CS 314
Project 2: Boolean Satisfiability Solver

1 Introduction

In this project, you will implement a Boolean satisfiability (SAT) solver for OCaml. The program takes a string representing a Boolean formula as input. This formula is in conjunctive normal form (CNF). CNF is a way of organizing logical statements used in mathematical logic and computer science. A statement is in CNF if it is a conjunction (AND) of one or more clauses, where each clause is a disjunction (OR) of literals, and a literal is either a variable or the negation of a variable.

For example, a statement in CNF might look like this:
(a OR NOT b) AND (b OR c OR NOT d) AND (d OR NOT e)

In our project, a literal also includes two boolean constants **TRUE** and **FALSE**. Your program should return a list of variable assignments that make the CNF formula true. For instance, to make (AND a b) true, both "a" and "b" need to be TRUE. To make (OR a b) true, there are three possible solutions, both "a" and "b" are TRUE, or "a" is TRUE, "b" is FALSE, or "b" is TRUE, "a" is FALSE.

The context-free grammar for the CNF formula is straightforward and defined in our project as below:

1. $< E > \ ::= \ (\ < W >\ )\ AND\ < E > \ |\ (\ < W >\ )$
2. $< W > \ ::= \ < T >\ OR\ < W > \ |\ < T >$
3. $< T > \ ::= \ < V > \ |\ NOT < V >$
4. $< V > \ ::= \ a \mid b \mid ... \mid z \mid TRUE \mid FALSE$

2 Input and Output of the Program

The input to the program is a string list. An example is below:
"(a OR b OR NOT c) AND (NOT a) AND TRUE"

The program's output is of the type (string, bool) list, which gives the first possible boolean value assignment of the variables to satisfy the CNF formula. An example is below:

[(“a”, false); (“b”, true); (“c”, false)]

If such an assignment doesn't exist, please return a list of one tuple, as in the example shown below:

[(“error”, true)]

3 Basic Function Implementations

First, you need to break down the input string into a string list by eliminating white spaces. This function is already provided to you in the “project2_driver.ml” file, called “tokensListFromString (str : string)”.

Next, you will need to implement a basic called “partition” in the code package given to you. The purpose of the “partition” function is to break down a string list with respect to a delimiter. It will help you get the input string list represented in a way that facilitates further processing.

Name	Input Type	Output Type
partition	(string list), string	string list list
getVariables	string list	string list
generateDefaultAssignments	string list	(string * bool) list
generateNextAssignments	(string * bool) list	(string * bool) list * bool
lookupVar	(string * bool) list, string	bool

Table 1: Required Functions

The input and output type of the “partition” function is given in Table 1. An example of the input and output of the “partition” function is below:

```
Name: partition
Input: ["("; "a"; "OR"; "NOT"; "b"; ")"; "AND"; "("; "b"; ")"] "AND"
Output: [["("; "a"; "OR"; "NOT"; "b"; ")"];[ "("; "b"; ")"]]
```

There are a few other basic functions you have to implement, listed below:

The next function you will have to implement is called “getVariables”. It takes a string list and gets the list of the variable names from the CNF. The input and output types are specified in Table 1. It filters out the left/right parenthesis, AND/OR, TRUE/FALSE, and NOT items out of the string list representing the input. The output should not have duplicates. An example is given below:

```
Name: getVariables
Input: ["("; "a"; "OR"; "NOT"; "b"; ")"; "AND"; "("; "b"; ")"]
Output: ["a"; "b"]
```

You will also have to implement the function called “generateDefaultAssignments”. It takes the list of variables, and outputs a list of tuples, each tuple being a variable name and the boolean value “false”. It is used to initialize the set of variables to the “false” value. An example is given below:

```
Name: generateDefaultAssignments
Input: ["a";"b"]
Output: [("a", false);("b", false)]
```

You will also have to implement the function called "generateNextAssignments". This function takes a list of string-boolean tuples, and returns an updated list of string-boolean tuples as if the singular binary number represented by the booleans is incremented by 1, as well as an additional boolean value *carry* for when it overflows. Essentially, starting with the rightmost variable, if the checked variable is assigned to false, it is set to true. If the checked variable is true, it is set to false and the algorithm carries to next variable to the left. The returned boolean value *carry* is true only if the resulting binary number overflows the given space, i.e. the algorithm reaches the leftmost variable and still carries.

```
Name: generateNextAssignments
Input: [("a", false);("b", false)]
Output: ([("a", false);("b", true)], false)

Input: [("a", false);("b", true)]
Output: ([("a", true);("b", false)], false)

Input: [("a", true);("b", true)]
Output: ([("a", false);("b", false)], true)
```

You will also have to implement the function called "LookupVar". This function takes a assignment list of string-boolean tuples as well as a string, and returns the boolean value associated with the given string in the assignment list. Behavior for string values not in the assignment list will not be tested.

```
Name: LookupVar
Input: [("a", false);("b", true)] "a"
Output: false
```

4 Advanced Function Implementations

Now, we will handle the boolean satisfiability evaluation function and data structures to represent the satisfiability function. There are three functions you will have to implement here. Their types are specified below.

1. buildCNF: string list → (string * string) list list
2. evaluateCNF: (string * string) list list → (string * bool) list → bool
3. satisfy: string list → (string * bool) list

We will describe each of the functions below.

4.1 The buildCNF function

This function builds a data structure to represent the CNF. It takes a string list as input, which you obtain from a string by calling "tokensListFromString" to break down a string. Next, you can take advantage of the helper function "partition" built earlier, plus some further implemetnation to construct a list of (string*string) list, which represents a list of clauses from the CNF. Examples are shown below:

```
Input: ["("; "a"; "OR"; "NOT"; "b"; "OR"; "c"; ")"; "AND"; "("; "b"; ")"]
Output: [[("a", ""); ("b", "NOT"); ("c", "")]; [("b", "")]]

Input: ["("; "NOT"; "a"; "OR"; "b"; ")"; "AND"; "("; "NOT"; "b"; ")"]
Output: [[("a", "NOT"); ("b", "")]; [("b", "NOT")]]
```

Note that one literal could be the negation of a variable or the variable itself. The tuple string*string in the list specifies that. If a NOT is decorating a variable, then the second item in the tuple is "NOT"; otherwise, it is an empty string. In the above example, there are two clauses, hence, there is a list of two lists, each list representing a clause. Each list representing a clause consists of the components divided by the "OR" operator in the clause.

4.2 The evalauteCNF function

This function takes the data structure of a CNF returned by the "buildCNF" function and an assignment list as input and returns a boolean variable. It evaluates the variable assignment on the CNF; if it satisfies, it returns true; otherwise, it returns false. An example is given below:

```
Input: [[("a", ""); ("b";"NOT")]; [("b", "")]] [("a", true), ("b, false")]
Output: false

Input: [[("a", ""); ("b";"NOT")]; [("b", "")]] [("a", true), ("b, true")]
Output: true

Input: [[("a", ""); ("b";"NOT")]; [("b", "")]] [("a", false), ("b, true")]
Output: false
```

4.3 The satisfy function

This function is the top-level function for the entire program. It takes a string list and returns the first variable assignment it finds that satisfies the CNF or an error (specified in Section 2). The order of trying variable assignment should be that you try the all-false assignment first, and then use "getNextAssignment" similar to a carry adder to update the next variable assignment. It should terminate when it exhausts all possible variable assignments or finds the first one that works. An example of input and output to the "satisfy" function is given below.

```
Name: satisfy
Input: ["("; "a"; "OR"; "NOT"; "b"; ")"; "AND"; "("; "b"; ")"]
Output: [("a", true); ("b", true)]
```

5 Code Package

A code package is given to you. You will need to implement everything in "project2.ml". A few helper functions are provided in the "project2_driver.ml". Please do not modify "project2_driver.ml". All your own helper functions should go into "project2.ml". We will test each function with independent inputs and outputs. For instance, when you implement "getVariable" wrong, you will get no credit for "getVariable". But we will use a correct "getVariable" function to create inputs when we test others functions. However, any functions called within other functions will behave defined by you, and are not guaranteed to work properly. For example, calling "getVariable" in the "satisfy" function used your implementation.

Please do not modify the type signature of each function that you are required to implement.

Testing on An Interpreter

You can test your code in the OCaml interpreter. If you can type "ocaml" in the terminal on an ilab machine, it will invoke the interpreter environment. You can load your program into the OCaml interactive toplevel, and invoke the functions from the toplevel. In the OCaml interpreter, enter the following commands in OCaml top-level:

```
# #use "project2.ml";;
```

It will load the code in "project2.ml" into the environment. After you load this file, you can then call functions you have implemented in "project2.ml". To use these functions in the "project2_driver.ml" file, enter the following commands in the interpreter:

```
# #mod_use "project2.ml";;
# #load "str.cma";;
# #use "project2_driver.ml";;
```

Submission

Please only submit the "project2.ml" file to GradeScope.