Project 2: Boolean Inference Checker

1. Data Model

A ***logical variable*** is a string of one or more lowercase letters *a,...,z,* not beginning with *v*. For example, the following are logical variables: "p", "foo", "qrs", "ivt".

A ***symbol*** is either a logical variable or one of the following eleven strings: "T", "F", "^", "v", "~", "=>", "<=>", "(", ")", ":.", "," Note that, by comparison with the Boolean expression evaluator, the new symbols are logical variables, the comma, and the *therefore* symbol ":.".

A ***symbol string*** is the concatenation of zero or more symbols and/or spaces. For example, the following are symbol strings:

- `"T :.  "`
- `"=> Tv~  F(( F"`
- `"T foo  => F ^ (F pr :. ) v F)"`
- `""`
- `"  "`
- `"Foo" --> ["F", "oo"]`

and the following are not symbols strings

- `"T X  "`
- `"  : .F => T"`
- `"TF p*^)"`
- `"FOO"`

The tokenizer should emulate the following algorithm. Your algorithm probably won't use the same process, but it must give the same results as this one:

Algorithm *Tokenize*(*string s*):

```
● list(string) tokens := [ ]
● i := 0
● // invariants:i in {0..len(s)}, tokens is a list of the complete tokens in s[0:i].
● while i < len(s)
    ○ If s[i:len(s)] has an initial segment that is a symbol
        ■ let t be the longest symbol that is an initial segment of s[i:len(s)]
        ■ push t onto the back of tokens
        ■ i := i + len(t)
    ○ else if s[i] is a space, i:= i+1
    ○ else halt and return an error message
● return tokens (and possibly a success message)
```

The hard part of the tokenizer is the underlined step. You can approach this as follows:

```
● if s[i] = 'T', then t = "T".
● if s[i] = '^' then t = "^"
● if s[i] = '=' and len(s)>i+1 and s[i+1] = '>' then t = "=>"
● etc.
● if s[i]  is a lowercase letter other than v
    ○ j := i
    ○ while j < len(s) and s[j] is a lowercase letter
        ■ j:=j+1
    ○ t := s[i:j]
```

For example, the string "foo ^ bar => T" should generate the tokens ["foo","^","bar","=>","T"], as follows:

i	*tokens*
0	[]
3	["foo"]
4	["foo"]
5	["foo","^"]
6	["foo","^"]
9	["foo","^","bar"]

....

A ***Boolean expression*** is a vector of symbols defined as in the previous assignment, with the exception that, in addition to other grammar rules, logical variables are also considered to be unbreakable expressions. So the only rule that chances is the rule for unbreakable expressions:

U `::= Const | "(" B ")" | Lvar`

A ***Boolean inference*** consists of one or more Boolean expressions separated by commas, followed by a *therefore* sign ":.", followed by a Boolean expression. Formally,

Ps ::= *Bexp* | *Bexp* `","` *Ps*

Inf ::= *Ps* `":."` *Bexp*

An AST is as in the previous assignment.

An ***inference*** is a struct with two attributes:

- *premises* -- a vector of AST's.
- *conclusion* -- an AST

2. Functions

The Boolean expressions p_1,...p_n are the ***premises*** of the Boolean inference

$p_1, \dots, p_n$ `:.` q

and q is its ***conclusion***.

A Boolean inference is ***invalid*** if there is a consistent substitution of truth values ("T" and "F") for its logical variables that makes all of its premises true and its conclusion false. Otherwise, it is ***valid.***

For example, the following strings represent valid inferences:

- `"p => q, ~q :. ~p"`
- `"p ^ ~q => r ^ ~r :. ~p v q"`
- `"p => q, q => r, r => s v t, ~s, p :. t"`

and the following strings represent invalid inferences:

- `"p => q, ~p :. ~q"`
- `"p => q, q => r  :. q => p ^ r"`

The following eight functions shall be written for this assignment:

`void Insert(string s, list<string> *L)`

- *Precondition: L* is a list of distinct logical variables, sorted alphabetically
- *Effects*:
    - if *s* is already in *L,* there is no effect.
    - If *s* is not already in *L*, it is added to *L* so that the length of *L* increases by 1 and *L* remains sorted.

For example, if L is ["ab","ac","boo"] and s is "ad", then after the call, L should be ["ab","ac","ad","boo"].

1. If s is an initial segment of t, and s is not equal to t, then s < t in lexical order.
2. If neither s nor t is an initial segment of the other, *i* is the smallest integer such that *s*[*i*] is not equal to *t*[*i*], and *s*[*i*] < *t*[*i*], then *s* < *t* in lexical order.

`list<string> vars(AST T)`

- If *T* is an AST of a Boolean expression, then *vars*(*T*) is a list of all logical variables occurring in *T*, sorted in alphabetical order, with duplicates eliminated.

For example, if T is the AST of "foo ^ bar => (bar ^ T)", then vars(T) is ["bar","foo"].

```
T ^ p => q v p     --->      ["p","q"]
  ● If T is a leaf node
      ○ if T.info is "T" or "F", return []
      ○ if T.info is a variable, return [T.info]
  ● If  T.info="~", return vars( *((T.children)[0]))
  ● Let A be the vars of the first child, B be the vars of the
    second child, and return A+B, merged (that is, sorted
    together with duplicates removed).
```

`list<string> vars(vector<AST> Ts)`

- If *Ts* is a list of AST's of Boolean expressions, then *vars*(*Ts*) is a list of all logical variables occurring in the members of *Ts*, sorted in alphabetical order, with duplicates eliminated.

For example, if Ts is [x,y], where x is the AST of "foo => bar" and y is the AST of "bar ^ T <=> dug", then vars(Ts) = ["bar","dug","foo"].

`list<bool> bits(int i, int N)`

- if $N > 0$ and *i* is an integer in $\{0,...2^N-1\}$, then *bits*(*i,N*) is the *N*-bit representation of *i* as an integer.

For example *bits*(0,3) = [0,0,0]; *bits*(3,3) = [0,1,1], *bits*(6,4) = [0,1,1,0].

`AST substitute(list<bool> vals, list<string> vars, AST Exp)`

- If *vars* and *vals* are the same length, then *substitute*(*vals, vars, Exp*) is the AST obtained from *Exp* by substituting "T" for *vars*[i] *w*henever *vals*[*i*]=1, and "F" for *vars*[*i*] *whenever vals*[*i*] = 0.

For example, substitute([0,1,0], ["p","q","r"], x), where x is the AST of "p => q ^ q ^ r" would return the ast of "F => T ^ T ^ F".

`bool witnessInvalid(list<bool> vals, list<string> vars, inference I)`

- If *eval*(*substitute*(*vals,vars,p)*) is true for every premise *p* of *I,* and *eval*(*substitute*(*vals, vars, I.conclusion*)) is false, then *witnessInvalid*(*vals, vars, I*) is true;
- Otherwise, *witnessInvalid*(*vals, vars, I*) is false

For example, if vals = [0,1,1], vars = ["p","q","r"], and I is the inference represented by "~p, p v q :. ~r", then witnesswInvald(vals, vars, I) should return true. But if vals = [0,0,1], vars = ["p","q","r"], and I is the inference represented by "~p, p v q :. ~r", then witnesswInvald(vals, vars, I) should return false.

`bool valid(inference I)`

- *valid*(*I*) is *true* if *I* is valid, and *false* otherwise.
- That is, if *Vars* is a is a list of the distinct variables occurring in *I,* and *N* = *length*(*Vars*), then
    - if there is an *i* in [0...2^N-1] such that *witnessInvalid*(*bits*(*i,N*),*Vars,I*) is true, then *valid*(*I*) is false.
    - otherwise, *valid*(*I*) is true.

For example, if I is the inference represented by "~p, p v q :. ~r", then valid(I) is false. If I is the inference represented by "p => ~p :. ~p", then valid(I) is true.

`string validInference(string s)`

- if *s* is not a symbol string, *validInference(s*) is `"symbol error"`
- if *s* is a symbol string, but the tokenization of s does not parse as an inference, *validInference*(*s*) is `"grammar error"`
- If *s* is a symbol string that parses as an inference, then *validInference*(*s*) is `"valid"` if the inference is valid, and `"invalid"` otherwise.
- For example,
    - *validInference*(`"=> u F *"`) is "symbol error"
    - *validInference*(`"u => :. :. F pr"`) is "grammar error"
    - *validInference*(`"u => p, u :. p"`) is "valid"
    - *validInference*(`"p v q, ~q :. p"`) is "valid"

- *validInference*(`"pa v q, ~pa v ~q :. pa <=> q"`) is "invalid".

For 100% credit at midnight on Thursday Nov. 9. Each week or part of a business week (not counting Thanksgiving week) after that counts off 10%, down to a minimum of 70% for students who submit a working program by the end of the semester.

Students are free to browse the Web for sample code that might help, or even to download and use libraries or publicly available code. However, it will be considered academic dishonesty to use, or look at code written by another student, or for one student to share or show code with another student. Therefore, you should not save your code on repl.it using a free account. Some additional C++ compilers are available online as applets:

- https://www.jdoodle.com/online-compiler-c++
- http://cpp.sh/

You may also use any other C++ compiler you can find. However, make sure your code runs on repl.it before you submit it.

```
/*   INSTRUCTIONS:


Only projects that follow the instructions below will receive credit:

1. Copy everything from below where it says TEST "FUNCTIONS START HERE".

2. Your program should include all the structures and functions given in the
specification as is. You may use as many helper functions but do not change the
specifications given in the project.

3. Pay close attention to the string value returned from validInference
function. That is, take care of uppercase,lowercase and spelling errors (e.g.
using grammer instead of grammar) in your code.

4. Do not include any cout statements in your code, nor any main function.

5. While submitting your project. Do not include the test harness in your code.
```

3. Test Harness

```
/*   INSTRUCTIONS:

Only projects that follow the instructions below will receive credit:

1. Copy everything from below where it says TEST "FUNCTIONS START HERE".

2. Your program should include all the structures and functions given in the
specification as is. You may use as many helper functions but do not change the
specifications given in the project.

3. Pay close attention to the string value returned from validInference
function. That is, take care of uppercase,lowercase and spelling errors (e.g.
using grammer instead of grammar) in your code.

4. Do not include any cout statements in your code, nor any main function.

5. While submitting your project. Do not include the test harness in your code.

6. Accepted submission formats: .txt, .cpp (or a zip file having either one of
those).

*/

// 11/06/2017: Minor correction in checkSubstitute function in the AST part



#include<iostream>
#include<string>
#include<list>
#include<vector>

using namespace std;

typedef struct AST* pNODE;

struct AST {string info; pNODE children[2]; };

struct inference
{
  vector<AST> premises;
  AST conclusion;
};

void prinTree(AST T)
{
// If both children are NULL, just print the symbol
    if (T.children[0]==NULL){
    cout<< T.info;
    return;
    }

    // print an opening paren, followed by the symbol of T, followed
    // by a space, and then the first child.
    cout << "(" << T.info<<" ";
    prinTree(*(T.children[0]));
    cout << " ";

    // print the second child if there is one, and then a right paren.
    if (T.children[1] != NULL)
    prinTree(*(T.children[1]));
    cout << ")";
}

string ASTtoString(AST T) //converts an AST to String
{
      string s;
      // If both children are NULL, just print the symbol
      if (T.children[0] == NULL) {
            s = s + T.info;
            return s;
      }

      // print an opening paren, followed by the symbol of T, followed
      // by a space, and then the first child.
      s = s + "(";
      s = s + T.info;
      s = s + " ";
      s += ASTtoString(*(T.children[0]));
      s = s + " ";

      // print the second child if there is one, and then a right paren.
      if (T.children[1] != NULL) {
            s += ASTtoString(*(T.children[1]));
      }
      s = s + ")";
      return s;
}

pNODE cons(string s,pNODE c1,pNODE c2)
```

```
{
    pNODE ret = new AST;
    ret->info = s; // same as (*ret).info = s
    ret->children[0]=c1;
    ret->children[1]=c2;
    return ret;
}

//******************** TEST FUNCTIONS START HERE ****************************
void checkInsert()
{
  list<string> mylist = {"bar", "foo"};

  Insert("boo",&mylist);

  list<string> test={"bar","boo","foo"};

   if(mylist==test)  //Insert  should  return  mylist  in  sorted  form  with  new
variable "boo" added
    cout<<"\nINSERT Function PASSED";

  else
    cout<<"\nINSERT Function FAILED";
}


//Checks list<string> vars(AST T)
void checkVars1()
{
    AST A,B,C,D,E;

    //Expression: p v q => T
    //Replace it with your own AST for your own test cases
    A= *cons("=>",&B,&C);
    B= *cons("v",&D,&E);
    C= *cons("T",NULL,NULL);
    D= *cons("p",NULL,NULL);
    E= *cons("q",NULL,NULL);


    list<string> allvars = vars(A);
     list<string> testvars= {"p","q"} ; //List of all variables that should be
in AST T, in sorted order


     if(allvars==testvars) //Insert should return mylist in sorted form with new
variable "boo" added
        cout<<"\nVars(AST T) PASSED";
    else
      cout<<"\nVars(AST T) Failed!!";

}

/*//Checks list<string> vars(vector<AST> Ts)
void checkVars2(inference inf)
{

    list<string> allvars = vars(inf.premises);

    list<string> test= {"",""}   //ADD all the variables here in your premises
    test.sort();


      if(allvars==test) //Insert should return mylist in sorted form with new
variable "boo" added
        cout<<"\nVars(vector<AST Ts>) PASSED";


    else
      cout<<"\nVars(vector<AST Ts>) Failed!!";

}*/


void checkBits()
{
  list<bool> num= {1,0,0,0,0,1,0,1,1}; //Number 267
  int i=267;
  list<bool> test1=bits(i,9);
  list<bool> test2=bits(i,5); //should return an empty list

  if(test1==num && test2.empty())
    cout<<"\nBits ALL test cases PASSED";

  else if(test1!=num) //Correction: should have been test1 instead of test
    cout<<"\nBits failed to generate sequence for number "<<i;

  else
    cout<<"\nBits ALL test cases Failed";

}


void checkSubstitute()
```

```
{
    AST A,B,C,D,E,F;
    //AST ~p ^ q v r
    A= *cons("v",&B,&C);
    B= *cons("^",&D,&E);
    C= *cons("r",NULL,NULL);
    D= *cons("~",&F,NULL);
    E= *cons("q",NULL,NULL);
    F= *cons("p",NULL,NULL);

    list<bool> b={0,0,1}; // p=0 (F), q=0 (F), r=1 (T)
    list<string> v={"p","q","r"};

    AST result=substitute(b,v,A);

    string output=ASTtoString(result); //Converting to string for comparison

    string subAST="(v (^ (~ F ) F) T)"; //Correction Made

    if(output == subAST)
      cout<<"\nSubstitute ALL test case passed";

    else
      cout<<"\nSubstitute failed for ~p ^ q v r";
}



void  checkValidInference()
{
    string t1="~ p v F => (ac v ~dc), ac ^ dc :. p";  //Should evaluate to
Invalid
   string t2="Foo V Bar, bar <=> Foo :. T";  //symbol error
   string t3="(ab ^ cd), cd => ~ab :. ^ cd";   //grammar error


   string op1,op2,op3;

   op1=validInference(t1);
   op2=validInference(t2);
   op3=validInference(t3);

   if(op1 == "invalid" && op2 == "symbol error" && op3 == "grammar error")
     cout<<"\nValid Inference all cases passed";

   else if(op1!= "invalid")
      cout<<"\n Valid Inference test case "<<t1<<" failed. EXPECTED: invalid,
OUTPUT generated: "<<op1;
```

```
   else if(op2!= "symbol error")
       cout<<"\n  Valid  Inference  test  case  "<<t2<<"  failed.  EXPECTED:  symbol
error, OUTPUT generated: "<<op2;

    else if(op3!= "grammar error")
      cout<<"\n  Valid  Inference  test  case  "<<t3<<"  failed.  EXPECTED:  grammar
error, OUTPUT generated: "<<op3;

    else
      cout<<"\n Valid Inference test ALL test cases FAILED";
}


int main()
{

  checkInsert();
      checkVars1();

      checkValidInference();
      return 0;
}
```