BoolInterpretation.h

```
#pragma once

class BoolInterpretation {
  private:
    static const size_t DATA_SIZE = 26;

  private:
    bool _grammer[DATA_SIZE]{false};

  public:
    bool getCharAt(char ...) const;
    void setCharAt(char, bool);
    size_t turnOnCount() const;
    void modify(size_t)
    (excludeAllTrueByMask)
};  void setAllTrue();
```

Tella 15

BoolInterpretation.cpp

```
#include "BoolInterpretation"

void BoolInterpretation::modify(size_t permutationNo){
    for(int i =DATA_SIZE-1 ; i>=0; --i){
    {
        if(_grammer[i])
        {
          if(permutationNo%2==0)
            {
              _grammer[i]= false;
            }
          permutationNo/=2;
        }
    }
}

bool BoolInterpretation::getCharAt(char ch)const
{return _grammer[ch-'a'];
}
void BoolInterpretation::setCharAt(char ch, bool val)
{_grammer[ch-'a']= val;
}
size_t BoolInterpretation::turnOnCount() const
{ size_t cnt=0;
   for(size_t i=0;i<DATA_SIZE;i++){
       if(_grammer[i])
           cnt++;
return cnt;}
```

```
void BoolInterpret::setAllTrue()
{
for(int i=0; i<=DATA_SIZE; i++)
  {
    grammar[i]= true;
  }
}
```

Expr.h

```
#pragma once
#include "BoolInterpretation.h"

class Expr {
public:
    Expr()=default;
    Expr(const Expr&)=delete
    Expr& op=(const Expr&)=delete;
virtual Expr* clone() const=0;
virtual bool eval(BoolInterpretation d) const=0;
virtual void getVariableInExpr(BoolInterpretation d)const
                                                const=0;

virtual ~Expr()=default;
```

Negotiate.h

```
#pragma once
#include "Expr.h"
#include "UnaryExpr.h"
class Negotiate:public UnaryExpr
{
public:
    Negotiate(Expr*);
    Expr* clone() const override;
    bool eval(const BoolInterpretation&) const override;
};
Negotiate::Negotiate(Expr* expr):UnaryExpr(expr){}
Expr* Negotiate::clone() const
{
    return new Negotiate(_expr->clone());
}
bool Negotiate::eval(const BoolInterpretation& Bi) const
{
    return !(_expr->eval(Bi));
}
```

BinaryExpr.h

```
#pragma once
#include "Expr.h"

class BinaryExpr : public Expr
{
protected:
    Expr* _left = nullptr;
    Expr* _right = nullptr;

public:
    BinaryExpr(Expr*, Expr*)
    ~BinaryExpr();
    void getVariableInExpr(BoolInterpretation & Bi) const override;
};

BinaryExpr::BinaryExpr(Expr* left, Expr* right):
        _left(left), _right(right) {}

BinaryExpr::~BinaryExpr()
{ delete _left;
  delete _right;
    _left = _right = nullptr;
}
void BinaryExpr::getVariableInExpr(BoolInterpretation & Bi) const
{ _left -> getVariableInExpr(Bi);
  _right -> getVariableInExpr(Bi);
```

Implies.h

```
#pragma once
#include "BinaryExpr.h"
class Implies:public BinaryExpr
    public:
    Implies(Expr*,Expr*);
    bool eval(const BoolInterpretation& i)const override;
    Expr* clone() const override;
};
Implies::Implies(Expr* l,Expr* r):BinaryExpr(l,r){}
bool Implies::eval(const BoolInterpretation& i)const
{
    return !left->eval(i)||_right->eval(i);
}
Expr* Implies::clone() const
{
    return new Implies(_left->clone(),_right->clone());
}
```

ExprFactory.h

```
#pragma once
#include "Expr.h"
#include "StringView.h"
#include "Variable.h"
#include "Negotiate.h"
#include "Conjunction.h"
#include "Disjunction.h"
#include "Implies.h"
class ExprFactory{
    public:
        static Expr* create(StringView str);
};
Expr* ExprFactory::create(StringView str)
{
StringView view= str.substr(1, str.GetSize()-2);
if(view.getSize()==1)
    return new Variable(view[0]);
size_t BracketCount=0;
for(size_t i=0; i< view.GetSize(); i++)
{ if(view[i]=='(')
    {BracketCount++;}
  else if(view[i]==')')
    {BracketCount--;}
  else if(BracketCount==0)
{
```

```
if(view[i]=='&')
    {return new Conjunction(create(view.substr(0,i)), create(
                                                        -i
    view.substr(i+1,view.GetSize()-1)));

else if(view[i]=='|')
{return new Disjunction(create(view.substr(0,i)),create(view.substr(i+1,view.GetSi
                                                        -i-1)));
else if(view=='!')
{return new Negotiate(create(view.substr(i+1, str.l  view.GetSize()-i-1));

else if(view=='>')
{return new Implies(create(view.substr(0,i)),create(view.substr(i+1,view.GetSize()-i
                                                        -1)));
    }
  }
}
throw std::invalid_argument("Invalid expression!");
}
```

Disjunction.h

```
#pragma once
#include "BinaryExpr.h"

class Disjunction:public BinaryExpr
{
  public:
   Disjunction(Expr*,Expr*);
   bool eval(const BoolInterpretation& i)const override;
   Expr* clone() const override;
};

Disjunction::Disjunction(Expr* l, Expr* r):BinaryExpr(l,r)
{}

Expr* Disjunction::clone() const
{return new Disjunction(_left->clone(),_right->clone());
}

bool Disjunction::eval(const BoolInterpretation& i)const
{return
  _left->eval(i)||_right->eval(i);
}
```

Conjuction.h

```
#pragma once
#include "BinaryExpr.h"

class Conjuction : public BinaryExpr
{
  public:
    Conjuction(Expr*, Expr*);
    bool eval(const BoolInterpretation& bi) const override;
    Expr* clone() const override;
};

Conjuction::Conjuction(Expr* left, Expr* right) : BinaryExpr(left, right)
{}

Expr* Conjuction::clone() const
{ return new Conjuction(_left->clone(), _right->clone());
}
bool Conjuction::eval(const BoolInterpretation& bi) const
{ _left->eval(bi) && _right->eval(bi);
}
```

Variable.h

```
#pragma once
#include "Expr.h"

class Variable: public Expr {
public:
    Variable(char ch);

    Expr* clone() const override;
    bool eval(const BoolInterpretation &) const override;
    void getVariableInExpr(BoolInterpretation &) const override;
private:
  char _ch;
};

Variable::Variable(char ch) :_ch(ch) {}
Expr*
Var::clone() const
{
 return new Variable(ch);
}

Var::bool Var::eval(const BoolInterpretation & bi, ) const
{
  return bi.getCharAt(_ch);
}
void Var::getVariableInExpr(BoolInterpretation & bi) const
{bi.setCharAt(_ch, true);
}
```

UnaryExpr.h

```
#pragma once
#include "Expr.h"

class UnaryExpr: public Expr
{
  protected:
     Expr* _expr=nullptr;

  public:
     UnaryExpr(Expr* expr);
     ~UnaryExpr();
     void getVariableInExpr(BodyInterpretation& bi) const override;

};
UnaryExpr::UnaryExpr(Expr* expr)
{ _expr=expr;
}
UnaryExpr::~UnaryExpr()
{ delete _expr;
  _expr=nullptr;
}
void UnaryExpr::getVariableInExpr(BodyInterpretation& bi) const
{ _expr->getVariableExpr(bi);
}
```

BooleanExprHandler.h

```
#pragma once
#include "MyString.h"
#include "BoolInterpretation.h"
class BooleanExprHandler
{
private:
    BoolInterpretation myVariables;
    Expr* expr=nullptr;

    void copyFrom(const BooleanExprHandler& other);
    void moveFrom(BooleanExprHandler&& other);
    void free();
    bool checkAllTruthAssignments(bool value) const;

public:
    BoolExprHandler(const MyString& str);
    BoolExprHandler(const BoolExprHandler& other);
    BoolExprHandler(BoolExprHandler&& other) noexcept;
    BoolExprH& op=(const BoolExprH& other);
    BoolExprH& op=(BoolExprH&& other) noexcept;

    ~BoolExprH();

    bool eval(const BoolInterpretation& bi) const;
    bool isTautology() const;
    bool isContradiction() const;
```

BooleanExprHandler.cpp

```
#include "BooleanExprHandler.h"

void BooleanExprH::copyFrom(const BoolExprH& other)
{ myVariables = other.myVariables;
  expr = other.expr->clone();
}
void BooleanExprH::moveFrom(BooleanExprH&& other
{ myVariables = std::move(other.myVariables);
  expr = other.expr;
other.expr = nullptr;
}
void BooleanExprH::free()
{
  delete expr;
  expr = nullptr;
}
bool BooleanExprH::checkTruthAssignments(bool value)con
{
  size_t varsCount = myVariables.trueVarCount();
  size_t powerOfTwo = 1 << varsCount;
  for(int i = 0; i < powerOfTwo; i++){
      BooleanInterpretation current = myVariables;
      current.excludeValuesByMask(i);
      if(expr->eval(current) != value)
          return false;
  }
  return true;
}
```

```
BoolExpr::BoolExpr(const MyString & str)
{
  expr = expressionFactory(str);
  myVariables.setAllTrue(); expr->populateVariables(&);
}
BoolExpr::BoolExpr(const BoolExpr& other)
{
  copyFrom(other);
}
BoolExpr::BoolExpr(BE&& other) noexcept
{
  moveFrom(std::move(other));
}
BoolExpr& BoolExpr::operator=(BE&& other)
{ if(this!=&other)
  { free();
    copyFrom(other);
  }
  return *this;
}
BoolExpr& BoolExpr::op=(BE&& other) noexcept
{
  if(this!=&other)
  { free()
    moveFrom(std::move(other));
  }return *this;
}
~BE(){free();}
```

```
bool BEH::eval(const BoolInterpret &Bi) BooleanExprHandle
{                                       №2
  return expr->eval(Bi);
}
bool BEH::isTautology() const
{
  return checkTruthAssignments(true);
}
bool BEH::isContradiction() const
{
  return checkTruthAssignments(false);
}
```