CS252 Fundamentals of Relational Databases — Solutions for Worksheet 4 Constraints, Catalog, and Virtual Relvars in *Rel*

1. *No questions asked.*

2. Write **Tutorial D** integrity constraints for the suppliers-and-parts database to express the following requirements:

   a. Every shipment tuple must have a supplier number matching that of some supplier tuple.

```
CONSTRAINT Ca IS_EMPTY ( SP NOT MATCHING S ) ;
```

   Relation comparison could alternatively be used:

```
CONSTRAINT Ca ( SP { S# } ) <= ( S { S# } ) ;
```

   but note the need to state the matching attribute name explicitly—this might be thought to be an advantage or a disadvantage. The first solution is neat and immune to changes in attribute names, but exposed to the possibility of inappropriately chosen attribute names. The second solution is exposed to the possible change in name of the S# attributes but is immune to all other attribute name changes. A compromise could be:

```
CONSTRAINT Ca IS_EMPTY ( SP { S# } NOT MATCHING S ) ;
```

   b. Every shipment tuple must have a part number matching that of some part tuple.

```
CONSTRAINT Cb IS_EMPTY ( SP NOT MATCHING P ) ;
```

   c. All London suppliers must have status 20.

```
CONSTRAINT Cc IS_EMPTY
  ( S WHERE City = 'London' AND Status <> 20 ) ;
```

   d. No two suppliers can be located in the same city.

   Add the following to the declaration of the relvar S:

```
KEY { CITY }
```

   Alternatively:

```
CONSTRAINT Cd
     COUNT ( S { City } ) = COUNT ( S ) ;
```

   e. At most one supplier can be located in Athens at any one time.

```
CONSTRAINT Ce
     COUNT ( S WHERE City = 'Athens' ) <= 1 ;
```

   f. There must exist at least one London supplier.

```
CONSTRAINT Cf
     COUNT ( S WHERE City = 'London' ) > 0 ;
```

   g. The average supplier status must be at least 10.

   One is tempted to write something like

```
CONSTRAINT Cg
     AVG ( S, Status ) >= 10.0 ;
```

but AVG is undefined on the empty set. If it is permissible for S to be empty, we could write

```
CONSTRAINT Cg
     AVG ( S { S#, Status }
       UNION
       RELATION { TUPLE { S# 'S1', Status 10 } },
     Status ) >= 10.0 ;
```

but not

```
CONSTRAINT Cg
     IS_EMPTY ( S ) OR AVG ( S, Status ) >= 10.0 ;
```

because **Tutorial D** assumes that (a) operands of OR can be evaluated in either order and (b) the system is permitted to evaluate both operands even when it has discovered one of them to be TRUE.

h. Every London supplier must be capable of supplying part P2.

```
CONSTRAINT Ch IS_EMPTY (
     ( S WHERE City = 'London' ) NOT MATCHING
     ( SP WHERE P# = 'P2' ) );
```

Which of your constraint definitions are rejected by *Rel* for being FALSE at the time of definition?

```
Cd
```

3. Create a virtual relvar named myvars giving the Name, Owner, and isVirtual of every relvar not owned by 'Rel'.

```
VAR myvars VIRTUAL ( sys.Catalog WHERE Owner <> 'Rel' )
                   { Name, Owner, isVirtual } ;
```

4. Load /local/java/Rel-*version*[1]/Scripts/OperatorsChar.d. into *Rel's* input pane and execute that script. As a result several useful user-defined operators will be available to you. One of the relvars mentioned in sys.Catalog is named sys.Operators. Display the contents of that relvar. How many attributes does it have? What is the declared type of the attribute named Implementations?

Two attributes: Name and Implementations

The declared type of Implementations is:

```
RELATION {Signature CHARACTER,
          ReturnsType CHARACTER,
          Definition CHARACTER,
          Language CHARACTER,
          CreatedByType CHARACTER,
          Owner CHARACTER,
          CreationSequence INTEGER}
```

Relation types aren't normally recommended for attributes of database relvars, but all updates to the system catalog are performed "under the covers" by the system itself, which should be capable of handling all the difficulties caused by relation-valued attributes. That said, queries against such relvars can be difficult to express, unless you begin them by ungrouping, as suggested in the next exercise.

[1] replace *version* by the version being used (was 0.3.17 in 2009)

5. Evaluate the expression

```
(sys.Operators ungroup (Implementations)
where Language = 'JavaF')
{ ALL BUT Language, CreatedByType, Owner, CreationSequence}
```

What are the "ReturnsTypes" of `LENGTH`, `IS_DIGITS`, and `SUBSTRING`?

> `INTEGER`, `BOOLEAN`, and `CHARACTER`, respectively.

6. Note that if *s* is a value of type `CHAR`, then `LENGTH`(*s*) gives the number of characters in *s,* `IS_DIGITS`(*s*) gives `TRUE` if and only if every character of *s* is a decimal digit. `SUBSTRING`(*s*,0,l) gives the string consisting of the first *l* characters of *s* (note that strings are considered to start at position 0, not 1). `SUBSTRING`(*s*,*f*) gives the string consisting of all the characters of *s* from position *f* to the end.

What is the result of `IS_DIGITS('')`? Is it what you expected? Is it consistent with the definition given above?

> `TRUE`. This is to be expected on the understanding that "everything is true of all elements of the empty set". The string `''` contains no characters and therefore does not contain a character that isn't a digit. $(\forall x)P(x)$ is logically equivalent to $\neg(\exists x)\neg P(x)$.

7. Using operators defined by OperatorsChar.d, define types for supplier numbers and part numbers, taking the example shown in lecture HACD.2, Slides 14 and 15. *Note:* Those two slides were revised *after* the 2009 presentation of that lecture.

```
TYPE SNO POSSREP { c CHAR CONSTRAINT
                          SUBSTRING(c,0,1) = 'S' AND
                          IS_DIGITS(SUBSTRING(c,1)) } ;

TYPE PNO POSSREP { c CHAR CONSTRAINT
                          SUBSTRING(c,0,1) = 'P' AND
                          IS_DIGITS(SUBSTRING(c,1)) } ;
```

Define relvars `Srev`, `Prev`, and `SPrev` as replacements for `S`, `P` and `SP`, using the types you have just defined as the declared types of attributes `S#` and `P#`.

```
VAR Srev BASE RELATION {S# SNO, Sname CHAR,
                        Status INTEGER, City CHAR}
                   KEY { S# } ;

VAR Prev BASE RELATION {P# PNO, Pname CHAR, Colour CHAR,
                        Weight RATIONAL, City CHAR}
                   KEY { P# } ;

VAR SPrev BASE RELATION {S# SNO, P# PNO, Qty INTEGER}
                    KEY { S#, P# } ;
```

Write relvar assignments to copy the contents of `S`, `P` and `SP` to `Srev`, `Prev`, and `SPrev`, respectively. Note that if `SNO` is the type name for supplier numbers in `S` and `Srev`, then `SNO(S#)` "converts" an `S#` value in `S` to one for use in `Srev`.

> We need to use `EXTEND` to add an attribute to contain the "converted" `S#` and/or `P#` values. The new attributes can't be named `S#` and `P#`, of course, but we do need them to have those names, so we "project away" the old attributes and then rename the new ones:

```
Srev := ( ( EXTEND S ADD ( XS# AS SNO(S#) ) )
          { ALL BUT S# } )
        RENAME ( XS# AS S# ) ;

Prev := ( ( EXTEND P ADD ( XP# AS PNO(P#) ) )
          { ALL BUT P# } )
        RENAME ( XP# AS P# ) ;

SPrev := ( ( EXTEND SP ADD ( XS# AS SNO(S#),
                             XP# AS PNO(P#) ) )
           { ALL BUT S#, P# } )
         RENAME ( XS# AS S#, XP# AS P# ) ;
```

End of Solutions