1. Introduction. `TBAN` is a sokobunny clone (with bugs added by Tássio), written in literate style and rendered using ncurses. Sokobunny is a game by Ruby (`@rubyclash`). At the end of this document there is an an index of sections and variable usage.

This first version of the program reads an input file (describing the sequence of levels) and lets you play `TBAN` in it. Here's a sample input file, with two levels, to give you an idea.

```
#######
#@.b.+#
#######

#######
#...,+#
#@,.,b#
###...#
#######
```

Here is a simplified explanation of the input file syntax.

- '`#`' represents a wall,
- '`.`' or a '` `' (a space) represents an empty space
- '`,`' are spikes (may be crossed using a magic field),
- '`+`' is a goal (bunny recipient),
- '`@`' is where the player starts,
- '`b`' is a bunny,
- '`B`' is a bunny over a '`+`',
- '`A`' is the player over a goal.

The player controls are the arrow keys to move and also

- '`x`' opens or closes the magic field,
- '`z`' undoes last action,
- '`r`' restarts level,
- '`q`' exits the game.

For convenience, we also include some debugging controls:

- '`J`' go to to next level
- '`K`' go to previous level
- '`D`' display board debugger
- '`G`' show game's state

2. The banner below should be changed whenever this program source is modified.

```
#define banner "**␣This␣is␣TBAN,␣version␣0.9.␣**\n"
⟨ Header files 3 ⟩
⟨ Global declarations 7 ⟩
int main(int argc, char **argv)
{
  ⟨ Local variables of main 12 ⟩
  printf(banner);
  assert(sizeof (board_square) ≡ 1);
  ⟨ Process arguments 4 ⟩
  ⟨ Initialize game 27 ⟩
  ⟨ Game loop 32 ⟩
end: ⟨ Clean up and exit 31 ⟩
}
```

3. ⟨ Header files 3 ⟩ ≡

```
#include <assert.h>
```

See also sections 5 and 13.

This code is used in section 2.

4. The only argument received by the program is the file from which to read the levels.

⟨ Process arguments 4 ⟩ ≡

```
if (argc < 2) {
  fprintf(stderr, "Please␣give␣me␣a␣file␣with␣the␣levels!\n");
  exit(1);
}
else if (argc > 2) {
  fprintf(stderr, "I'm␣confused,␣too␣many␣arguments!\n");
  exit(1);
}
```

This code is used in section 2.

5. ⟨ Header files 3 ⟩ +≡

```
#include <stdio.h>
#include <stdlib.h>
```

6. Data structures for levels. Our internal representation of a level is very simple. Each level is a matrix with 256 rows and 256 columns. Coordinate (i, j) corresponds to the i^{th} line and j^{th} column, and $(0, 0)$ corresponds to the top left corner of the level (increasing goes downwards and to the right). One hundred levels should be sufficient for now.

```
#define max_rows 256
#define max_cols 256
#define max_levels 100
```

7. Each position (square) in this matrix may be occupied by a wall, the player, spikes, a bunny, a goal or a field—and these are not all mutually exclusive. For example, we may have, in a single square, a bunny on a goal plus an open field with a bunny on it (that's right, two bunnies there). 'Invalid' configurations may also occur. For instance, a wall with spikes denotes rightmost entry of a line in a level's matrix. We list all cases in ⟨ Game loop 32 ⟩.

```
⟨ Global declarations 7 ⟩ ≡
  typedef struct {
    char wall:1;
    char player:1;
    char spikes:1;
    char bunny:1;
    char goal:1;
    char empty_field:1;      /* nothing on top of the field? */
    char bunny_on_field:1;     /* is there a bunny on top of the field? */
    char player_on_field:1;
  } board_square;
```

See also sections 8, 14, 28, and 29.

This code is used in section 2.

8. A variable of type *board* stores the state of the game at a given point. For convenience, we store the number of 'unfilled' goals (goal positions which do not yet have a rabbit in them), the player position. We also record the last field position and whether a field is being set or is open. (Note that the **bool** type is defined by *ncurses*.)

```
⟨ Global declarations 7 ⟩ +≡
  typedef unsigned char coordinate;
  typedef struct board {
    board_square squares[max_rows][max_cols];
    unsigned char n_rows;      /* board nas no square from row n_rows onwards */
    unsigned char n_cols;      /* board has no suqare from column n_cols onwards */
    bool setting_field;
    bool field_on;
    bool carrying_bunny;
    coordinate field_i;
    coordinate field_j;
    coordinate player_i;
    coordinate player_j;
    unsigned char empty_goals;      /* how many goals need bunnies? */
  } board;
  board levels[max_levels];
  unsigned int number_of_levels = 0;
```

9. The input file consists of levels, separated by blank lines. Each line describing a level is first copied into an array *buffer* before being loaded to the *levels* array. No line in the input file may be longer than *max_cols* bytes, including the (carriage return) line break. We exit if line is too long.

```
#define curr_input_lv levels[number_of_levels]
#define curr_n_rows levels[number_of_levels].n_rows
#define curr_n_cols levels[number_of_levels].n_cols
```

```
⟨ Load game levels 9 ⟩ ≡
  ⟨ Open input file 10 ⟩
  number_of_levels = 0;
  at_least_one_level = FALSE;
  ⟨ Initialize level 11 ⟩
  do {
    ⟨ Copy next level line into buffer; exit if invalid input 15 ⟩
    ⟨ Exit if blank file 21 ⟩
    ⟨ Insert buffer line in curr_input_lv (maybe after incrementing number_of_levels) 22 ⟩
  } while (¬reached_eof);
  printf("%d␣levels␣read.\n", number_of_levels);
  {
    int i = 0;
    while (i < number_of_levels) print_level(i++);
  }
```

This code is used in section 27.

10.

```
⟨ Open input file 10 ⟩ ≡
  if ((levels_input = fopen(argv[1], "r")) ≡ Λ) {
    fprintf(stderr, "Could␣not␣read␣levels!␣Wrong␣path,␣perhaps?\n");
    exit(1);
  }
```

This code is used in section 9.

11.

```
⟨ Initialize level 11 ⟩ ≡
  curr_n_rows = curr_n_cols = 0;
  curr_input_lv.empty_goals = 0;
  curr_input_lv.setting_field = FALSE;
  curr_input_lv.field_on = FALSE;
  curr_input_lv.carrying_bunny = FALSE;
```

This code is used in sections 9 and 22.

12.

```
⟨ Local variables of main 12 ⟩ ≡
  int reached_eof = FALSE;
  FILE *levels_input = Λ;
  bool at_least_one_level;
```

See also sections 16, 23, 33, and 46.

This code is used in section 2.

13. Ncurses defines `TRUE` and `FALSE`.

```
⟨ Header files 3 ⟩ +≡
#include <ncurses.h>
```

14.

```
#define levels_lij levels[l].squares[i][j]
⟨ Global declarations 7 ⟩ +≡
  void print_level(int l)
  {
    int i = 0;
    int j = 0;
    printf("%d-th level has %d rows and %d columns.\n", l, levels[l].n_rows, levels[l].n_cols);
    i = j = 0;
    while (i < levels[l].n_rows) {
      j = 0;
      while (TRUE) {
        if (levels_lij.wall ∧ levels_lij.spikes) {
          printf("\n");
          i++;
          fflush(stdout);
          break;
        }
        if (levels_lij.wall) printf("#");
        else if (levels_lij.spikes) printf(",");
        else if (levels_lij.player) printf("@");
        else if (levels_lij.bunny) {
          if (levels_lij.goal) printf("B");
          else printf("b");
        }
        else if (levels_lij.goal) printf("+");
        else printf(".");
        j++;
        fflush(stdout);
      }
    }
  }
```

15. When we are copying a line into *buffer*, we say that we reach a *stop* if one of three things happens: we reach the end of the line (i.e., line feed) or end of file (i.e., `EOF`) or if the line is too long.

```
⟨ Copy next level line into buffer; exit if invalid input 15 ⟩ ≡
  buf_ptr = buffer;
  skipped_blank_line = FALSE;
  while (TRUE) {
    ⟨ Advance buf_ptr to next nonblank or next stop 17 ⟩
    ⟨ Exit if line too long 18 ⟩
    if (*buf_ptr ≡ EOF) {
      reached_eof = TRUE;
      break;
    }
    else if (*buf_ptr ≡ '\n') {
      skipped_blank_line = TRUE;
      input_line_number++;
      continue;
    }
    else {      /* line not blank */
      ⟨ Scan remainder of input line; exit if line too long 19 ⟩
      if (*buf_ptr ≡ '\n') input_line_number++;
      break;
    }
  }
```

This code is used in section 9.

16.

```
⟨ Local variables of main 12 ⟩ +≡
  int *buf_ptr;
  bool skipped_blank_line;
  unsigned int input_line_number = 0;
```

17.

```
⟨ Advance buf_ptr to next nonblank or next stop 17 ⟩ ≡
  while ((*buf_ptr = fgetc(levels_input)) ≠ EOF ∧ *buf_ptr ≠ '\n' ∧ (buf_ptr − buffer) <
      max_cols ∧ *buf_ptr ≡ '␣') {
    buf_ptr++;
  }
```

This code is used in section 15.

18.

```
⟨ Exit if line too long 18 ⟩ ≡
  if (buf_ptr − buffer ≥ max_cols) {
    fprintf(stderr, "%s:%d:␣Input␣file␣has␣line␣exceeding␣%d␣characters\n", argv[1],
        input_line_number + 1, max_cols);
    fclose(levels_input);
    exit(1);
  }
```

This code is used in sections 15 and 19.

19. The next loop is a do-while rather than a while since *buf_ptr* points to the first non-blank character found in the current input line.

```
⟨ Scan remainder of input line; exit if line too long 19 ⟩ ≡
  do {
    buf_ptr++;
  } while ((*buf_ptr = fgetc(levels_input)) ≠ EOF ∧ *buf_ptr ≠ '\n' ∧ buf_ptr − buffer < max_cols);
  ⟨ Exit if line too long 18 ⟩
```

This code is used in section 15.

20. Storing levels. Most of what we need to do is copy non-blank of the input file to the board of the corresponding level. We also scan the board, counting goals without bunnies and storing the player's starting coordinates.

21. ⟨ Exit if blank file 21 ⟩ ≡
```
if (¬at_least_one_level ∧ reached_eof) {
  fprintf(stderr, "Blimey, a blank file! Are you a whitespace programmer?\n");
  exit(1);
}
```
This code is used in section 9.

22. Blank lines indicate a new level, but note that the first level may or *may not* be preceeded by blank lines. The last used position of a line in *curr_input_lv.squares* is a square with both *wall* and *spikes* set.

⟨ Insert *buffer* line in *curr_input_lv* (maybe after incrementing *number_of_levels*) 22 ⟩ ≡
```
buf_aux = buffer;
if (at_least_one_level ∧ (skipped_blank_line ∨ reached_eof)) {
  number_of_levels++;
  if (number_of_levels ≡ max_levels) {
    fprintf(stderr, "I'm sorry, I wasn't set to handle so many levels!\n");
    fclose(levels_input);
    exit(1);
  }
  if (¬reached_eof) {
    ⟨ Initialize level 11 ⟩
  }
}
at_least_one_level = TRUE;
j = 0;      /* current board column */
while (buf_aux ≠ buf_ptr) {
  ⟨ Set curr_input_lv.squares[curr_n_rows][j], maybe setting player coordinates and empty_goals 24 ⟩
  buf_aux++;
  j++;
}
curr_input_lv.squares[curr_n_rows][j].wall = TRUE;
curr_input_lv.squares[curr_n_rows][j].spikes = TRUE;
if (j > curr_input_lv.n_cols) curr_input_lv.n_cols = j;
curr_n_rows++;
```
This code is used in section 9.

23. ⟨ Local variables of *main* 12 ⟩ +≡
```
int buffer[max_cols + 2];      /* to store the input lines */
int *buf_aux;
unsigned int j;
```

24. The possible elements ocurring in a level are a the player, walls, spaces, spikes, bunnies and goals; moreover, either the player or a bunny may be on top of a goal (fields do not appear in level design).

```
#define input_board_wall '#'
#define input_board_space '␣'
#define input_board_space_alt '.'
#define input_board_spike ','
#define input_board_bunny 'b'
#define input_board_goal '+'
#define input_board_player '@'
#define input_board_bunny_on_goal 'B'
#define input_board_player_on_goal 'A'
```

⟨ Set *curr_input_lv.squares*[*curr_n_rows*][*j*], maybe setting player coordinates and *empty_goals* 24 ⟩ ≡

```
⟨ Set all fields of curr_input_lv.squares[curr_n_rows][j] to FALSE 25 ⟩
switch (*buf_aux) {
case input_board_wall: curr_input_lv.squares[curr_n_rows][j].wall = TRUE;
  break;
case input_board_space: break;
case input_board_space_alt: break;
case input_board_spike: curr_input_lv.squares[curr_n_rows][j].spikes = TRUE;
    /* TODO(tn): either spike or spikeS, not both! */
  break;
case input_board_bunny: curr_input_lv.squares[curr_n_rows][j].bunny = TRUE;
  break;
case input_board_goal: curr_input_lv.squares[curr_n_rows][j].goal = TRUE;
  curr_input_lv.empty_goals++;
  break;
case input_board_player: curr_input_lv.squares[curr_n_rows][j].player = TRUE;
  ⟨ Set player's initial position 26 ⟩
  break;
case input_board_bunny_on_goal: curr_input_lv.squares[curr_n_rows][j].bunny = TRUE;
  curr_input_lv.squares[curr_n_rows][j].goal = TRUE;
  break;
case input_board_player_on_goal: curr_input_lv.squares[curr_n_rows][j].player = TRUE;
  curr_input_lv.squares[curr_n_rows][j].goal = TRUE;
  curr_input_lv.empty_goals++;
  ⟨ Set player's initial position 26 ⟩
  break;
default: fprintf(stderr, "%s:%d:%d:␣Invalid␣character␣in␣level␣file.\n", argv[1],
      input_line_number, buf_aux − buffer);
  exit(1);
}
```

This code is used in section 22.

25. ⟨ Set all fields of *curr_input_lv.squares*[*curr_n_rows*][*j*] to `FALSE` 25 ⟩ ≡

```
curr_input_lv.squares[curr_n_rows][j].wall = FALSE;
curr_input_lv.squares[curr_n_rows][j].player = FALSE;
curr_input_lv.squares[curr_n_rows][j].spikes = FALSE;
curr_input_lv.squares[curr_n_rows][j].bunny = FALSE;
curr_input_lv.squares[curr_n_rows][j].goal = FALSE;
curr_input_lv.squares[curr_n_rows][j].empty_field = FALSE;
curr_input_lv.squares[curr_n_rows][j].bunny_on_field = FALSE;
curr_input_lv.squares[curr_n_rows][j].player_on_field = FALSE;
```

This code is used in section 24.

26. ⟨ Set player's initial position 26 ⟩ ≡

```
curr_input_lv.player_i = curr_n_rows;
curr_input_lv.player_j = j;
```

This code is used in section 24.

27. ⟨ Initialize game 27 ⟩ ≡

```
⟨ Load game levels 9 ⟩
curr_level = 0;
curr_state = levels[curr_level];
```

See also section 30.

This code is used in section 2.

28. ⟨ Global declarations 7 ⟩ +≡

```
unsigned int curr_level;
board curr_state;
```

29. Acting and undoing. To be able to unwind a player's moves, we need to choose a way to represent each one of them. This goes hand in hand with the description of current game state—the goal being that it is possible to reverse each move given the last move and the current game state. Broadly, a move may be of one of the following types (numbers in parenthesis explained below).

- **walk** (4),
- **push** (walk while pushing bunny) (4),
- **begin setting field** (1),
- **cancel setting field** (1),
- **open field** (8),
- **close field** (2).

Walking and pushing come in four flavours (each possible direction for the action); opening a field comes with both a direction and a flag to indicate whether a bunny was contained in the field, and closing a field just needs to indicate whether a bunny was in the field. The numbers between parenthesis indicate how many different flavours each action has. So, altogether, we have 16 actions.

Actually, this is not the whole story, we missed **undo** and **restart**, but those do not appear in the history.

History is just an array of actions with length at most *max_hist*.

```
#define walk_left 0
#define walk_up 1
#define walk_right 2
#define walk_down 3
#define push_left 4
#define push_up 5
#define push_right 6
#define push_down 7
#define begin_set_field 8
#define cancel_set_field 9
#define open_field_left 10
#define open_field_up 11
#define open_field_right 12
#define open_field_down 13
#define close_field_empty 14
#define close_field_with_player 15
#define close_field_with_bunny 16
#define max_hist (max_rows * max_cols * 2)
#define add_to_history(a) do
        {
          switch (a) {
          case walk_left: printf("add_to_history(walk_left)\n");
            break;
          case walk_up: printf("add_to_history(walk_up)\n");
            break;
          case walk_right: printf("add_to_history(walk_right)\n");
            break;
          case walk_down: printf("add_to_history(walk_down)\n");
            break;
          case push_left: printf("add_to_history(push_left)\n");
            break;
          case push_up: printf("add_to_history(push_up)\n");
            break;
          case push_right: printf("add_to_history(push_right)\n");
            break;
          case push_down: printf("add_to_history(push_down)\n");
```

```
      break;
    case begin_set_field: printf("add_to_history(begin_set_field)\n");
      break;
    case cancel_set_field: printf("add_to_history(cancel_set_field)\n");
      break;
    case open_field_left: printf("add_to_history(open_field_left)\n");
      break;
    case open_field_up: printf("add_to_history(open_field_up)\n");
      break;
    case open_field_right: printf("add_to_history(open_field_right)\n");
      break;
    case open_field_down: printf("add_to_history(open_field_down)\n");
      break;
    case close_field_empty: printf("add_to_history(close_field_empty)\n");
      break;
    case close_field_with_player: printf("add_to_history(close_field_with_player)\n");
      break;
    case close_field_with_bunny: printf("add_to_history(close_field_with_bunny)\n");
      break;
    default: printf("add_to_history(%d)\n", a);
      break;
    }
    if (hist_index < max_hist) {
      history[hist_index++] = a;
    }
    else history_disabled = TRUE;      /* you have gone too far! */
  }
  while (0)
```

⟨ Global declarations 7 ⟩ +≡

```
typedef unsigned char action;
bool history_disabled = FALSE;
action history[max_hist];
unsigned int hist_index = 0;      /* next unused slot in history */
```

30. Drawing on the screen. We use ncurses to draw the screen and intercept key presses.

```
⟨ Initialize game 27 ⟩ +≡
  initscr( );      /* start curses mode */
  raw( );      /* line buffering disabled */
  keypad(stdscr, TRUE);      /* we get F1, F2 etc.. */
  noecho( );      /* don't echo() while we do getch */
  curs_set(0);      /* invisible cursor */
  if (has_colors( ) ≡ FALSE) {
    fprintf(stderr, "Your␣terminal␣does␣not␣support␣color␣=(\n");
    endwin( );
    goto end;
  }
  start_color( );      /* Start color */
  ⟨ Initialize color pairs 44 ⟩
```

31. ⟨ Clean up and exit 31 ⟩ ≡

```
  endwin( );      /* End ncurses mode */
```

This code is used in section 2.

32. The current game state is stored in Besides rendering the current game

```
⟨ Game loop 32 ⟩ ≡
  quit_game = false;
  while (¬quit_game) {
    ⟨ Render game 34 ⟩
    ch = getch( );
    ⟨ Process user input 49 ⟩
  }
```

This code is cited in section 7.

This code is used in section 2.

33. ⟨ Local variables of *main* 12 ⟩ +≡

```
  int ch = '␣';
  bool quit_game = false;
```

34.

```
⟨ Render game 34 ⟩ ≡
  clear( );
  getmaxyx(stdscr, max_scr_row, max_scr_col);
  i = j = 0;
  while (i < curr_state.n_rows) {
    move(i + 1, (max_scr_col − curr_state.n_cols)/2);      /* center puzzle */
    ⟨ Render ith line of the board 35 ⟩
    i++;
  }
  ⟨ Render debug info 45 ⟩
  refresh( );
```

This code is used in section 32.

35.

#define *curr_board_ij* *curr_state.squares*[*i*][*j*]

⟨ Render *i*th line of the board 35 ⟩ ≡

```
j = 0;
while (TRUE) {
  if (board_inspector_on ∧ i ≡ debug_i ∧ j ≡ debug_j) attron(A_UNDERLINE);
  if (curr_state.setting_field ∧ ⟨ curr_board_ij is field candidate 37 ⟩) attron(A_REVERSE);
  if (⟨ curr_board_ij is an open field 38 ⟩) {
    ⟨ Render field 39 ⟩
  }
  else {
    ⟨ Render position without field 43 ⟩
  }
  if (curr_state.setting_field ∧ ⟨ curr_board_ij is field candidate 37 ⟩) attroff(A_REVERSE);
  if (board_inspector_on ∧ i ≡ debug_i ∧ j ≡ debug_j) attroff(A_UNDERLINE);
  j++;
  ⟨ Break if at end of line 36 ⟩
}
```

This code is used in section 34.

36. ⟨ Break if at end of line 36 ⟩ ≡

```
if (curr_board_ij.wall ∧ curr_board_ij.spikes) {
  break;
}
```

This code is used in section 35.

37. ⟨ *curr_board_ij* is field candidate 37 ⟩ ≡

$((i \equiv curr\_i \wedge ((j > curr\_j \wedge j \equiv curr\_j + 1) \vee (j < curr\_j \wedge j + 1 \equiv curr\_j))) \vee (j \equiv curr\_j \wedge ((i > curr\_i \wedge i \equiv curr\_i + 1) \vee (i < curr\_i \wedge i + 1 \equiv curr\_i)))) \wedge \neg(curr\_board\_ij.wall)$

This code is used in section 35.

38. ⟨ *curr_board_ij* is an open field 38 ⟩ ≡

$curr\_state.field\_on \wedge curr\_state.field\_i \equiv i \wedge curr\_state.field\_j \equiv j$

This code is used in section 35.

39. Whatever is underneath a field is only rendered if the field is empty.

⟨ Render field 39 ⟩ ≡

```
if (curr_board_ij.empty_field) {
  ⟨ Render empty field 40 ⟩
}
else if (curr_board_ij.bunny_on_field) {
  ⟨ Render bunny on field 41 ⟩
}
else if (curr_board_ij.player_on_field) {
  ⟨ Render player on field 42 ⟩
}
```

This code is used in section 35.

40. ⟨ Render empty field 40 ⟩ ≡

```
attron(A_BOLD | COLOR_PAIR(9));
if (curr_board_ij.goal) {
  if (curr_board_ij.bunny) printw("B");
  else printw("+");
}
else if (curr_board_ij.spikes) {
  printw(",");
}
else if (curr_board_ij.bunny) {
  printw("b");
}
else {
  printw("␣");
}
attroff(A_BOLD | COLOR_PAIR(9));
```

This code is used in section 39.

41. ⟨ Render bunny on field 41 ⟩ ≡

```
attron(COLOR_PAIR(7));
printw("b");
attroff(COLOR_PAIR(7));
```

This code is used in section 39.

42. ⟨ Render player on field 42 ⟩ ≡

```
attron(A_BOLD | COLOR_PAIR(8));
printw("@");
attroff(A_BOLD | COLOR_PAIR(8));
```

This code is used in section 39.

43. ⟨ Render position without field 43 ⟩ ≡

```
if (curr_board_ij.wall) {
  attron(COLOR_PAIR(2));
  printw("#");
  attroff(COLOR_PAIR(2));
}
else if (curr_board_ij.spikes) {
  attron(COLOR_PAIR(4));
  printw(",");
  attroff(COLOR_PAIR(4));
}
else if (curr_board_ij.player) {
  attron(A_BOLD | COLOR_PAIR(1));
  printw("@");
  attroff(A_BOLD | COLOR_PAIR(1));
}
else if (curr_board_ij.bunny) {
  attron(COLOR_PAIR(3));
  if (curr_board_ij.goal) printw("B");
  else printw("b");
  attroff(COLOR_PAIR(3));
}
else if (curr_board_ij.goal) {
  attron(A_BOLD);
  printw("+");
  attroff(A_BOLD);
}
else {
  attron(COLOR_PAIR(5));
  printw(".");
  attroff(COLOR_PAIR(5));
}
```

This code is used in section 35.

44. ⟨ Initialize color pairs 44 ⟩ ≡

```
init_pair(1, COLOR_RED, COLOR_BLACK);      /* player */
init_pair(2, COLOR_BLACK, COLOR_WHITE);    /* wall */
init_pair(3, COLOR_CYAN, COLOR_BLACK);     /* bunny */
init_pair(4, COLOR_YELLOW, COLOR_BLACK);   /* spikes */
init_color(COLOR_GREEN, 500, 500, 500);    /* green will be gray, TODO(tn) test if color changeable */
init_pair(5, COLOR_GREEN, COLOR_BLACK);    /* space */
init_pair(6, COLOR_BLACK, COLOR_GREEN);    /* empty field whatever is under is black */
init_pair(7, COLOR_CYAN, COLOR_GREEN);     /* bunny on field */
init_pair(8, COLOR_RED, COLOR_GREEN);      /* player on field */
init_pair(9, COLOR_WHITE, COLOR_GREEN);    /* stuff under field */
```

This code is used in section 30.

45.

#define *curr_debug_square* *curr_state.squares*[*debug_i*][*debug_j*]

⟨ Render debug info 45 ⟩ ≡

```
if (show_player_info) {
  ⟨ Render player info 47 ⟩
}
if (board_inspector_on) {
  ⟨ Render board debugger 48 ⟩
}
attron(A_BLINK | COLOR_PAIR(5));
mvprintw(max_scr_row − 1, 0, "Keys: ");
attroff(A_BLINK | COLOR_PAIR(5));
attron(A_BOLD);
printw("q");
attroff(A_BOLD);
attron(A_BLINK | COLOR_PAIR(5));
printw(": ends the game; ");
attroff(A_BLINK | COLOR_PAIR(5));
attron(A_BOLD);
printw("arrows");
attroff(A_BOLD);
attron(A_BLINK | COLOR_PAIR(5));
printw(": move; ");
attroff(A_BLINK | COLOR_PAIR(5));
attron(A_BOLD);
printw("x");
attroff(A_BOLD);
attron(A_BLINK | COLOR_PAIR(5));
printw(": magic field; ");
attroff(A_BLINK | COLOR_PAIR(5));
attron(A_BOLD);
printw("z");
attroff(A_BOLD);
attron(A_BLINK | COLOR_PAIR(5));
printw(": undo; ");
attroff(A_BLINK | COLOR_PAIR(5));
attron(A_BOLD);
printw("r");
attroff(A_BOLD);
attron(A_BLINK | COLOR_PAIR(5));
printw(": restart level");
attroff(A_BLINK | COLOR_PAIR(5));
```

This code is used in section 34.

46. ⟨ Local variables of *main* 12 ⟩ +≡

```
unsigned int i;
int max_scr_row, max_scr_col;
int debug_i = 0, debug_j = 0;
bool board_inspector_on = FALSE, show_player_info = FALSE;
```

47. ⟨ Render player info 47 ⟩ ≡

```
attron(COLOR_PAIR(5));
mvprintw(max_scr_row − 11, 0, "␣␣␣␣␣␣␣␣␣␣level:␣%d/%d␣(%d,␣%d)\n", curr_level + 1, number_of_levels,
    curr_state.n_rows, curr_state.n_cols);
printw("␣setting_field:␣%s\n", curr_state.setting_field ? "yes" : "no");
printw("␣␣␣␣␣␣␣␣player:␣(%d,␣%d)\n", curr_i, curr_j);
printw("␣␣␣␣␣␣field_on:␣%s\n", curr_state.field_on ? "yes" : "no");
printw("carrying_bunny:␣%s\n", curr_state.carrying_bunny ? "yes" : "no");
printw("␣␣␣␣␣␣␣␣␣field:␣(%d,␣%d)\n", curr_state.field_i, curr_state.field_j);
printw("␣␣␣empty_goals:␣%d\n\n", curr_state.empty_goals);
printw("␣␣␣␣␣␣␣history:␣%d␣actions␣(history␣%s␣disabled)\n\n", hist_index,
    history_disabled ? "IS" : "NOT");
attroff(COLOR_PAIR(5));
```

This code is used in section 45.

48. ⟨ Render board debugger 48 ⟩ ≡

```
mvprintw(max_scr_row − 11, max_scr_col − 30, "debugger␣position:␣(%d,␣%d)\n", debug_i, debug_j);
if (curr_debug_square.wall) {
  attron(COLOR_PAIR(4));
  mvprintw(max_scr_row − 10, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall:␣%s\n",
      curr_debug_square.wall ? "true" : "false");
  attroff(COLOR_PAIR(4));
}
else {
  attron(COLOR_PAIR(5));
  mvprintw(max_scr_row − 10, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣wall:␣%s\n",
      curr_debug_square.wall ? "true" : "false");
  attroff(COLOR_PAIR(5));
}
if (curr_debug_square.player) {
  attron(COLOR_PAIR(4));
  mvprintw(max_scr_row − 9, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣player:␣%s\n",
      curr_debug_square.player ? "true" : "false");
  attroff(COLOR_PAIR(4));
}
else {
  attron(COLOR_PAIR(5));
  mvprintw(max_scr_row − 9, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣player:␣%s\n",
      curr_debug_square.player ? "true" : "false");
  attroff(COLOR_PAIR(5));
}
if (curr_debug_square.spikes) {
  attron(COLOR_PAIR(4));
  mvprintw(max_scr_row − 8, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣spikes:␣%s\n",
      curr_debug_square.spikes ? "true" : "false");
  attroff(COLOR_PAIR(4));
}
else {
  attron(COLOR_PAIR(5));
  mvprintw(max_scr_row − 8, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣spikes:␣%s\n",
      curr_debug_square.spikes ? "true" : "false");
  attroff(COLOR_PAIR(5));
}
if (curr_debug_square.bunny) {
  attron(COLOR_PAIR(4));
  mvprintw(max_scr_row − 7, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣bunny:␣%s\n",
      curr_debug_square.bunny ? "true" : "false");
  attroff(COLOR_PAIR(4));
}
else {
  attron(COLOR_PAIR(5));
  mvprintw(max_scr_row − 7, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣bunny:␣%s\n",
      curr_debug_square.bunny ? "true" : "false");
  attroff(COLOR_PAIR(5));
}
if (curr_debug_square.goal) {
  attron(COLOR_PAIR(4));
```

```
    mvprintw(max_scr_row − 6, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣goal:␣%s\n",
        curr_debug_square.goal ? "true" : "false");
    attroff(COLOR_PAIR(4));
  }
  else {
    attron(COLOR_PAIR(5));
    mvprintw(max_scr_row − 6, max_scr_col − 30, "␣␣␣␣␣␣␣␣␣␣␣␣␣␣goal:␣%s\n",
        curr_debug_square.goal ? "true" : "false");
    attroff(COLOR_PAIR(5));
  }
  if (curr_debug_square.empty_field) {
    attron(COLOR_PAIR(4));
    mvprintw(max_scr_row − 5, max_scr_col − 30, "␣␣␣␣␣␣␣empty_field:␣%s\n",
        curr_debug_square.empty_field ? "true" : "false");
    attroff(COLOR_PAIR(4));
  }
  else {
    attron(COLOR_PAIR(5));
    mvprintw(max_scr_row − 5, max_scr_col − 30, "␣␣␣␣␣␣␣empty_field:␣%s\n",
        curr_debug_square.empty_field ? "true" : "false");
    attroff(COLOR_PAIR(5));
  }
  if (curr_debug_square.bunny_on_field) {
    attron(COLOR_PAIR(4));
    mvprintw(max_scr_row − 4, max_scr_col − 30, "␣␣␣bunny_on_field:␣%s\n",
        curr_debug_square.bunny_on_field ? "true" : "false");
    attroff(COLOR_PAIR(4));
  }
  else {
    attron(COLOR_PAIR(5));
    mvprintw(max_scr_row − 4, max_scr_col − 30, "␣␣␣bunny_on_field:␣%s\n",
        curr_debug_square.bunny_on_field ? "true" : "false");
    attroff(COLOR_PAIR(5));
  }
  if (curr_debug_square.player_on_field) {
    attron(COLOR_PAIR(4));
    mvprintw(max_scr_row − 3, max_scr_col − 30, "␣␣player_on_field:␣%s\n",
        curr_debug_square.player_on_field ? "true" : "false");
    attroff(COLOR_PAIR(4));
  }
  else {
    attron(COLOR_PAIR(5));
    mvprintw(max_scr_row − 3, max_scr_col − 30, "␣␣player_on_field:␣%s\n",
        curr_debug_square.player_on_field ? "true" : "false");
    attroff(COLOR_PAIR(5));
  }
```

This code is used in section 45.

49. User input.

```
#define curr_i (curr_state.player_i)
#define curr_j (curr_state.player_j)
#define curr_square curr_state.squares[curr_i][curr_j]
#define west_square curr_state.squares[curr_i][curr_j - 1]
#define north_square curr_state.squares[curr_i - 1][curr_j]
#define east_square curr_state.squares[curr_i][curr_j + 1]
#define south_square curr_state.squares[curr_i + 1][curr_j]
#define wwest_square curr_state.squares[curr_i][curr_j - 2]
#define nnorth_square curr_state.squares[curr_i - 2][curr_j]
#define eeast_square curr_state.squares[curr_i][curr_j + 2]
#define ssouth_square curr_state.squares[curr_i + 2][curr_j]
```

⟨ Process user input 49 ⟩ ≡

```
switch (ch) {
case 'q': quit_game = TRUE;
  break;
case KEY_LEFT:
  if (board_inspector_on) debug_j--;
  else ⟨ LEFT action 52 ⟩
  break;
case KEY_UP:
  if (board_inspector_on) debug_i--;
  else ⟨ UP action 53 ⟩
  break;
case KEY_RIGHT:
  if (board_inspector_on) debug_j++;
  else ⟨ RIGHT action 54 ⟩
  break;
case KEY_DOWN:
  if (board_inspector_on) debug_i++;
  else ⟨ DOWN action 55 ⟩
  break;
case 'x': ⟨ FIELD action 90 ⟩
  break;
case 'z':
  if (¬history_disabled) ⟨ UNDO action 93 ⟩
  break;
case 'r': ⟨ RESTART game 100 ⟩
  break;
case 'J': ⟨ Go to next level 50 ⟩
  break;
case 'K': ⟨ Go to previous level 51 ⟩
  break;
case 'D':
  if (board_inspector_on) board_inspector_on = FALSE;
  else {
    board_inspector_on = TRUE;
    debug_i = curr_i;
    debug_j = curr_j;
  }
  break;
case 'G':
```

```
    if (show_player_info) show_player_info = FALSE;
    else show_player_info = TRUE;
    break;
  }
```

This code is used in section 32.

50. ⟨ Go to next level 50 ⟩ ≡

```
  if (curr_level < number_of_levels − 1) {
    curr_level++;
    curr_state = levels[curr_level];
    hist_index = 0;      /* reset history */
  }
```

This code is used in sections 49, 60, 61, 62, 63, 84, 85, 86, and 87.

51. ⟨ Go to previous level 51 ⟩ ≡

```
  if (curr_level > 0) {
    curr_level−−;
    curr_state = levels[curr_level];
    hist_index = 0;      /* reset history */
  }
```

This code is used in section 49.

52. ⟨ LEFT action 52 ⟩ ≡

```
  if (curr_state.setting_field) {
    if (⟨ Opening a field left is possible 56 ⟩) {
      add_to_history(open_field_left);
      ⟨ Open field left 60 ⟩
    }
  }
  else if (⟨ Player can walk left 64 ⟩) {
    add_to_history(walk_left);
    ⟨ Walk left 68 ⟩
  }
  else if (⟨ Player can push left 72 ⟩) {
    add_to_history(push_left);
    ⟨ Push left 84 ⟩
  }
```

This code is used in section 49.

53. ⟨ UP action 53 ⟩ ≡

```
if (curr_state.setting_field) {
  if (⟨ Opening a field up is possible 57 ⟩) {
    add_to_history(open_field_up);
    ⟨ Open field up 61 ⟩
  }
}
else if (⟨ Player can walk up 65 ⟩) {
  add_to_history(walk_up);
  ⟨ Walk up 69 ⟩
}
else if (⟨ Player can push up 75 ⟩) {
  add_to_history(push_up);
  ⟨ Push up 85 ⟩
}
```

This code is used in section 49.

54. ⟨ RIGHT action 54 ⟩ ≡

```
if (curr_state.setting_field) {
  if (⟨ Opening a field right is possible 58 ⟩) {
    add_to_history(open_field_right);
    ⟨ Open field right 62 ⟩
  }
}
else if (⟨ Player can walk right 66 ⟩) {
  add_to_history(walk_right);
  ⟨ Walk right 70 ⟩
}
else if (⟨ Player can push right 78 ⟩) {
  add_to_history(push_right);
  ⟨ Push right 86 ⟩
}
```

This code is used in section 49.

55. ⟨ DOWN action 55 ⟩ ≡

```
if (curr_state.setting_field) {
  if (⟨ Opening a field down is possible 59 ⟩) {
    add_to_history(open_field_down);
    ⟨ Open field down 63 ⟩
  }
}
else if (⟨ Player can walk down 67 ⟩) {
  add_to_history(walk_down);
  ⟨ Walk down 71 ⟩
}
else if (⟨ Player can push down 81 ⟩) {
  add_to_history(push_down);
  ⟨ Push down 87 ⟩
}
```

This code is used in section 49.

56. ⟨ Opening a field left is possible 56 ⟩ ≡
```
curr_j > 0 ∧ west_square.wall ≡ FALSE
```
This code is used in section 52.

57. ⟨ Opening a field up is possible 57 ⟩ ≡
```
curr_i > 0 ∧ north_square.wall ≡ FALSE
```
This code is used in section 53.

58. ⟨ Opening a field right is possible 58 ⟩ ≡
```
curr_j < curr_state.n_cols ∧ east_square.wall ≡ FALSE
```
This code is used in section 54.

59. ⟨ Opening a field down is possible 59 ⟩ ≡
```
curr_i < curr_state.n_rows ∧ south_square.wall ≡ FALSE
```
This code is used in section 55.

60. ⟨ Open field left 60 ⟩ ≡
```
curr_state.setting_field = FALSE;
if (curr_state.carrying_bunny) {
  west_square.bunny_on_field = TRUE;
  curr_state.carrying_bunny = FALSE;
  if (west_square.goal ∧ ¬west_square.bunny) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else west_square.empty_field = TRUE;
curr_state.field_i = curr_i;
curr_state.field_j = curr_j − 1;
curr_state.field_on = TRUE;
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
```
This code is used in section 52.

61. ⟨ Open field up 61 ⟩ ≡
```
curr_state.setting_field = FALSE;
if (curr_state.carrying_bunny) {
  north_square.bunny_on_field = TRUE;
  curr_state.carrying_bunny = FALSE;
  if (north_square.goal ∧ ¬north_square.bunny) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else north_square.empty_field = TRUE;
curr_state.field_i = curr_i − 1;
curr_state.field_j = curr_j;
curr_state.field_on = TRUE;
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
```
This code is used in section 53.

62. ⟨ Open field right 62 ⟩ ≡
```
curr_state.setting_field = FALSE;
if (curr_state.carrying_bunny) {
  east_square.bunny_on_field = TRUE;
  curr_state.carrying_bunny = FALSE;
  if (east_square.goal ∧ ¬east_square.bunny) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else east_square.empty_field = TRUE;
curr_state.field_i = curr_i;
curr_state.field_j = curr_j + 1;
curr_state.field_on = TRUE;
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
```
This code is used in section 54.

63. ⟨ Open field down 63 ⟩ ≡
```
curr_state.setting_field = FALSE;
if (curr_state.carrying_bunny) {
  south_square.bunny_on_field = TRUE;
  curr_state.carrying_bunny = FALSE;
  if (south_square.goal ∧ ¬south_square.bunny) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else south_square.empty_field = TRUE;
curr_state.field_i = curr_i + 1;
curr_state.field_j = curr_j;
curr_state.field_on = TRUE;
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
```
This code is used in section 55.

64. If a player must push left, then we say they cannot *walk* left.

⟨ Player can walk left 64 ⟩ ≡
```
curr_j > 0 ∧ (west_square.empty_field ∨ ¬(west_square.wall ∨ west_square.spikes ∨ west_square.bunny ∨
    west_square.bunny_on_field))
```
This code is used in section 52.

65. ⟨ Player can walk up 65 ⟩ ≡
```
curr_i > 0 ∧ (north_square.empty_field ∨ ¬(north_square.wall ∨ north_square.spikes ∨ north_square.bunny ∨
    north_square.bunny_on_field))
```
This code is used in section 53.

66. ⟨ Player can walk right 66 ⟩ ≡
```
curr_j < curr_state.n_cols ∧ (east_square.empty_field ∨ ¬(east_square.wall ∨ east_square.spikes ∨
    east_square.bunny ∨ east_square.bunny_on_field))
```
This code is used in section 54.

67. ⟨ Player can walk down 67 ⟩ ≡

$curr\_i < curr\_state.n\_rows \wedge (south\_square.empty\_field \vee \neg(south\_square.wall \vee south\_square.spikes \vee south\_square.bunny \vee south\_square.bunny\_on\_field))$

This code is used in section 55.

68. ⟨ Walk left 68 ⟩ ≡

```
if (curr_square.player_on_field) {
  curr_square.player_on_field = FALSE;
  curr_square.empty_field = TRUE;
}
else curr_square.player = FALSE;
curr_state.player_j --;
if (curr_square.empty_field) {
  curr_square.player_on_field = TRUE;
  curr_square.empty_field = FALSE;
}
else curr_square.player = TRUE;
```

This code is used in sections 52, 84, and 93.

69. ⟨ Walk up 69 ⟩ ≡

```
if (curr_square.player_on_field) {
  curr_square.player_on_field = FALSE;
  curr_square.empty_field = TRUE;
}
else curr_square.player = FALSE;
curr_state.player_i --;
if (curr_square.empty_field) {
  curr_square.player_on_field = TRUE;
  curr_square.empty_field = FALSE;
}
else curr_square.player = TRUE;
```

This code is used in sections 53, 85, and 93.

70. ⟨ Walk right 70 ⟩ ≡

```
if (curr_square.player_on_field) {
  curr_square.player_on_field = FALSE;
  curr_square.empty_field = TRUE;
}
else curr_square.player = FALSE;
curr_state.player_j ++;
if (curr_square.empty_field) {
  curr_square.player_on_field = TRUE;
  curr_square.empty_field = FALSE;
}
else curr_square.player = TRUE;
```

This code is used in sections 54, 86, and 93.

71. ⟨ Walk down 71 ⟩ ≡

```
if (curr_square.player_on_field) {
  curr_square.player_on_field = FALSE;
  curr_square.empty_field = TRUE;
}
else curr_square.player = FALSE;
curr_state.player_i ++;
if (curr_square.empty_field) {
  curr_square.player_on_field = TRUE;
  curr_square.empty_field = FALSE;
}
else curr_square.player = TRUE;
```

This code is used in sections 55, 87, and 93.

72. ⟨ Player can push left 72 ⟩ ≡

$curr\_j > 1 \wedge \langle\, wwest\_square \text{ is free } 73 \,\rangle \wedge \langle\, west\_square \text{ has pushable bunny } 74 \,\rangle$

This code is used in section 52.

73. ⟨ *wwest_square* is free 73 ⟩ ≡

$(wwest\_square.empty\_field \vee \neg(wwest\_square.wall \vee wwest\_square.spikes \vee wwest\_square.bunny \vee wwest\_square.bunny\_on\_field))$

This code is used in section 72.

74. ⟨ *west_square* has pushable bunny 74 ⟩ ≡

$(west\_square.bunny\_on\_field \vee (west\_square.bunny \wedge \neg west\_square.empty\_field))$

This code is used in section 72.

75. ⟨ Player can push up 75 ⟩ ≡

$curr\_i > 1 \wedge \langle\, nnorth\_square \text{ is free } 76 \,\rangle \wedge \langle\, north\_square \text{ has pushable bunny } 77 \,\rangle$

This code is used in section 53.

76. ⟨ *nnorth_square* is free 76 ⟩ ≡

$(nnorth\_square.empty\_field \vee \neg(nnorth\_square.wall \vee nnorth\_square.spikes \vee nnorth\_square.bunny \vee nnorth\_square.bunny\_on\_field))$

This code is used in section 75.

77. ⟨ *north_square* has pushable bunny 77 ⟩ ≡

$(north\_square.bunny\_on\_field \vee (north\_square.bunny \wedge \neg north\_square.empty\_field))$

This code is used in section 75.

78. ⟨ Player can push right 78 ⟩ ≡

$curr\_j < curr\_state.n\_cols - 1 \wedge \langle\, eeast\_square \text{ is free } 79 \,\rangle \wedge \langle\, east\_square \text{ has pushable bunny } 80 \,\rangle$

This code is used in section 54.

79. ⟨ *eeast_square* is free 79 ⟩ ≡

$(eeast\_square.empty\_field \vee \neg(eeast\_square.wall \vee eeast\_square.spikes \vee eeast\_square.bunny \vee eeast\_square.bunny\_on\_field))$

This code is used in section 78.

80. ⟨ *east_square* has pushable bunny 80 ⟩ ≡

$(east\_square.bunny\_on\_field \vee (east\_square.bunny \wedge \neg east\_square.empty\_field))$

This code is used in section 78.

81. ⟨ Player can push down 81 ⟩ ≡
curr_i < *curr_state.n_rows* − 1 ∧ ⟨ *ssouth_square* is free 82 ⟩ ∧ ⟨ *south_square* has pushable bunny 83 ⟩
This code is used in section 55.

82. ⟨ *ssouth_square* is free 82 ⟩ ≡
(*ssouth_square.empty_field* ∨ ¬(*ssouth_square.wall* ∨ *ssouth_square.spikes* ∨ *ssouth_square.bunny* ∨
ssouth_square.bunny_on_field))
This code is used in section 81.

83. ⟨ *south_square* has pushable bunny 83 ⟩ ≡
(*south_square.bunny_on_field* ∨ (*south_square.bunny* ∧ ¬*south_square.empty_field*))
This code is used in section 81.

84. ⟨ Push left 84 ⟩ ≡

```
if (west_square.bunny_on_field) {
  if (west_square.goal ∧ ¬west_square.bunny) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  west_square.bunny_on_field = FALSE;
  west_square.empty_field = TRUE;
  wwest_square.bunny = TRUE;
  if (wwest_square.goal) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else {
  if (west_square.goal) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  west_square.bunny = FALSE;
  if (wwest_square.empty_field) {
    wwest_square.empty_field = FALSE;
    wwest_square.bunny_on_field = TRUE;
    if (wwest_square.goal ∧ ¬wwest_square.bunny) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
  else {
    wwest_square.bunny = TRUE;
    west_square.bunny = FALSE;
    if (wwest_square.goal) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
}
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
else {
  ⟨ Walk left 68 ⟩;
}
```

This code is used in section 52.

85. ⟨ Push up 85 ⟩ ≡

```
if (north_square.bunny_on_field) {
  if (north_square.goal ∧ ¬north_square.bunny) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  north_square.bunny_on_field = FALSE;
  north_square.empty_field = TRUE;
  nnorth_square.bunny = TRUE;
  if (nnorth_square.goal) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else {
  if (north_square.goal) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  north_square.bunny = FALSE;
  if (nnorth_square.empty_field) {
    nnorth_square.empty_field = FALSE;
    nnorth_square.bunny_on_field = TRUE;
    if (nnorth_square.goal ∧ ¬nnorth_square.bunny) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
  else {
    nnorth_square.bunny = TRUE;
    north_square.bunny = FALSE;
    if (nnorth_square.goal) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
}
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
else {
  ⟨ Walk up 69 ⟩;
}
```

This code is used in section 53.

86. ⟨ Push right 86 ⟩ ≡

```
if (east_square.bunny_on_field) {
  if (east_square.goal ∧ ¬east_square.bunny) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  east_square.bunny_on_field = FALSE;
  east_square.empty_field = TRUE;
  eeast_square.bunny = TRUE;
  if (eeast_square.goal) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else {
  if (east_square.goal) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  east_square.bunny = FALSE;
  if (eeast_square.empty_field) {
    eeast_square.empty_field = FALSE;
    eeast_square.bunny_on_field = TRUE;
    if (eeast_square.goal ∧ ¬eeast_square.bunny) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
  else {
    eeast_square.bunny = TRUE;
    east_square.bunny = FALSE;
    if (eeast_square.goal) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
}
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
else {
  ⟨ Walk right 70 ⟩;
}
```

This code is used in section 54.

87. ⟨ Push down 87 ⟩ ≡

```
if (south_square.bunny_on_field) {
  if (south_square.goal ∧ ¬south_square.bunny) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  south_square.bunny_on_field = FALSE;
  south_square.empty_field = TRUE;
  ssouth_square.bunny = TRUE;
  if (ssouth_square.goal) {
    ⟨ Decrease number of empty goals 88 ⟩
  }
}
else {
  if (south_square.goal) {
    ⟨ Increase number of empty goals 89 ⟩
  }
  south_square.bunny = FALSE;
  if (ssouth_square.empty_field) {
    ssouth_square.empty_field = FALSE;
    ssouth_square.bunny_on_field = TRUE;
    if (ssouth_square.goal ∧ ¬ssouth_square.bunny) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
  else {
    ssouth_square.bunny = TRUE;
    south_square.bunny = FALSE;
    if (ssouth_square.goal) {
      ⟨ Decrease number of empty goals 88 ⟩
    }
  }
}
if (curr_state.empty_goals ≡ 0) {
  ⟨ Go to next level 50 ⟩
}
else {
  ⟨ Walk down 71 ⟩;
}
```

This code is used in section 55.

88. ⟨ Decrease number of empty goals 88 ⟩ ≡

```
curr_state.empty_goals −−;
```

This code is used in sections 60, 61, 62, 63, 84, 85, 86, and 87.

89. ⟨ Increase number of empty goals 89 ⟩ ≡

```
curr_state.empty_goals ++;
```

This code is used in sections 84, 85, 86, and 87.

90.

#define *curr_field* *curr_state.squares*[*curr_state.field_i*][*curr_state.field_j*]

⟨ FIELD action 90 ⟩ ≡

```
if (curr_state.field_on) {
  ⟨ Close field if possible, updating history 91 ⟩
}
else {
  if (curr_state.setting_field) {
    add_to_history(cancel_set_field);
    curr_state.setting_field = FALSE;
  }
  else {
    add_to_history(begin_set_field);
    curr_state.setting_field = TRUE;
  }
}
```

This code is used in section 49.

91. ⟨ Close field if possible, updating history 91 ⟩ ≡

```
if (⟨ Field can be closed 92 ⟩) {
  add_to_history(curr_state.field_i);
  add_to_history(curr_state.field_j);
  curr_state.field_on = FALSE;
  if (curr_field.bunny_on_field) {
    add_to_history(close_field_with_bunny);
    curr_field.bunny_on_field = FALSE;
    curr_state.carrying_bunny = TRUE;
    if (curr_field.goal ∧ ¬curr_field.bunny) curr_state.empty_goals++;
  }
  else if (curr_field.player_on_field) {
    add_to_history(close_field_with_player);
    curr_field.player_on_field = FALSE;
    curr_field.player = TRUE;
  }
  else {      /* empty field */
    add_to_history(close_field_empty);
    curr_field.empty_field = FALSE;
  }
}
```

This code is used in section 90.

92. ⟨ Field can be closed 92 ⟩ ≡

```
(¬curr_field.player_on_field ∨ ¬(curr_square.wall ∨ curr_square.spikes ∨ curr_square.bunny))
```

This code is used in section 91.

93. ⟨ UNDO action 93 ⟩ ≡

```
if (hist_index > 0) {
  hist_index --;
  switch (history[hist_index]) {
  case walk_left: ⟨ Walk right 70 ⟩
    break;
  case walk_up: ⟨ Walk down 71 ⟩
    break;
  case walk_right: ⟨ Walk left 68 ⟩
    break;
  case walk_down: ⟨ Walk up 69 ⟩
    break;
  case push_left: ⟨ Undo push left 95 ⟩
    break;
  case push_up: ⟨ Undo push up 96 ⟩
    break;
  case push_right: ⟨ Undo push right 97 ⟩
    break;
  case push_down: ⟨ Undo push down 98 ⟩
    break;
  case begin_set_field: curr_state.setting_field = FALSE;
    break;
  case cancel_set_field: curr_state.setting_field = TRUE;
    break;
  case open_field_left: case open_field_up: case open_field_right: case open_field_down:
    ⟨ Undo open field 99 ⟩
    break;
  case close_field_empty: case close_field_with_player: curr_state.field_j = history[--hist_index];
    curr_state.field_i = history[--hist_index];
    curr_field.empty_field = true;
    curr_state.field_on = true;
    break;
  case close_field_with_bunny: curr_state.field_j = history[--hist_index];
    curr_state.field_i = history[--hist_index];
    curr_field.bunny_on_field = TRUE;
    curr_state.carrying_bunny = FALSE;
    curr_state.field_on = TRUE;
    if (curr_field.goal ∧ ¬curr_field.bunny) curr_state.empty_goals --;
    break;
  }
}
```

This code is used in section 49.

94. Undoing is much simpler than doing, since we don't have to check whether the move is legal. The bookeeping is the same as for normal moves: update the position of bunnies, the player and fields; update the count of empty goals as well.

95. If there are fields involved in the pushing, it is either under *curr_square*, under *west_square* or under *east_square*.

```
⟨ Undo push left 95 ⟩ ≡
  if (curr_square.player_on_field) {
    west_square.bunny = FALSE;
    curr_square.bunny_on_field = TRUE;
    if (west_square.goal) curr_state.empty_goals++;
    if (curr_square.goal ∧ ¬curr_square.bunny) curr_state.empty_goals--;
    curr_square.player_on_field = FALSE;
    east_square.player = TRUE;
  }
  else if (west_square.bunny_on_field) {
    west_square.bunny_on_field = FALSE;
    west_square.empty_field = TRUE;
    curr_square.bunny = TRUE;
    if (west_square.goal ∧ ¬west_square.bunny) curr_state.empty_goals++;
    if (curr_square.goal) curr_state.empty_goals--;
    curr_square.player = FALSE;
    east_square.player = TRUE;
  }
  else if (east_square.empty_field) {
    west_square.bunny = FALSE;
    curr_square.bunny = TRUE;
    if (west_square.goal) curr_state.empty_goals++;
    if (curr_square.goal) curr_state.empty_goals--;
    curr_square.player = FALSE;
    east_square.player_on_field = TRUE;
  }
  else {      /* no fields involved in this move */
    west_square.bunny = FALSE;
    curr_square.bunny = TRUE;
    if (west_square.goal) curr_state.empty_goals++;
    if (curr_square.goal) curr_state.empty_goals--;
    curr_square.player = FALSE;
    east_square.player = TRUE;
  }
  curr_j++;
```

This code is used in section 93.

96. ⟨ Undo push up 96 ⟩ ≡

```
if (curr_square.player_on_field) {
  north_square.bunny = FALSE;
  curr_square.bunny_on_field = TRUE;
  if (north_square.goal) curr_state.empty_goals++;
  if (curr_square.goal ∧ ¬curr_square.bunny) curr_state.empty_goals--;
  curr_square.player_on_field = FALSE;
  south_square.player = TRUE;
}
else if (north_square.bunny_on_field) {
  north_square.bunny_on_field = FALSE;
  north_square.empty_field = TRUE;
  curr_square.bunny = TRUE;
  if (north_square.goal ∧ ¬north_square.bunny) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  south_square.player = TRUE;
}
else if (south_square.empty_field) {
  north_square.bunny = FALSE;
  curr_square.bunny = TRUE;
  if (north_square.goal) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  south_square.player_on_field = TRUE;
}
else {      /* no fields involved in this move */
  north_square.bunny = FALSE;
  curr_square.bunny = TRUE;
  if (north_square.goal) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  south_square.player = TRUE;
}
curr_i++;
```

This code is used in section 93.

97. ⟨ Undo push right 97 ⟩ ≡

```
if (curr_square.player_on_field) {
  east_square.bunny = FALSE;
  curr_square.bunny_on_field = TRUE;
  if (east_square.goal) curr_state.empty_goals++;
  if (curr_square.goal ∧ ¬curr_square.bunny) curr_state.empty_goals--;
  curr_square.player_on_field = FALSE;
  west_square.player = TRUE;
}
else if (east_square.bunny_on_field) {
  east_square.bunny_on_field = FALSE;
  east_square.empty_field = TRUE;
  curr_square.bunny = TRUE;
  if (east_square.goal ∧ ¬east_square.bunny) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  west_square.player = TRUE;
}
else if (west_square.empty_field) {
  east_square.bunny = FALSE;
  curr_square.bunny = TRUE;
  if (east_square.goal) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  west_square.player_on_field = TRUE;
}
else {      /* no fields involved in this move */
  east_square.bunny = FALSE;
  curr_square.bunny = TRUE;
  if (east_square.goal) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  west_square.player = TRUE;
}
curr_j--;
```

This code is used in section 93.

98. ⟨ Undo push down 98 ⟩ ≡

```
if (curr_square.player_on_field) {
  south_square.bunny = FALSE;
  curr_square.bunny_on_field = TRUE;
  if (south_square.goal) curr_state.empty_goals++;
  if (curr_square.goal ∧ ¬curr_square.bunny) curr_state.empty_goals--;
  curr_square.player_on_field = FALSE;
  north_square.player = TRUE;
}
else if (south_square.bunny_on_field) {
  south_square.bunny_on_field = FALSE;
  south_square.empty_field = TRUE;
  curr_square.bunny = TRUE;
  if (south_square.goal ∧ ¬south_square.bunny) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  north_square.player = TRUE;
}
else if (north_square.empty_field) {
  south_square.bunny = FALSE;
  curr_square.bunny = TRUE;
  if (south_square.goal) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  north_square.player_on_field = TRUE;
}
else {      /* no fields involved in this move */
  south_square.bunny = FALSE;
  curr_square.bunny = TRUE;
  if (south_square.goal) curr_state.empty_goals++;
  if (curr_square.goal) curr_state.empty_goals--;
  curr_square.player = FALSE;
  north_square.player = TRUE;
}
curr_i--;
```

This code is used in section 93.

99. Note that nothing happens if the field was empty (other than turning on *setting_field* and turning off *field_on*.

```
#define curr_field_square curr_state.squares[curr_state.field_i][curr_state.field_j]
⟨ Undo open field 99 ⟩ ≡
  if (curr_field_square.bunny_on_field) {
    curr_field_square.bunny_on_field = FALSE;
    curr_state.carrying_bunny = TRUE;
    if (curr_field_square.goal ∧ ¬curr_field_square.bunny) curr_state.empty_goals++;
  }
  else if (curr_field_square.player_on_field) {
    curr_field_square.player_on_field = FALSE;
    curr_field_square.player = TRUE;
  }
  else {
    curr_field_square.empty_field = FALSE;
  }
  curr_state.setting_field = TRUE;
  curr_state.field_on = FALSE;
```

This code is used in section 93.

100.

```
⟨ RESTART game 100 ⟩ ≡
  curr_state = levels[curr_level];
  hist_index = 0;
  history_disabled = FALSE;
```

This code is used in section 49.

101. This is just here temporarily.

```
#define board_wall '#'
#define board_space '␣'
#define board_spikes ','
#define board_goal '+'
#define board_player '@'
#define board_bunny 'b'
#define board_bunny_on_goal 'B'
#define board_player_on_goal 'A'
#define board_field_on_space 'f'
#define board_field_on_spikes ';'      /* pillow over a spike? */
#define board_field_on_goal 'F'
#define board_field_on_bunny '='       /* the bunny is squashed under a pillow */
#define board_field_on_bunny_on_goal 'm'      /* oh well, bound to break at some point */
#define board_player_on_field_on_space '@'      /* this is never input ... */
#define board_player_on_field_on_spikes '@'
#define board_player_on_field_on_goal 'A'
#define board_player_on_field_on_bunny
#define board_player_on_field_on_bunny_on_goal
#define board_bunny_on_field_on_space
#define board_bunny_on_field_on_spikes
#define board_bunny_on_field_on_goal
#define board_bunny_on_field_on_bunny
#define board_bunny_on_field_on_bunny_on_goal
```

102. Check valid board. A move is valid if the resulting board is valid. We only check squares which have changed.

- **a goal and the player**.
- **spikes, a field and the player**.
- **spikes, a field and a bunny**.
- **two bunnies and a field**.

103. Build instructions. If you have the literate file `tban.w`, then you need both `ctangle` and a C compiler in order to build an executable file. Of course, if you already have the `tban.c` file, then you may skip tangling and just compile (don't forget to link ncurses). So, on a unix-like system, proceed as follows.

```
ctangle tban.w # generates tban.c
gcc -lncurses tban.c -o tban
```

104. Other mechanics. Multiplayer? Coloured bunnies? Two types of fields? Tiles which drag statues, moving them every turn? Toroidal board? Portals? Switches (entangled buttons: when a bunny statue is placed on one, the other goes down as well).

⟨ Advance *buf_ptr* to next nonblank or next stop 17 ⟩ Used in section 15.
⟨ Break if at end of line 36 ⟩ Used in section 35.
⟨ Clean up and exit 31 ⟩ Used in section 2.
⟨ Close field if possible, updating history 91 ⟩ Used in section 90.
⟨ Copy next level line into *buffer*; exit if invalid input 15 ⟩ Used in section 9.
⟨ DOWN action 55 ⟩ Used in section 49.
⟨ Decrease number of empty goals 88 ⟩ Used in sections 60, 61, 62, 63, 84, 85, 86, and 87.
⟨ Exit if blank file 21 ⟩ Used in section 9.
⟨ Exit if line too long 18 ⟩ Used in sections 15 and 19.
⟨ FIELD action 90 ⟩ Used in section 49.
⟨ Field can be closed 92 ⟩ Used in section 91.
⟨ Game loop 32 ⟩ Cited in section 7. Used in section 2.
⟨ Global declarations 7, 8, 14, 28, 29 ⟩ Used in section 2.
⟨ Go to next level 50 ⟩ Used in sections 49, 60, 61, 62, 63, 84, 85, 86, and 87.
⟨ Go to previous level 51 ⟩ Used in section 49.
⟨ Header files 3, 5, 13 ⟩ Used in section 2.
⟨ Increase number of empty goals 89 ⟩ Used in sections 84, 85, 86, and 87.
⟨ Initialize color pairs 44 ⟩ Used in section 30.
⟨ Initialize game 27, 30 ⟩ Used in section 2.
⟨ Initialize level 11 ⟩ Used in sections 9 and 22.
⟨ Insert *buffer* line in *curr_input_lv* (maybe after incrementing *number_of_levels*) 22 ⟩ Used in section 9.
⟨ LEFT action 52 ⟩ Used in section 49.
⟨ Load game levels 9 ⟩ Used in section 27.
⟨ Local variables of *main* 12, 16, 23, 33, 46 ⟩ Used in section 2.
⟨ Open field down 63 ⟩ Used in section 55.
⟨ Open field left 60 ⟩ Used in section 52.
⟨ Open field right 62 ⟩ Used in section 54.
⟨ Open field up 61 ⟩ Used in section 53.
⟨ Open input file 10 ⟩ Used in section 9.
⟨ Opening a field down is possible 59 ⟩ Used in section 55.
⟨ Opening a field left is possible 56 ⟩ Used in section 52.
⟨ Opening a field right is possible 58 ⟩ Used in section 54.
⟨ Opening a field up is possible 57 ⟩ Used in section 53.
⟨ Player can push down 81 ⟩ Used in section 55.
⟨ Player can push left 72 ⟩ Used in section 52.
⟨ Player can push right 78 ⟩ Used in section 54.
⟨ Player can push up 75 ⟩ Used in section 53.
⟨ Player can walk down 67 ⟩ Used in section 55.
⟨ Player can walk left 64 ⟩ Used in section 52.
⟨ Player can walk right 66 ⟩ Used in section 54.
⟨ Player can walk up 65 ⟩ Used in section 53.
⟨ Process arguments 4 ⟩ Used in section 2.
⟨ Process user input 49 ⟩ Used in section 32.
⟨ Push down 87 ⟩ Used in section 55.
⟨ Push left 84 ⟩ Used in section 52.
⟨ Push right 86 ⟩ Used in section 54.
⟨ Push up 85 ⟩ Used in section 53.
⟨ RESTART game 100 ⟩ Used in section 49.
⟨ RIGHT action 54 ⟩ Used in section 49.
⟨ Render board debugger 48 ⟩ Used in section 45.
⟨ Render bunny on field 41 ⟩ Used in section 39.
⟨ Render debug info 45 ⟩ Used in section 34.

⟨ Render empty field 40 ⟩ Used in section 39.
⟨ Render field 39 ⟩ Used in section 35.
⟨ Render game 34 ⟩ Used in section 32.
⟨ Render player info 47 ⟩ Used in section 45.
⟨ Render player on field 42 ⟩ Used in section 39.
⟨ Render position without field 43 ⟩ Used in section 35.
⟨ Render ith line of the board 35 ⟩ Used in section 34.
⟨ Scan remainder of input line; exit if line too long 19 ⟩ Used in section 15.
⟨ Set all fields of $curr\_input\_lv.squares[curr\_n\_rows][j]$ to `FALSE` 25 ⟩ Used in section 24.
⟨ Set player's initial position 26 ⟩ Used in section 24.
⟨ Set $curr\_input\_lv.squares[curr\_n\_rows][j]$, maybe setting player coordinates and *empty_goals* 24 ⟩ Used in section 22.
⟨ UNDO action 93 ⟩ Used in section 49.
⟨ UP action 53 ⟩ Used in section 49.
⟨ Undo open field 99 ⟩ Used in section 93.
⟨ Undo push down 98 ⟩ Used in section 93.
⟨ Undo push left 95 ⟩ Used in section 93.
⟨ Undo push right 97 ⟩ Used in section 93.
⟨ Undo push up 96 ⟩ Used in section 93.
⟨ Walk down 71 ⟩ Used in sections 55, 87, and 93.
⟨ Walk left 68 ⟩ Used in sections 52, 84, and 93.
⟨ Walk right 70 ⟩ Used in sections 54, 86, and 93.
⟨ Walk up 69 ⟩ Used in sections 53, 85, and 93.
⟨ *curr_board_ij* is an open field 38 ⟩ Used in section 35.
⟨ *curr_board_ij* is field candidate 37 ⟩ Used in section 35.
⟨ *east_square* has pushable bunny 80 ⟩ Used in section 78.
⟨ *eeast_square* is free 79 ⟩ Used in section 78.
⟨ *nnorth_square* is free 76 ⟩ Used in section 75.
⟨ *north_square* has pushable bunny 77 ⟩ Used in section 75.
⟨ *south_square* has pushable bunny 83 ⟩ Used in section 81.
⟨ *ssouth_square* is free 82 ⟩ Used in section 81.
⟨ *west_square* has pushable bunny 74 ⟩ Used in section 72.
⟨ *wwest_square* is free 73 ⟩ Used in section 72.

Listing TBAN

(version 0.1)