Plán na letný semester:
naprogramovať rozhranie medzi datalogom a relačnou algebrou na výpočet rekurzívnych dotazov implementovanou v jave

Parser gramatika:

<program> : <fact> {’,’ <fact>} | <fact> {’,’<fact>} <rule> {’,’<rule>} ;
<fact> : <predicate> ’(’ [<literal> {’,’<literal>}] ’)’ ’.’ ;
<rule> : <hlava> ’:-’ <telo> ’.’ ;
<telo> : <atom> {’,’<atom>}
<hlava> : <predicate> ’(’ [<argument> {’,’<argument>}] ’)’
<atom> : <predicate> ’(’ [<argument>{’,’<argument>}] ’)’ | ’/+’ <atom>
<argument> : <literal> | <variable> ;
tokens = {***’(’ , ’)’***, ***’variable’***, ***’literal’***, ***’predicate’***, ***’/+’***, ***’:-’***, ***’,’ , ’.’***, ***’eof’***, ***’unknown’***}
// pre nularne predikaty : <predicate> ’(’ ’)’

Hotová je syntaktická analýza ktorá kontroluje či datalogovský program patrí do predchádzajúcej gramatiky, ak nepatrí vráti chybovú hlášku s pozíciou, kde je chyba.
Ďalej je spravené parsovanie a ukladanie faktov ktoré sú ukladané ako Relation implementované v dataStructure.

V relačnej algebre sú implementované operátory Join, AntiJoin a Union, selekcia a projekcia sa robí pomocou TupleTransformation ktoré sa posiela operátorom v konštruktore.
Preklad pravidiel by mal fungovať nasledovne:

- uložia sa zvlášť pozitívne a negatívne podciele
- spravia sa Joiny pozitívnych podcieľov so selekciami a projekciami v TupleTransform
- spravia sa AntiJoiny negatívnych podcieľov so selekciami a projekciami v TupleTransform

Ďalej uvádzam nejaké ručne prekladané Datalogovské programy ktoré boli zároveň použité na testovanie top-down algoritmu na výpočet rekurzívnych dotazov:
1)

```
/*move(1,2)
move(2,1)
move(2,3)
move(3,4)
move(4,5)
move(5,6)
win(X) :- move(X,Y), \+win(Y)*/

win(X) = {5, 3}
/*==============================*/
String[][] m = {{"1","2"},{"2","1"},{"2","3"},{"3","4"},{"4","5"},{"5","6"}} ;
Relation move = new Relation(m);
RelationOperator r = move.operator();
RecursiveOperator win = new AntiJoin(r, this, (t1, t2) -> {
        if (t1.get(1) == t2.get(0)  {
            return null;
        }
        Tuple t3 = new Tuple();
        t3.add(t1.get(0));
        return t3;
    }
);
```

2)

```
/*R = {(0)}
P(x) :- R(x) , \+Q(x)
Q(x) :- R(x) , \+ P(x)


P={(0)}, Q = Φ
P = Φ, Q = {(0)}
*/
String[][] R1 = {{"0"}};
Relation R = new Relation(r1);

Operator r = R.operator();

RecursiveOperator P = new AntiJoin(r, Q, (t1,t2)-> {
                        if(t1.get(0) == t2.get(0)) return null;
                        return t1;
                     });
RecursiveOperator Q = new AntiJoin(r, P, (t1,t2)->{
                        if(t1.get(0) == t2.get(0)) return null;
                        return t1;
                     });
```

3)

```
/*
S = {(1)}
R(x) :- S(x) , \+ R(x)
*/

/*============================*/

Relation s = new Relation({{"1"}});
RelationOperator sOperator = s.operator();
RecursiveOperator r = new AntiJoin(sOperator, this, (t1,t2)->{
                                      if(t1.get(0)==t2.get(0)) return null;
                                      return t1;
                                   });
```

4)

```
/*meal(m1, white).
meal(m2, red).
meal(m3, white).
meal(m3, red).

vegetarian(m3).
vegetarian(X) :- meal(X, Y),  \+nonvegetarian(X), \+pork(X), \+beef(X).

nonvegetarian(X) :- meal(X, Y), beef(X), \+vegetarian(X).

pork(X) :- meal(X, Y), nonvegetarian(X),\+ beef(X).
pork(X) :- meal(X, white), nonvegetarian(X), \+ fish(X).

beef(X) :- meal(X, red), \+ vegetarian(X), \+ pork(X).

fish(X) :- meal(X, white), \+ pork(X), \+ beef(X).
*/

/*=====================================================*/
//meal(m1, white).
//meal(m2, red).
//meal(m3, white).
//meal(m3, red).
String[][] meal = {{"m1", "white"}, {"m2", "red"}, {"m3", "white"}, {"m3", "red"}};

//vegetarian(m3).
//vegetarian(X) :- meal(X, Y),  \+nonvegetarian(X), \+pork(X), \+beef(X).

String[][] v1 = {{"m3"}};
RecursiveOperator vegetarian = new Union(
                v1, new AntiJoin(
                        meal, new Union(
                               nonvegetarian, new Union(
                                           pork, beef ) ),
                        (t1,t2) -> {
                              if (t1.get(0) == t2.get(0)) return null;
                              Tuple t3 = new Tuple();
                              t3.add(t1.get(0));
                              return t3
                           }
                        )
                   );
//nonvegetarian :- meal(X, Y), beef(X), \+vegetarian(X).

RecursiveOperator nonvegetarian = new AntiJoin(new Join(meal, beef, (t1.t2) -> {
                                                    if (t1.get(0) != t2.get(0)) return null;
                                                    return t2;
                                                 }
                                              ),
                                       vegetarian, (t1,t2)->{
                                                    if (t1.get(0) == t2.get(0)) return null;
                                                    return t1;
                                                 }
                                       );

//pork(X) :- meal(X, Y), nonvegetarian(X),\+ beef(X).
//pork(X) :- meal(X, white), nonvegetarian(X), \+ fish(X).
RecursiveOperator pork = Union(new AntiJoin(new Join(meal, nonvegetarian, (t1,t2)->{
                                                                     if (t1.get(0) != t2.get(0)) return null;
                                                                     return t2;
                                                                  }),
                                    beef, (t1,t2) -> {
                                           if (t1.get(0) == t2.get(0)) return null;
```

```
                                     return t1;
                                   }),
                     new AntiJoin(new Join(meal, nonvegetarian, (t1,t2) -> {
                                                      if (t1.get(0) != t2.get(0)) return null;
                                                      if (t1.get(1) != "white") return null;
                                                      return t2;
                                                    }),
                              fish, (t1,t2) -> {
                                         if (t1.get(0) == t2.get(0)) return null;
                                         return t1;
                                      }
                     )
                  );

//beef(X) :- meal(X, red), \+ vegetarian(X), \+ pork(X).
RecursiveOperator beef = new AntiJoin(meal, new Union(vegetarian, pork), (t1,t2) -> {
                                     if (t1.get(0) == t2.get(0)) return null;
                                     if (t1.get(1) != "red") return null;
                                     Tuple t3 = new Tuple();
                                     t3.add(t1.get(0));
                                     return t3;
                                 }
                   );

//fish(X) :- meal(X, white), \+ pork(X), \+ beef(X).
RecursiveOperator fish = new AntiJoin(meal, new Union(pork, beef), (t1,t2) -> {
                                 if (t1.get(0) == t2.get(0)) return null;
                                 if (t1.get(1) != "white") return null;
                                 Tuple t3 = new Tuple();
                                 t3.add(t1.get(0));
                                 return t3;
                              }
                     );
```