
# CPS2000 - Compiler Theory & Practise

## Assignment Part 1

B.Sc Computer Science

Jacques Vella Critien - 97500L

# Contents

# 1 Task1: Boolean Satisfiability

In this task, we were required to write a command line program that accepts Boolean expressions in Conjunctive Normal Form (CNF). Moreover, this implementation should:

- accept symbols "w", "x", "y" and "z" to represent literals
- use the "!" symbol to represent negation
- use parenthesis to group clauses
- use commas to separate literals in clauses
- white-space is irrelevant and should be ignored while parsing

Moreover, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm had to be implemented to determine whether the expression is satisfiable. If the expression is satisfiable the program should display a truth assignment as a proof, while if it is unsatisfiable, the program should display "UNSAT".

Finally, we are to dedicate a section to describe the practical applications of SAT.

## 1.1 Solutions

### 1.1.1 SyntaxErrorException.java

This is an exception thrown when there is something wrong with the string passed to be parsed by the parser

### 1.1.2 Assignment.java

This is an enum to hold the assignment value and this can be either TRUE, FALSE or DONTCARE.

### 1.1.3 Symbol.java

The assignment states that the implementation should include the values "w", "x", "y" and "z" but in this enum class which contains the different symbols available for literals or variables, I also added "v" for more testing.

### 1.1.4 Literal.java

This is a class which represents a literal. It has the following member variables:

- **negated**: This is a boolean to hold if the literal is negated or not
- **symbol**: This is of type Symbol explained above and is used to hold the symbol or type of the literal.

In addition, this class consists of the following methods

1. **Literal(String)**: This is a constructor that when given a string it initialises the literal object from it. This is done by first checking if the length of the string passed is greater than 2 or if the length is 2 and that the first character is 2. If so, it means that the string passed is not of the correct format, therefore a **SyntaxErrorException** is thrown. Otherwise, it is checked whether the passed literal is negated by checking if the size is 2, if so meaning that it is negated and if not, it means that the literal is not negated. The last step involves obtaining the symbol of the literal from the string and setting it. In case of a symbol different from the accepted ones, a **SyntaxErrorException** is thrown.
2. **isNegated()**: This is a getter for the negated member variable.
3. **setNegated(boolean)**: This is a setter for the negated member variable.
4. **getSymbol()**: This is a getter for the symbol member variable.
5. **setSymbol(Symbol)**: This is a setter for the symbol member variable.
6. **equals(Literal)**: This is a method to check if the current literal is equal to the one passed. This is done by checking if both literals contain the same symbol and negation.
7. **clone()**: This is a method to clone a literal, i.e., getting its values, setting them to a new Literal object and returning it.

#### 1.1.5 Clause.java

This is a class which represents a clause. It has the following member variable:

- **literals**: This is an arraylist of Literals, used to hold the literals in the clause.

In addition, this class consists of the following methods

1. **Clause(ArrayList<Literal>)**: This is a constructor that when given an array-list of literals, they are set to member variable holding literals.
2. **Clause()**: This is an empty constructor for clause and it just initialises the array-list of literals.
3. **getLiterals()**: This is a getter for the literals member variable.
4. **addLiteral(Literal)**: This is a function which just adds a literal to the array-list holding literals.

### 1.1.6 Parser.java

This is the class used to represent the parser, hence which turns boolean expressions as strings and parses them into objects of clauses and literals. Moreover, this performs error checking to make sure that these expressions are entered in the correct format.

This class contains the following methods:

1. **parseString(String)**: This function is the main function used to parse a string into proper objects of clauses and literals by returning a set of clauses
   This is done by first checking making sure that the string is not empty and if so a **SyntaxErrorException** is thrown. The next step involves removing all white-spaces before checking whether the first character of the string is a '(' and the last character of the string is a ')' and if not, a **SyntaxErrorException** is thrown.

   After these checks, the parsing starts by removing the first and last characters which should be '(' and a ')' and then create a string tokenizer with the delimeter set to ')(' so that each clause is seperated. The next step involves going through each token stored by the clause tokenizer and for every token, a new clause object is created and each literal is obtained by creating another tokenizer, this time operating with the delimeter set to ','. Before moving to the next step, it is worth mentioning that, the DPLL class, which will be explained in the next section, contains a member variable of type hashmap with the a Symbol key and an Assignment enum value as value, called **assignments** to hold assignments, i.e. every literal symbol being used in the clause.

   After this, for every literal in each clause, a new Literal is created by calling the literal constructor accepting a string and if the **assignments** map does not contain the symbol, the symbol is added to the list with a DONTCARE assignment. Finally, the literal is added to the clause created in the loop containing the clauses and once all the literals of the clause are iterated through, the clause is added to the set of clauses to be returned.

   After all the literals and clauses are visited, the set of clauses which was filled is returned.

2. **printParsed(Set<Clause>**: This function is used to print the passed set of clauses in a parsed way. This is done by creating an iterator on the set of clauses and for each clause, the literals are obtained and are printed. If these are negated, a '¬' before the symbol is printed. The clauses and literals are also numbered to make it easier for the user to understand them, for an example, one can take a look at the figure below.

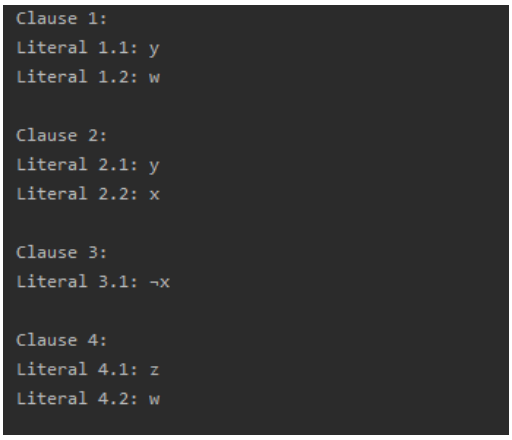


Figure 1: Result for (!x)(y, x)(z,w)(y, w)

#### 1.1.7 DPLL.java

This class contains methods for the DPLL algorithm and the actual DPLL function. Moreover, this class has the following global variable:

- **assignments**: This is a hashmap with a key of type Symbol and a value of type Assignment. This variable is ought to hold each symbol found in the expression and keep its assignments so that if it is satisfiable, the assignments, computed on run-time can be easily displayed.

This class contains the following methods:

1. **removeTriviallySat(Set <Clause>)**: This method removes the trivially satisfiable, of the form (x, !x), clauses from the passed set of clauses and returns a set of clauses without the satisfiable clauses. In order to do this, a boolean flag named removed is held and an iterator is initiated based on the elements inside the passed set of clauses.

   For every clause found in the iterator, removed is set to false, and the clause's literals are obtained. Apart from this, for every clause, there is an inner loop which goes through each literal in the clause until there is a removal or all literals have been iterated. Inside this loop, there is another nested loop to go through the literals one more time for comparison and if the literal of the outer loop and the literal of the inner loop are of the same symbol but their negation is different, hence in the form (x, !x), this clause is removed and removed is set to true. Finally, the remaining clauses are added. To help in the explanation the code snippet of this method can be found in the below image.

```
while(clauseItr.hasNext())
{
    //set removed to false
    removed = false;
    Clause clause = clauseItr.next();
    //get clause literals
    ArrayList<Literal> literals = clause.getLiterals();

    //nested loop for clause's literals
    for(int j =0; j < literals.size() && !removed; j++)
    {
        //get first literal
        Literal first = literals.get(j);
        for(int k =0; k < literals.size(); k++)
        {
            //get second literal
            Literal second = literals.get(k);
            //if both clause's literals have the same symbol but they're negation is different (xor)
            if(first.getSymbol().equals(second.getSymbol()) && (first.isNegated() ^ second.isNegated()))
            {
                //remove clause
                clauseItr.remove();
                //set flag
                removed = true;
                //break and check next clause
                break;
            }
        }
    }

}
```

Figure 2: Code snipped for how to remove trivially satisfiable clauses

2. **checkTriviallyUnSat(Set <Clause>)**: This is a method which checks if there are any trivially unsatisfiable clauses, of the form (x)(!x), from a set of clauses. This is done by looping through all clauses and comparing them with each other.

   To do this, an iterator is used for the first loop which goes on until the iterator has another value. The first thing done in the first loop is to check if the the amount is literals inside the clause is 1 because if not, it cannot be an unsatisfiable clause, therefore, the loop proceeds to the next clause. Inside this loop, another iterator is obtained for the same clauses to compare and a nested loop is created to go through the clauses again.

   Inside the nested loop, the clause is obtained and the check for the amount of literals is done once again and if the size is not 1, this clause is skipped. If not, the literal from the first clause (outer loop) and the literal from the second clause (inner loop) are obtained. If these literals are of the same symbol and they have opposite negation, it must mean that they are of the form (x)(!x), hence, trivially unsatisfiable and true is returned. If no two clauses are found to be trivially unsatisfiable, false is returned.

3. **checkEmptyClause(Set <Clause>)**: This is a function which checks if there are any empty clauses in the set of clauses passed as parameters. This is done by iterating through all the clauses and if the clause has no literal, true is returned.

4. **getUnitClause(Set <Clause>)**: A unit clause is a clause containing only 1 literak, This is a function which returns a unit clause if there is one from the passed clauses and null if there is not. This is done by going through the clauses using an iterator and if the clause has only 1 literal, a copy of the clause is cloned and returned.

5. **getPureLiteral(Set <Clause>)**: This is a function which returns a pure literal from the list of passed clauses. A pure literal is a literal which appears either positive or negative throughout all clauses. This is started by keeping a boolean flag named searching, initiating an arraylist of literals and adding all literals in all clauses in it.

   After this, there is a loop through all these literals. Inside this loop, the first thing we do is set the boolean flag searching to true and get the current literal. Then there a nested loop to go through the literals again for comparison and if the symbol from the first literal, is the same as the symbol from the literal in the nested loop and their negation is opposite, searching is set to false and we break out of the loop because it means that the outer literal is not pure. After the nested loop, there is a check for searching and if this is true, a clone of the literal in the outer loop is returned because it means that it is pure. If searching is always false, null is eventually returned.

6. **exhaustivelyApply1LiteralRule(Clause, Set <Clause>)**: This is a function which given a unit clause and a set of clauses, returns a set of clauses after performing the 1 literal rule exhaustively on the passed set of clauses using the passed clause.

   This is done by looping through all clauses and doing the following:

   - Getting current clause and its literals from iterator and getting the first literal from the passed unit clause
   - Looping through each literal from the literals of the current clause and inside this loop:
     - The current literal is obtained
     - The unit clause literals negation value is inverted (so if it's not negated, negate it and vice-versa)
     - If the current literal is equal to the unit clause inverted, the current literal is removed from the set of literals of its parent clause.
     - The unit clause literal is re-inverted to its original negation
     - It is checked if the current clause is equal to the unit clause and if so, the clause is removed and the loop is broken.

   Finally, the remaining clauses with their remaining literals are returned

7. **applyPureLiteralRule(Literal, Set <Clause>)**: This is a function which returns a set of clauses after applying the pure literal rule exhaustively on the set of clauses passed using the passed pure literal. This is done by going through each clause in the passed clauses and then for each, clause there is another loop for each literal in this clause. If this literal is equal to the pure literal, the current clause in the loop is removed and the loop is broken. Finally the clauses are returned once all the clauses are traversed.

8. **chooseLiteral(Set <Clause>)**: This function is a heuristic and from my research, I decided to implement it based on the **Dynamic Largest Individual Sum(DLIS)**, meaning that each literal is binned and the variable with maximal max(CP, CN) is selected, with CP and CN being the number of positive and negative occurrences. So this function, given a set of clauses, returns the most frequent literal.

   This is done by first initialising a hashmap with a key of type Literal and a value of type Integer, named bins to hold the number of occurrences. Then, the all the literals in all the clauses are stored in an arraylist and each literal from this arraylist is traversed and if the bins contained the literal, the value of the bin for that literal would be incremented, otherwise a new entry in the bin hashmap is created with the currently visited literal. After this, the maximum from the bins is obtained and a new literal with the symbol of the most frequent literal is returned.

9. **DPLL(Set <Clause>)**: This function is the function that actually implements the Davis-Putnam-Logemann-Loveland algorithm which can be found below. This recursive function is passed a set of clauses and returns whether the boolean expression made up of the passed clauses is satisfiable.

```
BOOL DPLL(Clauses)
{
        // This part is the same as DP.
        Remove trivially satisfiable clauses from Clauses.

        If a clause in Clauses is trivially unsatisfiable return false.

        // This is the backtracking step.
        If Clauses contains an empty clause return false.

        If Clauses is an empty set return true.

        For each unit clause u in Clauses:
                Clauses = ExhaustivelyApply1LiteralRule(u, Clauses)

        For every pure literal p in Clauses:
                Clauses = ApplyPureLiteralRule(p, Clauses)

        // This is DPLL. Note that we have a decision
        // strategy in selecting the most efficient literal.
        l = ChooseLiteral(Clauses) // Heuristic?

        return DPLL(Clauses+{l}) OR DPLL(Clauses+{~l})
}
```

Figure 3: DPLL algorithm from the notes

This is done by first removing trivially satisfiable clauses by calling **removeTriviallySat()** with the passed clauses as parameters. Then, it is checked if there are any trivially unsatisfiable clauses by calling **checkTriviallyUnSat()** with the passed clauses as parameters and if so, 'UNSAT' is printed and false is returned. Otherwise, it is checked if there are any empty clauses by calling **checkEmptyClause()** with the passed clauses as parameters and if so, "UNSAT" is printed and false is returned. Then, if not, it is checked whether clauses is empty because if this is the case, we can say that the boolean expression is satisfiable and therefore, the assignments stored in the assignments global variable are printed one by one and true is returned.

The above were the base cases, and if none were reached, it means that we still have to perform some more operations before we get to know whether the expression is satisfiable or not. This is done by getting unit clauses one by one by calling **getUnitClause()** with the passed clauses as parameters and for each one, an assignment is set with the unit clause only literal's symbol as key and if it is negated the assignment is FALSE, otherwise, it is true. Moreover, for every unit clause, the 1-literal rule is applied exhaustively by calling **exhaustivelyApply1LiteralRule()** with the unit clause and clauses as parameters.

The next step involves getting the pure literals one by one by calling **getPureLiteral()** with the passed clauses as parameters and for each one, an assignment is set with the pure literals's symbol as key and if it is negated the assignment is FALSE, otherwise, it is true. Moreover, for every pure literal, the pure literal rule is applied exhaustively by calling **applyPureLiteralRule()** with the pure literal and clauses as parameters.

The final step involves getting a literal by calling the heuristic function **chooseLiterals()** with the passed clauses as parameters and the literal returned is added to the list of clauses. These clauses are then used as parameters to recall this function recursively until a base case is reached.

## 1.2 Testing

My implementation was tested by having unit tests for the methods in the literal and parser class. Moreover some integration tests were done to make sure that the DPLL method returns the expected the results.
It was made sure that all the code is hit by tests and this can be confirmed by having 100% test coverage as can be seen in the following figure.

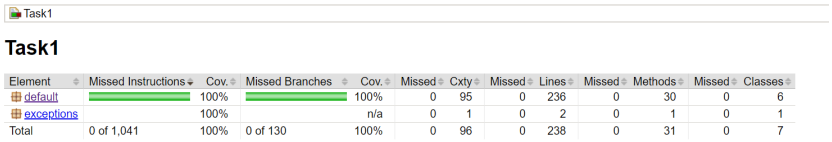


Figure 4: Code coverage for task 1

A breakdown of these tests can be found in the explanation below.

### 1.2.1 LiteralTest.java

This testing class contains tests for the constructor method by passing some valid and invalid strings and it is asserted that the correct Literal object or exception is returned. Moreover, tests for the overridden **equals(Literal)** method were implemented to test all the possible cases.

### 1.2.2 ParserTest.java

This testing class contains tests for the **parseString(String)** method by passing correct and incorrect strings and then asserting that the expected set of clauses are returned or that a **SyntaxErrorException** is thrown in case a string is not of the correct format.

### 1.2.3 DPLLTest.java

This testing class includes unit tests to test all the functions used by the DPLL algorithm and it also contains integration tests to make sure that the DPLL algorithm returns the expected results. It should also be noted that in order to test the implementation better, another symbol "v" was implemented. Below, one can find the integration tests performed:

```
v TRUE
x TRUE
y DONTCARE
z DONTCARE
w DONTCARE
```

Figure 5: Test result for (x, !y, z)(v, w, y)(v, !x, !z)

```
UNSAT
```

Figure 6: Test result for (x, y, z)(x, !y)(y, !z)(z, !x)(!x, !y, !z)

```
UNSAT
```

Figure 7: Test result for (x)(!x)

```
z DONTCARE
x FALSE
y DONTCARE
```

Figure 8: Test result for (!x)(!x,y)(!x,z)

```
z DONTCARE
x FALSE
y DONTCARE
```

Figure 9: Test result for (!x)(!x,y)(!x,z)

```
y TRUE
w TRUE
x TRUE
z TRUE
```

Figure 10: Test result for (w)(x, !y)(z,x)(y,!x)(x, !x)(z, !y)

```
w TRUE
x FALSE
y TRUE
z DONTCARE
```

Figure 11: Test result for (x,y)(w)(!x, z)

```
z DONTCARE
x TRUE
y FALSE
```

Figure 12: Test result for (x,y)(x,z)(!y, !z)

```
x TRUE
y TRUE
```

Figure 13: Test result for(x)(!x, y)

```
x FALSE
y DONTCARE
```

Figure 14: Test result for(!x)(!x, y)

## 1.3 Practical Applications of SAT

Just like everything else, anything which is not practical is not very useful. Boolean Satisfiability have been going through improvements throughout the years in order to be used in a practical environment to help in producing a better end product. Moreover, in some applications, the use of SAT solvers improves performance remarkably [1]. The most common applications can be split into

- **product configuration** which involves formulating model lines, dependencies between components and adhering to customer's restrictions [2].

hardware verification which includes checking the correctness of hardware designs at gate-level and properties related to temporal logic[3].

The most well known practical applications include:

1. **Combinational Equivalence Checking** : One of the uses of SAT applications checking the equivalence of two circuits, namely, combinational circuits[1].

2. **ATPG and SSF** : SAT is used in Automatic Test-Pattern Generation and Single Stuck-at Fault models to be able to identify fabrication defects in integrated circuits respectively[1]. These defects introduce a possibility of of circuit failure and hence, as explained above, SAT helps in making these models better, hence being able to identify and represent the faults ore robustly.

3. **Bounded Model Checking**: A model checking question asks to show that property always holds while a bounded model checking question asks whether there is a path which contradicts the property, or in other terms, a path which satisfies a failing property. SAT is used in the bounded model checking algorithms which are very common nowadays and are accepted by big companies such as Intel, AMD and IBM[3].

4. **Planning**: SAT is also used in systems when it comes to planning. In fact, Artificial Intelligence planning was the first push to SAT being used as a practical application[1] and from this, a lot of other opportunities were opened for SAT to be used in the areas explained above. This works in a similar way to BMS, hence, increasing the length of paths until the SAT is unsatisfiable. Different types of planning applications include timetables, air-traffic control and telegraph routing[3]. In addition, scheduling of events can also be considered as a practical application of SAT.

   To conclude, as can be seen from the above descriptions, usually SAT algorithms must be able to prove unsatisfiability. The best thing about is as time goes by better variable selection heuristics will be found and the implementation of SAT will be even more efficient and optimized.

### 1.4 How to run

1. Go to the directory "src/main/java"
2. Run "javac ParserExecutor.java"
3. Run "java ParserExecutor <anyExpression>"

## References


[1] J. Marques-Silva, "Practical applications of boolean satisfiability." http://www.icsd.aegean.gr/lecturers/konsterg/teaching/KR/SATapplications.pdf. Accessed on 1-05-2020.

[2] R. Michel, A. Habaux, and V. Ganesh, "An smt-based approach to automated configuration." https://smt2012.loria.fr/paper9.pdf. Accessed on 1-05-2020.

[3] C. Sinz, "Practical applications of sat." http://www.carstensinz.de/talks/RISC-2005.pdf. Accessed on 1-05-2020.

[4] "Practical sat solver." https://baldur.iti.kit.edu/sat/files/2016/l05.pdf. Accessed on 15-03-2020.

[5] "Dpll algorithm." https://en.wikipedia.org/wiki/DPLL_algorithm. Accessed on 17-03-2020.