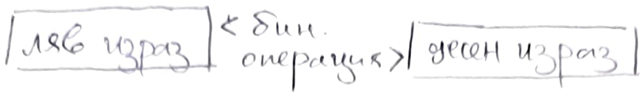
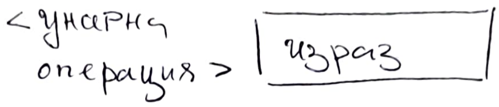
~~Тема~~ Тема 15

1. Какво е булев/логически израз?

Всеки логически израз е - променлива
- бинарна операция с ляв и десен логически израз



- унарна операция с друг логически израз

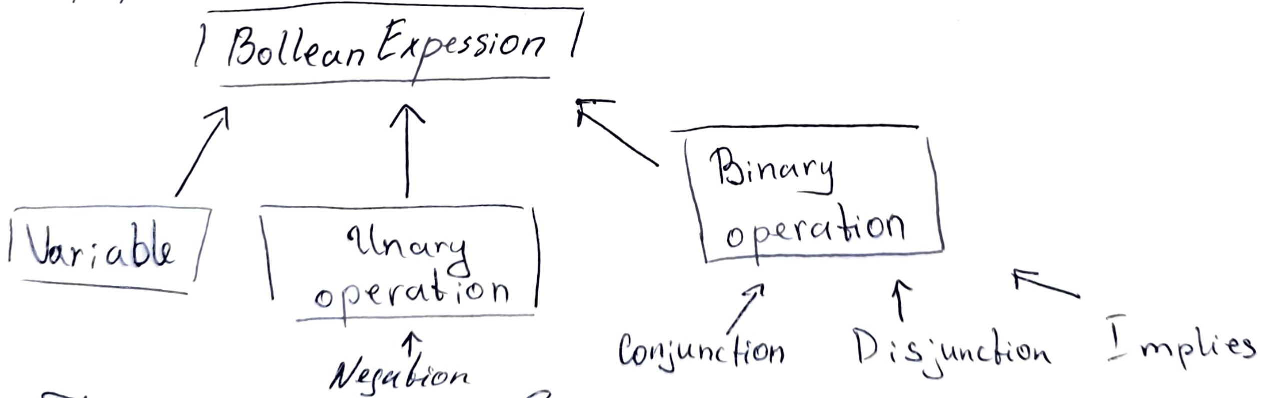


2. Как се оценява един израз?

→ булева интерпретация - оценка на всяка участваща променлива

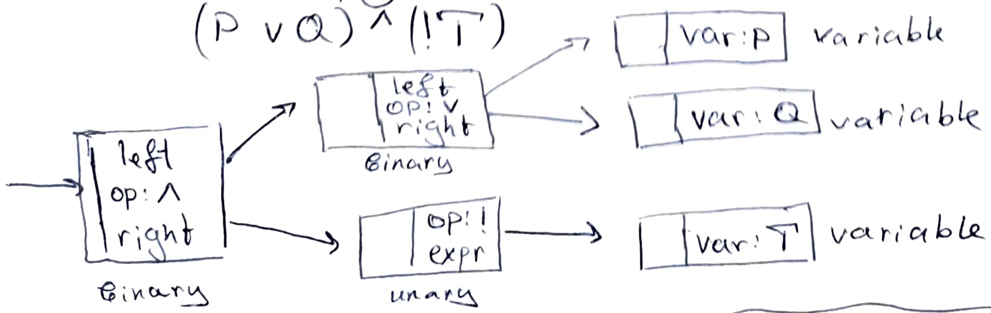
f: variable → {true, false}

Йерархия:



Пример за представяне на израз:

(P v Q) ^ (!T)



Оценката на изразите се изпълнява чрез чисто виртуалната ф-я evaluate( ), предефинирана от трите наследника по следните начини:
- пром. връща оценката от интерпретацията
- ун. оп оценява своя подизраз и връща резултат спрямо операцията си

бин. оп ~~д~~ оценява подизразите си и връща резултат спрямо операцията си

Тавтология - винаги верен израз, независимо от булевата интерпретация

Противоречие - винаги лъжа, =//=

За да можем да направим проверка за тавтология/противоречие, ще генерираме всички булеви интерпретации на израза. За целта при построяването на всеки ~~изр~~ израз ще пазим кои променливи участват. Това става по следните начини във всеки от трите вида изрази:
- в пром. участва само една - самата тя
- в ун. оп. участват променливите от подизраза
- в бин. оп. участват променливите от левия и десния подизрази

k променливи дават 2^k булеви интерпретации.

3. Как създаваме булев израз по стринг?

Ще четем изрази със скоби около всяка операция

Пример: $(((P \vee Q) \wedge (T \vee R)) \vee (!P))$

Определяме реда на изпълнение по следния начин:

1.) премахваме първата и последната скоба

2.) Обхождаме стринга като ~~броим~~ използваме брояч. При срещане на отваряща скоба брояч++, при затваряща брояч --.

3). Символът за операция, който срещнем при брояч със ст-т 0, е операцията, която трябва да се приложи първа.

Извикваме рекурсивно ~~ф-ята~~ за аргументите на операцията.

eanExpressionHandler.h

#include Expressionfactory.h
MyString.h
BooleanInterpretation.h
BooleanExpression.h

```
class BooleanExpressionHandler{
public:

 //Big6
 BEH(const MyString& str)

 bool evaluate(const BooleanInterpretation&bi) const;
 bool isTautology() const;
 bool isContradiction const()

private:
  bool checkAllTruthAssignments(bool value) const;
  void free(), copyFrom() moveFrom()
  BooleanInterpretation myVariables;
  BooleanExpression* expr = nullptr;
};
BEH.cpp
 BEH::BEH(const MyString & str){
  expr = expressionFactory(str);}
bool BEH::evaluate(const BooleanInterpretation & bi) const {
   return expr→eval(bi);}
bool BEH::isTautology() const {
   return checkAllThruthAssignments(true);}
bool BEH::isContradition() const{
   return checkAllThruthAssignments(false);}
bool BEH::check...(bool value) const {
  size_t varsCount = myVariables.getTrueCount();
  size_t powerOfTwo = 1 << varsCount;
```

2

```
for ( int i=0; i< powerOfTwo; i++) {
 BooleanInterpretation current = myVariables;
 current.excludeValuesByMask(i);
if(expr->eval(current) != value
      return false;
}
return true;
}

                                        BEH::
void  BEH:: free(){ {           void  copyFrom (const BEH& other){
    delete expr;}                     expr = other.expr -> clone(); }

void  BEH:: moveFrom(){
   expr = other.expr;
   other.expr = nullptr; }
```

```
Boolean Interpretation.h
 #include "MyString.h"
 #include "StringView.h"

 class BooleanInterpretation {
 public:
      void  ' set (char ch, bool value);
      bool operator()(char ch) const;
      size_t getTrueCount() const;
      void excludValuesByMask(unsigned mask) ;
  private:
     bool values [26]{false};
 };
```

cpp
void BI::
val

```cpp
cpp constexpt size_t availableVariables = 26; // трябва за .h
void BI::set(char ch, bool value) {
  values[ch-'a'] = value; }
bool BI::operator()(char ch) const {
   return values[ch - 'a'];
 size_t BI::getTrueCount() const {
   size_t count = 0;
  for(int i = 0; i < ~~26~~ availableVar; i++) {
        if(values[i])
            count++; }
   return count;
void BI::excludValuesByMask(unsigned mask) {
    for(ind i = 25; i >= 0; i--) {
       if(values[i]) {
          if(mask % 2 == 0)
              values[i] = false;
          mask /= 2;
       }
   }
   }
```

```cpp
BooleanExpression.h
 #include "BI".h
 struct BooleanExpression {
    BooleanExpression() = default;
   BooleanExpression(const BE& other) = delete;
   BE& op=(const BE& other) = delete;

   virtual ~BE() = default;
   virtual bool eval(const BI& interpret) const = 0;
   virtual BE* clone() const = 0;
   virtual void populateVariables(BI& interpret) const = 0; };
```

3

```
~~sd~~ Var.cpp // требва да се раздели на .h и .cpp
struct Var : BE
   Var(char ch) : ch(ch {}
  bool eval(const BI & interpret )const override {
       return interpret(ch) ;}
  ~~virtual~~ BE* clone() const override {
      return new Var (ch); }
  void populateVariables (BooleanI & interpret ) const override
  { interpret.set(ch, true); }
 private:
   char ch; };
```

```
UnaryOperation.cpp
  struct UnaryOperation : BE {
   UnaryOperation (BE* expr ) : expr(expr){}
  void populateVariables ( BI & interpret ) const override {
   expr-> populateVariables (interpret); }

  ~ UnaryOperation() {
         delete expr ; }

   protected:
       BE* expr; };
```

```
Negation.cpp
   struct Negation:UnaryOperation {
 Negation(BooleanE * expr):UO(expr){}
 BE* clone() const override {
  return new Negation (~~[illegible]~~ expr -> clone()); }
 bool eval (constBI & interpret ) const override {
 return ! expr -> eval(interpret); }
```

```
naryOperation.cpp
#include BooleanExpression.h

struct BinaryOperation : BooleanExpression {
  BO(BE *left , BE* right) : left(left), right(right) {}

void populateVariables (PsI & interpret ) const override {
    left -> populate Variables (interpret);
    right -> populateVariables (interpret); }

~BinaryOperation() {
 delete left;
 delete right; }
protected :
     BooleanExpression *left;
     BE* right; };
```

```
                Conj.cpp
Conjunction: BinaryOp {
  Conjunction (Boolean Expression *left, BE* right): BO(left, right)
        {}

  virtual BE* clone() const override {
      return new Conjunction(left -> clone(), right->clone());
  }

  bool eval(const PSI interpret) const override {
      return left ->eval(interpret) & right -> eval( interpret);
  }
 };
```

Disjunction.cpp

```
struct Disjunction : BinaryOperation {
    Disjunction (BooleanExpression *left, BE right):
                BO( left, right) {}

    virtual BE* clone() const override {
  return new Disjunction(left->clon() ,right->clone());
   bool eval(const BI & interpret ) const override {
    return left->eval( interpret) || right ->eval(interp
   }};
```

```
struct Implies : BinaryOp {
   Implies( BE* left ,BE* right ): BO(left,right) {}
  BE* clone () const override {
      return new Implies( left -> clone(), right -> clone());}
  bool eval (const BI & interpret) const override {
   return(!left->eval(interpret)) || right ->eval (interpret)
  }};
```

```
oolean Expression* expressionFactory (StringView str){
    ~~str = str.substr (1, str.lenght() = (1)~~
    ~~( return (new~~
    str = str.substr(1, str.lenght()-2);

      if (str.lenght()==1)
          return new Var (str[0]);

    unsighed count=0;
    ~~for(int i=0; i<str.lengs~~
    unsigned len= str.length();
   for (int i=0; i<len; i++){
      if(str[i]== '(')
          count++;
  else if(str[i]== ')')
          count--;

  else if (count==0){
     switch (str[i]) {

       case '!' : return new Negation(
       exprFact(str.substr(i+1, str.length()-i-1)));
       case '&': return new Conjunction(exprFact (str.substr(0,i)),
       exprFact(str. substr(i+1, str lenght()-i-1)));
       case '|': return new Disjunction(
       exprFact(str.substr(0,i)),
       expr Fact (str.substr (i+1, str.length()-i-1)));
       case '>': return new Implies(
       exprFact(str, substr(0,i)),
       exprFact(str.substr(i+1, str.lenght() -i-1)));
     }}}throw std::invalid_argument("InvalidExpression!"); }
```
5