— grid —
The griddler (nonogram) solver

Version: 1.2, November 2003

Mirek Olšák + Petr Olšák (`mirek@olsak.net`, `petr@olsak.net`)

This program solves the "griddler puzzles" (somebody call it as "nonogram puzzles"). The rectangle of square puzzles is given and the lengths of black (or colored) blocks is given around sides of the rectangle. You have to color the puzzles in such way to all data around puzzles stays correct. You can get more info from:

```
www.griddlers.net
```

The program `grid` solves the black+white or colored square puzzles. The triangular puzzles in rectangle grid are possible too. Moreover, the program can solve "triddlers" (triangular grid with hexagonal circumference) in black+white and colored version.

The program `grid` gives you the possibility to solve all puzzles or it can read the partially solved puzzles and give you only a hint about next one step. Moreover, program can check your partially solution and show you your errors in it (but it cannot show you the whole solution).

How to install

You can compile the program from source file by the command:

```
cc -O2 -o grid grid.c
strip grid
```

The option `-O2` is supported by GNU `gcc` (speed optimization). If your compiler does not support this option then you cannot use it. The command `strip grid` is optional (may be not implemented on every platforms). It removes the debug info from binary code.

The source code is based only on standard C library. It means, that the program will be compiled without problems at *arbitrary computer platform.*

How to run the program

You can run the program by the command:

```
grid file
```

where `file` is the name of the input file where the task with block lengths in rows and columns of a grid is declared. The format of input file is described below.

A several cases can be occur during solution:

• The solution is found. In this case, program prints OK and prints the solution on terminal. Moreover, it saves the solution as graphical output in `file.xpm`. You can see and print this solution using Gimp, for example.

• There is a conflict in the task. Program simple checks the consistency of the task before it starts the solution: it checks if the number of puzzles of each color is the same from "row point of view" as from "column point of view". It this is not true, the error message occurs and program terminates.

• There is a conflict in the task, but solution was started. The unsolved puzzles are printed by question mark. Program prints KO. This case occurs only if there is a conflict in the task but the simple test (see previous item) did not detect this problem. The program points to the row/column where the problem is and terminates.

• There exist more than one solution. Program prints all solutions and it saves only the first solution to XPM graphic file. This behavior can be changed by command line options, see below.

Command line options

`grid [options] mainfile`

`-help`
Program prints a short help about options to the terminal and terminates.

`-p` ⟨*number*⟩ default value: `-p 0`
Program enters to the "pause mode" after ⟨*number*⟩ steps. If it is in pause mode then it pauses after each step and prints the partial solution on the terminal. The new solved puzzles (from the last step) is printed by `#` (color) and `-` (background). If the puzzles are colored and `-out 3` is given then program prints all "color layers" in pause mode. The `-p 0` means that the pause mode will be never reached, `-p 1` means that pause mode is active immediately after first step. The word "step of program" will be explained at the end of this chapter.

`-stop` ⟨*number*⟩ default value: `-stop 0`
The meaning is the same as in `-p` ⟨*number*⟩ option. The only difference is that the program does not pauses but terminates. If `-stop 0` is given then program terminates after all solutions are found or after conflict is found or after `-total` solutions are found.

`-total` ⟨*number*⟩ default value: `-total 30`
If the ⟨*number*⟩ different solutions of one task is found then program terminates and it does not find another solutions. If `-total 0` is given then all solutions will be found.

`-xpm` ⟨*number*⟩ default value: `-xpm 1`
The first ⟨*number*⟩ solutions will be saved to `*.xpm` files. If `-xpm 1` is given then the program creates the file with the same base-name as `mainfile` but appends the `.xpm` extension. If `-xpm 2` or more is given then the suffix ⟨*solution number*⟩`.xpm` is appended to each name of XPM file. It means that more than one XPM file can be created. The ⟨*solution number*⟩ has the same number of digits as the ⟨*number*⟩ given by `-xpm` option. It means that left trailing zeros can be appended to ⟨*solution number*⟩. If `-xpm 0` is given then no XPM output is created.

`-i`
The program will use only intensive algorithms and tests.

`-log` ⟨*number*⟩ default value: `-log 2`
The ⟨*number*⟩ means the verbosity level of the output to stdout.
`-log 0` ... Only solutions are printed. The output format of solutions can be controlled by `-out` option.
`-log 1` ... Program prints the number of solutions and the final statistic about number of steps and successful/all calls of various line solvers.
`-log 2` ... Moreover, the number of steps are printed during solving.
`-log 3` ... Moreover, the state of solved lines are printed.
`-log 4` ... Moreover, the internal information from line solvers is printed.

`-lf` ⟨*file*⟩ default value: `stdout`
The log output will be printed to ⟨*file*⟩ instead to terminal. If the ⟨*file*⟩ exists then it will be removed when program starts.

`-out` ⟨*number*⟩ default value: `-out 2`
The solution printing format is controlled by `-out` option.
`-out 0` ... nothing is printed.
`-out 1` ... solution without numbers of rows/columns around it.
`-out 2` ... solution including numbers of rows/columns around it.
`-out 3` ... no-definitive solutions (during pause mode for example) are printed in all color layers.
`-out 4` ... the uncompleted solutions will be printed in every step.

`-of` ⟨*file*⟩ default value: `stdout`

The solution output will be printed to ⟨*file*⟩ instead to terminal. If the ⟨*file*⟩ exists then it will be removed when program starts.

`-cmp` ⟨*file*⟩

The ⟨*file*⟩ includes the partially solved puzzles in the same format as printed by program if `-out 2` is given. More details about this format is described below. If `-cmp` ⟨*file*⟩ is given then program finds the first solution but does not print it and does not save it to XPM. Program only checks the differences between solution and partially solved puzzles in ⟨*file*⟩ and prints (by `#` and `-` characters) the differences. Question marks are unchanged. This is a "hint mode" because program prints only a hint about bugs in your partially solved puzzles.

`-ini` ⟨*file*⟩

Program starts the solution from the state given in partially solved puzzles in ⟨*file*⟩. In this case program sets `-stop 1` in order to only hint about one next step is printed. If you want to run more steps then you have to use `-stop` option explicitly after `-ini` option. Example: `grid -ini my -stop 0 problem.g`

`-bl` ⟨*number*⟩ default value: `-bl 7`

First, program examines only rows and columns where less or equal than ⟨*number*⟩ blocks are present. If this limit causes no new solved puzzles then program enter the "full mode" where all rows and columns are examined.

You can write options in arbitrary order separated by space. If you use the same option more than once then the last one is significant. If you write `-` character instead of name of file then standard input is used. It is possible, for example, the following fitting of input files:

```
cat main.g inifile | grid -xpm 0 -log 0 -out 1 -ini - - > hint-file
```

Now, we describe the word **step of the program**. One step is the pass through all rows (step type **r**) or through all columns (step type **c**). These two types of steps alternates. The fast (so called left-right) line solver is used in this type of steps. This line solver is not able to find all new solved puzzles. This implies that the steps of type **r** and **c** can fail (it means they give no new solved puzzles). In such case the steps of type **R** and **C** is invoked. The little bit slower but elaborate line solver is used in these steps. As soon as the program finishes this type of steps then it enters the "normal" type of steps **r** and **c**. These normal steps can fail again and then the intensive type of steps is entered again. If the intensive steps fail then program enters to step type **t** (test). In this case the program tries to substitute some unsolved puzzle to a definitive color (or background) and run the normal steps **r** or **c** again. If conflict occurs in this situation then program tries to substitute unsolved puzzle by another color and run normal steps again. And so on, and so on...

If triddlers are active then three "normal" step types are in progress: **r** – rows, **c** – columns from bottom, **e** – columns from top of the hexagonal. If these steps fail then the steps of type **R**, **C**, **E** and (may be) type **t** are invoked in the same manner as in griddlers.

The format of main input file

If you have only two colors (black and white) puzzles then the format of the input file is simple:

```
Arbitrary text in zero or more lines. This text is used for comments
only and it is ignored if no colon, no hash is present as first
character of the line.
The first character of the whole file cannot be the decimal digit.
: the colon at the first position of the line starts row declaration
... data from rows
    each line represents one row data with block lengths
: the colon at the first position of the line starts column declaration
```

```
... data from columns
    each line represents one column data with block lengths
: the colon at the first position of the line ends the input
The arbitrary text here will be ignored.
```

The rows and columns data includes decimal numbers separated by space (or more spaces or tabulators). The arbitrary spaces and tabulators can be before the first number too. The numbers denotes the lengths of the blocks. The empty line is essential: it denotes the row/column without any blocks. The example of this type of input is in the file `kocka.g`.

Colored puzzles have the similar format of input file, but the colors declaration have to be present. An example including detail description of this format can be found in `oko.g` and `ruze.g` files.

The triangle puzzles (triangles by www.griddles.net) have the same input file format as the colored puzzles. The example including the description of this format can be found in `alladin.g` file.

The triangle puzzles (triangles by the journal “Malovane krizovky”, Silentium s.r.o.) have the similar format as colored puzzles, but you have to declare the left-glue and right-glue triangles by `<` or `>` characters. The example including detail format description can be found in `brontik.g` file.

The triddlers are declared by `#T` or `#t` at the first column before color declaration. The six data groups (separated by colons) are expected instead two data groups in rectangular puzzles. The first group means rows from side A, second rows from side B, third columns from side C, fourth columns from side D, fifth columns from side E and sixth columns from side F. The sides of the hexagonal are labeled in `tkocka.g` file. Simply speaking you begin to read the data at left upper corner and go counterclockwise around the hexagonal. Warning: the block lengths of columns, which begin at the bottom of hexagonal, have to be read *from underneath upstairs*. See the examples in `tkocka.g` and `vcely.g` files.

The program is able to read the black+white puzzles format used by `mk` program (see the URL `http://frix.fri.utc.sk/~johny/mk43frm.php`). The input mode for such format is activated automatically if the first character of the input file is a decimal digit. See the example in the file `levikral.mk` or in another `*.mk` files.

Format of input file with partial solution

These files are used by `-ini` and `-cmp` options. The format is compatible with the grid output on the terminal:

```
arbitrary text
:::: four colons say that the following line starts data input
  : the data lines
  : the number of these lines have to be the same as the number of rows
arbitrary text
```

Each data line has the format:

⟨*arbitr.text*⟩⟨*colon*⟩⟨*ignored char*⟩⟨*data characters*⟩⟨*arbitr.text*⟩

where the number of ⟨*data characters*⟩ have to be the same as the number of columns.

The ⟨*data character*⟩ is one of the following:

question mark or period — unsolved puzzle
space or minus — the background color
asterisk or hash mark — black color
the ⟨*outchar*⟩ from color declaration — this color

You can create the partial solution file very simply:

```
grid -stop 1 task.g > task.p
```

And you can go on:

```
grid -ini task.p task.g > task2.p
```

If you are solving triddlers then you can use the same file format only with the following difference. Each data line has the format:

⟨*arbitr.text*⟩⟨*(back)slash*⟩⟨*ignored char*⟩⟨*data characters*⟩⟨*arbitr.text*⟩

where ⟨*(back)slash*⟩ is the normal slash (`/`) or the backslash (`\`) character.

The example of usage

In UNIX shell:

```
for i in *.g *.mk; do grid -out 0 $i; done
gimp *.xpm
```

The insides

of the program is described in source `grid.c` in detail. Of course, *very detailed* description is here – the amount of comments are great than the amount of code like in `tex.web` source.

We assume that the usage of this program without usage of your own head brings no enjoyment. More enjoyment occurs if you are solving these puzzles manually. Most enjoyment occurs if you are studying of the puzzle solvers principles and of possibility of implementation these principles into the computer program. You can do this, it is sufficient to use some text editor, open the `grid.c` source and read...

Sorry, the comments in `grid.c` are only in our mother tongue. It means Czech, no English.

We spended many hours of time optimization of our program. We rejected the *brutal force* method and used the *brutal intelligence* method. We assume that our program belogs to the fastest programs in its category and to the best documented programs.

The another advantage of this program is its independence of the computer platform. We never used MS Windows because we need not it. But we are sure that the program is simply compilable at this obscured platform too.