SAARLAND UNIVERSITY
COMPUTER SCIENCE

Sebastian Hack
Christoph Weidenbach

December 02, 2008

Tutorials for "Advanced C"
Exercise sheet 6

Exercise 6.1: (*2 P*)

Extend your Makefile with:

1. a target `tags` to build tag file.
2. a variable `SATDEBUG` that causes a debug version of the program to be built. When the user says `make SATDEBUG=1` a preprocessor variable `SATDEBUG` shall be defined in all source files which causes debug macros to be activated. Furthermore, optimizations shall be deactivated and debug symbols shall be included.
3. a variable `SATCHECK` that causes invariant checking code to be compiled into the program. When the user says `make SATCHECK=1` the preprocessor variable `SATCHECK` shall be defined in all source files which causes check macros to be activated. Checking works completely independent of debugging.

Exercise 6.2: (*3 P*)

Extend your SAT solver with debug macros as presented in this week's lecture. Add the following extension: On each `DBG(...)` invocation, the programmer has to specify an importance of the message. For example:

```
DBG((MOD_PARSER, 3, "read a number\n"));
```

Smaller numbers mean higher importance. Later on, when executing your program, the user shall be able to specify that only messages with an importance higher than a certain level are printed. The format of the debug messages should resemble:

```
module_name filename(line_number): message
```

For example:

```
PARSER parse.c(45): read a character
```

Add command line flags for the user to activate certain debug modules and to set the importance level. If the importance level is not specified by the user, `0` is assumed. If no module is given, messages from all modules are printed.

The flags should be usable as follows:

- `-m PARSER,SOLVER` to select modules.
- `-v 2` to set the importance.

Hint: Take a look at the POSIX function `getopt(3)`

Exercise 6.3: (*5 P*)

Implement the checking macros depicted in the lecture. Add to your SAT project an invariant check that checks the consistency of your overall data structures before the selection of the next propositional variable (and after backtracking). Document the invariant check. Demonstrate your approach by letting it catch a bug you put in the program. So your deliverables are

- a `.tgz` archive of a consistent program including the macro module and the invariant check
- a `.tgz` archive of a buggy version of the same program with an input file where the bug is caught by the check

Submit your solution until the lecture on December 09.

Note: Joint solutions are not permitted.