```c
/* Program to evaluate candidate routines for Robotic Process Automation.

  Skeleton program written by Artem Polyvyanyy, artem.polyvyanyy@unimelb.edu.au,
  September 2020, with the intention that it be modified by students
  to add functionality, as required by the assignment specification.

  Student Authorship Declaration:

  (1) I certify that except for the code provided in the initial skeleton
  file, the  program contained in this submission is completely my own
  individual work, except where explicitly noted by further comments that
  provide details otherwise.  I understand that work that has been developed
  by another student, or by me in collaboration with other students, or by
  non-students as a result of request, solicitation, or payment, may not be
  submitted for assessment in this subject.  I understand that submitting for
  assessment work developed by or in collaboration with other students or
  non-students constitutes Academic Misconduct, and may be penalized by mark
  deductions, or by other penalties determined via the University of
  Melbourne Academic Honesty Policy, as described at
  https://academicintegrity.unimelb.edu.au.

  (2) I also certify that I have not provided a copy of this work in either
  softcopy or hardcopy or any other form to any other student, and nor will I
  do so until after the marks are released. I understand that providing my
  work to other students, regardless of my intention or any undertakings made
  to me by that other student, is also Academic Misconduct.

  (3) I further understand that providing a copy of the assignment
  specification to any form of code authoring or assignment tutoring service,
  or drawing the attention of others to such services and code that may have
  been made available via such a service, may be regarded as Student General
  Misconduct (interfering with the teaching activities of the University
  and/or inciting others to commit Academic Misconduct).  I understand that
  an allegation of Student General Misconduct may arise regardless of whether
  or not I personally make use of such solutions or sought benefit from such
  actions.

   Signed by: JAMES LA FONTAINE    1079860
   Dated:     30/10/2020

*/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include <string.h>

/** #define's *****************************************************************/

#define DEBUG 0
#if DEBUG
#define DUMP_DBL(x) printf("line %d: %s = %.6f\n", __LINE__, #x, x)
#define DUMP_INT(x) printf("line %d: %s = %d\n", __LINE__, #x, x)
#define DUMP_CHR(x) printf("line %d: %s = %c\n", __LINE__, #x, x)
#define DUMP_STR(x) printf("line %d: %s = %s\n", __LINE__, #x, x)
#define DEBUG_PRINT(x) printf("%s",x)
#else
#define DUMP_DBL(x)
```

```
#define DUMP_INT(x)
#define DUMP_CHR(x)
#define DUMP_STR(x)
#define DEBUG_PRINT(x)
#endif

#define ASIZE 26              // the maximum size of a state array
#define LOWERCASE_OFFSET 97   // used to line up state array with ASCII codes

#define MUSTTRUE 1            // indicates value must be true in precon
#define EITHER 0              // indicates either value is fine in precon
#define MUSTFALSE -1          // indicates value must be false in precon
#define SETTRUE 1             // indicates value is set to true by action
#define SAME 0                // indicates value is kept same by action
#define SETFALSE -1           // indicates value is set to false by action
#define TRUE 1                // represents a true state for the variables
#define FALSE 0                      // represents a false state for the variables


#define TRUEPRECON 0          // track the current action definition
#define FALSEPRECON 1         // section being defined in stage 0 input
#define ACTIONNAME 2
#define TRUEEFFECT 3
#define FALSEEFFECT 4

#define VALID 1               // used to indicate that a trace is valid

#define EQUAL 1               // used to indicate that a subsequence and
                                           // candidate routine have an equal
effect


/** type definitions **********************************************************/

// state (values of the 26 Boolean variables)
typedef int state_t[ASIZE];

// action
typedef struct action action_t;
struct action {
    char name;        // action name
    state_t precon;   // precondition
    state_t effect;   // effect
};

// step in a trace
typedef struct step step_t;
struct step {
    action_t *action; // pointer to an action performed at this step
    step_t   *next;   // pointer to the next step in this trace
};

// trace (implemented as a linked list)
typedef struct {
    step_t *head;     // pointer to the step in the head of the trace
    step_t *tail;     // pointer to the step in the tail of the trace
} trace_t;

/** function prototypes *******************************************************/
```

```
trace_t* make_empty_trace(void);
trace_t* insert_at_tail(trace_t*, action_t*);
void free_trace(trace_t*);

/** my function prototypes ****************************************************/

int mygetchar(void);
action_t* create_new_action_struct(void);
trace_t* add_to_ordered_actions_list(trace_t*, trace_t*, int*, char);
trace_t* simulate_actions(step_t*, state_t, int);

void stage0(void);
trace_t* create_definitions_list(int*);
void print_stage0(trace_t*, trace_t*, state_t, int, int);
int validate_trace(trace_t*, trace_t*, state_t, int*);

void stage_1_and_2(trace_t*, trace_t*, int);
void identify_subsequences(trace_t*, trace_t*, int, int);
void print_candidate_routine(trace_t* canactions);
void print_subsequence(trace_t*, int, int);
int equal_cumulative_effect(trace_t*, state_t*, state_t*, int);

/** where it all happens ******************************************************/

int
main(int argc, char *argv[]) {

    stage0();

    /* print the required newline at end of output */
    printf("\n");

    return EXIT_SUCCESS;        // we are done !!! algorithms are fun!!!
}

/** function definitions ******************************************************/

// Adapted version of the make_empty_list function by Alistair Moffat:
// https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/listops.c
// Data type and variable names changed
trace_t
*make_empty_trace(void) {
    trace_t *R;
    R = (trace_t*)malloc(sizeof(*R));
    assert(R!=NULL);
    R->head = R->tail = NULL;
    return R;
}

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

// Adapted version of the insert_at_foot function by Alistair Moffat:
// https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/listops.c
// Data type and variable names changed
trace_t
*insert_at_tail(trace_t* R, action_t* addr) {
    step_t *new;
    new = (step_t*)malloc(sizeof(*new));
    assert(R!=NULL && new!=NULL);
    new->action = addr;
    new->next = NULL;
    if (R->tail==NULL) { /* this is the first insertion into the trace */
        R->head = R->tail = new;
    } else {
        R->tail->next = new;
        R->tail = new;
    }
    return R;
}

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

// Adapted version of the free_list function by Alistair Moffat:
// https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/listops.c
// Data type and variable names changed
void
free_trace(trace_t* R) {
    step_t *curr, *prev;
    assert(R!=NULL);
    curr = R->head;
    while (curr) {
        prev = curr;
        curr = curr->next;
        free(prev);
    }
    free(R);
}

/** my function definitions ***************************************************/

// The mygetchar function by Alistair Moffat:
// https://people.eng.unimelb.edu.au/ammoffat/teaching/10002/ass1/
// throws away CR characters to prevent program disruption when testing
// in different environments
int
mygetchar() {
      int c;
      while ((c=getchar())=='\r') {
      }
      return c;
}

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

/* mallocs a new action struct that will be inserted into linked lists
*/
action_t
*create_new_action_struct() {
      action_t *newaction;
      int i;

      newaction = (action_t*)malloc(sizeof(*newaction));
      assert(newaction!=NULL);
      DEBUG_PRINT("a newaction struct has been created\n");
      /* initialise the state arrays to a default state */
      for (i=0; i<ASIZE; i++) {
            newaction->precon[i] = EITHER;
            newaction->effect[i] = SAME;
      }
      return newaction;
}

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

/* this function is called when a trace action / candidate routine action
is input in any stage and simply adds an input action to the relevant ordered
actions list
*/
trace_t
*add_to_ordered_actions_list(trace_t* definitions, trace_t* orderedactions,
                                                                              int*
numactions, char ch) {
      step_t *currentdefinition;

      currentdefinition = definitions->head;
      while (currentdefinition) {
            /* find the matching action in the definition list and record it in
            the ordered actions list */
            if (ch==currentdefinition->action->name) {
                  orderedactions = insert_at_tail(orderedactions,
                                                         currentdefinition-
>action);
                  DEBUG_PRINT("a new orderedactions step has been inserted\n");
                  *numactions += 1;
                  return orderedactions;
            /* otherwise we haven't found the matching action and need to keep
            looking */
            }else {
                  currentdefinition = currentdefinition->next;
                  DEBUG_PRINT("no match here, looking at next definition\n");
            }
      }
      /* avoid warnings about potential memory leak and control reaching end of
      non-void function - only erroneous input would result in this anyway */
      return orderedactions;
}

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

/* creates a linked list by simulating all the sequence of actions, for stage 1
it also includes in the effect arrays whether a variable has been altered by
using SETFALSE (-1) to denote a variable being set to false
*/
trace_t
*simulate_actions(step_t* currentdefstep, state_t I, int isstage1) {
      trace_t *simactions = make_empty_trace();
      action_t *currentaction, *prevaction;
      int i, simtracelen = 0;

      while (currentdefstep) {
            currentaction = create_new_action_struct();
            /* we apply the first action to the initial state */
            if (simtracelen==0) {
                  for (i=0; i<ASIZE; i++) {
                        /* check if action definition says we need to change value
*/
```

```
                if (currentdefstep->action->effect[i]==SETTRUE) {
                    currentaction->effect[i] = SETTRUE;
                }else if (currentdefstep->action->effect[i]==SETFALSE) {
                    if (isstage1) {
                        currentaction->effect[i] = SETFALSE;
                    }else {
                        currentaction->effect[i] = FALSE;
                    }
                /* otherwise it is same value as it was before the action
*/
                }else {
                    currentaction->effect[i] = I[i];
                }
            }
        /* this isn't the first action so we apply the action to the
        previous state */
        }else {
            for (i=0; i<ASIZE; i++) {
                /* check if action definition says we need to change value
*/
                if (currentdefstep->action->effect[i]==SETTRUE) {
                    currentaction->effect[i] = SETTRUE;
                }else if (currentdefstep->action->effect[i]==SETFALSE) {
                    if (isstage1) {
                        currentaction->effect[i] = SETFALSE;
                    }else {
                    currentaction->effect[i] = FALSE;
                    }
                /* otherwise it is same value as it was before the action
*/
                }else {
                    currentaction->effect[i] = (prevaction->effect[i]);
                }
            }
        }
        /* now shift to the next action definition and store the state after
the
        previous action */
        simactions = insert_at_tail(simactions, currentaction);
        simtracelen += 1;
        prevaction = currentaction;
        currentdefstep = currentdefstep->next;

    }
    return simactions;
}

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

/* handles all the flow of stage 0 and leads into stage 1 and 2 if required
*/
void
stage0() {
    trace_t *definitions;
    trace_t *simtrace;
    trace_t *trcactions = make_empty_trace();
    state_t I = {FALSE};                                /* initial state array */
    int numdefinitions = 0, tracelen = 0;
    int *numdefptr = &numdefinitions, *tracelenptr = &tracelen;
```

```
char ch;

/* set up the initial state of the variables in an array */
while (scanf("%c", &ch)) {
      if (ch == '#') {
            break;
      }else if (ch=='\r' || ch=='\n') {
            continue;
      }
      I[ch-LOWERCASE_OFFSET] = TRUE;
}

/* read the definitions and record them in a linked list */
definitions = create_definitions_list(numdefptr);

/* remove the previous newline */
mygetchar();

/* now we will read the trace and create a linked list of ordered
action definitions */
while (scanf("%c", &ch)) {
      DUMP_CHR(ch);
      if (ch=='#' || ch=='\n') {
            break;
      }else if (ch=='\r') {
            continue;
      }else {
      /* we must have received an action name in the input trace */
      trcactions = add_to_ordered_actions_list(definitions, trcactions,

   tracelenptr, ch);
      }

}

/* using the ordered actions list we will now simulate the
trace and record all the states along the way so we can display them and
also validate the trace */
simtrace = simulate_actions(trcactions->head, I, FALSE);
DEBUG_PRINT("have created a simulated trace linked list\n");

/* now we will display this information as specified */
print_stage0(simtrace, trcactions, I, numdefinitions, tracelen);
fflush(stdout);
free_trace(simtrace);
simtrace = NULL;

/* check if we need to go to stage 1 */
if ((ch=mygetchar())=='#') {
      /* remove last newline from stage 0 */
      mygetchar();
      stage_1_and_2(definitions, trcactions, tracelen);
}
/* we are all done, free up the remaining linked lists */
free_trace(trcactions);
trcactions = NULL;
free_trace(definitions);
definitions = NULL;
return;
```

```
}

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

/* creates a linked list of the action definitions in stage 0
*/
trace_t
*create_definitions_list(int* numdefinitions) {
      trace_t *definitions = make_empty_trace();
      action_t *newaction;
      int currsubaction = TRUEPRECON, made_new_struct = FALSE;
      char ch;


      while (scanf("%c", &ch)) {
            DUMP_CHR(ch);
            /* once we encounter a #, we are finished defining actions */
            if (ch=='#') {
                  return definitions;
            /* ignore \r characters */
            }else if (ch=='\r') {
                  continue;
            /* if we encounter a newline, we will now record a new action */
            }else if (ch=='\n') {
                  currsubaction = TRUEPRECON;
                  made_new_struct = FALSE;
                  continue;
            }
            /* otherwise we must have received an action definition */

            /* check if we need to allocate memory for a new action struct */
            if (!made_new_struct) {
                  definitions = insert_at_tail(definitions,

      create_new_action_struct());
                  newaction = definitions->tail->action;
                  made_new_struct = TRUE;
                  *numdefinitions += 1;
                  DEBUG_PRINT("new definition added to definition list\n");
            }

            /* first record the true preconditions until a colon is encountered */
            if (currsubaction==TRUEPRECON) {
                  if (ch==':') {
                        currsubaction=FALSEPRECON;
                        continue;
                  }
                  newaction->precon[ch-LOWERCASE_OFFSET] = MUSTTRUE;

            /* now the false preconditions */
            }else if (currsubaction==FALSEPRECON) {
                  if (ch==':') {
                        currsubaction = ACTIONNAME;
                        continue;
                  }
                  newaction->precon[ch-LOWERCASE_OFFSET] = MUSTFALSE;

            /* now the action name */
            }else if (currsubaction==ACTIONNAME) {
                  if (ch==':') {
                        currsubaction = TRUEEFFECT;
                        continue;
                  }
                  newaction->name = ch;

            /* now the true effects */
            }else if (currsubaction==TRUEEFFECT) {
                  if (ch==':') {
                        currsubaction = FALSEEFFECT;
                        continue;
                  }
                  newaction->effect[ch-LOWERCASE_OFFSET] = SETTRUE;
            /* finally the false effects */
            }else if (currsubaction==FALSEEFFECT) {
                  newaction->effect[ch-LOWERCASE_OFFSET] = SETFALSE;
            }
      }
      /* avoid warnings about potential memory leak and control reaching end of
      non-void function - only erroneous input would result in this anyway */
      return definitions;
}

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

/* displays the relevant information as specified for stage 0
*/
void
print_stage0(trace_t* simtrace, trace_t* trcactions, state_t I,
                                                                   int
numdefinitions, int tracelen) {
      step_t *currentnamestep, *currentstatestep;
      int i, j, validtracelen = 0;
      int *validtracelenptr = &validtracelen;

      printf("==STAGE 0===============================\n");
      printf("Number of distinct actions: %d\n", numdefinitions);
      printf("Length of the input trace: %d\n", tracelen);
      if (validate_trace(simtrace, trcactions, I, validtracelenptr)) {
            printf("Trace status: valid\n");
      }else {
            printf("Trace status: invalid\n");
      }
      printf("----------------------------------------\n");
      printf("  abcdefghijklmnopqrstuvwxyz\n");
      printf("> ");
      for (i=0; i<ASIZE; i++) {
            printf("%d", I[i]);
      }
      currentnamestep = trcactions->head;
      currentstatestep = simtrace->head;
      /* print trace states up until the trace was found to be invalid or if it
      was valid then until the end of the trace */
      for (i=0; i<validtracelen; i++) {
            printf("\n");
            printf("%c ", currentnamestep->action->name);
            for (j=0; j<ASIZE; j++) {
                  /* print the state specified in the simulated trace */
                  printf("%d", currentstatestep->action->effect[j]);
            }
            currentnamestep = currentnamestep->next;
            currentstatestep = currentstatestep->next;
      }
      return;
}
/******************************************************************************/

/* checks that the actions occur with each state fulfilling the
precondition or, if they don't, finds out when the trace becomes invalid
*/
int
validate_trace(trace_t* simtrace, trace_t* trcactions, state_t I,

int* validtracelen) {
      int i;
      state_t *currentprecon, *prevstate = NULL;
      step_t *currentdefstep, *currentsimstep;

      /* current step in simulated trace */
      currentsimstep = simtrace->head;
      /* current step in ordered list of trace actions */
      currentdefstep = trcactions->head;

      /* if this is the first action, we need to check its preconditions
      with the initial state */
      while (currentsimstep) {
      assert (currentdefstep);
      currentprecon = &(currentdefstep->action->precon);
      assert (currentprecon);
            if (*validtracelen==0) {
                  for (i=0; i<ASIZE; i++) {
                        /* check if value must be true and is actually not true */
                        if ((*currentprecon)[i]==MUSTTRUE && !(I[i])) {
                              return !VALID;
                        /* check if value must be false and is actually true */
                        }else if ((*currentprecon)[i]==MUSTFALSE && (I[i])) {
                              return !VALID;
                        }
                  }
            /* otherwise we are checking the previous state with the next actions
            preconditions */
            }else {
                  assert(prevstate);
                  for (i=0; i<ASIZE; i++) {
                        /* check if value must be true and is actually false */
                        if ((*currentprecon)[i]==MUSTTRUE && (*prevstate)[i]!
=SETTRUE) {
                              return !VALID;
                        /* check if value must be false and is actually true */
                        }else if ((*currentprecon)[i]==MUSTFALSE &&

(*prevstate)[i]==SETTRUE) {
                              return !VALID;
                      }
                  }
            }
            *validtracelen += 1;
            prevstate = &(currentsimstep->action->effect);
            currentsimstep = currentsimstep->next;
            currentdefstep = currentdefstep->next;
      }

      /* all actions occur with their preconditions fulfilled so the trace is
      valid */
      return VALID;
}

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

/* entry point into stage 1, handles the input of the candidate routine and
leads to searching for subsequences and printing them. If stage 2 is required
then isstage1 variable is simply changed to false and the simulate_actions
function will take this into account when it is needed
*/
void stage_1_and_2(trace_t* definitions, trace_t* trcactions, int tracelen) {
      /* now we will read the candidate trace and create a linked list of ordered
      action definitions */
      trace_t* canactions = make_empty_trace();
      int canlen = 0;
      int* canlenptr = &canlen;
      int firstcandidate = TRUE, firstaction = TRUE, isstage1 = TRUE;
      char ch;

      DEBUG_PRINT("\nnow looking at candidate routines\n");
      printf("\n==STAGE 1==============================\n");

      /* read the input for the candidate routine */
      while (scanf("%c", &ch)) {
            DUMP_CHR(ch);
            if (ch=='#') {
                  fflush(stdout);
                  isstage1 = FALSE;
                  firstcandidate = TRUE;
                  /* remove the newline from the previous stage */
                  mygetchar();
                  printf("\n==STAGE 2==============================\n");
            }else if (ch=='\r') {
                  continue;
            }else if (ch=='\n') {
                  /* if we encounter another newline before we start reading
another
                  candidate routine then we are completely finished */
                  if (canlen==0) {
                        if (!isstage1) {
                              printf("\n==THE END===============================");
                              fflush(stdout);
                              free_trace(canactions);
                              canactions = NULL;
                              return;
                        }
                        free_trace(canactions);
                        canactions = NULL;
                        return;
                  }
                  /* otherwise we have just finished reading a candidate routine
                  and can start searching for subsequences and printing information
*/
                  print_candidate_routine(canactions);
                  identify_subsequences(canactions, trcactions, tracelen,
isstage1);
                  /* now we are now done with this candidate routine and have to
                  prepare for the possibility of another candidate routine */
                  free_trace(canactions);
                  canactions = make_empty_trace();
                  canlen = 0;
                  firstcandidate = FALSE;
                  firstaction = TRUE;

            /* we must have received a candidate routine action name */
            }else {
                  /* if this is a new candidate routine but isn't the first
candidate
                  routine of the stage then print the delimiter line above it */
                  if (firstaction && !firstcandidate) {
                        printf("\n----------------------------------------\n");
                        firstaction = FALSE;
                  }
                  canactions = add_to_ordered_actions_list(definitions, canactions,

         canlenptr, ch);
            }
      }

      /* prevent potential memory leak warning - this would only occur if
      there was erroneous input anyway */
      if (canactions) {
            free_trace(canactions);
            canactions = NULL;
      }
      return;
}

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

/* finds the smallest non-overlapping subsequences in the trace with the same
cumulative effect as the given candidate routine
*/
void
identify_subsequences(trace_t* canactions, trace_t* trcactions, int tracelen,

       int isstage1) {
      trace_t *simcan, *simsubseq;
      state_t I = {FALSE};   /* blank slate initial state for comparing effects */
      state_t *caneffectptr, *subseqeffectptr;
      step_t *subseqfirststep, *subseqeffectstep;
      int startpos, endpos;
      int startdepth = 0, enddepth = 0;
      int match_found = FALSE;

      DEBUG_PRINT("now identifying subsequences\n");

      /* first we will simulate and store the cumulative effect of the candidate
      routine */
      simcan = simulate_actions(canactions->head, I, isstage1);
      assert(simcan->tail);
      caneffectptr = &(simcan->tail->action->effect);

      /* now iterate through the trace and find smallest non-overlapping
      subsequences with the same cumulative effect as the candidate routine */

      DEBUG_PRINT("now iterating through subsequences of the trace\n");
      /* iterate through each starting position */
      subseqfirststep = trcactions->head;
      for (startpos=0; startpos<tracelen;) {
            DEBUG_PRINT("\n");
            DUMP_INT(startpos);
            /* line up what will be the first action in the subsequence simulation
            with the current action we are up to in the subsequence search */
            for (; startdepth<startpos; startdepth++) {
                  subseqfirststep = subseqfirststep->next;
            }

            /* create a simulation of a full sequence of trace actions starting
            from the startpos step and ending at the end of the trace */
            simsubseq = simulate_actions(subseqfirststep, I, isstage1);

            /* iterate through all subsequences of the trace starting from startpos
            */
            for (endpos=startpos; endpos<tracelen; endpos++) {
                  DUMP_INT(endpos);
                  /* have to go the step in the subsequence's simulation specified
by
                  the endpos to find and access the appropriate cumulative effect
*/
                  subseqeffectstep = simsubseq->head;
                  subseqeffectptr = &(subseqeffectstep->action->effect);
                  for (enddepth=startdepth; enddepth<endpos; enddepth++) {
                        subseqeffectstep = subseqeffectstep->next;
                        subseqeffectptr = &(subseqeffectstep->action->effect);
                  }
                  /* check if the subsequence achieves the same cumulative effect
                  as the candidate routine and also satisfies the stage conditions
*/
                  if (equal_cumulative_effect(trcactions, caneffectptr,

subseqeffectptr, startpos)) {
                        DEBUG_PRINT("a matching subsequence has been found\n");
                        print_subsequence(trcactions, startpos, endpos);
                        match_found = TRUE;
                        break;
                  }
            }
            /* if we found a match then restart the search from 1 to the right of
            the endpos, or if we didn't find anything then just shift right by 1*/
            if (match_found) {
                  startpos = endpos+1;
                  match_found = FALSE;
                  free_trace(simsubseq);
                  simsubseq = NULL;
            }else {
                  startpos += 1;
                  free_trace(simsubseq);
                  simsubseq = NULL;
            }
      }
      free_trace(simcan);
      simcan = NULL;
      return;
}

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

/* prints out the line for a candidate routine including the action names
*/
void
print_candidate_routine(trace_t* canactions) {
      step_t *currentstep;

      printf("Candidate routine: ");
      currentstep = canactions->head;
      /* print out each action name in our candidate routine ordered actions list
      */
      while (currentstep) {
            printf("%c", currentstep->action->name);
            currentstep = currentstep->next;
      }
      return;
}

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

/* prints out the line for a matching subsequence including the action names and
position in the trace that the subsequence begins at
*/
void
print_subsequence(trace_t* trcactions, int startpos, int endpos) {
      int trcdepth;
      step_t *currentstep;

      printf("\n");
      printf("%5d: ", startpos);
      currentstep = trcactions->head;
      /* we get the action names from the original ordered trace actions list (as
      the simulated list does not have them) and use startpos and endpos as our
      guides to find the right actions in the list */
      for (trcdepth = 0; trcdepth<startpos; trcdepth++) {
            currentstep = currentstep->next;
      }
      /* print out each action name in our simulated subsequence */
      for (;startpos<=endpos; startpos++) {
            printf("%c", currentstep->action->name);
            currentstep = currentstep->next;
      }
      return;
}

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

/* checks the values of variables based on 2 given effect states and
returns whether an equal effect was produced or not
*/
int
equal_cumulative_effect(trace_t* trcactions, state_t* caneffect,
                                                           state_t* subseqeffect,
int startpos) {
      int i;

      for (i=0; i<ASIZE; i++) {
            if ((*caneffect)[i]!=(*subseqeffect)[i]) {
                  return !EQUAL;
            }
      }
      DEBUG_PRINT("matching subsequence found\n");
      return EQUAL;
}

/******************************************************************************/
/* --------------------------- ta-da-da-daa!!! ------------------------------ */
/******************************************************************************/
/* ------------------------ algorithms are fun!!! --------------------------- */
/******************************************************************************/
```