

Lab 2

CS 5233 – Fall 2007
Tom Bylander, Instructor

assigned September 26, 2007
due midnight, October 31, 2007

In Lab 2, you will program a player for a 4x4 Sudoku game. Your grade on the lab will depend on the quality of the player that is implemented.

Sudoku 4

This version of the game consists of an 4×4 board that contains 1 number in each square. Each square can contain a number from 1 to 4. Each row, column, and “block” must contain all 4 numbers. There are four blocks, each of which is a 2×2 corner of the board.

Initially, you start with a board with some of the numbers filled in:

```

. 4 | . .
3 . | . .
-----+-----
. . | . 2
1 . | . .

```

In this high-resolution image, the horizontal and vertical lines separate the blocks. A “.” indicates an unfilled square. In this board, there is a 4 in location 1,2 (row 1, column 2); a 3 in location 2,1; a 2 in location 3,4; and a 1 in location 4,1. The remaining squares are blank.

The first message from the `sudoku4` environment will list the above information in one line:

```
0 4 0 0 3 0 0 0 0 0 0 2 1 0 0 0
```

A “0” is used to denote a blank square.

In this board, a 2 can be inferred at location 1,1. There is already a 4 in row 1, and a 3 and 1 in column 1. This leaves a 2 as the only choice at that location.

To communicate this assertion to the `sudoku4` environment, you would enter a “1 1 2”. Then, `sudoku4` will indicate that you are correct or you just lost the game. With appropriate settings, `sudoku4` will also redisplay the board.

```

2 4 | . .
3 . | . .
-----+-----
. . | . 2
1 . | . .

```

You win if you fill all the blank squares correctly.

Environment

The environment program for Sudoku 4 is in `/home/bylander/agents/linux/bin/sudoku4`. The `/home/bylander/agents/src/sudoku` directory contains the source for this program and other programs. Do `sudoku4 -h` to get an overview of the messages that it produces and expects.

To run the environment for yourself, do the following:

```
setenv A /home/bylander/agents/linux/bin
or A=/home/bylander/agents/linux/bin
$A/sudoku4 -v
```

In this mode, you can see the board, and you can put numbers in blank squares as described above. That is, the protocol for an agent program works as follows. First, the agent program will receive a message describing the initial board, e.g., `0 4 0 0 3 0 0 0 0 0 0 2 1 0 0 0`, where `0 4 0 0` is row 1, `3 0 0 0` is row 2, and so on. Then the agent sends an `x y v` message to place value `v` in square (x, y) . The response to an `x y v` message is `lose` if the value is incorrect, `correct` if the value is correct and the board still has empty squares, or `win` if the value is correct and all squares are filled.

Agent

Your task is to write a Sudoku 4 player that will infer the numbers in the squares without making any mistakes. This will be implemented by “telling” logical sentences to Prover9 and “asking” Prover9 if a given sentence is true or false. The basic routines can be implemented as follows (based on `wagent.c` in the `src/wump` subdirectory)

```
/* The disadvantage of the following two functions below is a
 * potential memory leak. They don't free the space that they
 * allocate. Hopefully, the number of calls will be small enough
 * so that it won't a problem.
 */

/* Create a literal from a predicate and its arguments.
 * Negation is added if the sign argument is negative or zero.
 */

char *make_lit(int sign, char *predicate, char *args) {
    char *literal;
    literal = (char *) calloc(strlen(predicate) + strlen(args) + 4, sizeof(char));
    sprintf(literal, "%s%s(%s)", ((sign <= 0) ? "-" : ""), predicate, args);
    return literal;
}

/* Concatenate two strings with a separator between them.
 * To concatenate two literals (or disjunctions of literals) into a
 * single disjunction, use " | " as the separator.
```

```
* To concatenate two terms (or lists of terms) into a single
* list of terms, use "," as the separator.
*/

char *concat_list(char *list1, char *separator, char *list2) {
    char *list;
    list = (char *) calloc(strlen(list1) + strlen(separator)
                          + strlen(list2) + 1, sizeof(char));
    strcpy(list, list1);
    strcat(list, separator);
    strcat(list, list2);
    return list;
}

/* Assert that the sentence is true. */

void tell(char *sent) {
    printf("tell %s\n", sent);
    fflush(stdout);
}

/* These variables are for inputting a response from the environment,
* or from the Prover9 logical inference program.
*/

int num_fields;
char **response;

/* Get a response from standard input, freeing the old response. */

void get_response() {
    int i;
    if (response != NULL) {
        free_split_stdin(response, num_fields);
    }
    response = split_stdin(" ", &num_fields);
}

/* Need to distinguish unknown from true and false */

#define FALSE 0
#define TRUE 1
#define UNKNOWN -1

/* Detect whether the substring is in the response. */
```

```

int sense(char *perception) {
    int i;
    for (i = 0; i < num_fields; i++)
        if (0 == strcmp(perception,response[i])) return TRUE;
    return FALSE;
}

/* Ask whether the the sentence is true, false, or unknown. */

int ask(char *sent) {
    printf("ask %s\n", sent);
    fflush(stdout);
    get_response();
    if (sense("true") == TRUE)
        return TRUE;
    else if (sense("false") == TRUE)
        return FALSE;
    else return UNKNOWN;
}

```

The interface to the Prover9 program is through the `tell` and `ask` subroutines. The `tellask` program takes care of input to and output from the Prover9 program. Note that `ask` has three possible results: `TRUE`, `FALSE`, and `UNKNOWN`, so you need to program conditions carefully. `make_lit` and `concat_list` are two subroutines to help you construct sentences.

Here are examples of various kinds of sentences that would be useful to tell Prover9.

- Every square contains a 1, 2, 3, or 4.
`square(x,y,1) | square(x,y,2) | square(x,y,3) | square(x,y,4).`
- A square cannot contain both a 1 and a 2.
`-square(x,y,1) | -square(x,y,2).`
- A row cannot have the same number in columns 1 and 2.
`-square(x,1,z) | -square(x,2,z).`
- A column cannot have the same number in rows 1 and 2.
`-square(1,y,z) | -square(2,y,z).`
- For you to figure out, two squares in a block cannot have the same number. [You may need to explicitly list each case.]

The `tellask` program takes care of input to and output from the Prover9 program. `tellask` is like the `interact` program you used in Lab 1, but in addition provides an interface to the Prover9 program so that you can perform logical inference for your agent. You should be able to run your lab 2 as follows:

```
A=/home/bylander/agents/linux/bin
or setenv A /home/bylander/agents/linux/bin
PROVER9=/home/bylander/prover9/linux/bin/prover9
or setenv PROVER9 /home/bylander/prover9/linux/bin/prover9
$A/tellask $A/sudoku4 lab2 $PROVER9
```

The level of performance of your lab will be determined by the percentage of boards correctly solved from a sample of boards. ***To get credit, you must perform the inference using tell-ask.***

Turning in Your Lab

Somewhere in your directory, you should create a `lab2` subdirectory. This directory *must* have a `Makefile` (or *must* have an executable file `make_lab2`) that compiles and links your agent programs, which *must* be named `lab2`. Any source code that is linked to these programs *must* be in this directory. That is, you should be able to copy the files in your `lab2` directory to another directory and be able to run the following command sequence.

```
setenv A /home/bylander/agents/sun/bin or A=/home/bylander/agents/sun/bin
/bin/rm lab2
source /etc/.login revert to default values for shell variables
make or ./make_lab2
$A/tellask $A/sudoku4 ./lab2
```

Zip or tar the entire directory and submit it using WebCT. If you submit multiple times, only the last one is kept. Be sure that WebCT has registered your submission.