Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

Project 3

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

The boolean satisfiability (SAT) problem is one of the oldest and still most important problems in computer science. It is probably the most prominent representative of the P versus NP millennium problem. And its application is widely used in many fields of computer science, e.g. theorem proving or software verification. The problem statement is to check for a given propsitional formula if it satisfiable or not. A propositional formula is a collection of variables combined by logical operators. These variables can be assigned to true or false. If there is at least one assignment so that the resulting term is equivalent to the logical truth, we say that the formula is satisfiable. The SAT problem is the task of determining for a given propositional formula if it is satisfiable. While noone was ever able to come up with an efficient solution to this problem, it is comparably easy to come up with a basic solution. In this project you will implement such a basic solution (SAT solver) in `C`.

To do so you have to 1) read input files of formulas in Reverse Polish notation (RPN), 2) use the Tseytin transformation to transform the task in conjuctive normal form and 3) solve it via the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. You will respectively find a detailed description for each subtask in the chapters 1,2,3. In the following, we will explain basic setup of the project.

An important part of this project is learning handling pre-existing code. Please first take a look at the provided code structure and read through the appended comments. This will help instead you implementing it from scratch.

To obtain the project use

```
git clone ssh://git@git.prog2.de:2222/project3/$NAME.git $FOLDER
```

where `$NAME` specifies your CMS username and `$FOLDER` is the folder the project should be placed in.

You can edit the files in the `src` folder using any text editor. We recommend using Visual Studio Code as it is already installed within in the virtual machine. To compile the project use

```
make
```

in the root project of the repository. After that you can use the following command to run your program:

```
bin/satsolver [flags] formula.in
```

to run your program. And the command

```
./tests/run_tests.py
```

to run the local tests for your project. Make sure to first build your project using `make`. `make check` will build the project and run all local tests. You can use the `-t` flag to run a specific test. For example,

```
./tests/run_tests.py -t "public.stack.emptyclear"
```

to run the test `public.stack.emptyclear`. You can use `-f` to run all tests that match a pattern. For example,

```
./tests/run_tests.py -f 'public\.stack\..*'
```

runs all tests with the prefix `public.stack.`. Existing parameters (flags) are documented at relevant places in this document.

Your project will be compiled with sanitizers that, among other things, check for correct memory handling and undefined behavior. They give you more helpful error messages in case of crashes, but will also lead to failed tests if your memory management is faulty. Make sure to read the notes in Section 4 about the test system.

You have two weeks to complete this project, meaning that we evaluate the last: `git push` until 06.06.2023, 23:59, MESZ.

Happy Coding :)

1 Reading propositional formulas from a file (2 + 3 Points)

1.1 Input specification

This chapter describes the process of building propositional formulas from an input file. The formulas consist of boolean *variables* (e.g. A, $x1$, vAr, ...) and *operators* ($\neg$, $\vee$, ...). Formulas in the input files are guarantee to be in *Reverse Polish Notation* (RPN). The following grammar defines the set of propositional formulas in RPN.

```
 formula  =  variable | formula unaryop | formula formula binaryop
 unaryop  =  !
binaryop  =  && | || | => | <=>
variable  =  [a-zA-Z0-9]+
```

`!` is the unary negation operator ($\neg$), `&&` is a binary conjunction ($\wedge$), `||` is a binary disjunction ($\vee$), `=>` is a (binary) implication ($\Rightarrow$) and `<=>` represents a (binary) equivalence ($\Leftrightarrow$). Your program should accept strings of upper and lower case letters and digits as variable names, for example `v4l1D` is a valid variable name. If you create internal variables later in the program (see 2.2), the `$` character is also allowed in the variable name.

Your program should be able to read the logical formulas as just specified. The file has to contain only one formula and may contain additional white space (spaces, tabs or line breaks). In a correct input, variables and operators are separated by one or more spaces. There may also be additional white space at the beginning and end of each formula. The variables and operators themselves must not be split by whitespace. Here are some examples of valid inputs:

1. `a   b && c ||`
2. `a b !  david Putnam && && && c ||`

1. defines the propositional formula $(a \wedge b) \vee c$ where a, b, and c are variables, and 2. defines the formula $(a \wedge (\neg b \wedge (david \wedge Putman))) \vee c$ where a, b, $david$, $Putman$, and c are variables.

The following input files are invalid. Your program should output an error when reading the following files.

1. `ab && c ||` (the conjuction is unary in this example, but should be binary)
2. `a b & & c ||` (the conjuction operator is split into to separate &)
3. `a b && c` (the last variable is unused)

1.2 Stack Implementation

For handling the input and later subtasks in the project, the *stack* is an important data structure. A stack is a collection of elements that can be accessed according to the LIFO principle (*Last In - First Out*). It provides the following operations for this purpose:

push to add a new element on top of the stack

peek to return the the topmost element of the stack

pop to delete the topmost element of the stack

isEmpty to check if the stack contains an element

clear to remove all elements

The topmost element here refers to the last added element that was not yet removed. Calling `peek` or `pop` on an empty stack can lead to undefined behavior.
In this project you will implement a stack as a linked list. To do so take a look at the file `src/list.c`. Put your implementation into the empty function bodies. Be sure to allocate space using `malloc` and also free them again using `free` where needed.

1.3 Freeing propositional formulas

For the internal representation of propositional formulas, we provide a suitable data structure in the files `src/propformula.*`. Familiarize yourself with this data structure and implement the function `freeFormula`, which frees all memory used to represent a propositional formula.
Note that variables are managed independently of this (in `src/variables.c`) and accordingly should not be freed by `freeFormula`.

1.4 Reading the input

To read the input formulas, implement the `parseFormula` function in `src/parser.c`. Also implement the `toKind` function in the same file for this purpose.

Remarks:

- Think about how you can use your stack implementation for this task.
- Use the `nextToken` function from `include/lexer.h` to read strings separated by spaces from a file into strings. Make sure to free these strings.
- Use the `err` function from `include/err.h` to abort the program when an invalid input is detected.
- To understand how variables are managed, it may be helpful to read the documentation in `include/variables.h`.
- To test your implementation, you can call your program with the argument `--printformula` or `-p` to output an infix representation of the formula you created. If you want to print a propositional formula at another point in your program, you can use the `prettyPrintFormula()` function from the `include/propformula.h` file for this purpose.

2 Transforming the formula into Conjunctive Normal Form (7 Points)

2.1 Conjunctive Normal Form

Your program should check the satisfiability of propositional formulas with the help of the DPLL algorithm. In order to apply the DPLL algorithm to the read in formulas, they must first be put into *conjunctive normal form* (*CNF*). A propositional formula in conjunctive normal form is a conjunction of *clauses*. Each clause is a disjunction of *literals*. A literal can be a non-negated variable or a negated variable. An example of a formula in conjunctive normal form is:

$$(A \lor B \lor C) \land (\neg A \lor B) \land (\neg C)$$

In this representation there are no operators for logical implication and equivalence. Furthermore, the negation operator must only be applied to variables and no longer to arbitrary formulas. As indicated in the example, conjunction and disjunction can be considered here as operators that allow any number of operands. Because of these differences you will use a different represenation for CNFs.

2.2 Tseytin Tansformation

The *Tseytin transformation*[1] is an algorithm to convert arbitrary propositional formulas into conjunctive normal form. In contrast to the naive equivalent transformation using De Morgan's laws, Tseytin's method does not lead to an exponential increase of the number of conjunctions in the formula. For this purpose, the Tseytin method introduces additional propositional variables for all non-trivial partial formulas.

A new variable is created for each non-trivial partial formula. Non-trivial partial formulas are all formulas that contain at least one operator, including the complete formula. (In your implementation you should use the method `mkFreshVariable()` from the file `variables.h` to create new variables with new names and assign them to the partial formulas in post order.) This, essentially, means a new variable is created for each operator appearing in the formula, i.e., Tseytin transformation produces as many new variables as there are operators in the given formula.

[1] `https://en.wikipedia.org/wiki/Tseytin_transformation`

To see how the transformation works in a more detail, consider the formula $a \wedge ((\neg b \vee c) \Rightarrow d)$ that can be expressed in the reverse polish notation as `a b ! c || d => &&`. The Tseytin transformation creates a fresh new variable (which we denote x_i here) for each operator $\neg$ (`!`), $\vee$ (`||`), $\wedge$ (`&&`), and $\Rightarrow$ (`=>`). So, we have x_1 representing the *partial* formula $\neg b$, x_2 representing $x_1 \vee c$, x_3 representing $x_2 \Rightarrow d$, and x_4 representing the whole input formula $a \wedge x_3$ (see the following table).

Variable	Partial formula	Expanded partial formula
x_1	$\neg b$	$\neg b$
x_2	$x_1 \vee c$	$\neg b \vee c$
x_3	$x_2 \Rightarrow d$	$(\neg b \vee c) \Rightarrow d$
x_4	$a \wedge x_3$	$a \wedge ((\neg b \vee c) \Rightarrow d)$

Now the question is how to put everything together and construct a formula in CNF using $x_1, \dots, x_4$ that is equivalent to the input formula, i.e., we need to construct the formula so that it is true if and only if the input formula is true. The first observation is that we have chosen x_4 to represent the whole input formula. So, we want to construct our CNF so that the input formula is true if and only if x_4 is true. Since x_4 was chosen to represent $a \wedge x_3$, we want x_4 to be true if and only if $a \wedge x_3$ is true. Now we need to make sure that x_3 is true if and only if $x_2 \Rightarrow d$ is true. Next, we need x_2 to be true if and only if $x_1 \vee c$ is true. And finally, x_1 needs to be true if and only if $\neg b$ is true. So, put everything together we end up with the formula

$$\begin{array}{ll} x_4 & \wedge \\ x_4 \Leftrightarrow (a \wedge x_3) & \wedge \\ x_3 \Leftrightarrow (x_2 \Rightarrow d) & \wedge \\ x_2 \Leftrightarrow (x_1 \vee c) & \wedge \\ x_1 \Leftrightarrow \neg b & \end{array}$$

which is clearly true if and only the input formula $a \wedge ((\neg b \vee c) \Rightarrow d)$ is true, because this is how we constructed the formula in the first place. It is also easy to see that the new formula can be transformed to the conjunctive normal form using De Morgan's laws without exponential blow-up. In fact, we can derive a set of simple rules that can be directly used to generate formulas in CNF when applying the Tseytin transformation:

$$\begin{aligned} & x_i \Leftrightarrow (\neg a) \\ \equiv & (x_i \Rightarrow \neg a) \wedge (\neg a \Rightarrow x_i) \\ \equiv & (\neg x_i \vee \neg a) \wedge (a \vee x_i) \end{aligned}$$

$$\begin{aligned} & x_i \Leftrightarrow (a \wedge b) \\ \equiv & (x_i \Rightarrow (a \wedge b)) \wedge ((a \wedge b) \Rightarrow x_i) \\ \equiv & (\neg x_i \vee (a \wedge b)) \wedge (\neg(a \wedge b) \vee x_i) \\ \equiv & (\neg x_i \vee (a \wedge b)) \wedge (\neg a \vee \neg b \vee x_i) \\ \equiv & (\neg x_i \vee a) \wedge (\neg x_i \vee b) \wedge (\neg a \vee \neg b \vee x_i) \end{aligned}$$

$$\begin{aligned} & x_i \Leftrightarrow (a \vee b) \\ \equiv & (x_i \Rightarrow (a \vee b)) \wedge ((a \vee b) \Rightarrow x_i) \\ \equiv & (\neg x_i \vee (a \vee b)) \wedge (\neg(a \vee b) \vee x_i) \\ \equiv & (\neg x_i \vee a \vee b) \wedge ((\neg a \wedge \neg b) \vee x_i) \\ \equiv & (\neg x_i \vee a \vee b) \wedge (\neg a \vee x_i) \wedge (\neg b \vee x_i) \end{aligned}$$

$$\begin{aligned} & x_i \Leftrightarrow (a \Rightarrow b) \\ \equiv & x_i \Leftrightarrow (\neg a \vee b) \\ \equiv & (\neg x_i \vee \neg a \vee b) \wedge (a \vee x_i) \wedge (\neg b \vee x_i) \end{aligned}$$

$$\begin{aligned}&x_i \Leftrightarrow (a \Leftrightarrow b)\\ \equiv&(\neg x_i \vee \neg a \vee b) \wedge (\neg x_i \vee \neg b \vee a) \wedge (x_i \vee \neg a \vee \neg b) \wedge (x_i \vee a \vee b)\end{aligned}$$

That is, whenever we need to generate a (partial) formula $x_i \Leftrightarrow (\neg a)$ during the transformation, we can directly generate $(\neg x_i \vee \neg a) \wedge (a \vee x_i)$ which is already in CNF. Whenever we need to generate $x_i \Leftrightarrow (a \vee b)$ we can generate $(\neg x_i \vee a \vee b) \wedge (\neg a \vee x_i) \wedge (\neg b \vee x_i)$ (which is already in CNF) instead, and so on.

For example, recall our formula from above $x_4 \wedge (x_4 \Leftrightarrow (a \wedge x_3)) \wedge (x_3 \Leftrightarrow (x_2 \Rightarrow d)) \wedge (x_2 \Leftrightarrow (x_1 \vee c)) \wedge (x_1 \Leftrightarrow \neg b)$. Here, we can use the aforementioned rules and replace $x_1 \Leftrightarrow \neg b$ with $(\neg x_1 \vee \neg b) \wedge (b \vee x_1)$. And we can replace $x_2 \Leftrightarrow (x_1 \vee c)$ with $(\neg x_2 \vee x_1 \vee c) \wedge (\neg x_1 \vee x_2) \wedge (\neg c \vee x_2)$, and so on, resulting in the formula

$$\begin{array}{ll}x_4 & \wedge\\ (\neg x_4 \vee a) \wedge (\neg x_4 \vee x_3) \wedge (\neg a \vee \neg x_3 \vee x_4) & \wedge\\ (\neg x_3 \vee \neg x_2 \vee d) \wedge (x_2 \vee x_3) \wedge (\neg d \vee x_3) & \wedge\\ (\neg x_2 \vee x_1 \vee c) \wedge (\neg x_1 \vee x_2) \wedge (\neg c \vee x_2) & \wedge\\ (\neg x_1 \vee \neg b) \wedge (b \vee x_1) & \end{array}$$

which is already in CNF.

2.3 Creating a CNF formula

Your project template already contains data structures for the representation of CNF formulas in `include/cnf.h`. Please familiarize yourself with them. Note that clauses in this data structure can contain a maximum of three literals. Make sure you understand why this is not a limitation for the Tseytin transformation as it always generates clauses with at most three literals.
Implement the Tseytin procedure by implementing the function `addClauses` in `src/tseitin.c`. This is to return a variable x for a valid P and insert clauses into a passed CNF according to the above scheme. After calling `addClauses`, a satisfying assignment with $x \mapsto$ *true* shall exist for the CNF if and only if P is satisfiable.
This function is called within the function `getCNF`. The returned variable is inserted in a clause as a positive literal in the CNF to get the desired formula in conjunctive normal form.

Remarks:

- It is highly recommended to make the `addClauses` function recursive.
- Use the helper functions from `src/Tseytin.c` to insert new clauses into a CNF.
- To understand how variables are managed, it may be helpful to read the documentation in `include/variables.h`.
- To test your implementation, you can call your program with the `--printcnf` or `-c` argument to output an infix representation of the converted CNF. If you want to print a formula in conjunctive normal form at another point in your program, you can use the `prettyPrintCNF()` function from the `include/cnf.h` file for this purpose.

3 DPLL-Algorithm (6 Points)

3.1 Overview

The DPLL algorithm checks whether a propositional formula in conjunctive normal form can be fulfilled. For this purpose, it systematically tries to find an assignment that satisfies the formula. The basic version of the algorithm works as follows: A variable is selected and assigned the value *true*. Then it is checked recursively if there exists a fulfilling assignment for the remaining variables. If this is the case, a fulfilling assignment for the formula was found. If not, the selected variable is set to the value *false* and a new search is started for a valid assignment of the remaining variables. If no valid assignment is found for the new value, the formula is unsatisfiable. Additionaly, we consider optimization called *unit-propagation*, which in certain cases allows to immediately infer the only valid value for a variable.
You will implement this procedure iteratevily. In each iteration, the first statement with a matching precondition shall be executed:

abort (satisfiable) If the formula evaluates to *true*, the algorithm is aborted. The formula can be fulfilled.

abort (unsatisfiable) If the formula evaluates to *false* and no other truth value can be assigned to all variables set so far, the algorithm is aborted. The formula is unsatisfiable.

reset If the formula evaluates to *false* and there is a variable for which an alternative value can be tested, the last iterations of the algorithm are undone until the variable is unassigned again. Then the variable is assigned the alternative value and the next iteration of the algorithm is started.

unit-propagation If there is a *unit-clause*, it is satisfied and the algorithm proceeds to the next iteration. A unit clause is a clause in which all literals, except one, are already assigned and to which no truth value can yet be assigned. Such a clause can only be satisfied if the last unoccupied literal is assigned such that the clause *true*.

assign a free variable A free variable is selected and set to the value *true*.

The following represents this description as pseudo-code:

```
while not terminated:
    if all clauses are fulfilled:
        abort (sat)
    if one clause is false
        if reset possible:
            Reset
            terminate iteration
        else:
            abort (unsat)
    if unit clause exists:
        fulfill any unit clause
        end iteration
    select next free variable abd set to true
```

3.2 Example

We consider the following example:

$$(A \lor B \lor C) \land (\neg A \lor B) \land (\neg C)$$

In this example, $\neg C$ is a unit clause. It has only one unoccupied variable left, namely C. So we assign C the value *false* to satisfy the clause. There are now no more unit clauses and we choose an unoccupied variable, A, and set it to *true*. Now the remaining unsatisfied clause $\neg A \lor B$ is a unit clause and we set B to the only possible value, *true*. Thus all clauses are satisfied and the algorithm terminates with the satisfying assignment

$$A \mapsto true, B \mapsto true, C \mapsto false$$

.

3.3 Implementation

3.3.1 Backtracking

Your implementation of the DPLL algorithm should use *backtracking* to reset variables. During the search for a valid variable assignment it is always necessary to reverse a previously made assignments in order to search for the next assignment. To do so, you need to know for which variables there is an alternative assignment and for which variables the only permissible value was inferred. We will use the before implemented stack to track these assignments. Each element on the stack is a pair. The first element is the corresponding variable name, the second element indicates if there is an alternate assignment for the variable (`CHOSEN`) or if the assignment is the only allowed one (`IMPLIED`). In the following example, backtracking is necessary.

$$(B \lor \neg C) \land (\neg B \lor \neg C) \land (\neg B \lor D) \land (C \lor \neg D) \land A$$

Since clause A is a unit clause, we set A to the only possible value *true* and place the pair (A, `IMPLIED`) on the stack. Since there now exists no unit clauses, we select the next free variable B and set it to *true*, the first of two possible values. We place the pair(B, `CHOSEN`) on the stack. Next, $(\neg B \vee \neg C)$ is a unit clause. The only valid assignment for C is *false*. We therefore place (C, `IMPLIED`) on the stack. Moreover, $\neg B \vee D$ is a unit clause that enforces that D must take the value *true*. We place (D, `IMPLIED`) on the stack.

D	`IMPLIED`
C	`IMPLIED`
B	`CHOSEN`
A	`IMPLIED`

By the assignment of D, the last clause of the formula, $(C \vee \neg D)$, is unsatisfiable. We therefore need to apply backtracking to look for another assignment. We backtrack the last assignments until we find a variable for which an alternative assignment is possible - in this case B. Now we assign B the only remaining value *false* and put (B, `IMPLIED`) on the stack.

B	`IMPLIED`
A	`IMPLIED`

3.3.2 Implementation details

Implement the function `iterate` in `src/dpll.c`, which should perform one iteration of the DPLL algorithm as shown above.
You can refer to several pre-implemented functions for this task:

- `getNextUndefinedVariable` in `include/variables.h` is used to get the next free variable for the DPLL algorithm.
- The function `getUnitLiteral` in `include/cnf.h` checks if a clause contains exactly one literal with `UNDEFINED` variable. If so, this literal is returned, otherwise `0` is returned.
- Use the `pushAssignment` and `popAssignment` functions from `src/dpll.c` to modify the stack for variable assignments. They are dependent on your stack implementation.
- Use only functions from `include/variables.h` to access variables. These ensure that changes to the truth values of variables are propagated to all clauses that include them. To do this, a variable maintains a list of *parent clauses*. This list contains all clauses in which the variable occurs.

4 Evaluation

As in the previous project, for every task there are public, daily as well as eval tests, and you have to pass all public tests to get any points for this project. The daily tests are not available locally. These are scheduled to run after you upload your project to our server (git push). Finally, the eval tests will be used to rate the projects upon completion. These tests are unavailable to you.
The test system will only use `.c` files from the `src` subfolder of your project that were provided with the project. In particular, new files as well as changes to headers (`.h` files), Makefile and the test scripts will be ignored. So, make sure that your delivery is executable under these conditions to avoid failure of all tests. Also, make sure that your program does not write any additional output to `stderr` and that the existing command line parameters are not modified. You should write any debug output to `stdout` using `printf()` In particular, do not use existing output functions that end in `Eval` as these print the results to `stderr` for evaluation.

In this project, you will obtain points based on the successfull implementation of subtasks according to to the following table:

Subtask	**Points**	**Public Tests**
Stack	2	public.stack.singlepush, public.stack.singlepeek, public.stack.singlepop, public.stack.empty, public.stack.emptyclear
Reading input	3	public.parser.simple01_valid, public.parser.simple02_valid, public.parser.simple03_invalid, public.parser.simple04_invalid
CNF-Construction	7	public.cnf.variable, public.cnf.tseitin01
DPLL Algorithm	6	public.solver.simple01_sat, public.solver.complex00_sat, public.solver.complex00_unsat, public.solver.valid_output_sat, public.solver.minisudoku01_sat

We test both individual functions of your program using unit tests and the entire module using integration testing. For your project, it may be helpful to add your own tests to the test framework included in your project repository. You can find more information about this in the `run_tests.py` script in your project repository.