CS 314 Principles of Programming Languages

Project 2: Boolean Satisfiability

1 Introduction

In this project you will implement a Boolean satisfiability (SAT) solver in OCaml. The program takes as input a string representing a Boolean formula. This formula may involve constants (`TRUE` and `FALSE`) and variables (represented by lowercase letters `a` through `z`). Each variable may be either `TRUE` or `FALSE`. The program should return a list of variable assignments that make the 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.

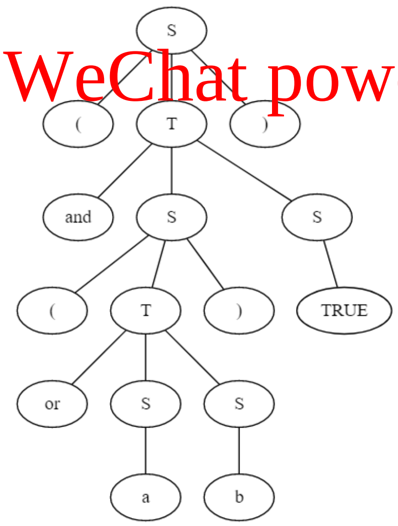
The grammar for the logical formula in our project is defined below:

< S > ::= `TRUE`
< S > ::= `FALSE`
< S > ::= `a` | `b` | · · · | `z`
< S > ::= `(` <T> `)`
< T > ::= `not` <S>
< T > ::= `and` <S> <S>
< T > ::= `or` <S> <S>

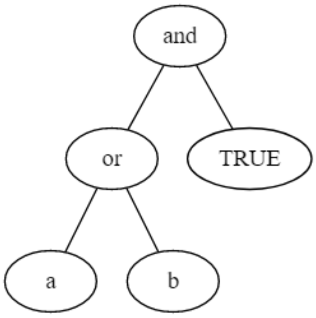
1.1 Tree representation of Boolean formula

To begin with, you will first convert the *string list* into a parse tree. You will write a recursive descent parser to extract the parse tree from the string list.

For example, the parse tree for string list `( and ( or a b ) TRUE )` with respect to the grammar defined above is the following:



The parse tree will be further converted into an abstract syntax tree (AST). The AST is succinct form of the parse tree. A parent node represents a boolean operator, and its child(ren) node(s) represent the operands. In the AST, there will be no node that represents a non-terminal. Each node corresponds to a terminal in the input sentence. The AST example corresponding to the parse tree above is below:



1.2 Evaluating the AST

The main idea for the SAT solver is to enumerate all possible assignments for the variables and evaluate the tree with each assignment. It then only keeps the assignments that make the given boolean formula `TRUE`. Evaluating the AST representation is more efficient than evaluating the raw parse tree representation.

1.3 Generating the parse tree

You will implement an LL(1) parser for the grammar using OCaml, using the recursive descent parsing method. In our class, Lecture 6–8 covered the basics of recursive descent parsing. However, there is a global variable in the recursive descent parsing code skeleton. The global variable is used for the next input token and its position in the input string list. Global variables are not allowed in pure functional languages as it introduces mutable state. You are not supposed to have mutable global variables in this project.

You will need to implement it in the pure functional style. The general idea is, whenever we perform a function call to parse a non-terminal, we pass the non-terminal and the string list, and the function returns the (sub) parse tree for this non-terminal and the remaining string list (after the matching substring list for the non-terminal has been removed). Therefore, given the same input to each function, it always returns the same value no matter how many times it is called. It ensures referential transparency.

1.4 Generating boolean value assignments: mimicking the binary adder

After obtaining the tree representation, the next step is to generate all possible combinations of variable assignments and try them one by one. Because there are exponentially many possible combinations, you may not want to generate them and store them all at once. Instead, you can generate one combination at one time, evaluate it, and move on to the next combination.

You will implement the boolean value assignment list in a way that mimics the carry adder for binary. Suppose the formula contains n variables. Each value assignment can be seen as an n-digit binary number. The leftmost digit is 1 if the first variable is assigned `true`, 0 if assigned `false`. The second variable corresponds to the second digit, etc.

We begin by setting each variable to `false`. This corresponds to the binary number 0. Each time, it generates the next assignment based on the previous assignment, as if the binary number is incremented by 1.

When you implement this, you do not need to actually convert boolean value to integer value. Instead, you can change `true` to `false` or `false` to `true` as if 1 is flipped to 0 or 0 is flipped to 1.

NOTE that you will treat the rightmost variable in the variable list as the least significant bit when generating a next assignment. When we do grading, this is how we are going to test it. DO NOT implement it the other way that treats the leftmost variable as the least significant bit.

The get next assignment function will take an existing assignment as input and return the next assignment as well as a carry value (boolean) that may affect the next value of the significant bit.

If all variables are assigned `true`, then it has reached the last possible assignment in the list, and the solver should terminate.

2 Project Specification

2.1 Data Types

We use the same OCaml type to represent both parse trees and ASTs. The type definition is:

```
type tree = TreeNode string * tree list
```

That is, each tree node consists of a string (representing the lexeme of itself) and a list of its children. Each of its children is also a *TreeNode*. A leaf *TreeNode* is the one whose children node list is empty.

Below we give the OCaml representation of the parse tree and AST for the example given in Section 1.1:

Parse tree:

```
TreeNode ("S",[
 TreeNode ("(", []);
 TreeNode ("T",[
  TreeNode ("and", []);
  TreeNode ("S",[
   TreeNode ("(", []);
   TreeNode ("T",[
    TreeNode ("or", []);
    TreeNode ("S", [TreeNode ("a", [])]);
    TreeNode ("S", [TreeNode ("b", [])])]);
   TreeNode (")", [])]);
  TreeNode ("S", [TreeNode ("TRUE", [])])]);
 TreeNode (")", [])])
```


AST:

```
TreeNode ("and",[
 TreeNode ("or", [
  TreeNode ("a", []);
  TreeNode ("b", [])]);
 TreeNode ("TRUE", [])])
```


In the following we specify each function that you should implement.

2.2 Basic Functions

2.2.1 Function scanVariable

Signature: `scanVariable (input : string list) : string list`
Description: This function takes as input a list of tokens representing a Boolean formula. It should return a list of variables contained in the formula. The returned list can be in any order, but it should not contain duplicates.

Examples:

```
scanVariable ["(";"or";"a";"b";")"] = ["a";"b"]
scanVariable ["(";"or";"a";"a";")"] = ["a"]
```

2.2.2 Function generateInitialAssignList

Signature: `generateInitialAssignList (varList : string list) : (string * bool) list`
Description: This function takes a list of variable names, and generates an initial value assignment for them. Value assignments are represented as a list of `string` and `bool` pairs. Each pair contains a variable name and the value assigned to that variable. The initial value assignment should assign `false` to each variable.

Example:

```
generateInitialAssignList ["a";"b"] = [("a",false);("b",false)]
```

2.2.3 Function generateNextAssignList

Signature: `generateNextAssignList (assignList : (string * bool) list) : ((string * bool) list * bool)`
Description: This function takes a value assignment list, and returns the next assignment list as if a binary number is incremented by 1. It also returns a Boolean value called *carry*. In the base case, when the input assignment list is empty, it should return an an empty list and a carry value of `true`. If the input assignment list is not empty, it then calls itself on the rest of the list, and return a tuple of assigned variable list and carry value. If the returned carry bit is `true`, it needs to flip the variable at head of the input assignment list, and depending on the head variable is `true` or `false`, it needs to return a carry value correspondingly.

Examples:
```
generateNextAssignList [("a",false);("b",false)] = ([("a",false);("b",true)], false)
generateNextAssignList [("a",false);("b",true)] = ([("a",true);("b",false)], false)
generateNextAssignList [("a",true);("b",true)] = ([("a",false);("b",false)], true)
```

2.2.4 Function lookupVar

Signature: `lookupVar (assignList : (string * bool) list) (str : string) : bool`
Description: This function takes a value assignment list and a variable name. It returns the value assigned to this variable.

Example:
```
lookupVar [("a",false);("b",true)] "a" = false
```

2.2.5 Function evaluateTree

Signature: `evaluateTree (t : tree) (assignList : (string * bool) list) : bool`
Description: This function takes an abstract syntax tree and a value assignment list. It evaluates the Boolean formula represented by the tree, and returns the result.

Examples:
Let `t` be the AST tree shown in Section 2.1,
```
evaluateTree t [("a",false);("b",false)] = false
evaluateTree t [("a",true);("b",false)] = true
```

2.3 Parse and SAT Solver Functions

Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

2.3.1 Function buildParseTree

Signature: `buildParseTree (input : string list) : tree`
Description: This function takes as input a list of tokens representing a Boolean formula. It should return a parse tree for the input.

You can implement your own helpers functions that can help *buildParseTree*. You might need to implement separate functions for parsing the $<S>$ and $<T>$ non-terminals. NOTE that it is mutually recursive. You can use the "let .... and ..." to implement mutually recursive functions in OCaml. You can find the documentation here: `https://ocaml.org/learn/tutorials/labels.html#Mutually-recursive-functions`

Example output:
See Section 2.1.

2.3.2 Function buildAbstractSyntaxTree

Signature: `buildAbstractSyntaxTree (input : tree) : tree`
Description: This function takes as input a parse tree for a Boolean formula. It should return an abstract syntax tree for that same formula.

Example output:
See section 2.1.

2.3.3 Function satisfiable

Signature: `satisfiable (input : string list) : (string * bool) list list`
Description: In this function you will put together all functions implemented above to build a satisfiability solver. The input is a list of tokens representing a Boolean formula. The output is a list of all value assignments that would make the formula evaluate to true. The list can be in any order.

Example:
Let `input` be the formula shown in Section 1.1, then
```
satisfiable input = [[("a",false);("b",true)];[("a",true);("b",false)];[("a",true);("b",true)]]
```

3 Testing

3.1 Testing in the 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 toplevel:

```
# #mod_use "proj2_types.ml";;
# #use "proj2.ml";;
```

It will load the "proj2_types.ml" as a module into the top-level. It will load the code in "proj2.ml" into the environment. Please DO NOT modify anything in "proj2_types.ml". Your implementation must go to "proj2.ml". After you load these two files, you can then call functions you have implemented in "proj2.ml".

We also provide several helper functions to help you debug your program. These functions are included in `proj2_driver.ml`. To use these functions, enter the following commands in the interpreter:

```
# #mod_use "proj2_types.ml";;
# #mod_use "proj2.ml";;
# #load "str.cma";;
# #use "proj2_driver.ml";;
```

The functions provided by `proj2_driver.ml` are:

1. `tokenListFromString`: This function takes a string and splits it using the white space delimiter to generate a list of strings representing a list of tokens.

2. `parseTreeFromString`: This is a wrapper around `buildParseTree` function that invokes `tokenListFromString` to process input.

3. `astFromString`: This is a wrapper around `buildAbstractSyntaxTree`.

4. `satisfiableFromString`: This is a wrapper around `satisfiable`.

5. `printTree`: This function takes a tree and prints it.

3.2 Testing by Compilation to Native code

OCaml files can be compiled into native code and executed using command line in the terminals. We have provided a Makefile for you in which there are commands for compiling the "proj2.ml" file and running it.

NOTE that you do not have to invoke OCaml interpreter in this case. You can simply type "make test" at the terminal of any ilab machine. It will report the test results for each test case. If your functions are not completely implemented in "proj2.ml", it will report test cases failed for these functions. The test functions are implemented in "proj2_test.ml". Feel free to reuse the functions in "proj2_test.ml" in the interpreter. You can copy and paste the useful functions in "proj2_test.ml" into a "yourtest.ml" file, and use #use "yourtest.ml" to load its code into the interpreter for testing. Be careful about the dependence between different functions and make sure you load the code into the interpreter in the correct order.

All grading will be done on ilab. The OCaml installed on ilab is at version 4.05.

4 Submission & Grading

You should submit a compressed tarball with filename `proj2_$(NETID).tar.gz`. For example, if your netid is `ab123`, then the filename is `proj2_ab123.tar.gz`. This tarball contains a single directory `proj2`, and inside this directory there is a single file `proj2.ml`.

There is an automatic script that prepares the tarball for submission. To invoke this script, enter `make submit` from the terminal (not in OCaml toplevel).

You should **not** use any module in the OCaml standard library other than `List`. You should **not** use any imperative or objective features of OCaml. The autograder includes a script that detects their usage.

5 Questions

If you have questions about this project, please post them on Sakai forum.