MANIPULATION OF EXPRESSIONS IN A RELATIONAL ALGEBRA

J. W. M. STROET† and R. ENGMANN

Twente University of Technology, Dept. of Applied Mathematics, P.O. Box 217, 7500 AE Enschede, The Netherlands

(Received 6 October 1978; revised 23 March 1979)

Abstract—This paper describes a syntax for expressions based on the relational algebra. A tree representation is generated when an expression is analyzed. Transformations on the tree representations of expressions are applied in order to obtain improvements with respect to the speed of evaluation in a data base environment.

1. INTRODUCTION.

The present work is part of the investigations on the implementation of a small experimental data base system, based on the relational model. The data base system contains an interface which accepts expressions based on the relational algebra. A syntax is presented which generates such expressions. Syntactic analysis and code generation produce a tree structure, which represents the relational expression.

The tree representation is analyzed for transformations which might decrease the effort needed for evaluation of the expression, e.g. by combination of operations, reordering of operations and recognition of equivalent subexpressions. The main objective of these transformations is the minimization of the numbers of tuple retrieval operations and expected disk accesses.

2. RELATIONAL EXPRESSIONS.

The discussed data base system should be capable to accept queries in the form of expressions in the relational algebra[1]. It is supposed, that the relations to be operated upon are in first normal form[2], such that the domains are not relations themselves but sets of unstructured values. A relational expression may be conceived as a definition of a new relation, which may be evaluated from existing relations by application of a set of operators defined in the relational algebra. In the sequel these operators will be indicated as relational operators. A syntax for relational expressions is described in the appendix.

The relational operators which are incorporated in the relational expressions are:

—The traditional set operators: union (+), intersection (.) and set difference (−), which are defined for two operands, being union-compatible relations[1].

—The join (*) which is conceived as a full quadratic join or cartesian product of two relations. The join of two relations R and S with degrees m and n , respectively, yields a relation with degree $m + n$. The join contains tuples which are formed by concatenating tuples from R with tuples from S .

—The division (/): if R and S are relations, α and β are complementary domain lists of R , γ and δ are complementary domain lists of S , then the division is defined as:

$$R[\beta/\gamma]S = \{r \in R[\alpha] \mid \forall s \in S[\gamma] [(r, s) \in R]\}.$$

The result of the division is a relation with a degree equal to the number of domains in α .

—The projection (%) selects from a relation specific domains mentioned in the domain list.

—The selection (:) selects from a relation those tuples which meet the selection condition.

The union, intersection, difference, join and division are dyadic operators; the projection and selection are monadic operators.

Though users may introduce arbitrarily chosen names for relations and domains we will assume that such names will be coded such that relation names are denoted as " R_m ", with m being an integer which could be an index in a table with user relation names. Domain names are denoted as " D_j ", such that j denotes the j th domain of the relation which is described by the preceding relational expression, e.g. $R_5 \% D_3$ denotes the projection of relation R_5 on its third domain.

When a relational expression is evaluated the result is conceived as a relation with one single continuous list of domains; e.g. a join of two relations R with degree m and S with degree n yields a relation $R * S$ with degree $m + n$, and the domains are referred to with D_j ($1 \leq j \leq m + n$).

Relational expressions are evaluated according to the following rules:

Parenthesized expressions are evaluated first.

Expressions are evaluated from left to right.

All relational operators have equal priority.

Some examples of relational expressions are:

Example 1:

$$R_5 : D_1 = D_3 \wedge \neg D_8 = 20 \% D_2, D_3 * R_4$$

This expression may be described as follows: select from relation R_5 those tuples where domain value D_1 equals

†Present address: Nymegen University.

domain value $D3$ and domain value $D8$ is not equal to 20. Project these tuples on domains $D2$ and $D3$; perform a join operation on the resulting relation and relation $R4$.

Example 2:

$$R5 \% D3, D1 * R4: D1 = D5 \% D2, D6$$

The interpretation of this expression is as follows: relation $R5$ is projected on the third and first domains; the result is joined with relation $R4$. From this result the tuples are selected such that the first and the fifth domain values are equal (this conforms to an equi-join[1]); project the result on the second and sixth domains.

Example 3:

$$R5 \% D3, D1 * (R4: D1 = D5) \% D2, D6$$

This expression is different from the previous one as the subexpression $R4: D1 = D5$ is evaluated before it is joined with $R5 \% D3, D1$.

Expressions in the relational algebra are relation oriented, i.e. the relational algebra doesn't contain operators which deal with individual tuples within a relation. The evaluation of the expressions is implemented by a tuple-oriented system, which can identify, read and store individual tuples of relations and may use access paths which are hidden for higher level (more abstract) user interfaces. The lower level tuple-oriented interface will not be discussed in this paper.

3. SYNTACTICAL ANALYSIS OF RELATIONAL EXPRESSIONS.

The acceptor which analyses relational expressions consists of a set of syntactic procedures. For each non terminal in the syntax a procedure has been written which accepts a derivation of the non terminal, provided that it is syntactically correct.

For an expression which is accepted a tree representation is constructed. Conceptually the relational and Boolean operators of the expression are represented in the expression tree by non terminal nodes. Stored relations (e.g. $R5$) and simple Boolean predicates (e.g. $D1 = D5$) are represented by terminal nodes.

The system has been implemented in SIMULA-67. The language SIMULA-67, though originally designed for simulation purposes is a very general programming language[3]. SIMULA-67 is a derivative of ALGOL-60 and includes facilities for input, output and file processing.

The main feature of SIMULA-67 is the class concept which provides the user with extensive facilities for defining abstract data types. The data structuring capabilities include references or pointers, record structures and standard list processing facilities. The class concept also appeared to be a powerful tool for structuring software systems. Large software systems as e.g. data base management systems may be designed and implemented as leveled structures or classes, each class containing its own procedures, data structures and parameters.

The data structures representing expressions are described using the class construct; e.g. the nodes of

expression trees are described in a dedicated class. Each node contains components which refer to the father node, the leftson node and the rightson node. When a node has only one subtree (e.g. $\neg$) then it is taken as the right subtree. For a selection operator the selection condition is represented by the left subtree of the node representing the selection operator ($:$); likewise, for a projection operator the domain list is represented by the left subtree of the node representing the projection operator ($\%$).

Figure 1 shows the expression:

$$R2 + R5: D1 = D4 \vee D5 = 27$$

The arcs emanating from a node represent references to its left and right subtree; the references to father nodes are not shown in Fig. 1.

Fig. 1.

The expression tree may contain nodes which represent " $\neg$ " operators; these nodes are removed from the tree by application of De Morgan's laws recursively on trees containing " $\neg$ " nodes. The removal of " $\neg$ " nodes simplifies the algorithms to be described in the sequel.

4. DISPLACEMENT OF SELECTION OPERATIONS.

The evaluation of expressions may be optimized by changing the order of operations and by combination of operations. Such optimizations are represented by transformations of the expression tree such that the transformed tree represents an equivalent expression but which may be evaluated more efficiently in terms of numbers of tuple retrieval operations and expected disk accesses. Presently, as the knowledge about costs of evaluation of expressions is very limited only global, heuristic reasoning can be applied for improving the efficiency of evaluation.

One of the main improvements of expression trees is the propagation of nodes which represent selections towards the terminal nodes of the expression tree. The effect is that during evaluation of the expression selection operations are performed as early as possible; this has the following advantages:

(1) In many cases selections can be moved into a position such that they operate on terminal nodes which represent stored relations. In this way available index tables can be used for evaluation of the selection operation.

(2) The early evaluation of selections may reduce considerably the cardinality of the operands of operator nodes higher up in the tree. The reduction depends heavily upon the selectivity of the selection condition.

The propagation of a selection node over nodes representing a union, intersection, difference or projection has been discussed in [4] and [5]. The propagation of a selection node over a join node is considerably more complicated. A theoretical treatment of this problem has been given in [6]; this section describes the algorithm which was used to implement the propagation of a selection node over a join node. Finally the propagation of a selection node over a division node will be discussed.

We consider an expression consisting of a selection condition C on a join of two relational expressions L and R :

$$L * R : C$$

L has m domains and R has n domains; the join $L * R$ then has $m + n$ domains $D1, \dots, Dm, D[m+1], \dots, D[m+n]$, which is the concatenation of the lists of domains of L and R . The selection condition C is a Boolean expression containing domain names Di of $L * R$. ($Di \in \{D1, \dots, D[m+n]\}$)

The selection condition C may be factorized as follows:

$$C = C_L \wedge C_R \wedge C_B,$$

such that C_L only contains the domain names $D1 \dots Dm$ and C_R only contains the domain names $D[m+1] \dots D[m+n]$. It is supposed that the factorization of C is maximal such that C_B has no Boolean factors which only contain domain names of either L or R . The expression $L * R : C$ then can be transformed into:

$$L * R : C = L * R : (C_L \wedge C_R \wedge C_B) \\ = (L : C_L) * (R : C_R) : C_B$$

in which condition components (C_L , C'_R and C_B) may be empty, i.e. identically true. C'_R is derived from C_R such

that a domain name Di in C_R , which is a condition on $L * R$, agrees with a domain name $D[i-m]$ in C'_R , which is a condition on R only. The transformation is shown in Fig. 2.

The partial selection conditions C_L and C'_R may be shifted downwards in the left and right subtrees, respectively.

In the sequel an algorithm is described which decomposes a selection condition C into two partial conditions success and residue, such that

$$C = \text{success} \wedge \text{residue}.$$

The partial condition "success" only contains predicates which refer to one (either the left or the right) subtree of the join node. The partial condition "residue" doesn't contain partial conditions to that same subtree which can be split off by decomposition. The transformation is illustrated in Fig. 3.

The algorithm may be used to split off a partial condition "success" for L , yielding C_L , next split off a partial condition "success" for R from the residue of the previous split, yielding C_R . The residue of the second decomposition will yield C_B . As a result the original condition C is decomposed into three components C_L , C_R and C_B .

The algorithm which decomposes a selection condition is described in the sequel. The parameter "node" is a reference to the root of a tree representing the initial selection condition, "branch" specifies the operand (left or right) of the join for which a partial condition has to be split off, "success" will refer to this partial condition and "residue" to the residual partial condition; the output parameters "success" and "residue" are called by name. The local parameters successleft, successright, residueleft and residueright are references to nodes representing a partial selection condition.

```

Algorithm:
decompose (node, branch, success, residue);
begin if node represents a simple predicate
  then if node refers to the branch operand only
    then success := node
    else residue := node
  else
    begin decompose (node, leftson, branch, successleft, residueleft);
      decompose (node, rightson, branch, successright, residueright);
      if node represents " ^ "
        then begin success := successleft ^ successright;
              residue := residueleft ^ residueright
            end
        else {node represents " v "}
          begin success := successleft ^ successright;
              residue := (successleft ^ residueright)
                        ^ (residueleft ^ successright)
                        ^ (residueleft ^ residueright)
          end
    end
  end
end

```

Fig. 2.

Fig. 3.

In the algorithm subtrees which represent partial conditions may appear to be empty which means that the corresponding partial condition is identically true. The tree may be simplified in such cases according to:

$$\begin{aligned} \text{condition} \wedge \text{true} &= \text{condition} \\ \text{condition} \vee \text{true} &= \text{true}. \end{aligned}$$

The algorithm decomposes the tree representation of a condition into two trees, denoted by success and residue such that success denotes a tree which only contains predicates pertaining to the branch operand. The partial condition success is maximal, i.e. it is not possible to split off a condition from the residue such that it only pertains to the branch subtree. This is obvious for a tree with depth $d = 1$. By induction on d the contention can be proven for arbitrary d .

The propagation of a selection over a division node is specified by the following transformation:

$$R[\beta/\gamma]S: C_\alpha = R: C_\alpha[\beta/\gamma]S$$

where the symbols have the same meaning as in the definition of the division in Section 2, and C_α is a condition on the α domain list of R .

Proof:

$$\begin{aligned} X \in R[\beta/\gamma]S: C_\alpha &\Leftrightarrow \\ \forall s \in S[\gamma] \exists r \in R[\alpha] [(r, s) \in R \wedge x = r] \wedge x \text{ satisfies } C_\alpha &\Leftrightarrow \\ \forall s \in S[\gamma] \exists r \in R[\alpha] [(r, s) \in R \wedge x = r \wedge x \text{ satisfies } C_\alpha] &\Leftrightarrow \\ \forall s \in S[\gamma] \exists r \in R[\alpha] [(r, s) \in R \wedge x = r \wedge r \text{ satisfies } C_\alpha] &\Leftrightarrow \\ \forall s \in S[\gamma] \exists r \in (R[\alpha]: C_\alpha) [(r, s) \in (R: C_\alpha) \wedge x = r] &\Leftrightarrow \\ x \in R: C_\alpha[\beta/\gamma]S. & \end{aligned}$$

5. SIMPLIFICATION OF EXPRESSION TREES.

Simplification of expression trees may be obtained by three techniques:

(1) Reduction of relational expressions based on idempotency and zero equivalence.

(2) Reduction of Boolean expressions by recognition of idempotency and predicates which are identically true or false.

(3) Recognition of equivalent subexpressions.

Table 1 shows the reductions which are applied in the simplification process. ("R" stands for a relation, "p" stands for a selection predicate, " ϕ " is the empty relation.)

Reductions are possible when an operator is superfluous, e.g. when its left and right operands are equivalent subexpressions or one of the operands is the empty relation. Superfluous operators may be removed from the expression; in the sequel they will be denoted as dummy operators. Reductions of Boolean operators and relational operators will be combined in one process. As has been explained in [5] the reduction of operands and the recognition of equivalent subexpressions are techniques which should be applied simultaneously on relational expressions, because on each level of the expression tree equivalent subexpressions and dummy operators may be discovered.

An algorithm for the recognition of equivalent subexpressions has been described by Hall in Ref. [7]. The method is aimed at recognition of equivalent nodes in the expression tree and removal of duplicates from the tree.

Equivalence of two nodes P and Q in the expression tree is defined as follows:

(1) P and Q should represent either the same relation, the same relational operator or the same predicate, and
(2) If P and Q represent the same relational or Boolean operator Ω then P and Q are equivalent if:

$$\begin{aligned} (P. \text{leftson} = Q. \text{leftson} \wedge P. \text{rightson} = Q. \text{rightson}) \\ \vee \\ (P. \text{rightson} = Q. \text{leftson} \wedge P. \text{leftson} \\ = Q. \text{rightson} \wedge \Omega \text{ is commutative}) \end{aligned}$$

When nodes P and Q are equivalent, Q may be removed from the expression tree by changing all arcs entering Q to arcs entering P . In this way the tree representing the relational expression is converted into a lattice in which all nodes are unique with respect to equivalence. For an easy implementation of the reduction process a special node ROOT is added to the tree, such that the original tree is the (single) subtree of ROOT.

In the algorithm which recognizes equivalent nodes

and dummy operators the nodes are grouped into levels N_k . The levels are constructed as follows:

$$\begin{aligned} N_1 &= \{r: r \text{ is a terminal node of the tree}\} \\ N_k &= \{r: r \text{ is a node of the tree} \wedge \end{aligned}$$

$$r \notin \bigcup_{i=1}^{k-1} N_i \wedge$$

$$r. \text{leftson} \in \bigcup_{i=1}^{k-1} N_i \wedge$$

$$r. \text{rightson} \in \bigcup_{i=1}^{k-1} N_i\}.$$

For the construction of N_k it is necessary to check if the descendants of a node are contained in the levels N_1 through N_{k-1} . Practically this can be done by adding a special attribute LEVEL to the nodes, such that $r. \text{LEVEL} = k$ if $r \in N_k$. Levels N_k have been implemented as lists of references to nodes of the expression tree. The search for the nodes of N_k can be limited to the father nodes of the nodes in N_{k-1} , because N_k may be defined also as:

$$N_k = \{r: q \in N_{k-1} \wedge q. \text{father} = r \wedge$$

$$r. \text{leftson} \in \bigcup_{i=1}^{k-1} N_i \wedge$$

$$r. \text{rightson} \in \bigcup_{i=1}^{k-1} N_i\}.$$

Hall's algorithm[7] doesn't reduce the expression tree in case of dummy operations. An algorithm which at the same time recognizes equivalent subexpressions and reduces expressions in case of dummy operations is described in the sequel.

Algorithm:

T = expression tree
 k = number of current level
 q, r denote nodes of T
 N_k = set of nodes on level k of T

- (1) $N_1 := \{r: r \text{ is a terminal node of the tree}\};$
- (2) remove nodes representing dummy predicates from T and N_1 ;
- (3) $k := 1$;
- (4) while $\text{ROOT} \notin N_k$ do
- (5) begin $N_{k+1} := \{r: q \in N_k \wedge q. \text{father} = r \wedge$

$$r. \text{leftson} \in \bigcup_{i=1}^k N_i \wedge$$

$$r. \text{rightson} \in \bigcup_{i=1}^k N_i\};$$

- (6) remove duplicate nodes from N_k and T ;
- (7) remove nodes representing dummy operators from N_{k+1} and T ;
- (8) $k := k + 1$
- end

Explanation:

Step 2:

If N_1 contains nodes which represent predicates which are identically true or false, then these nodes are removed from N_1 and T . If resulting dummy operator

Table 1.

expression:	equivalent with:
$R + R$	R
$R \cdot R$	R
$R - R$	$\emptyset$
$R : p - R$	$\emptyset$
$R - (R : p)$	$R : \neg p$
$(R : p) + R = R + (R : p)$	R
$(R : p) \cdot R = R \cdot (R : p)$	$R : p$
$(R : p1) \cdot (R : p2) = (R : p2) \cdot (R : p1)$	$R : p1 \wedge p2$
$(R : p1) + (R : p2) = (R : p2) + (R : p1)$	$R : p1 \vee p2$
$R + \emptyset = \emptyset + R$	R
$R \cdot \emptyset = \emptyset \cdot R$	$\emptyset$
$R - \emptyset$	R
$\emptyset - R$	$\emptyset$
$\emptyset \neq \text{domainlist}$	$\emptyset$
$\emptyset : p$	$\emptyset$
$R : \text{true}$	R
$R : \text{false}$	$\emptyset$
$R * \emptyset = \emptyset * R$	$\emptyset$
$A[\beta/\gamma]\emptyset$	$A \neq \alpha$
$\emptyset[\beta/\gamma]A$	$\emptyset$
$p \wedge \text{true} = \text{true} \wedge p$	p
$p \wedge \text{false} = \text{false} \wedge p$	false
$p \vee \text{true} = \text{true} \vee p$	true
$p \vee \text{false} = \text{false} \vee p$	p
$p \wedge p$	p
$p \vee p$	p
$D_i \geq D_i, D_i = D_i, D_i \leq D_i$	true
$D_i > D_i, D_i \neq D_i, D_i < D_i$	false

nodes are recognized they also will be removed from the expression tree. This implies searching the tree from the dummy predicate node following the father references towards the ROOT node until an ancestor node is found which doesn't represent a dummy operator.

Step 7:

Deletion of dummy operators in N_{k+1} is feasible because levels N_1 through N_k only contain unique subexpressions. If a node represents a dummy operator the following actions are taken:

The dummy operator node is removed from T and N_{k+1} . The dummy operator may introduce an empty operand, e.g. $R2 - R2$ yields the empty relation, which means that ancestors of the deleted node also have to be checked if they represent dummy operators. This implies searching the tree from the dummy operator node following the father references towards the ROOT node until an ancestor node is found which doesn't represent a dummy operator. All dummy operators found in this way are removed from the tree. Step 7 may be described formally as follows:

```

for all nodes  $q \in N_{k+1}$  do
  if  $q$  represents a dummy operator
  then
    begin  $r := q$ . father;
       $T := T - q$ ;
       $N_{k+1} := N_{k+1} - q$ ;
      while  $r$  represents a dummy operator do
        begin  $q := r$ ;
           $r := r$ . father;
           $T := T - q$ 
        end
    end
end

```

end

Now r denotes a non-dummy operator node which possibly should be added to one of the levels $N_1 \dots N_{k+1}$. The following situations may occur:

- (1) r . leftson $\in \bigcup_{i=1}^k N_i \wedge r$. rightson $\in \bigcup_{i=1}^k N_i \wedge$
 $(r$. leftson $\in N_k \vee r$. rightson $\in N_k)$

In this case r should be added to N_{k+1} .

- (2) If r . leftson $\notin \bigcup_{i=1}^k N_i \wedge r$. rightson $\notin \bigcup_{i=1}^k N_i$ then presently r cannot be added to any of the levels $N_1 \dots N_{k+1}$, but will be added in a later stage to some level N_j ($j > k+1$).

- (3) If r . leftson $\in \bigcup_{i=1}^{j-1} N_i \wedge r$. rightson $\in \bigcup_{i=1}^{j-1} N_i \wedge$
 $(r$. leftson $\in N_{j-1} \vee r$. rightson $\in N_{j-1})$ (with $j < k$) then r has to be added to N_j . This implies that also the ancestors of r possibly have to be added to $N_{j+1} \dots N_{k+1}$. This process finishes when a level N_m ($m \geq k+1$) or the ROOT node is reached.

Theorem:

The algorithm converts an expression tree into a lattice which doesn't contain equivalent subexpressions and dummy operations.

Proof:

The algorithm terminates because the while statement (4) is executed only once for each level of the tree.

Equivalent subexpressions are represented by equivalent nodes on the same level N_k . Because duplicate nodes are eliminated on level N_k levels $N_1 \dots N_k$ have a lattice structure which doesn't contain equivalent subexpressions.

Fig. 4a

Fig. 4b

Fig. 4c

Fig. 4d

Fig. 4e

Fig. 4f

Fig. 4.

Because levels $N_1 \dots N_k$ don't contain equivalent subexpressions dummy operators can be removed from level N_{k+1} , such that levels $N_1 \dots N_{k+1}$ don't contain dummy operators.

From induction on k it follows that the resulting lattice doesn't contain equivalent subexpressions and dummy operators.

The present algorithm doesn't account for associativity of operators, e.g. the syntactic analysis conceives $R1 + R2 + R3$ as $(R1 + R2) + R3$ and $R1 + R3 + R2$ as $(R1 + R3) + R2$. These expressions yield different tree structures which cannot simply be transformed into each other by interchanging left and right subtrees.

As an illustration of the described technique we consider the following example:

$(R6:D2 = 3 - (R6:D2 = 3) + R2):D1 = "NY"$
 $+ (R2:D1 = "NY")$

This relational expression is represented by the expression tree shown in Fig. 4(a). The reduction process is shown in Fig. 4.(b-f).

Explanation: (a) The original expression tree, (b) On levels N_1 and N_2 the duplicates and dummy operators have been deleted. (c) The dummy operation on level N_3 has been deleted. (d) The father node (+) of the dummy (-) is also a dummy operator and has been deleted from

the tree. In this branch of the tree the first non-dummy ancestor is the selection node which now has to be added to level N_2 . (e) A duplicate of the selection node exists on level N_2 such that one of the duplicates is deleted. (f) The union then belongs to level N_3 but can be deleted; this causes the recognition of ROOT on level N_3 which ends the process.

REFERENCES.

- [1] E. F. Codd: Relational Completeness of Data Base Sublanguages Data Base Systems (Ed. by R. Rustin), pp. 65-98. Prentice-Hall, Englewood Cliffs, New Jersey, (1972).
- [2] E. F. Codd: Further Normalisation of the Data Base Relational Model. Ibid. pp. 33-64 (1972).
- [3] G. M. Birtwistle e.a.: SIMULA Begin, Studentlitteratur, Lund (1974).
- [4] J. M. Smith and P.Y.-T. Chang: Optimizing the performance of a relational algebra database interface. *Commun. ACM* 18(10), 568-579 (1975).
- [5] P. A. V. Hall: Optimization of single expressions in a relational data base system. *IBM J. Res. Develop.* 20(3), 244-257 (1976).
- [6] P. A. V. Hall and S. J. P. Todd: Factorisations of algebraic expressions. *Rep. UKSC 0055, IBM UKSC, Peterlee* (1974).
- [7] P. A. V. Hall: Common subexpression identification in general algebraic systems. *Rep. UKSC 0060, IBM UKSC, Peterlee* (1974).

APPENDIX: SYNTAX OF RELATIONAL EXPRESSIONS.

<relational

expression> ::= <operand><monadic operation>|
 <operand><monadic operation>
 <dyadic operator symbol>
 <relational expression>

<monadic operation> ::= <projection>|
 <projection><monadic operation>|
 <selection>|
 <selection><monadic operation>|
 <empty>

<projection> ::= <projection symbol><domain list>

$\langle \text{selection} \rangle ::= \langle \text{selection symbol} \rangle \langle \text{condition} \rangle$

$\langle \text{operand} \rangle ::= \langle \text{relation name} \rangle |$
 $\quad (\langle \text{relational expression} \rangle)$

$\langle \text{domain list} \rangle ::= \langle \text{domain name} \rangle |$
 $\quad \langle \text{domain name} \rangle, \langle \text{domain list} \rangle$

$\langle \text{condition} \rangle ::= \langle \text{Boolean term} \rangle |$
 $\quad \langle \text{Boolean term} \rangle \vee \langle \text{condition} \rangle$

$\langle \text{Boolean term} \rangle ::= \langle \text{Boolean factor} \rangle |$
 $\quad \langle \text{Boolean factor} \rangle \wedge \langle \text{Boolean term} \rangle$

$\langle \text{Boolean factor} \rangle ::= \langle \text{Boolean primary} \rangle |$
 $\quad \neg \langle \text{Boolean primary} \rangle$

$\langle \text{Boolean primary} \rangle ::= (\langle \text{condition} \rangle) |$
 $\quad \langle \text{domain name} \rangle \langle \text{comparator} \rangle$
 $\quad \langle \text{right part} \rangle$

$\langle \text{right part} \rangle ::= \langle \text{domain name} \rangle |$
 $\quad \langle \text{constant} \rangle$

$\langle \text{comparator} \rangle ::= = | \neq | < | \leq | > | \geq$

$\langle \text{dyadic operator} \rangle ::= \langle \text{union symbol} \rangle |$
 $\quad \langle \text{intersection symbol} \rangle |$
 $\quad \langle \text{difference symbol} \rangle |$
 $\quad \langle \text{join symbol} \rangle |$
 $\quad \langle \text{division operation} \rangle$

$\langle \text{division operation} \rangle ::= [\langle \text{dividend domain list} \rangle$
 $\quad \langle \text{division symbol} \rangle$
 $\quad \langle \text{divisor domain list} \rangle]$

$\langle \text{dividend domain list} \rangle ::= \langle \text{domain list} \rangle$

$\langle \text{divisor domain list} \rangle ::= \langle \text{domain list} \rangle$

$\langle \text{projection symbol} \rangle ::= \pi$

<condition symbol> ::= :

<union symbol> ::= +

<intersection symbol> ::= .

<difference symbol> ::= -

<join symbol> ::= *

<division symbol> ::= /