Evolution of A Query Translation System

Jyh-Sheng Ke
Institute of Information Science
Academia Sinica, Taipei, Taiwan
R. O. C.

Shi-kuo Chang
Department of Information Engineering
University of Illinois at Chicago Circle
U. S. A.

This paper presents the motivation, history, and idiosyncrasy of a query translation system. Detail of the translation process has also been described.

key words: relational database; database skeleton; fuzzy query; conceptual graph; query graph

Introduction

In 1975, we decided to implement a relational database management system for the application of medical information. It quickly became clear that it's impossible to convince physicians of accommodating themselves to using an interactive terminal without providing an efficient user interface. Firstly, the physicians don't know (and are reluctant to know) anything about programming or access path. Secondly, not every physician is good at keyboard typing. So, our goal was set to work out an easy-to -use user interface with the concept of access path being transparent to the users.

One year later, we finished developing the elementary query language (EQL). An example of elementary query (EQ) is given below:

Q : Who is the mayor of Taipei? (user thinking)
EQ : Get MAYOR ; CITY equal 'Taipei'. (user input query)

The user's elementary query is then translated into a sequence of relational algebra commands which is equivalent to the user's original query. Using the elementary query language to input retrieval request, the concept of files is transparent to the user. In other words, the user need not know how data is stored in the database and how the files are linked.

The expressive power of EQL was soon found to be limited. A new version, say, extended elementary query language, was then proposed. The basic idea of extended elementary query is to use a variable name to distinguish descriptors which have different semantic meanings. For example, X.COLOR and Y.COLOR have different semantic meanings though they have the same underlying domain. An example of extended elementary query (EEQ) is given as follows:

Q : Find the names of employees who have salary more than their manager's salary.
EEQ : get ENAME; ENAME of E#; SALARY greater than X.SALARY; X.SALARY of MGR; MGR similar to E#.

In this example, "SALARY" is the salary of employee, and "X.SALARY" is the salary of manager. However, SALARY and X.SALARY have the same underlying domain, MGR and E# also have the same underlying domain.

The extended elementary query language is translated into a sequence of operations which involves the operators contained in the relational operator set {restriction, join, projection, division, difference}.

Using the extended elementary query to input retrieval request , the user still has to know the descriptor name of each data entity. In order to facilitate casual users, we design a query language which allows the user to enter his queries using a natural-like language: We call this query language Extensible Query Language (XQL). The XQL will be first transformed into a form of EEQL and then translated into a sequence of relational algebra commands of the underlying database management system.

In the following, Section 1 describes the general architecture of the query translator. Section 2 describes the syntactic parser. Section 3 describes the defuzzifier. Section 4 describes the QG translator. Section 5 describes the GR translator.

1. The Query Translation System

Translation of XQL is based on a database skeleton. The database skeleton contains a conceptual schema and a relational schema. The conceptual schema is a set of conceptual graphs which represents the user's view of the problem domain in the real world. The relational schema is a logical database description of the relational model from the systems view. Detail of database skeleton can be found in [CHANG78] and [CHANG79]. A sample database skeleton has been shown in Appendix 1. All examples throughout this paper are based on that database skeleton.

The XQL translator includes a parser, a defuzzifier, a QG translator and a GR translator. The parser recognizes word strings and transforms them into standard internal forms. The defuzzifier is a semantic analyzer which constructs query graph at concept level (Q_c). The task of semantic defuzzification is often interspersed throughout the syntactic analysis. Since the user may ask a question with incomplete information, the defuzzifier is used to resolve the ambiguity of user's query. There could be multiple paths for generating the results for answering a user's query, and the translation process might stumble upon a path that answers a question to be different from the one the user asked. The problem of disambiguation will be solved using heuristics. The QG translator translates the query graph at conceptual level (Q_c) into a query graph at file level (Q_f) which is a spanning tree of files and descriptors that are relevant to the request. The GR translator translates Q_f into relational algebra commands of the underlying relational database management systems—RAIN[CHANG75].

2. The Parser

The XQL parser is very simple, since the input language of XQL was chosen in order to guarantee the simplicity of the parser and to provide the user an easy-to-use interface. The parser is driven by the conceptual schema in the database skeleton. It not only recognizes the word strings but also does syntactic defuzzification.

2.1 Syntax Structure of XQL

In XQL, a query is a sequence of short statements separated by semicolons and terminated by a period. In the way of translation process, variation of case grammar [Bruce75] and genus-specializer pair [SZOLO77] has been used to represent the intermediate form of concepts.

2.2 Syntactic Defuzzification of XQL

The major task of syntactic defuzzification is to recognize word strings in the user's query Q, and to transform them into the internal representation. The following rules will be applied repeatedly until all word strings in Q have been transformed into some internal form.

```
Rule 1 : A have B; ----→ B(ch A);

Rule 2 : B of A; ----→ B(ch A);

Rule 3 : A verb B [from C] [to D] [with E];
         ----→ A(agnt verb); B(ptnt verb);
               [C(sou verb);]  [D(des verb);]
               [E(inst verb)]

                               /*B is a concept*/
Rule 4 : A op B; ----→ A(ch A) op B(ch B);

                               /*B is a species of A*/
                               A(ch A) op B(s A);

                               /*B is an instance of A*/
                               A(ch A) op B(i A);

Rule 5 : A verb all B; ----→ A(agnt verb);
                             B(ptnt verb) contain B;

Rule 6 : all A verb B; ----→ A(agnt verb) contain A;
                             B(ptnt verb);
```

The task of syntactic defuzzification is always interspersed with the task of semantic defuzzification. Particularly, when a fuzzy concept is met, the semantic defuzzification will be called to resolve its semantic ambiguity.

3. Semantic Defuzzification of XQL

The task of the semantic defuzzification is to resolve the ambiguity of fuzzy concepts in Q, and to relate all data entities in Q. Since the conceptual schema (CS) in the database skeleton represents the interrelationships among data entities, our problem is to formulate a query graph basing on the set of basic conceptual graphs (BCGs) and the generic hierarchy in the conceptual schema. In other words, we want to formulate a complicated single-level conceptual schema. The final query graph will cover all data entities metioned in Q. The following rules are applied repeatedly until the semantic meaning of each fuzzy concept has been resolved.

Rule 1 : if B(ch A) is in Q but not in CS, and if A > C and B(ch C) is in CS, then B(ch A) is replaced by B(ch C).

Rule 2 : if B(ch A) is in Q but not in CS, and if B > C and C(ch A) is in CS, then B(ch A) is replaced by C(ch A).

Rule 3 : if B(ch A) is in Q but not in CS, and if A < C and B(ch C) is in CS, then B(ch A) is replaced by B(ch C) and some constraint on C. The constraint on C in B(ch C) can have three cases :

(3.1) if A appears in some BCG, then B(ch C) is replaced by X.B(ch C), , and "X.C similar to A" is added into Q, where X is a unique variable name.

(3.2) A is a fuzzy concept of C and is described by a D-type conceptual graph [CHANG79] which represents some constraint on C.

(3.3) A is an instance of C. Add "C == 'A';" into Q.

Rule 4 : if B(ch A) is in Q but not in CS, and if B < C and C(ch A) is in CS, then B(ch A) is replaced by C(ch A) and some constraints on C. The constraint on C in C(ch A) can have three cases :

(4.1) if B appears in some BCG, then C(ch A) is replaced by X.C(ch A), and "X.C similar to B" is added into Q, where X is a unique variable name.

(4.2) B is a fuzzy concept of C and is described by a D-type conceptual graph which represents some constraint on C.

(4.3) B is an instance of C. Add "B == 'C';" into Q.

For Rule 5 and Rule 6, '*' means any case element in agnt, ptnt, sou, des, inst .

Rule 5 : if A(* verb) is in Q but not in CS, and if C < A and C(* verb) is in CS, then A(* verb) is replaced by C(* verb).

Rule 6 : if A(* verb) is in Q but not in CS, and if A < C and C(*verb) is in CS, then A(*verb) is replaced by C(* verb) and some constraint on C. The constraint on C in C(* verb) can have three cases :

(6.1) if A appears in some BCG, then C(* verb) is replaced by X.C(* verb), and "X.C similar to A" is added into Q, where X is a unique variable name.

(6.2) A is a fuzzy concept of C and is described by a D-type conceptual graph which represents some constraint on C.

(6.3) A is an instance of C. Add "C == 'A';" into Q.

Rule 7 : if "A op B;" is in Q, and A is a fuzzy concept which is defined as a genus concept followed by a condition, then replace "A op B;" with "GEN(A) op B; <conditional statement>;". Similarly, if B is a fuzzy concept, replace "A op B;" with "A op GEN(B); <conditional statement>;". The <conditional statement> is the condition associated with the fuzzy concept. Here GEN(A) means "genus concept of A" [SZOLO77].

Rule 8 : if "X.A similar to B" and "Y.A similar to B" are in Q, then X and Y can be unified together, i.e. change Y to X.

Rule 1-8 identify a subset of basic conceptual graphs which covers the user's query. These BCGs must be projected and joined together to form a connected query graph by using query formation rules [CHANG79]. AI researchers have developed some heuristic search strategies [NILSS71] which can be applied to connect the set of BCGs to form a query graph at concept level. To apply conceptual join operation, the following conditions must be satisfied :

(1) if A is joined with B, then A and B must be comparable.
(2) no two non-kernel concepts can be joined together.
(3) if A is a concept of G1 and B is a concept of G2, and if "A contain B' is in Q, then no BCG can be joined with both G1 and G2 directly or indirectly.

Rule 9 : if the conceptual join operation is applied to join concept A and concept B, then add a "A similar to B" statement to Q.

If there are more than one choice in any replacement rule, then select one which is already in Q. For example, if A(ch B) and A(ch C) can be used to replace A(ch A), and A(ch B) is already in Q, then select A(ch B) to replace A(ch A). It should be noted that the process of applying replacement rules is not always deterministic in the sense that more than one pass of defuzzification may be required to resolve semantic ambiguities.

4. QG Translator

In Section 3, the user's original query has been transformed into a query graph at concept level (Qc). The query graph Qc is the connection of a set of basic conceptual graphs (or their projections). Each concept in a basic conceptual graph can be mapped into one or more attribute descriptors. Our next step of query translation is to map each concept into one and only one descriptor, and to find a spanning subgraph of relational files in RS. Each BCG in Qc is mapped into its associated access graph (AG) (An access graph I = (R,E) is a nondirected graph, where R is the set of nodes, $E \subset R \times R$ is the set of edges. Each node in R corresponds a relational file in Rs, and accordingly, a basic conceptual graph in CS.) [KE80]. The set of AGs is then joined together to form a query graph at file level (Q_f). In other words, the QG translator enumerates access paths based on access graphs in RS. It should be noted that if R(X,Y,Z) = R1(X,Z) * R2(X,Y), then the projection of R(X,Y,Z) over X and Z is the same as R1(X,Z). Therefore, if Y contains no essential descriptors, then R2 is redundant to the query. In the access graph, there may be more than one relation containing the same descriptor. Our purpose is to find an access path which has no redundant connection. We will employ some strategies in graph theory to enumerate the access path as a minimum spanning tree of relational files.

For each access graph, we can construct a distance matrix [DEO73] to represent the minimum distances between relational files.

Algorithm 1 : Find a minimum spanning subgraph for a subset X of nodes in an access graph $\underline{G}$ = (R,E).

Let $X \subset R$ be a subset of nodes in $\underline{G}$, and P be the distance matrix of $\underline{G}$, P_{ij} the minimum distance between R_i and R_j, N = $\emptyset$, M = $\emptyset$.

Step 1 : Select R_i and R_j from X, $i \neq j$, such that P_{ij} is minimum. Let N = $\{R_i , R_j\}$ and M = $\{(R_i , R_j)\}$.

Step 2 : Select R_m and R_n, $R_m \in X - N$, $R_n \in N$, such that P_{mn} is minimum. N = N $\cup$ $\{R_m\}$ and M = M $\cup$ $\{(R_m , R_n)\}$.

Step 3 : If X - N is empty, then stop. Else go to Step 2.

Step 4 : Enumerate the path of length P_{mn} for every (R_m , R_n) in M, using Flament's algorithm [SCHAE73].

The total path length is $L(X) = \sum_{(R_m,R_n) \in M} P_{mn}$

Theorem 1 : The spanning subgraph found by Algorithm 1 is a tree, and L(X) is globally minimum.

Proof : Since the minimum length path between any two nodes is unique [KE80] the proof of this theorem triviously follows Prim's theorem [PRIM57].

Q.E.D.

Let S_i be a subset of nodes in an access graph. Denote $L(S_i)$ as the total spanning path length of S_i.

Theorem 2 : Let S_1 and S_2 be two subsets of nodes in an access graph G, and they cover the same set of descriptors. If $S_1 \subset S_2$, then $L(S_1) <= L(S_2)$.

Proof : Let $S_1 = \{R_1, R_2, R_3, \ldots, R_k\}$, $S_2 = \{R_1, R_2, R_3, \ldots, R_k, R_{k+1}\}$. If G is a tree, then $L(S_1) < L(S_2)$, the proof is immediate. In the following we assume that G is not a tree. Since S_1 and S_2 cover the same set of descriptors, R_{k+1} must have at least one descriptor identical to either of R_1, R_2, ..., R_k. Let's assume that R_k and R_{k+1} have one identical descriptor. This implies that R_{k+1} is adjacent to R_k, i.e. R_k and R_{k+1} have an edge connect them. If there does not exist any other path between R_k and R_{k+1}, then either R_{k+1} is contained in the spanning subgraph of S_1 or R_{k+1} is redundant. On the other hand, if there exists one or more paths other than the edge $R_k - R_{k+1}$ connect R_k and R_{k+1}, then all of the nodes in these paths constitute a complete subgraph, and all these nodes have distance 1 between each other.
Thus R_{k+1} is either redundant or contained in the spanning subgraph of S_1. These imply that $L(S_1) <= L(S_2)$.

Q.E.D.

Definition 1 : A concept C is called an essential concept, if it satisfies any of the following conditions :

(1) C appears in 'get' statement.
(2) C appears in a conditional statement.
(3) C appears in a 'similar to' statement.
(4) C is the kernel of a BCG which has been identified in Q.

The following algorithm is used to enumerate the access paths for a user's query. The result of applying this algorithm is a query graph at file level.

Algorithm 2 : Find the minimum spanning subgraph of relational files for a query Q_c.

Let $W = \{CG_i\}$ be the set of BCG_s (or their projections) in Q_c. For each CG_i in W, there is an access graph AG_i associated with it.

The following rules are applied for each CG_i in W :

Step 1 : Identify the set of essential concepts.

Let $E_i = \{C_{ij}\}$ be the set of essential concepts in CG_i.

Step 2 : Map each essential concept into its associated descriptor, and find the relations which contain that descriptor.
Let U_{ij}, $i <= j <= m$, be the set of relations which contain the descriptor associate with the concept C_{ij} in E_i.

Step 3 : Find a set of relations which covers all of the essential descriptor.
Let $Y_i = U_{i1} \times U_{i2} \times \ldots \times U_{im} = \{(u_{i1}, u_{i2}, \ldots, u_{im})\}$.
where $u_{i1} \in U_{i1}, u_{i2} \in U_{i2}, \ldots, u_{im} \in U_{im}$.
Each element in Y_i is a collection of relations which covers all of the essential concepts in CG_i.
Let y_{i1} and y_{i2} be two elements in Y_i, if $y_{i1} \subset y_{i2}$, then delete y_{i2} from y_i.

Step 4 : Calculate the total path length for each candidate set of relations, and select the one with minimum length.
For each $y_{ij} \in Y_i$, apply Step 1-3 of Algorithm 1. Select y_{in}, the element with minimum total path length, from Y_i.

Step 5 : Enumerate access path.
Apply Step 4 of Algorithm 1 to y_{in}.

Step 6 : From the result of Step 5, if R_1 and R_2 are two adjacent relations, generate a "$R_1.K_i$ similar to $R_2.K_i$" statement, where K_i is the descriptor exists in both R_1 and R_2.

After the access paths for all CG_i's in W have been found, the join of these access paths is the query graph at file level Q_f, which is called the fully query.

5. GR Translator

The GR translator translates from the query graph at file level into a sequence of RAIN statements. The sequence of RAIN statements is equivalent to the user's original query.

Definition 2 : A descriptor K is called an essential descriptor, if it is associated with an essential concept.

Definition 3 : A descriptor K is called a key descriptor, if (1)it is an essential descriptor, and (2)it appears in a "similar to" statement of "contain" statement.

Let M be the set of relations in the full query Q_f.

Step 0 : If $R_i.A_1$, $R_i.A_2$, ..., $R_i.A_n$ are the descriptors associated with some virtual entities, then generate a RAIN extend statement

ENTEND R_i TO D BY (A1,FUN1), (A2,FUN2), ..., (An,FUNn)
where FUN_i's are defined by D-type conceptual graphs.
In M, change R_i to D.

Step 1 : For each conditional statement in Q_f, which contains only descriptors covered by the same file $R_i \in M$, do :
Generate a RAIN restriction statement

$C_i = R_i$ *conditions*

Delete this conditional statement from Q_f.
In M, change R_i to C_i.
end;
If no more conditions in Q_f and, M contains only one file name, go to Step 4.
Else, goto Step 2.

Step 2 : If ($R_i.K_n$, similar to, $R_j.K_m$) is in Q_f , then do :
Generate a RAIN join statement

J_i = R_i (essential desc.)(*key desc.) R_j (essential desc.)

Delete this "similar to" statement from Q_f.
In M, change R_i, R_j to J_i. Go to Step 1.
end;
Else goto Step 3.

Step 3 : If either of the following two rules have been applied, then go to Step 1. Otherwise, go to Step 3.

Rule 1 : If ($R_i.K_j$, verb, $R_j.K_i$) and (R_i, K_i, contain, $R_n.K_m$) in Q_f, R_i , $R_n \in$ M, and there is neither linking nor conditional statement for R_n, generate a RAIN division statement followed by a RAIN join statement

D_i = $R_i(K_j,K_i)(/K_i;K_m)R_n(K_m)$
D_j = R_i(essential desc.)(*K_j)D_j (K_j)

Delete this "contain" statement from Q_f.
In M, change R_i to D_j. Go to Step 1.

Rule 2 : If ($R_i.K_j$, verb, $R_j.K_i$) and ($R_i.K_i$, not contain, $R_n.K_m$) is in Q_f, R_i, $R_n \in$ M, and there is neither linking for conditional statement for R_n, generate a RAIN division statement followed by a RAIN difference statement and a RAIN join statement

D_i = $R_i(K_j,K_i)(/K_i;K_m)R_n(K_m)$
D_j = $R_i(K_j)$ - $D_i(K_j)$
D_k = R_i(essential desc.)(*K_j)D_j (K_j)

Delete this "not contain" statement from Q_f.
In M, change R_i to D_k. Go to Step 1.

Step 4 : Generate a RAIN projection statement to project all descriptors in get statement

R_i = R_i(descriptors in get statement)

Step 5 : Generate a RAIN print statement to print out the final response relation

print R_i

Step 6 : If there is a "into filename" statement in Q_f, generate a RAIN rename statement

rename R_i to filename

Step 7 : Erase all temporary files created in query translation

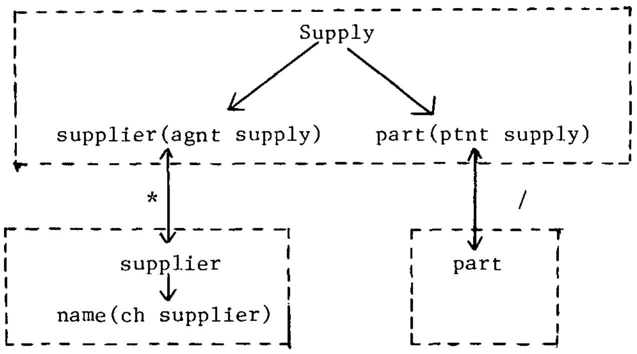
erase R_1, R_2,...

Example : Find the names of suppliers who supply all parts.

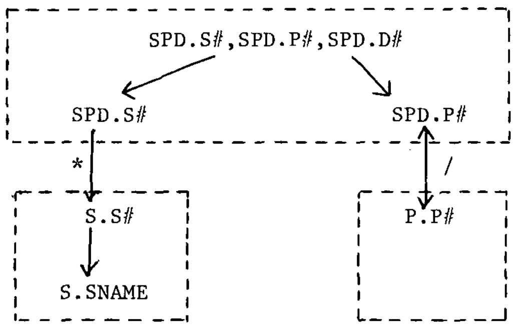
XQL : get name of supplier; supplier supply all part.

(1) get name(ch supplier) ; supplier(agnt supply) supply part(ptnt supply) , part(ptnt supply) contain part.
(2) get name(ch supplier); supplier similar to supplier(agnt supply); supplier(agnt supply) supply part(ptnt supply) ; part(ptnt supply) contain part.

Q_c



Q_f



(3) get S. SNAME ; S.S# similar to SPD.S# ; SPD.S# supply SPD.P#; SPD.P# contain P.P#.
(4) R_1 = S(S#,SNAME)(*S#)SPD(S#,P#)
R_2 = R_1(S#,P#)(/P#)P(P#)
R_3 = R_1(S#,SNAME)(*S#)R_2(S#)
R_4 = R_3(SNAME)
print R_4
erase R_1, R_2, R_3, R_4

Conclusions

In this paper we have described the evolution of a query translation system. The three generations, EQL, EEQL, XQL, of this query translation system reflect not only the evolution of machine intelligence but also the improvement of human engineering. Translation of XQL has been described in detail. Relational algebra has been selected as the target data manipulation language in query translation. The expressive power of XQL is at least relational complete, however, we are extending it by incorporating the capacity of partial matching. This extension is very useful to library application. The underlying database management system is relational model. However, with little modification, it's not difficult to couple XQL to other data models (e.g. hierarchical model, network model).

References

BRUCE75 Bruce, Bertram, "Case System for Natural Language," Artificial Intelligence, Vol.6, No.4, Winter 1975.

CHANG76 Chang, S. K., "Design Considerations of a Database System in a Clinical Network Environment," Proc.of NCC,N.Y., June 1976.

CHANG78 Chang, S. K. and Ke, J. S., "Database Skeleton and its Application to Fuzzy Query Translation," IEEE Trans. Software Eng. Vol.SE-4, Jan. 1978.

CHANG79 Chang, S. K. and Ke, J. S., "Translation of Fuzzy Queries for Relational Database System," IEEE Trans. Pat. Ana. and Mac. Int. Vol.PANI-1, No.3, July 1979.

KE80 Ke, J. S., "On Modeling Relational Databases, Proc. of ICS, Taipei, Dec. 1980.

PRIM57 Prim R. C., "Shortest Connection Networks and Their Some Generalizations," Bell System Tech. J., 36(1957), p1389-1401.

SZOLO77 Sxolorits, P., etal., "An Overview of OWL, A Language for Knowledge Representation," Laboratory for Computer Science, MIT, June 1977.

APPENDIX 1 : An Example Database Skeleton

```
/*Data Base Skeleton for an Example Company
//GENERIC-HIERARCHY
     manager < employee < person;
     secretary < employee;
     department-manager=manager(ch department)
     number(ch employee)=employee-number;
/*'='means 'equivalent' or 'identical'
     supplier-name=name(ch part);
     department < location;
     city < location;
     electrical-part < part;
     mechanical-part < part;
     engineer < employee;
     salesman < employee;
     clerk < employee;
     supplier-city=city(ch supplier);
     visit-city=city(ch salesman);
/*supplier has number, name, and city
/*No comment statement can be inserted within each
/*graph
//P-TYPE CG
      supplier : = S

              $number: S#(9(3));
              name : SNAME(X(20));
              city : SCITY(X(20));

/*part has number, color, and name
//P-TYPE CG
      part : = P

              $number : P#(9(3));
              name : PNAME(X(20));
              color : COLOR(X(10));

/*department has number and manager
//P-TYPE CG
      department : = D

              $number : P#(9(2));
              manager : MGR(9(2));

/*employee has number, name, age(virtual entity),
/*birth, salary, and department
//P-TYPE CG

      employee : = EMP

              $number : E#(9(4));
              name : ENAME(X(20));
              *age : AGE(9(2));
              birth : BIRTH(9(6));
              salary : SALARY(9(5));
              department : D#(9(2));

/*electrical-part has number, voltage and current
//P-TYPE CG
      electrical-part : = ELEC

              $number : EP#(9(3));
              voltage : VOLT(9(2));
              current : CURNT(9(3));

/*mechanical-part has number and load
//P-TYPE CG
      mechanical-part : = MECH

              $number : MP#(9(3));
              load : LOAD(9(3));

/*engineer has number, license and specialization
//P-TYPE CG
      engineer : = ENGR

              $number : ER#(9(3));
              specialization : SPEC(9(1));
              license : LICE(9(6));

/*salesman has number, visiting-city, and languages
//P-TYPE CG
      salesman : = SALES

              $number : ES#(9(3));
              city : CITY(X(20));
              language : LANG(9(2));

/*clerk has number, typing speed, and writing speed
//P-TYPE CG
      clerk : = CLERK

              $number : EC#(9(3));
              type-speed : TYPE(9(2));
              write-speed : WRITE(9(2));

/*city has name, population, and mayor
//P-TYPE CT
      city : = CITY

              $name : CNAME(X(20));
              population : POP(9(6));
              mayor : MAYOR(X(20));
```

```
/*supplier supply part to department with some quan-
/*tity
//R-TYPE CG
      supply : = SPD

              $supplier(agnt) : S#(9(3));
              $part(ptnt) : P#(9(3));
              $department(des) : D#(9(2));
              quantity(inst) : QTY(9(3));

/*virtual entity age = diff(data, birth)
//DV-TYPE CG
      diff

              age(ch employee);
              data;
              birth(ch employee);

/*graphical representation of fuzzy concept
//DF-TYPE CG
       dark-color

              color;
              color = brown(!);
              color = black(!);
```

After compilation, the following relation definition statements will be generated;

```
DEFINE EFILE S(S#(9(3)),SNAME(X920)),SCITY(X(20))))
DEFINE EFILE P(P#(9(3)),PNAME(X(20)),COLOR(X(10))))
DEFINE EFILE D(D#(9(2)),MGR(9(2)))
DEFINE EFILE EMP(E#(9(4)),ENAME(X(20)),AGE(9(2)),
       BIRTH(9(6)),SALARY(9(5)),D#(9(2)))
DEFINE EFILE ELEC(EP#(9(3)),VOLT(9(2)),CURNT(9(3)))
DEFINE EFILE MECH(MP#(9(3)),LOAD(9(3)))
DEFINE EFILE ENGR(ER#(9(3)),SPEC(9(1)),LICE(9(6)))
DEFINE EFILE SALES(ES#(9(3)),VCITY(X(20)),LANG(9(2)))
DEFINE EFILE CLERK(EC#(9(3)),TYPE(9(2)),WRITE(9(2)))
DEFINE EFILE CITY(CNAME(X(20)),POP(9(6)),MAYOR(X(20)))
DEFINE EFILE SPD(S#(9(3)),P#(9(3)),D#(9(2)),QTY(9(3)))
```