCharacter == EOF)
        {
          fprintf(stderr, "The '=' at line %d, char %d, seems misplaced
 \n", iLineMark, iLineCharacterMark);
          exit(2);
        }

        /* the test == is used to determine if the workspace is
           the same as the current tape cell */
        if (iCharacter == '=')
        {
          if (strlen(instruction->argument1) != 0)
          {
            fprintf(stderr,
              "syntax error: The tape test '==' at line %d, char %d alr
eady has an argument \n",
              iLineMark, iLineCharacterMark);
            exit(2);
          }

          instruction->command = TESTTAPE;
          program->size++;
          instruction++;
          break;
        }

        while ((iCharacter != EOF) && (iCharacter != '=') && (iTextLeng
th < MAXARGUMENTLENGTH))
        {
          /* handle the escape sequence */
          if (iCharacter == '\\')
          {
            iCharacter = getc(inputstream);
            if (iCharacter == EOF)
            {
              fprintf(stderr, "unterminated test (=...=), and backslash
 starting at line %d, char %d",
                iLineMark, iLineCharacterMark);
              exit(2);
            }
          }

          sprintf(sText, "%s%c", sText, iCharacter);
          iTextLength++;
          iCharacter = getc(inputstream);
          if (iCharacter == '\n')
           { iLineCount++; }
          iCharacterCount++;
        }

        if (iCharacter == EOF)
        {
          fprintf(stderr, "unterminated test (=...=) at line %d, char %
d \n",
             iLineMark, iLineCharacterCount);
          exit(2);
        }

        if (iTextLength >= MAXARGUMENTLENGTH)
        {
          fprintf(stderr, "the test (==) at line %d, char %d \n", iLine
Mark, iLineCharacterMark);
          fprintf(stderr, "is too long. The maximum is %d characters \n
", MAXARGUMENTLENGTH);
          exit(2);
        }

        if (iCharacter != '=')
        {
          fprintf(stderr, "error parsing test at line %d, char %d\n", i
LineMark, iLineCharacterMark); exit(2);
          fprintf(stderr, "this error indicates a bug in the code 'libr
ary.c' near line 1160 \n");
          exit(2);
        }

        if (strlen(instruction->argument1) == 0)
        {
            instruction->command = TESTLIST;
            strcpy(instruction->argument1, sText);
            program->size++;
            instruction++;
        }
        else
        {
            fprintf(stderr, "syntax error: The test (==) at line %d alr
eady has an argument \n",
                    iLineMark);
            fprintf(stderr, " \n");
            exit(2);
        }
        break;
      /*-------------------------------------------*/
      case '/':
        switch(instruction->command)
        {
          case UNDEFINED:
            break;
          default:
            fprintf(stderr,
               "Line %d, char %d: syntax error before '/' character. \n
",
                iLineCount, iLineCharacterCount);
            fprintf(stderr, "(missing semi-colon?)\n");
            exit(2);
        }

        iLineMark = iLineCount;
        iLineCharacterMark = iLineCharacterCount;

        strcpy(sText, "");
        iTextLength = 0;
        iCharacter = getc(inputstream);
        iCharacterCount++;
        if (iCharacter == EOF)
        {
          fprintf(stderr, "The '/' at line %d, char %d, seems misplaced
 \n", iLineMark, iLineCharacterMark);
          exit(2);
        }

        /*
        if (iCharacter == '/')
        {
          fprintf(stderr, "empty test (//) at line %d, char %d \n", iLi
neMark, iLineCharacterMark);
          exit(2);
        }
        */

        while ((iCharacter != EOF) && (iCharacter != '/') && (iTextLeng
th < MAXARGUMENTLENGTH))
        {
          /* handle the escape sequence */
          if (iCharacter == '\\')
          {
            iCharacter = getc(inputstream);
            if (iCharacter == EOF)
            {
              fprintf(stderr, "unterminated test, and backslash startin
g at line %d, char %d",
                iLineMark, iLineCharacterMark);
              exit(2);
            }
          }

          sprintf(sText, "%s%c", sText, iCharacter);
          iTextLength++;
          iCharacter = getc(inputstream);
          if (iCharacter == '\n')
           { iLineCount++; }
          iCharacterCount++;
        }

        if (iCharacter == EOF)
        {
          fprintf(stderr, "unterminated test (//) at line %d, char %d \
n", iLineMark, iLineCharacterCount);
          exit(2);
        }

        if (iTextLength >= MAXARGUMENTLENGTH)
        {
          fprintf(stderr,
            "the test (//) at line %d, char %d \n",
            iLineMark, iLineCharacterMark);
          fprintf(stderr, "is too long. The maximum is %d characters \n
",
            MAXARGUMENTLENGTH);
          exit(2);
        }

        if (iCharacter == '/')
        {
          if (strlen(instruction->argument1) == 0)
          {
            instruction->command = TESTIS;
            strcpy(instruction->argument1, sText);
            program->size++;
            instruction++;
          }
          else
          {
            fprintf(stderr, "The test (//) at line %d already has an ar
gument \n",
                    iLineMark);
            fprintf(stderr, " \n");
            exit(2);
          }
        }
        else
        {
          fprintf(stderr, "error parsing test at line %d, char %d\n", i
LineMark, iLineCharacterMark); exit(2);
          fprintf(stderr, "this error indicates a bug in the code 'libr
ary.c' \n");
          exit(2);
        }
        break;
      /*----------------------------------------*/
      case '\n':
        iLineCount++;
        iLineCharacterCount = 1;
        break;
      /*----------------------------------------*/
      case '!': //negations only before tests or a while command
        switch(instruction->command)
        {
          case UNDEFINED:
            if (instruction->isNegated == TRUE)
              { instruction->isNegated = FALSE; }
            else if (instruction->isNegated == FALSE)
              { instruction->isNegated = TRUE; }
            break;
          case WHILE:
            if (instruction->isNegated == TRUE)
              { instruction->isNegated = FALSE; }
            else if (instruction->isNegated == FALSE)
              { instruction->isNegated = TRUE; }
            break;
         default:
            fprintf(stderr,
              "Line %d, char %d: syntax error before '!' character. \n"
,
              iLineCount, iLineCharacterCount);
            fprintf(stderr, "\n");
            exit(2);
        }
        break;
      /*----------------------------------------*/
      case ';':
        switch (instruction->command)
        {
          case UNDEFINED:
            fprintf(stderr,
             "The semi-colon (;) at line %d, char %d seems misplaced. \
n",
             iLineCount, iLineCharacterCount);
            exit(2);
          case ADD:
          case WHILE:
          case UNTIL:
            if (strlen(instruction->argument1) == 0)
            {
              fnCommandToString(sCommandName, instruction->command);
              fprintf(stderr,
                 "The command '%s' requires an argument: line %d, char
%d \n",
                  sCommandName, iLineCount, iLineCharacterCount);
              exit(2);
            }
            program->size++;
            instruction++;
            break;
          case LABEL:
            iLabelLine = program->size;
            program->size++;
            instruction++;
            break;
          case CHECK:
            //-- convert 'checks' to 'jumps' and set the jump line
            instruction->command = JUMP;
            if (iLabelLine == -1)
            {
              fprintf(stderr,
                 "The check must be preceded by the '@@@' label: line %
d, char %d \n",
                  iLineCount, iLineCharacterCount);
              exit(2);
            }
            instruction->trueJump = iLabelLine;
            program->size++;
            instruction++;
            break;
          default:
            program->size++;
            instruction++;
        } // switch
        break;
      /*----------------------------------------*/
      case '{':
        // assign jumps
        if (instruction->command != UNDEFINED)
        {
          fprintf(stderr,
            "Line %d, char %d: syntax error before '{' \n", iLineCount,
 iLineCharacterCount);
          exit(2);
        }
        instruction->command = OPENBRACE;
        iOpenBraceCount++;
        if (program->size == 0)
        {
          fprintf(stderr, "error: A script cannot start with '{' \n");
          exit(2);
        }

        instruction--;
        switch (instruction->command)
        {
          case TESTIS:
          case TESTBEGINS:
          case TESTENDS:
          case TESTCLASS:
          case TESTEOF:
          case TESTTAPE:
          case TESTLIST:
            if (instruction->isNegated)
              { instruction->falseJump = program->size; }
            else
              { instruction->trueJump = program->size; }

            if (program->size == 1)
            {
              *pBraceStackPointer = program->size;
              pBraceStackPointer++;
              program->size++;
              instruction = &program->instructionSet[program->size];
              break;
            }
            instruction--;
            iTestPointer = program->size - 1;
            while ((instruction->command == TESTIS) ||
                   (instruction->command == TESTBEGINS) ||
                   (instruction->command == TESTENDS) ||
                   (instruction->command == TESTLIST) ||
                   (instruction->command == TESTTAPE) ||
                   (instruction->command == TESTEOF) ||
                   (instruction->command == TESTCLASS))
            {
              if (instruction->isNegated)
              {
                instruction->falseJump = program->size;
                instruction->trueJump = iTestPointer;
              }
              else
              {
                instruction->falseJump = iTestPointer;
                instruction->trueJump = program->size;
              }
              iTestPointer--;
              if (iTestPointer < 0) { break; }
              instruction--;
            } //-- while


            /* load the brace stack for calculated jumps */
            *pBraceStackPointer = program->size;
            pBraceStackPointer++;
            program->size++;
            instruction = &program->instructionSet[program->size];

            break;
          default:
            fprintf(stderr,
              "script error: The '{' character at line %d, char %d is n
ot preceded by a test \n",
              iLineCount, iLineCharacterCount);
            exit(2);
            break;
        } //-- switch
        break;
      /*-----------------------------------------*/
      case '}':
        iCloseBraceCount++;
        if (iCloseBraceCount > iOpenBraceCount)
        {
          fprintf(stderr,
            "script error: the '}' character at line %d, char %d seems
misplaced. \n",
            iLineCount, iLineCharacterCount);
          fprintf(stderr,
            "The are more close braces than open braces \n");
          exit(2);
        }

        if (instruction->command != UNDEFINED)
        {
          fnPrintInstruction(*instruction);
          fprintf(stderr,
            "script error: The '}' character at line %d, char %d seems
misplaced. \n",
             iLineCount, iLineCharacterCount);
          exit(2);
        }
        instruction->command = CLOSEBRACE;
        /* set the jumps for the test of the current brace pair, using
the brace stack
         * to find the corresponding open brace */
        pBraceStackPointer--;
        instruction = &program->instructionSet[*pBraceStackPointer - 1]
;
        if (instruction->isNegated)
        {
          instruction->trueJump = program->size;
          //instruction->trueJump = *pBraceStackPointer;
        }
        else
        {
          //instruction->falseJump = *pBraceStackPointer;
          instruction->falseJump = program->size;
        }
        program->size++;
        instruction = &program->instructionSet[program->size];
        break;

      /*-----------------------------------------*/
      // commands
      default:
        strcpy(sText, "");

        if (iCharacter == '\0')
        { break; }

        if (!islower(iCharacter) && (iCharacter != '+') && (iCharacter
!= '-') && (iCharacter != '@'))
        {
          fprintf(stderr, "line %d: illegal character '%c' (%d) \n",
                  iLineCount, iCharacter, iCharacter);
          fprintf(stderr, "  this character may only occur between quot
es");
          fprintf(stderr, "  or within tests.");
          exit(2);
        }

        while ((islower(iCharacter) || (iCharacter == '+') || (iCharact
er == '@') || (iCharacter == '-'))
                && (strlen(sText) < TEXTBUFFERSIZE))
        {
          sprintf(sText, "%s%c", sText, iCharacter);
          iCharacter = getc(inputstream);
          iCharacterCount++;
          iLineCharacterCount++;
        } //-- while

        if (strlen(sText) >= TEXTBUFFERSIZE)
        {
          fprintf(stderr, "syntax error: unrecognized command %s, line
%d, char %d",
            sText, iLineCount, iLineCharacterCount);
          exit(2);
        }
        iCommand = -1;
        iCommand = fnCommandFromString(sText);
        if (iCommand == UNKNOWN)
        {
          fprintf(stderr, "line %d: unrecognized command '%s'",
                  iLineCount, sText);
          exit(2);
        }

        if (instruction->command != UNDEFINED)
        {
          fprintf(stderr, "line %d: syntax error before command '%s'",
                  iLineCount, sText);
          exit(2);
        }

        if (iCharacter == EOF)
        {
          fprintf(stderr, "script error: script ends badly");
          exit(2);
        }

        instruction->command = iCommand;

       /* process the character currently in iCharacter */
        continue;
        /* fnPrintInstruction(*instruction); */

    } //-- switch


    iCharacter = getc(inputstream);
    iCharacterCount++;
    iLineCharacterCount++;

    int bDebug = 0;
    if (bDebug)
    {
      printf("current char=%c \n", iCharacter);
      fnPrintProgram(program);
    }

  } //-- while

  //fnPrintInstruction(*instruction);

  if (iOpenBraceCount != iCloseBraceCount)
  {
    printf("error: unbalanced braces: \n", iLineCount);
    printf("open braces=%d, ", iOpenBraceCount);
    printf("close braces=%d \n", iCloseBraceCount);
    exit(2);
  }

  if (instruction->command != UNDEFINED)
  {
    fnCommandToString(sText, instruction->command);
    fprintf(stderr, "line %d: unfinished command '%s'.",
            iLineCount, sText);
    exit(2);
  }

  /* add a final read and jump(0) command so that the script loops */
  instruction->command = READ;
  program->size++;
  instruction = &program->instructionSet[program->size];
  instruction->command = JUMP;
  instruction->trueJump = 0;
  program->size++;
  instruction = &program->instructionSet[program->size];

  /* compute the compile time */
  tEndCompile = clock();
  int iCompileTime = (int) (((tEndCompile - tBeginCompile) * 1000)/ CLO
CKS_PER_SEC);
  program->compileTime = iCompileTime;
  //printf("------------------- \n", iLineCount);
  //printf("     Lines parsed: %d \n", iLineCount);
  //printf("Characters parsed: %d \n\n", iCharacterCount);
  //printf("--Program Listing-- \n");

  //fnPrintProgram(program);

} //-- fnCompile

/* -------------------------------------------*/
int fnExecuteInstruction (Program * program, Machine * machine,
   FILE * inputstream)
{
  Instruction * instruction =
    &program->instructionSet[program->instructionPointer];

  FILE * fListFile;     //-- for the list file test
  char * sClass; //--
  char sTemp[TEMPSTRINGSIZE];
  char sTemp2[TEMPSTRINGSIZE];
  char sFileLine[MAXFILELINELENGTH];
  char * pTemp;
  Element * ee;
  pTemp = sTemp;
  int ii;
  int iOldStackSize = 0;
  Element * eCurrentTapeElement;

  switch (instruction->command)
  {
    /* ------------------------------------*/
    case ADD:
      machine = appendToWorkspace(machine, instruction->argument1);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case CLEAR:
      *machine->workspace = '\0';
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case PRINT:
      printf("%s", machine->workspace);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case STATE:
      fnPrintMachine(machine);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case REPLACE:
      // fnStringReplace(machine->workspace);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case INDENT:
      if (strlen(machine->workspace) >= TEMPSTRINGSIZE)
      {
        pTemp = (char *) realloc(pTemp, strlen(machine->workspace) * siz
eof(char) + GROWFACTOR);
      }

      if (pTemp == NULL)
      {
        printf ("\nError reallocating memory for a temporary string \n")
;
        exit (1);
      }

      strcpy(pTemp, machine->workspace);
      strcpy(machine->workspace, "  ");

      for (ii = 0; ii < strlen(pTemp); ii++)
      {
        sprintf(sTemp2, "%c", pTemp[ii]);
        machine = appendToWorkspace(machine, sTemp2);
        if (pTemp[ii] == '\n')
        {
           machine = appendToWorkspace(machine, "  ");
        }

      } //-- for

      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case CLIP:
      if (strlen(machine->workspace) > 0)
        { machine->workspace[strlen(machine->workspace) - 1] = '\0'; }
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case CLOP:
      if (strlen(machine->workspace) > 0)
      {
        for (ii = 0; ii < strlen(machine->workspace); ii++)
        {
          machine->workspace[ii] = machine->workspace[ii + 1];
        }
      }
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case NEWLINE:
      strcat(machine->workspace, "\n");
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case PUSH:
      if (*machine->workspace == '\0')
      {
        program->instructionPointer++;
        break;
      }

      machine->workspace++;
      while ((*machine->workspace != '*') &&
             (*machine->workspace !='\0'))
      {
        machine->workspace++;
      }

      if (*machine->workspace == '*')
      {
        machine->workspace++;
      }

      // printf("machine->tapepointer = %d \n", machine->tapepointer);
      // printf("&machine->tape[MAXTAPELENGTH] =x %d \n", &machine->tape
[MAXTAPELENGTH]);
      if (machine->tapepointer < &machine->tape[MAXTAPELENGTH - 1])
      {
         machine->tapepointer++;
      }
      else
      {
        printf("Maximum tape length (%d) exceeded \n", MAXTAPELENGTH);
        printf("The possible remedies are: \n");
        printf(" a. increase the MAXTAPELENGTH constant in 'library.c' a
nd recompile \n");
        printf(" b. rewrite the script to use less tape elements. \n");
        printf(" c. use the -d switch to view a trace of the script. \n"
);
        printf("Below is shown the final state of the virtual machine \n
\n");
        fnPrintMachineState(machine);
        exit(2);
      }

      machine->stacksize++;
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case POP:
      if (machine->workspace == machine->stack)
      {
        program->instructionPointer++;
        break;
      }

      machine->workspace--;
      if (machine->workspace == machine->stack)
      {
        machine->tapepointer--;
        machine->stacksize--;
        program->instructionPointer++;
        break;
      }

      if (*machine->workspace == '*')
       { machine->workspace--; }

      while ((*machine->workspace != '*') &&
             (machine->workspace != machine->stack))
      {
        machine->workspace--;
      }

      if (*machine->workspace == '*')
        { machine->workspace++; }

      if (machine->tapepointer > &machine->tape[0])
       { machine->tapepointer--; }

      machine->stacksize--;
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case PUT:
      if (strlen(machine->workspace) >= (machine->tapepointer->size) - 1
)
      {
        machine->tapepointer->size = strlen(machine->workspace) + GROWFA
CTOR;
        machine->tapepointer->text =
          (char *) realloc(machine->tapepointer->text, machine->tapepoin
ter->size * sizeof(char));
      }

      if (machine->tapepointer->text == NULL)
      {
         printf ("\nError reallocating memory for a tape element \n");
         exit (1);
      }

      strcpy(machine->tapepointer->text, machine->workspace);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case GET:

      machine = appendToWorkspace(machine, machine->tapepointer->text);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case INCREMENT:
      if (machine->tapepointer >= &machine->tape[MAXTAPELENGTH])
      {
        printf("maximum tape length exceeded (%d)\n", MAXTAPELENGTH);
        printf("change the MAXTAPELENGTH constant and recompile \n");
        exit(2);
      }
      machine->tapepointer++;
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case DECREMENT:
      if (machine->tapepointer > &machine->tape[0])
       { machine->tapepointer--; }
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case READ:
      if (machine->peep == EOF)
      {
        return ENDOFSTREAM;
      }
      sprintf(sTemp, "%c", machine->peep);
      machine = appendToWorkspace(machine, sTemp);
      machine->peep = getc(inputstream);
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case UNTIL:
      if (machine->peep == EOF)
      {
        program->instructionPointer++;
        break;
      }

      int bLoop = TRUE;
      sprintf(sTemp, "%c", machine->peep);
      machine = appendToWorkspace(machine, sTemp);
      machine->peep = getc(inputstream);

      if (fnStringEndsWith(machine->workspace, instruction->argument1) =
= TRUE)
      {
        bLoop = FALSE;
      }

      while (bLoop == TRUE)
      {

        sprintf(sTemp, "%c", machine->peep);
        machine = appendToWorkspace(machine, sTemp);
        machine->peep = getc(inputstream);
        if (machine->peep == EOF)
        {
          program->instructionPointer++;
          break;
        }

        if (fnStringEndsWith(machine->workspace, instruction->argument1)
 == TRUE)
        {
          bLoop = FALSE;
          if ((fnStringEndsWith(machine->workspace, instruction->argumen
t2) == TRUE) &&
              (strlen(instruction->argument2) > 0))
          {
            bLoop = TRUE;
          }
        }
      } //-- while

      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case WHILE:
      if (machine->peep == EOF)
      {
        program->instructionPointer++;
        break;
      }

      bLoop = TRUE;
      sClass = instruction->argument1;
      if ((fnIsInClass(sClass, machine->peep) == FALSE) &&
            (instruction->isNegated == FALSE))
      {
        program->instructionPointer++;
        break;
      }
      if ((fnIsInClass(sClass, machine->peep) == TRUE) &&
            (instruction->isNegated == TRUE))
      {
        program->instructionPointer++;
        break;
      }


      while (bLoop)
      {

        sprintf(sTemp, "%c", machine->peep);
        machine = appendToWorkspace(machine, sTemp);
        machine->peep = getc(inputstream);

        if (machine->peep == EOF)
        {
          bLoop = FALSE;
        }

        if ((fnIsInClass(sClass, machine->peep) == FALSE) &&
            (instruction->isNegated == FALSE))
        {
          bLoop = FALSE;
        }
        if ((fnIsInClass(sClass, machine->peep) == TRUE) &&
            (instruction->isNegated == TRUE))
        {
          bLoop = FALSE;
        }
     } //-- while


      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case WHILENOT:
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case TESTIS:
      if (strcmp(machine->workspace, instruction->argument1) == 0)
        { program->instructionPointer = instruction->trueJump; }
      else
        { program->instructionPointer = instruction->falseJump; }
      break;
    /* ------------------------------------*/
    case TESTLIST:
      fListFile = fopen(instruction->argument1, "r");
      strcpy(program->listFile, instruction->argument1);
      if (fListFile == NULL)
      {
        program->fileError = TRUE;
        program->instructionPointer++;
        break;
      }

      program->instructionPointer = instruction->falseJump;
      while (fgets(sFileLine, MAXFILELINELENGTH, fListFile) != NULL)

      {
        fnStringTrim(sFileLine);
        // printf ("sFileLine=[%s]\n", sFileLine);
        if (strcmp(sFileLine, machine->workspace) == 0)
          {program->instructionPointer = instruction->trueJump; }
      }

      fclose(fListFile);
      break;
    /* ------------------------------------*/
    case TESTBEGINS:
      if (fnStringBeginsWith(machine->workspace, instruction->argument1)
)
         { program->instructionPointer = instruction->trueJump; }
      else
        { program->instructionPointer = instruction->falseJump; }
      break;
    /* ------------------------------------*/
    case TESTENDS:
      if (fnStringEndsWith(machine->workspace, instruction->argument1))
         { program->instructionPointer = instruction->trueJump; }
      else
        { program->instructionPointer = instruction->falseJump; }
      break;
    /* ------------------------------------*/
    case TESTCLASS:
      sClass = instruction->argument1;
      program->instructionPointer = instruction->falseJump;
      if (fnIsInClass(sClass, *machine->workspace))
        { program->instructionPointer = instruction->trueJump; }

      break;
    /* ------------------------------------*/
    case TESTTAPE:
      if (strcmp(machine->workspace, machine->tapepointer->text) == 0)
        { program->instructionPointer = instruction->trueJump; }
      else
        { program->instructionPointer = instruction->falseJump; }
      break;
    /* ------------------------------------*/
    case TESTEOF:
      if (machine->peep == EOF)
        { program->instructionPointer = instruction->trueJump; }
      else
        { program->instructionPointer = instruction->falseJump; }
      break;
    /* ------------------------------------*/
    case COUNT:
      /* add the counter to the workspace */
      /* add the text to the workspace 'count' times */
      if (strlen(instruction->argument1) == 0)
      {
        sprintf(sTemp, "%d", machine->counter);
        machine = appendToWorkspace(machine, sTemp);
      }
      else
      {
        strcpy(sTemp, "");
        for (ii = 0; ii < machine->counter; ii++)
         { strcat(sTemp, instruction->argument1); }
        machine = appendToWorkspace(machine, sTemp);

      }

      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case INCC:
      machine->counter++;
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case DECC:
      machine->counter--;
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case CRASH:
      program->instructionPointer++;
      return ENDOFSTREAM;
      break;
    /* ------------------------------------*/
    case JUMP:
      program->instructionPointer = instruction->trueJump;
      break;
    /* ------------------------------------*/
    case LABEL:
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case UNDEFINED: /* the default */
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case NOP:     /* no operation */
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case ZERO:    /* set the counter to zero */
      machine->counter = 0;
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case OPENBRACE:
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    case CLOSEBRACE:
      program->instructionPointer++;
      break;
    /* ------------------------------------*/
    default:
      printf(
        "runtime error: unexpected instruction at instruction %d",
        program->instructionPointer);
      exit(2);
      break;


  } //-- switch

  machine->lastoperation = instruction->command;
  return TRUE;

} //-- fnExecuteInstruction
```