20875 Software Engineering – Assignment 1

Due Tuesday, September 24th 2024, 23:59 CEST

The assignment consists in implementing a program that displays truth tables for Boolean formulas. Grading will be partially automated, and you are asked to follow precisely the instructions given in this document. In particular, both the input and the output must conform to specific formats detailed below. The assignment is to be uploaded as a single archive file (either `.zip` or `.tgz`) on BlackBoard by the end of September 24th.

Rules. This is an individual assignment. You are allowed to talk about the assignment with your classmates. However, you must write all your code on your own. As a consequence, you will be able to explain and modify any part of your code upon request.

Invocation. This assignment may be completed in Python or in C. The program will be invoked with the following commands:

Python: `python3 table.py input.txt`

C: `./table input.txt`

Compilation. [Only for assignments completed in the C language.] If a file called "`Makefile`" is present in the submitted archive, the code will be compiled with the command: `make`
Otherwise, it will be compiled with the command `clang -Wall -O3 -o table *.c`

Grading.

- [3 marks] The program accepts all files that conform to the specification and parses them correctly.
- [3 marks] The program correctly rejects malformed files (no crash or uncaught exception/error).
- [3 marks] For valid input files, the program prints the correct output as specified in this document.
- [1 mark] The program can print the one entries of the truth table (see "`show_ones`" below) for formulas with up to 24 variables (of sizes similar to `ag24_00`, ..., `ag24_15`, for which the "`ones`" table has 0-200 entries) under a minute. Bonus point for under a second.

1 Input

The input must conform precisely to the following specification. Any program that cannot be tokenized or parsed according to these rules must be rejected with an error message. Crashes and uncaught exceptions are not considered appropriate error messages. It is acceptable to reject (with an appropriate error message) programs that declare more than 64 variables in total.

1.1 Tokenization

1. **Comments.** Anything on a line after the character "`#`" is ignored.
2. **Special characters.** There are 4 special characters: "`(`", "`)`", "`=`" and "`;`".

3. **Words.** A *word* is any sequence of (one or more) consecutive *letters* ("A", ..., "Z" and "a", ..., "z"), *digits* ("0", ..., "9") and *underscores* ("_") that starts with a letter or an underscore.

4. **Blanks.** Spaces (" "), tabs ("\t"), carriage returns ("\r") and newlines ("\n") are considered *blank* characters. All blank characters are ignored except for the fact that they can separate two words.

5. **Keywords and identifiers.** There are 8 *keywords*: "`var`", "`show`", "`show_ones`", "`not`", "`and`", "`or`", "`True`", "`False`". Any other word is an identifier.

1.2 Parsing

We define here how a program is to be parsed, once it has been tokenized into a sequence of keywords, identifiers and special characters. A program is a sequence of instructions, each of which being either (a) a declaration, or (b) an assignment, or (c) a "show" instruction:

(a) Variable declaration: (i) the keyword `var`, followed by (ii) a sequence of identifiers, then (iii) a semicolon ("`;`"). Identifiers cannot have been previously declared (neither as a variable, nor by an assignment).

(b) Assignment: (i) an identifier, followed by (ii) the special character "`=`" and (iii) an expression, then (iv) a semicolon ("`;`"). The identifier in (i) cannot have been previously declared (neither as a variable, nor by an assignment). Before the assignment, every identifier present in the expression must have been either declared as a variable, or defined by an earlier assignment. In particular, the identifier in (i) cannot be used in the expression.

(c) "Show" instruction: (i) the keyword `show` or `show_ones`, followed by (ii) a list of identifiers, then (iii) a semicolon ("`;`"). Each identifier must have been defined by a prior assignment.

The expression (b.iii) is a Boolean formula involving *elements*. An *element* can be either the keyword "`True`", or the keyword "`False`", or any identifier that was defined previously (either by a variable declaration or by an assignment). An expression is either an element, or the negation ("`not`") of a sub-expression, or a conjunction ("`and`") of 2 or more sub-expressions, or a disjunction ("`or`") of 2 or more sub-expressions. If those sub-expressions are not elements themselves, then they *must* be surrounded by parentheses. As a result, there is no need for operator priorities.

BNF grammar:

```
<element>     ::= "True" | "False" | <identifier>
<paren-expr>  ::= <element> | "(" <expr> ")"
<negation>    ::= "not" <paren-expr>
<conjunction> ::= <paren-expr> "and" <paren-expr> | <paren-expr> "and" <conjunction>
<disjunction> ::= <paren-expr>  "or" <paren-expr> | <paren-expr>  "or" <disjunction>
<expr>        ::= <negation> | <conjunction> | <disjunction> | <paren-expr>

<id-list>     ::= <identifier> | <identifier> <id-list>
<instruction> ::= "var" <id-list> ";" | <identifier> "=" <expr> ";"
                  | "show" <id-list> ";" | "show_ones" <id-list> ";"
<program>     ::= <instruction> | <instruction> <program>
```

2 Output

The output must conform precisely to the following specification. In the output, anything on a line after the character "`#`" is ignored. All spaces and tab characters are ignored. Beyond that, the output consists only of rows of ones ("`1`", standing for True) and zeros ("`0`", standing for False).

The "`show`" and "`show_ones`" instructions print the truth table of the specified identifiers. The "`show`" instruction prints all the rows of the truth table, while "`show_ones`" only prints those rows for which at least one of the identifiers takes a value `1`.

A row of the truth table starts with the values of all the variables declared so far ("`0`" or "`1`"), in the order in which they were declared. Then, the row continues with the corresponding value ("`0`" or "`1`") of all the specified identifiers, in the order in which they are listed. The rows must be enumerated in lexicographic order. In other words, if we have n variables and we view the first n columns as a binary integer (with the first column corresponding to the first-declared variable and to the most significant bit), then this integer is monotonously increasing from one row to the next.

3 Example

A Boolean formula for `XOR` is: `x xor y = (x or y) and (not (x and y))`. A valid input file that describes it could be:

```
# We declare two variables:  x and y
var x y;

# We assign (x xor y) to z
z = (x or y) and (not (x and y));

# We show the truth table of z
show z;
```

If we store it in a file called `xor.txt` and run the command

Python: `python3 table.py xor.txt`

C: `./table xor.txt`

then a valid output could be:

```
#  x  y  z
   0  0  0
   0  1  1
   1  0  1
   1  1  0
```