ISO/IEC JTC 1/SC 21 N 11159

Date: 1997-10-21

ISO/IEC CD 13249-4:199x (E)

ISO/IEC JTC 1/SC 21/WG 3

Secretariat: Canada

Information Technology — Database Languages —

SQL Multimedia and Application Packages —

Part 4: General Purpose Facilities

Document type: International Standard
Document subtype: Not applicable
Document stage: (30) Committee
Document language: E

Contents Page

Foreword v

Introduction vi

1 Scope 1

2 Normative references 2

3 Definitions, notations, and conventions 3

3.1 Definitions 3

3.2 Notations 3

3.3 Conventions 3

4 Concepts 4

4.1 USAGE Privileges on User-defined Types 4

4.2 UNDER Privileges on User-defined Types 4

4.3 EXECUTE Privileges on Routines 4

5 Angle Abstract Data Type 5

5.1 Definitions 5

5.2 Notations 6

5.3 Conventions 6

5.4 Concepts 6

5.5 Angle Abstract Data Type 8

5.6 SQL Syntax Extensions 15

5.6.1 Arithmetic operations 15

5.6.2 C**omparison predicates** 15

5.7 Conformance 16

5.8 Status Codes 17

6 Complex Number Abstract Data Type 18

6.1 Definitions 18
6.2 Notations 19
6.3 Conventions 19
6.4 Concepts 19
6.5 Complex Number Abstract Data Type 20
6.6 SQL Syntax Extensions 24
6.6.1 Arithmetic operations 24
6.6.2 Comparison predicates 25
6.7 Conformance 26
7 Numerics Schema 27
7.1 Definitions 27
7.2 Notations 27
7.3 Conventions 27
7.4 Concepts 27
7.4.1 Elementary numeric functions 27
7.4.2 Combinatorial functions 29
7.4.3 Transcendental functions 30
7.5 Numerics Schema 36
7.6 Conformance 85
7.6.1 Elementary functions conformance 85
7.6.2 Transcendental function conformance 86
7.6.3 SQLMM_NUMERICS schema conformance 86
7.7 Status Codes 87
Index 92

Tables

Page

Table 1 — SQLSTATE class and subclass values 17
Table 2 — Input and Return Types for Elementary Numeric Functions 28
Table 3 — Input and Output Types for Transcendental Functions 32
Table 4 — SQLSTATE class and subclass values 88

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC SQL/MM was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*.

This document is based on the content of ISO Working Draft Database Language (SQL3).

This is the first edition of ISO/IEC SQL/MM—Part 4: General Purpose Facilities.

Introduction

The purpose of this International Standard is to define multimedia and application specific objects and their associated methods (object packages) using the object-oriented features in SQL3 (ISO/IEC Project 1.21.3.4)

SQL/MM is structured as a multi-part standard. At present it consists of the following parts:

Part 1: Framework

Part 2: Full-Text

Part 3: Spatial

Part 4: General Purpose Facilities

Part 5: Still Image

Information Technology — Database Languages — SQL Multimedia and Application Packages — Part 4: General Purpose Facilities

1 Scope

This part of ISO SQL/MM:

a) specifies SQL/MM ADTs that have general purpose applicability in a number of different application areas. These may include numeric functions, especially trigonometric, exponential, and logarithmic, as well as various other general purpose ADTs such as tree, graphs, stacks and queues.

2 Normative references

The following standards contain provisions that, through reference in this text, constitute provisions to this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 9075-1:199x, *Information Technology — Database languages — SQL — Part 1: Framework (SQL/Framework).*

ISO/IEC 9075-2:199x, *Information Technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation).*

ISO/IEC 9075-4:1996, *Information Technology — Database languages — SQL — Part 4: Persistent Stored Modules (SQL/PSM).*

ISO/IEC 13249-2:199x, *Information Technology — Database languages — SQL Multimedia and Application Packages — Part 2: Full-Text.*

ISO/IEC 13249-3:199x, *Information Technology — Database languages — SQL Multimedia and Application Packages — Part 3: Spatial.*

ISO/IEC 13249-5:199x, *Information Technology — Database languages — SQL Multimedia and Application Packages — Part 5: Still Image.*

3 Definitions, notations, and conventions

3.1 Definitions

To be supplied.

3.2 Notations

To be supplied.

3.3 Conventions

To be supplied.

4 Concepts

4.1 USAGE Privileges on User-defined Types

ISO 9075 specifies that a user must have the USAGE privilege on the domain or user-defined type before they can use it for defining other objects such as SQL-invoked routines, tables, views, domains, or user-defined types. This International Standard does not include the GRANT USAGE statements for the domains and user-defined types defined in this International Standard. For each object defined by this International Standard, a GRANT statement granting USAGE privilege to an implementation-defined set of grantees shall be effectively executed when these domains and user-defined types are created, except when explicitly noted by the Definitional Rules in this International Standard. It is implementation-defined whether the GRANT statement includes WITH GRANT OPTION.

4.2 UNDER Privileges on User-defined Types

ISO 9075 specifies that a user must have the UNDER privilege on the user-defined type before they can use it for defining subtypes. This International Standard does not include the GRANT UNDER statements for the user-define types defined in this International Standard. For each object defined by this International Standard, a GRANT statement granting UNDER privilege to an implementation-defined set of grantees shall be effectively executed when these user-defined types are created, except when explicitly noted by the Definitional Rules in this International Standard. It is implementation-defined whether the GRANT statement includes WITH GRANT OPTION.

4.3 EXECUTE Privileges on Routines

ISO 9075 specifies that users must have the EXECUTE privilege on the SQL-invoked routine before they can execute it. This International Standard does not include the GRANT EXECUTE statements for the SQL-invoked defined in this International Standard. For each such SQL-invoked routine, a GRANT statement granting EXECUTE privilege to an implementation-defined set of grantees shall be effectively executed when the SQL-invoked routines are created, except when explicitly noted by the Definitional Rules in this International Standard. It is implementation-defined whether the GRANT statement includes WITH GRANT OPTION.

More to be supplied as required.

5 Angle Abstract Data Type

5.1 Definitions

angle: A term from geometry, or engineering, used for circular measurements, or to measure the degree of separation of two intersecting lines. An angle is a real number that can be evaluated in degrees, minutes, and seconds, or in radians. Both measures are supported by the ANGLE abstract data type. Instances of ANGLE(BaseType), for any given approximate numeric BaseType, form a vector space over that BaseType.

degree: A real number unit of measurement for angles. A degree is subdivided into minutes and seconds. Each degree consists of 60 minutes and each minute consists of 60 seconds. Seconds are further subdivided into fractions of a second with arbirary fractional precision. Degrees (d) are related to radians (r) by the expression d = (180*r)/pi.

radian: A real number unit of measurement for angles. Radians (r) are related to degrees (d) by the expression r = (pi*d)/180.

pi: A real number mathematical constant that represents the circumference of a circle with unit diameter. This number is transcendental and cannot be represented exactly in any algebraic form.

5.2 Notations

To be supplied.

5.3 Conventions

To be supplied.

5.4 Concepts

The ANGLE abstract data type template, ANGLE(BaseType), models the geometric and engineering concept of an angle measurement, with varying degrees of precision. BaseType is the data type of the underlying angle representation in either degrees or radians, and the binary precision of this numeric data type determines the precision of the resulting ANGLE ADT. The type template produces a family of generated angle data types, each with a different precision. The following are supported as BaseTypes in this specification: REAL, FLOAT with user specified precision, and DOUBLE PRECISION.

For convenience and simplicity, ANGLE is defined to be a syntactic shorthand for ANGLE(REAL), and ANGLE_DOUBLE is defined to be a syntactic shorthand for ANGLE(DOUBLE PRECISION). Other convenient syntactic shorthands are defined in the ANGLE ADT subclause below.

All operations on angle instances are through functions defined as part of the ADT definitions. These functions include:

Angle(d,m,s)	to create a new angle instance from degree (d), minute (m), and second (s) input. Each input parameter is a real number of the BaseType for angle with no sign, magnitude, or fractional restrictions.
Angle(r)	to create a new angle instance from radian (r) input. Each input parameter is a real number of the BaseType for angle with no sign, magnitude, or fractional restrictions.
IsEqual(u,v)	to test equality of two angles u and v returning BOOLEAN.
IsLessThan(u,v)	to compare two angles u and v to determine if u is less than v, returning BOOLEAN.
Plus(u)	to perform the identity operation on an angle, returning +u.
Add(u,v)	to add two angles returning u + v.
Minus(u)	to find the additive inverse of an angle, returning -u.
Subtract(u,v)	to subtract two angles returning u - v.
Multiply(x,u)	to multiply a scalar, x, times an angle, u, returning x * u. This function has the specific name LeftScalarMult.
Multiply(u,x)	to multiply a scalar, x, times an angle, u, returning x * u. This function has the specific name RightScalarMult.

Abs(u)	to return the absolute value of an angle.
Sign(u)	to return a SMALLINT, -1, 0, or +1, depending on whether an angle is negative, zero, or positive, and returns null if the angle is null.
Degree(u)	to return a non-negative INTEGER representing the absolute value of the number of degrees in an angle. The result is not scaled or normalized and may be arbitrarily large.
Minute(u)	to return a non-negative SMALLINT, less than 60, representing the number of minutes in a fractional degree of an angle.
Second(u)	to return a non-negative SMALLINT, less than 60, representing the number of seconds in a fractional minute of an angle.
Fraction(u)	to return a non-negative real number, less than 1, of the BaseType of an angle, representing the fractional number of seconds in that angle.
Radian(u)	to return a real number of the BaseType of an angle representing its equivalent angular measure in radians.
ToVarChar(u)	to cast an angle to a variable length character string representation.
ToRealAngle(u)	to cast an angle to an angle with REAL as the underlying BaseType.
ToFloat20Angle(u)	to cast an angle to an angle with FLOAT(20) as the underlying BaseType.
ToFloat47Angle(u)	to cast an angle to an angle with FLOAT(47) as the underlying BaseType.
ToDoubleAngle(u)	to cast an angle to an angle with DOUBLE PRECISION as the underlying BaseType.

An angle variable or parameter may assume any of the valid SQL null values, but only for instances of the ADT itself. The angle ADT definition is designed to prohibit null values for any of the components of its underlying representation.

The SQL special operators for addition (+), subtraction or negation (-), and multiplication (*) are overloaded to support angle addition, angle subtraction or negation, and scalar multiplication, respectively, with the normal mathematical rules for precedence. The SQL comparison operands for equality (=), not equal (<>), less than (<), less than or equal (<=), greater than (>), and greater than or equal (>=) are overloaded to support angle comparison in any SQL comparison predicate. This implicit comparison for angles is also used in all other SQL clauses that depend upon comparison criteria, including ORDER BY, GROUP BY, HAVING, UNIQUE, and DISTINCT.

5.5 Angle Abstract Data Type

Function

The ADT type template for ANGLE specified below is self-contained with several exceptions; the functions "Abs", "Sign", and "Floor" from Subclause 7.4.1, "Elementary numeric functions", and the "Radian" function from Subclause 7.4.3.2, "Trigonometric functions", are used in the routine body of several of the routine members of this ADT. If an implementation claiming support for processing the ANGLE ADT does not also support elementary numeric functions, then that implementation must supply routines for "Abs", "Sign", and "Floor" defined on the supported approximate numeric BaseTypes. The sole use of the "Radian" function is to calculate the value of the mathematical constant "pi", accurate to within the numeric precision of the BaseType of the Angle ADT. Any other substitute for calculating the value of "pi", with the expected accuracy, is acceptable.

****Editor's Note 4-014****

Angle Fuzzy Equality. The ANGLE data type in SQL/MM introduces new concerns about the nature of approximate comparisons. For example, the comparison predicate, Angle(45,0,0) = Angle(44,60,0), will likely return FALSE because of round-off errors in the calculation of the underlying radian representation of both angle values. This is not unlike the current situation with approximate numerics where the predicate, 3/4 = 0.75, may also return FALSE because of some round-off error in the 64th bit, but it is more of a concern in ANGLE because the user may not be aware that conversion from degrees to radians is happening during construction of the angle instance. Some suggest that we might define a special "fuzzy equals" function of the form IsAlmostEqual(x, y, delta), where delta is some very small positive number, to return a Boolean result equivalent to Abs(x-y) < delta; the drawback with "fuzzy equal" is that the axioms of arithmetic (e.g. transitivity) fall by the wayside. This might be one more argument in support of an exact ANGLE data type with base representation using SQL Exact Numeric data types.

****Editor's Note 4-015****

Bounded Angle Measurements. The ANGLE data type in SQL/MM interprets an angle as a point on the real number line -- it is not bounded to be between 0 to 360 degrees or -180 to +180 degrees. Such boundings are critical to certain applications, e.g. -90 to +90 degrees for Latitudes, and -180 to +180 degrees for Longitudes. Such applications require the definition of angular arithmetic to return values within these same ranges. One approach might be to allow the bounds to be declared at type creation time. For example:

```
CREATE TYPE TEMPLATE ANGLE (BaseType TYPE, Lower BaseType, Upper BaseType)
```

where Lower and Upper determine the range of values for angle instances. Then we could define modulo arithmetic to always return a value within those ranges, thereby allowing Latitude and Longitude definitions as follows:

```
CREATE DISTINCT TYPE LATITUDE AS ANGLE (REAL, -90, +90)
CREATE DISTINCT TYPE LONGITUDE AS ANGLE (REAL, -180, +180)
```

Other considerations to be addressed at the same time include whether or not both endpoints in a range of angular values represent the same angle (clearly No for Latitude, but Yes for Longitude), or if, instead, angles should only assume values on the half-open interval [Lower, Upper), e.g. the International Date Line would be -180 instead of +180 degrees Longitude.

****Editor's Note 4-016****

Alternative Degree, Minute, Second Functions. The Degree function in the ANGLE ADT of SQL/MM Part 4 returns only the absolute value of the whole number of degrees in an angle. Often it is desirable to return a signed approximate numeric value to represent the total number of degrees in an angle. Similarly, the Minute function returns only the whole number of minutes in a fractional degree of an angle, whereas sometimes it is desirable to return a signed approximate numeric to represent the total number of minutes in a degree. Similarly for the Second and Fraction functions. These functions were intended originally to be used with the Sign function to break a degree into its Sign, Degree, Minute, Second, and Fraction components. Instead, it might be better to have these functions return total values as indicated above — and then define a new function, say DegreeParts, to return a ROW of 5 values to represent the above components of a degree.

Definition

```
CREATE TYPE TEMPLATE ANGLE
   (BaseType   TYPE)
   (-- !! Possible problem with PROTECTED
    PROTECTED radian BaseType DEFAULT 0  NOT NULL,
```

****Editor's Note 4-036****

PROTECTED is no longer supported.

```
   FUNCTION Angle
      (deg     BaseType,
       min     BaseType,
       sec     BaseType)
      RETURNS ANGLE(BaseType)
      LANGUAGE SQL
      BEGIN
         DECLARE ang             ANGLE(BaseType) DEFAULT Angle();
         DECLARE pi              BaseType
            DEFAULT Radian(CAST(180 AS BaseType));
         IF deg IS NULL OR min IS NULL OR sec IS NULL
            THEN RETURN NULL;
         END IF;
         SET ang>>radian = pi*(deg + min/60 + sec/3600)/180;
         RETURN ang;
      END,

   FUNCTION Angle(rad BaseType)
      RETURNS ANGLE(BaseType)
      LANGUAGE SQL
      BEGIN
         DECLARE ang             ANGLE(BaseType) DEFAULT Angle();
         IF rad IS NULL
            THEN RETURN NULL;
         END IF;
         SET ang>>radian = rad;
         RETURN ang
      END,
```

```
FUNCTION IsEqual
   (ang1    ANGLE(BaseType),
    ang2    ANGLE(BaseType))
   RETURNS BOOLEAN
   LANGUAGE SQL
   BEGIN
      IF ang1 IS NULL OR ang2 IS NULL
         THEN RETURN NULL;
      END IF ;
   RETURN ang1>>radian = ang2>>radian ;
   END,

FUNCTION IsLessThan
   (ang1    ANGLE(BaseType),
    ang2    ANGLE(BaseType))
   RETURNS BOOLEAN
   LANGUAGE SQL
   BEGIN
      IF ang1 IS NULL OR ang2 IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN ang1>>radian < ang2>>radian;
   END,

FUNCTION Plus(ang ANGLE(BaseType))
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN ang;
   END,

FUNCTION Add
   (ang1    ANGLE(BaseType),
    ang2    ANGLE(BaseType))
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE ang             ANGLE(BaseType);
      IF ang1 IS NULL OR ang2 IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang>>radian = ang1>>radian + ang2>>radian;
      RETURN ang;
   END,

FUNCTION Minus(ang ANGLE(BaseType))
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE ang1            ANGLE(BaseType);
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = -ang>>radian;
      RETURN ang1;
   END,
```

```
FUNCTION Subtract
   (ang1    ANGLE(BaseType),
    ang2    ANGLE(BaseType))
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE ang             ANGLE(BaseType);
      IF ang1 IS NULL OR ang2 IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang>>radian = ang1>>radian - ang2>>radian;
      RETURN ang;
   END,

FUNCTION Multiply
   (scalar  BaseType,
    ang     ANGLE(BaseType)
   )
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   SPECIFIC LeftScalarMult
   BEGIN
      DECLARE ang1            ANGLE(BaseType);
      IF scalar IS NULL OR ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = scalar * ang>>radian;
      RETURN ang1;
   END,

FUNCTION Multiply
   (ang     ANGLE(BaseType),
    scalar  BaseType)
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   SPECIFIC RightScalarMult
   BEGIN
      DECLARE ang1            ANGLE(BaseType);
      IF scalar IS NULL OR ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = scalar * ang>>radian;
      RETURN ang1;
   END,

FUNCTION Abs(ang ANGLE(BaseType))
   RETURNS ANGLE(BaseType)
   LANGUAGE SQL
   BEGIN
      IF ang>>radian < 0
         THEN RETURN Minus(ang)
         ELSE RETURN ang;
      END IF;
   END,
```

```
FUNCTION Sign(ang ANGLE(BaseType))
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN Sign(ang>>radian);
   END,

FUNCTION Degree(ang ANGLE(BaseType))
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE x               BaseType;
      DECLARE pi              BaseType
         DEFAULT Radian(CAST(180 AS BaseType));
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET x = 180*Abs(ang>>radian)/pi;
      RETURN Floor(x);
   END,

FUNCTION Minute(ang ANGLE(BaseType))
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      DECLARE x               BaseType;
      DECLARE pi              BaseType
         DEFAULT Radian(CAST(180 AS BaseType));
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET x = 180*Abs(ang>>radian)/pi;
      RETURN Floor(60*(x - Floor(x)));
   END,

FUNCTION Second(ang ANGLE(BaseType))
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      DECLARE x               BaseType;
      DECLARE pi              BaseType
         DEFAULT Radian(CAST(180 AS BaseType));
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET x = 60*(180*Abs(ang>>radian)/pi - Degree(ang));
      RETURN Floor(60*(x - Floor(x)));
   END,
```

```
FUNCTION Fraction(ang ANGLE(BaseType))
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      DECLARE x               BaseType;
      DECLARE pi              BaseType
         DEFAULT Radian(CAST(180 AS BaseType));
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET x = 60*(60*(180*Abs(ang>>radian)/pi - Degree(ang)) -
      Minute(ang));
      RETURN x - Floor(x);
   END,

FUNCTION Radian(ang ANGLE(BaseType))
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN ang>>radian;
   END,

FUNCTION ToVarChar(ang ANGLE(BaseType))
   RETURNS CHARACTER VARYING(30)
   LANGUAGE SQL
   BEGIN
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      IF Sign(ang) = 0
         THEN RETURN '0:00:00';
      END IF;
      RETURN
         CASE Sign(ang) WHEN -1 THEN '-' WHEN +1 THEN '+' END ||
         CAST (Degree(ang) AS CHARACTER VARYING(20)) || ':' ||
         CAST (Minute(ang) AS CHARACTER(2)) || ':' ||
         CAST (Second(ang) AS CHARACTER(2)) ||
         CASE Sign(Fraction(ang)) WHEN 0 THEN '' WHEN +1 THEN '.' ||
         SUBSTRING(CAST(Fraction(ang) AS CHARACTER VARYING(25)) FROM
            POSITION('.' IN CAST(Fraction(ang) AS
               CHARACTER VARYING(25)))) END;
   END,

FUNCTION ToRealAngle(ang ANGLE(BaseType))
   RETURNS ANGLE(REAL)
   LANGUAGE SQL
   BEGIN
      DECLARE ang1            ANGLE(REAL);
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = CAST(ang>>radian AS REAL);
      RETURN ang1;
   END,
```

```
FUNCTION ToFloat20Angle(ang ANGLE(BaseType))
   RETURNS ANGLE(FLOAT(20))
   LANGUAGE SQL
   BEGIN
      DECLARE ang1            ANGLE(FLOAT(20));
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = CAST(ang>>radian AS FLOAT(20));
      RETURN ang1;
   END,

FUNCTION ToFloat47Angle(ang ANGLE(BaseType))
   RETURNS ANGLE(FLOAT(47))
   LANGUAGE SQL
   BEGIN
      DECLARE ang1            ANGLE(FLOAT(47));
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = CAST(ang>>radian AS FLOAT(47));
      RETURN ang1;
   END,

FUNCTION ToDoubleAngle(ang ANGLE(BaseType))
   RETURNS ANGLE(DOUBLE PRECISION)
   LANGUAGE SQL
   BEGIN
      DECLARE ang1            ANGLE(DOUBLE PRECISION);
      IF ang IS NULL
         THEN RETURN NULL;
      END IF;
      SET ang1>>radian = CAST(ang>>radian AS DOUBLE PRECISION);
      RETURN ang1;
   END,

EQUALS IsEqual LESS THAN IsLessThan,

CAST (ANGLE(BaseType) AS CHARACTER VARYING(30) WITH ToVarChar),
CAST (ANGLE(BaseType) AS ANGLE(REAL) WITH ToRealAngle),
CAST (ANGLE(BaseType) AS ANGLE(FLOAT(20)) WITH ToFloat20Angle),
CAST (ANGLE(BaseType) AS ANGLE(FLOAT(47)) WITH ToFloat47Angle),
CAST (ANGLE(BaseType) AS ANGLE(DOUBLE PRECISION) WITH ToDoubleAngle)
)

CREATE DISTINCT TYPE ANGLE AS ANGLE(REAL)

CREATE DISTINCT TYPE ANGLE_DOUBLE AS ANGLE(DOUBLE PRECISION)

CREATE DISTINCT TYPE ANGLE_PRECISION_20 AS ANGLE(FLOAT(20))

CREATE DISTINCT TYPE ANGLE_PRECISION_47 AS ANGLE(FLOAT(47))
```

****Editor's Note 4-017****

The numbers 20 and 47 play a special role in approximate numeric calculations because they are conservative estimates for maximum approximate numeric precision (i.e. binary precision of the mantissa) in 32-bit and 64-bit environments, respectively.

5.6 SQL Syntax Extensions

5.6.1 Arithmetic operations

Database Language SQL (March '95 SQL3 WD) identifies the following predefined standard operators in Subclause 4.16, "Operators": Plus (+), Minus (-), Multiply (*), Add(+), and Subtract (-). When a predefined standard operator is applied to one or more abstract data values, the operator expression is implicitly transformed into a routine invocation. The effect is that the SQL predefined standard operators are overloaded to invoke ANGLE ADT operators as follows:

Prefix Plus	+u	invokesPlus(u)
Prefix Minus	-u	invokesMinus(u)
Infix Multiply	s*u	invokesMultiply(s,u) ⇒ specific LeftScalarMult(s,u)
Infix Multiply	u*s	invokesMultiply(u,s) ⇒ specific RightScalarMult(u,s)
Infix Add	u+v	invokesAdd(u,v)
Infix Subtract	u-v	invokesSubtract(u,v)

****Editor's Note 4-018****

Arithmetic for angles having different base precision.

5.6.2 C**omparison predicates**

Database Language SQL (March '95 SQL3 WD) defines the effect of EQUALS and LESS THAN ordering in the General Rules of Subclause 11.48, "<abstract data type body>". These two operators define the ordering to be used for all SQL comparison predicates (=, <>, <, >, <=, >=), the ORDER BY clause, the GROUP BY and HAVING clauses, and UNIQUE and DISTINCT comparison requirements. The effect is that the SQL comparison predicates are overloaded to invoke ANGLE ADT comparisons. If x and y are angles having the same BaseType, then:

u = v	invokes	IsEqual(u,v)
u < v	invokes	IsLessThan(u,v)
u <= v	invokes	NOT IsLessThan(v,u)
u > v	invokes	IsLessThan(v,u)
u <> v	invokes	NOT IsEqual(u,v)
u >= v	invokes	NOT IsLessThan(u,v)

****Editor's Note 4-018****

Comparison for angles having different base precision.

5.7 Conformance

An implementation may claim conformance to the ANGLE ADT in several different ways. The minimum requirements for support are as follows:

The implementation shall support ANGLE as a valid declaration of <data type> in any situation where an SQL <predefined type> is supported.

All functions defined in the angle ADT definition shall be callable as SQL-invoked routines in any situation where an SQL <value expression> of the RETURNS <data type> is supported.

The SQL special symbols for arithmetic operations, Plus (+), Minus (-), Add(+), Subtract (-), and Multiply (*), are overloaded to support Plus, Minus, Add, Subtract, and Scalar Multiply, respectively, for angles.

The SQL special symbols for comparison predicates (=, <, <=, <>, >, >=) are overloaded to support comparisons for angles. All other SQL clauses that depend upon the ordering of angles use the ordering implied by these comparisons.

An implementation may claim support for the functions defined in the ANGLE ADT if it effectively produces the specified result, without necessarily employing the specified algorithm.

In addition, an implementation may claim support for other values of BaseType in the ADT type template ANGLE (BaseType), or may support modification or enhancement of this ADT specification. Implementors may choose to support any combination of the following pre-defined conformance alternatives:

a) SQL/MM angle. Minimal support for ANGLE ADT as specified above.

b) SQL/MM double precision angle. Minimal SQL angle support plus support for ANGLE_DOUBLE as a valid declaration of <data type> in any situation where an SQL <predefined type> is supported.

c) SQL/MM multi-precision angle. Minimal SQL angle support plus support for ANGLE (BaseType), where BaseType is taken from any of the following built-in numeric types: REAL, FLOAT(p), DOUBLE_PRECISION. Support for the syntactic shorthand declarations: ANGLE, ANGLE_DOUBLE, ANGLE_PRECISION_20, and ANGLE_PRECISION_47.

d) SQL/MM angle type template. Minimal SQL angle support plus inclusion of type template descriptor information in the INFORMATION_SCHEMA as if the type template definition for ANGLE (BaseType) from any of the above supported items had been processed with authorization _SYSTEM, with schema name SQLMM_NUMERICS, and with the following privilege declaration: GRANT USAGE ON ANGLE (BaseType) TO PUBLIC.

e) SQL/MM angle subtyping. Minimal SQL angle support plus support for defining subtypes of supported ANGLE ADTs by allowing their specification in the <supertype clause> of any <abstract data type body>.

f) SQL/MM angle as ADT. Support for compilation and processing of the complete ADT <type template definition> for ANGLE (BaseType) as specified in Clause 5, "Angle Abstract Data Type", with arbitrary authorization instead of "_SYSTEM" authorization, and with implementation-supplied routines for any non-supported "Abs", "Sign", "Floor", or "Radian" routines in the ADT body.

5.8 Status Codes

Table 1 — SQLSTATE class and subclass values

Category	Condition	Class	Subcondition	Subclass
X	None	H4	None	F01

6 Complex Number Abstract Data Type

The algebraic field of complex numbers is very important in many engineering and physics applications. Since it is likely that several different Parts of SQL/MM will have a need for complex number arithmetic, it is best if the structure and methods of a COMPLEX data type are specified once, in this part, for subsequent use in other Parts.

This subclause specifes a complex number as having two components, a real component and an imaginary component. It permits the user to specify the base data type of these two parts as one of the existing SQL built-in data types, or possibly as some other existing ADT. The necessary requirements are that the base data type contain an additive and a multiplicative identity, support the arithmetic operations of addition, subtraction, multiplication, and division, and satisfy the associative, commutative, and distributive properties of an algebraic field. If a complex number data type is referenced without an explicit base type for each component, then the built-in data type REAL is assumed as the default.

6.1 Definitions

additive group: A term taken from classical mathematics to define a mathematical structure <G; +, ->, where G is a set of values, 0 is an identified value in G, + is a binary operator over G, and - is a unary operator over G, that satisfy the following axioms:

associative	1) x + (y + z) = (x + y) + z
commutative	2) x + y = y + x
identity	3) x + 0 = x
inverse	4) x + (-x) = 0

The following SQL data types are known to form additive groups under the addition operator, with negation as the additive inverse: NUMERIC, DECIMAL, INTEGER, SMALLINT, FLOAT, REAL, DOUBLE PRECISION.

field: A term taken from classical mathematics to define a mathematical structure <F; +, *, -, 1/>, where F is a set of values, 0 and 1 are identified values in F, + and * are binary operators over F, - is a unary operator over F, and 1/ is a unary operator over F (except 1/0 is not defined), that satisfy the following axioms:

associative	1) x+ (y + z) = (x + y) + z	x * (y * z) = (x * y) * z
commutative	2) x + y = y + x	x * y = y * x
identity	3) x + 0 = x	x * 1 = x
inverse	4) x + (-x) = 0	x * (1/x) = 1
distributive	5) x * (y + z) = x*y + x*z	

The following SQL data types are known to form a field under the addition and multiplication operators, with negation as the additive inverse and inversion as the multiplicative inverse: FLOAT, REAL, DOUBLE PRECISION. NUMERIC and DECIMAL data types often successfully model field operations, but the requirements for exact arithmetic (see Subclause 6.12, "<numeric value expression>", Syntax Rule 1 and General Rule 5, of SQL-1992) imply that these data types are not always closed under multiplication and division, possibly resulting in exception conditions if assigned to a variable of the same type.

6.2 Notations

To be supplied.

6.3 Conventions

To be supplied.

6.4 Concepts

The COMPLEX abstract data type template, COMPLEX(BaseType), models the mathematical field of complex numbers with varying degrees of precision. BaseType is the data type of both the real part and the imaginary part of the complex number. The family of generated complex number data types offers a choice of SQL primitive numeric types to use as the underlying type for BaseType. The following are supported in this specification: REAL, FLOAT, and DOUBLE PRECISION. If COMPLEX is specified without any template data type parameters for the underlying fields, then the resulting data type is COMPLEX(REAL). If COMPLEX_DOUBLE is specified without any template data type parameters for the underlying fields, then the resulting data type is COMPLEX(DOUBLE PRECISION).

None of the generated COMPLEX data types have any attributes. Instead, all operations on complex instances are through functions defined as part of the ADT definitions. These functions include:

Complex(x,y)	to create a new complex instance from real numbers x and y.
IsEqual(u,v)	to test equality of two complex numbers u and v returning BOOLEAN.
Plus(u)	to perform the identity operation on a complex number, returning +u
Add(u,v)	to add two complex numbers returning u + v.
Minus(u)	to find the additive inverse of a complex number returning -u.
Subtract(u,v)	to subtract two complex numbers returning u - v.
Multiply(u,v)	to multiply two complex numbers returning u * v.
Invert(u)	to find the multiplicative inverse of a complex number returning 1/u.
Divide(u,v)	to divide two complex numbers returning u / v.

****Editor's Note 4-033****

In SQL/MM LHR-013, Clause 1.8 Handling NULLs in component references identifies a problem with defining the functions RealPart and ImagPart with the same signature as the observer functions for the attributes of the same name. Similar problem with the Radian function in the ANGLE Type Template.

RealPart(u)	to return the real part of a complex number.
ImagPart(u)	to return the imaginary part of a complex number.
Conjugate(u)	to return the complex conjugate of a complex number.

Conjugate(x)	an identity function on a real number to return x.
Abs(u)	to find the absolute value of a complex number, returning a non-negative real number.
RealToComplex(x)	to cast a real number x to a complex number.
ToVarChar(u)	to cast a complex number to a variable length character string representation.

There is no LESS THAN comparison predicate function for complex numbers so it is not possible to order complex values or to invoke comparison predicates for inequalities. Any such attempt to order or compare for inequality of complex numbers will return an exception condition.

A complex variable or parameter may assume any of the valid SQL null values, but only for instances of the ADT itself. The complex number ADT definition is designed to prohibit null values for either the real part or the imaginary part of any non-null complex instance.

The SQL special operators for addition (+), subtraction or negation (-), multiplication (*), and division (/) are overloaded to support complex number arithmetic, with the same rules for precedence.

6.5 Complex Number Abstract Data Type

Function

The Complex Number type provides functions to construct and manipulate complex numbers.

Definition

```
CREATE TYPE TEMPLATE COMPLEX(BaseType TYPE)
   (PRIVATE RealPart BaseType DEFAULT 0,
    PRIVATE ImagPart BaseType DEFAULT 0,
```

****Editor's Note 4-037****

PRIVATE is no longer supported.

```
   FUNCTION Complex
      (realin  BaseType,
       imagin  BaseType)
      RETURNS COMPLEX(BaseType)
      LANGUAGE SQL
      BEGIN
         DECLARE z              COMPLEX(BaseType) DEFAULT Complex();
         IF realin IS NULL OR imagin IS NULL
            THEN RETURN NULL;
         END IF;
         SET z>>RealPart = realin;
         SET z>>ImagPart = imagin;
         RETURN z;
      END,
```

```
FUNCTION IsEqual
   (complex1      COMPLEX(BaseType),
    complex2      COMPLEX(BaseType))
   RETURNS BOOLEAN
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL OR complex2 IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN RealPart(complex1) = RealPart(complex2)
         AND ImagPart(complex1) = ImagPart(complex2));
   END,

FUNCTION Plus
   (complex1      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN complex1;
   END,

FUNCTION Add
   (complex1      COMPLEX(BaseType),
    complex2      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL OR complex2 IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN Complex(RealPart(complex1)+RealPart(complex2),
         ImagPart(complex1)+ImagPart(complex2));
   END,

FUNCTION Minus
   (complex1      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN Complex(-RealPart(complex1),-ImagPart(complex1));
   END,

FUNCTION Subtract
   (complex1      COMPLEX(BaseType),
    complex2      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN Add(complex1,Minus(complex2));
   END,
```

```
FUNCTION Multiply
   (complex1      COMPLEX(BaseType),
    complex2      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL OR complex2 IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN Complex(
         CAST(RealPart(complex1)*RealPart(complex2) -
         ImagPart(complex1)*ImagPart(complex2) AS BaseType),
         CAST(RealPart(complex1)*ImagPart(complex2) +
         RealPart(complex2)*ImagPart(complex1) AS BaseType));
   END,

FUNCTION Invert
   (complex1      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE ComplexZeroDivide EXCEPTION FOR SQLSTATE '22012';
      IF complex1 IS NULL
         THEN RETURN NULL;
      END IF;
      IF RealPart(complex1)=0 AND ImagPart(complex1)=0
         THEN SIGNAL ComplexZeroDivide;
      END IF;
      RETURN Complex(
         CAST(RealPart(complex1)/(RealPart(complex1)*
         RealPart(complex1)+ImagPart(complex1)*
         ImagPart(complex1)) AS BaseType),
         CAST(-ImagPart(complex1)/(RealPart(complex1)*
         RealPart(complex1)+ImagPart(complex1)*
         ImagPart(complex1)) AS BaseType));
      EXCEPTION
         WHEN ComplexZeroDivide THEN RETURN NULL;
   END,

FUNCTION Divide
   (complex1      COMPLEX(BaseType),
    complex2      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN Multiply(complex1,Invert(complex2));
   END,

FUNCTION RealPart
   (complex1      COMPLEX(BaseType))
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL
         THEN RETURN NULL;
         ELSE RETURN complex1.RealPart;
      END IF;
   END,
```

```
FUNCTION ImagPart
   (complex1      COMPLEX(BaseType))
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL
         THEN RETURN NULL;
         ELSE RETURN complex1.ImagPart;
      END IF;
   END,

FUNCTION Conjugate
   (complex1      COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      IF complex1 IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN Complex(RealPart(complex1),-ImagPart(complex1));
   END,

FUNCTION Conjugate
   (realval BaseType)
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      RETURN realval;
   END,

FUNCTION Abs
   (complex1      COMPLEX(BaseType))
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      DECLARE squarevalue     BaseType;
      IF complex1 IS NULL
         THEN RETURN NULL;
      END IF;
      SET squarevalue = RealPart(complex1)*RealPart(complex1)+
         ImagPart(complex1)*ImagPart(complex1);
      RETURN SQLMM_NUMERICS.Sqrt(squarevalue);
   END,

FUNCTION RealToComplex
   (realval BaseType)
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN Complex(realval,0);
   END,
```

```
      FUNCTION ToVarChar
         (complex1      COMPLEX(BaseType))
         RETURNS CHARACTER VARYING(53)
         LANGUAGE SQL
         BEGIN
            DECLARE xval            CHARACTER VARYING(25);
            DECLARE yval            CHARACTER VARYING(25);
            IF complex1 IS NULL
               THEN RETURN NULL;
            END IF;
            SET xval = CAST(RealPart(complex1) AS CHARACTER VARYING(25));
            SET yval = CAST(ImagPart(complex1) AS CHARACTER VARYING(25));
            RETURN '(' || xval || ',' || yval || ')';
         END,

      EQUALS IsEqual LESS THAN NONE,

      CAST(BaseType AS COMPLEX(BaseType) WITH RealToComplex),
      CAST(COMPLEX(BaseType) AS CHARACTER VARYING(53) WITH ToVarChar)
      )

   CREATE DISTINCT TYPE COMPLEX AS COMPLEX(REAL)

   CREATE DISTINCT TYPE COMPLEX_DOUBLE AS
      COMPLEX(DOUBLE PRECISION)
```

6.6 SQL Syntax Extensions

6.6.1 Arithmetic operations

Database Language SQL (March '95 SQL3 WD) identifies the following predefined standard operators in Subclause 4.16, "Operators": Plus (+), Minus (-), Multiply (*), Divide(/), Add(+), and Subtract (-). When a predefined standard operator is applied to one or more abstract data values, the operator expression is implicitly transformed into a routine invocation. The effect is that the SQL predefined standard operators are overloaded to invoke COMPLEX ADT operators as follows:

Prefix Plus	+u	invokes	Plus(u)
Prefix Minus	-u	invokes	Minus(u)
Infix Multiply	u*v	invokes	Multiply(u,v)
Infix Divide	u/v	invokes	Divide(u,v)
Infix Add	u+v	invokes	Add(u,v)
Infix Subtract	u-v	invokes	Subtract(u,v)

****Editor's Note 4-024****

Arithmetic for complex numbers having different base data types.

6.6.2 Comparison predicates

Database Language SQL (March '95 SQL3 WD) defines the effect of EQUALS and LESS THAN ordering in the General Rules of Subclause 11.48, "<abstract data type body>". These two operators define the ordering to be used for all SQL comparison predicates (=, <>, <, >, <=, >=), the ORDER BY clause, the GROUP BY and HAVING clauses, and UNIQUE and DISTINCT comparison requirements. The complex numbers have EQUALS defined, but not LESS THAN inequality. The effect is that only the SQL equals (=) and Not Equals (<>) symbols are overloaded, with appropriate extensions to the semantics of GROUP BY, HAVING, DISTINCT, and UNIQUE clauses. Any attempt to apply the SQL inequality predicates to complex numbers will result in an exception condition. If x and y are complex numbers having the same BaseType, then:

u = v	invokes	IsEqual(u,v)
u <> v	invokes	NOT IsEqual(u,v)

****Editor's Note 4-025****

Equality comparison for complex numbers having different base data types.

6.7 Conformance

An implementation may claim conformance to the SQL/MM ADT for complex numbers in several different ways. The minimum requirements for support are as follows:

The implementation shall support COMPLEX as a valid declaration of <data type> in any situation where an SQL <predefined type> is supported.

All functions defined in the complex number ADT definition shall be callable as SQL-invoked routines in any situation where an SQL <value expression> of the RETURNS <data type> is supported.

The SQL special symbols for arithmetic operations (+,-,*,/) are overloaded to support complex number arithmetic.

In addition, an implementation may claim support for other combinations of input parameters to the ADT type template COMPLEX(BaseType), or may support modification or enhancement of this ADT specification. Implementors may choose to support any combination of the following pre-defined conformance alternatives:

a) SQL/MM complex

   The minimal support for COMPLEX as specified above.

b) SQL/MM double precision complex

   Support for the following specific ADT definitions: COMPLEX, and COMPLEX_DOUBLE.

c) SQL/MM complex type template

   Support for COMPLEX(BaseType) with choices for BaseType from the following: REAL, FLOAT(p), or DOUBLE PRECISION. Inclusion of type template descriptor information in the INFORMATION_SCHEMA as if the type template definition for COMPLEX had been processed with authorization _SYSTEM and with the following privilege declaration: GRANT USAGE ON COMPLEX TO PUBLIC.

d) SQL/MM complex subtyping

   Support for defining subtypes of supported complex ADTs by allowing their specification in the <supertype clause> of any <abstract data type body>.

e) SQL/MM complex as ADT

   Support for compilation and processing of the complete ADT <type template definition> for COMPLEX(BaseType TYPE) as specified in Clause 5.5, "Complex Number Abstract Data Type".

7 Numerics Schema

This subclause defines numeric functions commonly implemented on a scientific calculator (e.g. exponentials, trig functions, and simple numerics). This subclause specifies a system owned schema, SQLMM_NUMERICS, that contains SQL functions to invoke a collection of elementary numeric functions. In each case, a numeric function accepts a numeric value as input and returns a numeric value of the same or greater precision. If the input is a null value, then the output is a null value. If the input is an invalid numeric value, then a standardized exception condition is returned.

7.1 Definitions

algebraic function: Any function f(x) that can be expressed as a root of a polynomial P(f(x),x) in two variables having rational coefficients. The algebraic functions include the polynomial functions, the rational functions, and algebraic expressions involving rational roots and powers. If a function is not algebraic, then it is said to be transcendental.

periodic function: A term from classical mathematics to identify a function f that satisfies the property f(x+p) = f(x) for all x in the domain of f, where p is a constant. The smallest such constant p is said to be the period of f.

rational function: Any function f that can be expressed as the quotient of two polynomial functions having rational coefficients.

7.2 Notations

To be supplied.

7.3 Conventions

To be supplied.

7.4 Concepts

7.4.1 Elementary numeric functions

Commonly used numeric functions on the real numbers include the absolute value function, negations, reciprocols, floor and ceiling functions, integer powers, and various helpful functions from number theory and probability. A collection of such SQL functions are defined in the NUMERICS schema for the convenience of SQL users.

In many cases, the returns data type of a numeric function has the same data type as the input value. In other cases, for example reciprocol, an exact numeric input value is operated on to produce an approximate numeric returns value. Each elementary numeric function returns a single output value that is a real number with precision greater or equal to the precision of each input real number. If any of the input values is a null value, then the output is a null value. Unless otherwise indicated, an elementary numeric function satisfies the following input and returns type constraints:

Table 2 — Input and Return Types for Elementary Numeric Functions

Real Input Type	Returns Type
SMALLINT	REAL
INTEGER	REAL
NUMERIC	REAL
DECIMAL	REAL
REAL	REAL
FLOAT(p)	FLOAT(p)
DOUBLE PRECISION	DOUBLE PRECISION

In each case, the returns value is accurate within the precision of the Returns Type.

Abs(x) Defined for all real values x. Returns the absolute value of x, with the same data type as the data type of x.

Ceiling(x) Defined for all real values x whose integer part can be expressed as an integer data type. If possible, returns the smallest integer that is greater than or equal to x; otherwise, an exception condition is raised: *invalid input value - ceiling*.

Exponent(x) Defined for all approximate numeric values. Returns a signed integer that specifies the magnitude of Mantissa(x). Satisfies the expression Sign(x)*Mantissa(x)*Power(10,Exponent(x)) as an approximation of x.

Floor(x) Defined for all real values x whose integer part can be expressed as an integer data type. If possible, returns the greatest integer that is less than or equal to x; otherwise, an exception condition is raised: *invalid input value - floor*.

Invert(x) Defined for all non-zero real values x. Returns 1/x, the multiplicative inverse of x. If x = 0, then an exception condition is raised: *invalid input value - invert*.

Mantissa(x) Defined for all approximate numeric values. Returns a non-negative integer. Satisfies the expression Sign(x)*Mantissa(x)*Power(10,Exponent(x)) as an approximation of x.

Mod(m,n) Defined for all integers m and for positive integers n. Returns the non-negative integer remainder when m is divided by n. If m = nk + r for some integer k, and for non-negative r satisfying r < n, then r is returned. If n = 0 or n < 0, then an exception condition is raised: *invalid input value - modulo*.

Minus(x) Defined for all real values x. Returns -x, the additive inverse of x, with the same data type as the data type of x.

Plus(x) Defined for all real values x. Returns +x, the identity operator on x, with the same data type as the data type of x.

Power(x,n) Defined for all real values x and all non-negative integers n, except not both equal to zero. Returns x^n, the n-th power of x, with the same data type as the data type of x. If x = 0 and n = 0, then an exception condition is raised: *invalid input value - power*.

Sign(x) Defined for all real values x. Returns +1 if x is positive, -1 if x is negative, and 0 if x is zero.

Sqrt(x) Defined for non-negative real values x. Returns the positive square root of x. If x < 0, then an exception condition is raised: *invalid input value - square root*.

As with any numeric function, the precision of the output can be controlled by specifying the precision of the input. For example, to get greater precision for Sqrt(2), one might specify Sqrt(CAST(2 AS DOUBLE PRECISION)) to obtain a returns value accurate within the precision of the DOUBLE PRECISION data type.

7.4.2 Combinatorial functions

Combinatorial functions for combinations, permutations, and factorials are commonly used in elementary probability and statistics applications. Both exact and approximate calculations of these functions are defined in the Numerics Schema for the convenience of SQL users. The exact functions return an INTEGER result and can be used when the input values are relatively small; they will produce overflow exceptions if the input values are too large. The approximate functions return a REAL or DOUBLE PRECISION value and can be used when an exact result is not needed; they may also produce overflow exceptions if the input values are too large. In all cases, if any one of the input values is the null value, then a null value is returned.

Fact(n) Calculates n-factorial. Defined for non-negative integers n, and returns an INTEGER value. If n is zero, then 1 is returned; otherwise, Fact(n) is defined recursively as Fact(n+1) = (n+1)* Fact(n). If n < 0, then an exception condition is raised: *invalid input value - factorial*.

FactX(n) Approximates n-factorial as a real number. Defined for non-negative integers n, and returns a REAL value. If n < 0, then an exception condition is raised: *invalid input value - factorial*.

FactXX(n) Approximates n-factorial as a double precision real number. Defined for non-negative integers n, and returns a DOUBLE PRECISION value. If n < 0, then an exception condition is raised: *invalid input value - factorial*.

Stirling(n) Approximates n-factorial as a double precision real number. For n < 8, defined to be Fact(n). For n >= 8, defined to be Sqrt(2*Radian(180)*n)* Power(n/Exp(1), n), and returns a DOUBLE PRECISION value. Discovered by James Stirling in 1730. If n < 0, then an exception condition is raised: *invalid input value - factorial*.

Perm(n,r) Calculates the permutations of n things taken r at a time. Defined for integers n and r satisfying n >= 0 and 0 <= r <= n, and returns an INTEGER value equal to Fact(n)/Fact(n-r). If n < 0, r < 0, or r > n, then an exception condition is raised: *invalid input value - permutations*.

PermX(n,r)	Approximates the permutations of n things taken r at a time as a real number. Defined for integers n and r satisfying n >= 0 and 0 <= r <= n, and returns a REAL value. If n < 0, r < 0, or r > n, then an exception condition is raised: *invalid input value - permutations*.
PermXX(n,r)	Approximates the permutations of n things taken r at a time as a double precision real number. Defined for integers n and r satisfying n >= 0 and 0 <= r <= n, and returns a DOUBLE PRECISION value. If n < 0, r < 0, or r > n, then an exception condition is raised: *invalid input value - permutations*.
Comb(n,r)	Calculates the combinations of n things taken r at a time. Defined for integers n and r satisfying n >= 0 and 0 <= r <= n, and returns an INTEGER value equal to Fact(n)/(Fact(r)*Fact(n-r)). If n < 0, r < 0, or r > n, then an exception condition is raised: *invalid input value - combinations*.
CombX(n,r)	Approximates the combinations of n things taken r at a time as a real number. Defined for integers n and r satisfying n >= 0 and 0 <= r <= n, and returns a REAL value. If n < 0, r < 0, or r > n, then an exception condition is raised: *invalid input value - combinations*.
CombXX(n,r)	Approximates the combinations of n things taken r at a time as a double precision real number. Defined for integers n and r satisfying n >= 0 and 0 <= r <= n, and returns a DOUBLE PRECISION value. If n < 0, r < 0, or r > n, then an exception condition is raised: *invalid input value - combinations*.

In a 16-bit INTEGER data type, Fact(8) may cause numeric overflow, but Comb(n,r) is safe up to n = 17 for all values of r. In a 32-bit INTEGER data type, Fact(13) may cause numeric overflow, but Comb(n,r) is safe up to n = 33 for all values of r. In a 64-bit INTEGER data type, Fact(18) may cause numeric overflow, but Comb(n,r) is safe up to n = 65 for all values of r.

7.4.3 Transcendental functions

In classical mathematics there is an important collection of functions that cannot be expressed as algebraic functions. Such functions are said to be transcendental. They include all exponential and trigonometric functions defined over real or complex variables. The most basic of these is the exponential functon, denoted by e^z or Exp(z), which is defined as the power series

$$1 + z/1! + z^2/2! + \ldots + z^n/n! + \ldots .$$

It is well known that this series converges for all real and complex values of z. It is also known that a special case of this function, Exp(iz), where i = Complex(0,1), is a periodic function with period 2*pi*, where *pi* is a real constant equal to the circumference of a circle with unit diameter. In addition, it can be shown that for every complex value, z, there exists a unique real number, Arg(z), satisfying 0 <= Arg(z) < 2*pi*, such that z can be represented as Abs(z) * Exp(i*Arg(z)).

Using tools from elementary calculus, if z=Complex(x,y), then Arg(z) can be calculated from the following expression:

$$\text{Arg}(z) = \begin{cases} 0 & \text{if } y = 0 \text{ and } x >= 0 \\ pi & \text{if } y = 0 \text{ and } x < 0 \\ \text{Cot}^{-1}(x/y) & \text{if } y > 0 \\ pi + \text{Cot}^{-1}(x/y) & \text{if } y < 0 \end{cases}$$

There are many other ways to calculate Arg(z), some undoubtedly much more efficient. The remainder of this specification assumes the existence of a means to calculate this function to a degree of precision greater or equal to the precision of the input real variables x and y.

It can be shown that Exp(z) has an inverse function, Log(z). The calculation of Log(z) depends upon the calculation of the natural logarithm of a positive real number. If x is a real number, then e^x is a positive, monotonically increasing function, and it's inverse function, ln x, exists and is defined for all positive real values x. Using tools from elementary calculus, it follows that ln x can be calculated, with arbitrary precision, by various numerical methods. The remainder of this specification assumes the existence of a means to calculate ln x to a degree of precision greater or equal to the precision of the input real variable x.

From the theory of complex variables, it follows that the inverse exponential function, Log(z), exists for all non-zero complex values z, i.e. except for z = Complex(0,0), and satisfies the expression

Log(z) = ln(AbsoluteValue(z)) + i Arg(z)

All of the remaining transcendental exponential and trigonometric functions of real or complex variables can be expressed as algebraic functions of Exp(z) or Log(z). SQL syntax for invoking all of the exponential, trigonometric, and hyperbolic transcendental functions is presented in subsequent subclauses.

All of the Transcendental functions have a single input value and a single output value. If the input is a null value, then the output is a null value. All Transcendental functions satisfy the following input and output type constraints:

Table 3 — Input and Output Types for Transcendental Functions

Input Type	Returns Type
SMALLINT	REAL
INTEGER	REAL
NUMERIC	REAL
DECIMAL	REAL
REAL	REAL
FLOAT(p)	FLOAT(p)
DOUBLE PRECISION	DOUBLE PRECISION
COMPLEX	COMPLEX
COMPLEX_DOUBLE	COMPLEX_DOUBLE
COMPLEX(Float(p))	COMPLEX(Float(p))

In each case, the returns value is accurate within the precision of the Returns Type.

7.4.3.1 Exponential functions

The following functions are defined as SQL invokable functions:

Arg(z) Defined for all real and complex values. If z is real, then z is implicitly recast as complex. Returns the argument of the complex number z.

Exp(z) Defined for all real and complex values z. Returns the result of the power series defined above. If z is real, returns a positive real value.

Log(z) Defined for positive real values and non-zero complex values. Returns ln z if z is real and Complex(ln(AbsoluteValue(z)), Arg(z)) if z is complex. If z is a negative real value, or if z = 0, then an exception condition is raised: *invalid input value - logarithm*.

7.4.3.2 Trigonometric functions

The following trigonometric and inverse trigonometric functions are defined as SQL invokable functions.

Radian(x) Defined for all real values. The value x is interpreted as the measurment of an angle in degrees. Returns the real radian approximation of x as (2*pi*/360) times x. The precision of the result is equal to or greater than the precision of x.

Sin(z) Defined for all real and complex values z. Returns $(e^{iz} - e^{-iz})/2i$. If z is real, returns a real value between -1 and 1.

Cos(z) Defined for all real and complex values z. Returns $(e^{iz} + e^{-iz})/2$. If z is real, returns a real value between -1 and 1.

Tan(z) Returns Sin(z)/Cos(z). If Cos(z) is zero, then an exception condition is raised: *invalid input value - tangent*.

Sec(z) Returns 1/Cos(z). If Cos(z) is zero, then an exception condition is raised: *invalid input value - secant*.

Csc(z) Returns 1/Sin(z). If Sin(z) is zero, then an exception condition is raised: *invalid input value - cosecant*.

Cot(z) Returns Cos(z)/Sin(z). If Sin(z) is zero, then an exception condition is raised: *invalid input value - cotangent*.

ArcSin(z) Defined for real values between -1 and 1, and defined for all complex values. Returns -iLog(iz + Sqrt(1-z^2). For real input returns a value between -*pi*/2 and +*pi*/2. If z is real with z > 1 or z < -1, then an exception condition is raised: *invalid input value - inverse sine*.

ArcCos(z) Defined for real values between -1 and 1, and defined for all complex values. Returns -iLog(z + Sqrt(z^2-1). For real input returns a value between 0 and *pi*. If z is real with z > 1 or z < -1, then an exception condition is raised: *invalid input value - inverse cosine*.

ArcTan(z) Defined for all real values, and defined for all complex values except z = i or z = -i. Returns (i/2)Log((i+z)/(i-z)). For real input returns a value between -*pi*/2 and +*pi*/2. If z is complex with z = i or z = -i, then an exception condition is raised: *invalid input value - inverse tangent*.

ArcSec(z) Defined for real values z >= 1 or z <= -1, and defined for all non-zero complex values. Returns -iLog(1/z + Sqrt(1/z^2 - 1)). For real input returns a value between 0 and *pi*. If z is real with -1 < z < 1, or if z is complex with z = 0, then an exception condition is raised: *invalid input value - inverse secant*.

ArcCsc(z) Defined for real values z >= 1 or z <= -1, and defined for all non-zero complex values. Returns -iLog(1/z + Sqrt(1 - 1/z^2)). For real input returns a value between -*pi*/2 and +*pi*/2. If z is real with -1 < z < 1, or if z is complex with z = 0, then an exception condition is raised: *invalid input value - inverse cosecant*.

ArcCot(z) Defined for all real values, and defined for all complex values except z = i or z = -i. Returns (-i/2)Log((iz-1)/(iz+1)). For real input returns a value between 0 and *pi*. If z is complex with z = i or z = -i, then an exception condition is raised: *invalid input value - inverse cotangent*.

7.4.3.3 Hyperbolic functions

The following hyperbolic and inverse hyperbolic functions are defined as SQL invokable functions.

Sinh(z) Defined for all real and complex values z. Returns (e^z - e^{-z})/2.

Cosh(z) Defined for all real and complex values z. Returns (e^z + e^{-z})/2. For real input returns a real value >= 1.

Tanh(z) Returns Sinh(z)/Cosh(z). Cosh(z) is never zero. For real input returns a real value between -1 and +1.

Sech(z) Returns 1/Cosh(z). Cosh(z) is never zero. For real input returns a real value between 0 and 1.

Csch(z) Returns 1/Sinh(z). If Sinh(z) is zero, then an exception condition is raised: *invalid input value - hyperbolic cosecant*. For real input returns a non-zero real value.

Coth(z) Returns Cosh(z)/Sinh(z). If Sinh(z) is zero, then an exception condition is raised: *invalid input value - hyperbolic cotangent*. For real input returns a real value having absolute value >= 1.

InvSinh(z) Defined for all real and complex values z. Returns Log(z + Sqrt(z^2+1)). For real input, the output ranges over the entire real line.

InvCosh(z) Defined for real values z >= 1, and defined for all complex values. Returns Log(z + Sqrt(z^2-1)). For real input returns a non-negative real value. If z is real with z < 1, then an exception condition is raised: *invalid input value - inverse hyperbolic cosine*.

InvTanh(z) Defined for real values > -1 and < +1, and defined for all complex values except z = 1 and z = -1. Returns (1/2)Log((1+z)/(1-z)). For real input returns values that range over the entire real line. If z is real with -1 <= z <= 1, then an exception condition is raised: *invalid input value - inverse hyperbolic tangent*.

InvSech(z) Defined for real values > 0 and <= 1, and defined for all non-zero complex values. Returns Log(1/z + Sqrt($1/z^2$-1)). For real input returns a non-negative real value. If z is a real value with z < 0 or z > 1, or if z = 0, then an exception condition is raised: *invalid input value - inverse hyperbolic secant*.

InvCsch(z) Defined for all non-zero real and complex values. Returns Log(1/z + Sqrt($1/z^2$+1)). For real input returns a non-zero real value. If z = 0, then an exception condition is raised: *invalid input value - inverse hyperbolic cosecant*.

InvCoth(z) Defined for real values > 1 and < -1, and defined for all complex values except z =1 and z = -1. Returns (1/2)Log((z+1)/(z-1)). For real input returns a non-zero real value. If z is a real value with -1 < z < 1, or if z = 1 or z = -1, then an exception condition is raised: *invalid input value - inverse hyperbolic cotangent*.

7.4.3.4 Properties of transcendental functions

The exponential, trigonometric, and hyperbolic functions are all derived from algebraic expressions in e^z; it follows that these functions are closely related and satisfy a number of algebraic identities. The following identities are well known from classical mathematics. Unless otherwise specified, they are valid for both real and complex, non-null variables. If null input variables are considered, then some identities, e.g. (1) and (7) below, fail.

1) $\sin^2 z + \cos^2 z = 1$

2) $\sin(z_1 + z_2) = \sin z_1 \cos z_2 + \cos z_1 \sin z_2$

3) $\cos(z_1 + z_2) = \cos z_1 \cos z_2 - \sin z_1 \sin z_2$

4) $\sin(-z) = -\sin z$

5) $\cos(-z) = \cos z$

6) $\cos z = \sin(pi/2 - z)$

7) $\cosh^2 z - \sinh^2 z = 1$

8) $\sinh(z_1 + z_2) = \sinh z_1 \cosh z_2 + \cosh z_1 \sinh z_2$

9) $\cosh(z_1 + z_2) = \cosh z_1 \cosh z_2 + \sinh z_1 \sinh z_2$

10) $\sinh(-z) = -\sinh z$

11) $\cosh(-z) = \cosh z$

12) $\sinh(iz) = i \sin z$

13) $\cosh(iz) = \cos z$

14) $\sin(iz) = i \sinh z$

15) $\cos(iz) = \cosh z$

16) $\sinh(x + iy) = \sinh x \cos y + i \cosh x \sin y$, for x and y as real values.

17) $\cosh(x + iy) = \cosh x \cos y + i \sinh x \sin y$, for x and y as real values.

18) $|\sinh(x + iy)|^2 = \sinh^2 x + \sin^2 y$, for x and y as real values.

19) $|\cosh(x + iy)|^2 = \sinh^2 x + \cos^2 y$, for x and y as real values.

20) $\text{Exp}(z) = \text{Exp}(x)*(\text{Cos}(y) + i\,\text{Sin}(y))$, for $z = x + iy$.

21) $\text{Exp}(\text{Conjugate}(z)) = \text{Conjugate}(\text{Exp}(z))$

22) $\text{Exp}(z + i\,2pi) = \text{Exp}(z)$

23) $\text{Log}(z_1 * z_2) = \text{Log}\, z_1 + \text{Log}\, z_2$

24) $\text{Log}(z_1/z_2) = \text{Log}\, z_1 - \text{Log}\, z_2$

7.5 Numerics Schema

Function

The Numerics schema provides numeric functions.

Definition

```
CREATE SCHEMA SQLMM_NUMERICS
   AUTHORIZATION "_SYSTEM"
   DEFAULT CHARACTER SET SQL_CHARACTER

   --    The following functions are implementation-dependent
   --    functions to return real and complex exponential and
   --    square root, truncation, natural log, and argument,
   --    with a specific precision.
   --    If precision is expressed as a positive integer, p,
   --    then the results of each of these functions is accurate
   --    within binary precision p.  No permissions are granted on
   --    these functions, so they are not directly available to
   --    non_SYSTEM users.
```

****Editor's Note 4-034****

The functions: TruncateToInteger, RealExp, RealLog, RealSqrt, ComplexExp and ComplexArg are defined with implementation dependent functions or by using superscripts in the PSM code.

```
   CREATE FUNCTION TruncateToInteger(x DOUBLE_PRECISION)
      RETURNS INTEGER
      LANGUAGE SQL
      BEGIN
         IF x IS NULL
            THEN RETURN NULL;
         END IF;
         RETURN truncate_to_integer(x);
         -- Note: May cause a numeric overflow exception.
      END

   CREATE FUNCTION RealExp(x DOUBLE_PRECISION, p SMALLINT)
      RETURNS DOUBLE_PRECISION
      LANGUAGE SQL
      BEGIN
         IF x IS NULL
            THEN RETURN NULL;
         END IF;
         RETURN e^x;
         -- accurate within precision p, for p within system limits.
      END
```

```
CREATE FUNCTION RealLog(x DOUBLE_PRECISION, p SMALLINT)
   RETURNS DOUBLE_PRECISION
   LANGUAGE SQL
   BEGIN
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN ln x;
      -- accurate within precision p, for p within system limits.
   END

CREATE FUNCTION RealSqrt(x DOUBLE_PRECISION, p SMALLINT)
   RETURNS DOUBLE_PRECISION
   LANGUAGE SQL
   BEGIN
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN square_root(x);
      -- accurate within precision p, for p within system limits.
   END

CREATE FUNCTION ComplexExp(z COMPLEX_DOUBLE, p SMALLINT)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      IF z IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN e^z;
      -- accurate within precision p, for p within system limits.
   END

CREATE FUNCTION ComplexArg(z COMPLEX_DOUBLE, p SMALLINT)
   RETURNS DOUBLE_PRECISION
   LANGUAGE SQL
   BEGIN
      IF z IS NULL
         THEN RETURN NULL;
      END IF;
      RETURN argument(z);
      -- accurate within precision p, for p within system limits.
   END
```

```
--    The following ADT for NUMBER, and its included functions,
--    are implementation-dependent facilities for identifying
--    data types and precisions of input variables to numeric
--    functions.
--    No permissions are granted on these functions, so they
--    are not directly available to non_SYSTEM users.
```

****Editor's Note 4-011****

In following the instructions of SOU-010 Section 3.1.3, I have deleted the WITHOUT OID clause from the NUMBER TYPE TEMPLATE and added an <abstract data type value> with an <abstract data type name> of NUMBERNAME. It is not clear to me what the <abstract data type name> should actually be. A similar change was made for other TYPE TEMPLATEs in this part.

Type Name with Type Templates. The Working Draft SQL3 (March '95) productions for <type template definition> (Part 2, Subclause 11.50) and <abstract data type body> (Subclause 11.48) are inconsistent in that a <type template name> and an <abstract data type name> are both required. Only one type name is needed! This error was probably introduced by the Southampton (July '94) paper (DBL SOU-xxx/X3H2-94-xxx) that restructured VALUE and OBJECT declarations to be part of <abstract data type body>, but did not address the effect of these changes on the <type template definition>.

```
CREATE TYPE TEMPLATE NUMBER(InputType TYPE)
   (FUNCTION GetType(intype InputType)
      RETURNS CHARACTER(18)
      LANGUAGE SQL
      BEGIN
         RETURN type_name(intype);
         -- Just returns the name, not template attributes.
      END,

   FUNCTION GetBaseType(intype InputType)
      RETURNS CHARACTER(18)
      LANGUAGE SQL
      BEGIN
         RETURN type_name(BaseType(intype));
      END,

   FUNCTION GetBinPrec(intype InputType)
      RETURNS SMALLINT
      LANGUAGE SQL
      BEGIN
         RETURN binary_precision(intype);
      END,

   FUNCTION GetDecPrec(intype InputType)
      RETURNS SMALLINT
      LANGUAGE SQL
      BEGIN
         RETURN decimal_precision(intype);
      END,

   FUNCTION GetMaxValue(intype InputType)
      RETURNS InputType
      LANGUAGE SQL
      BEGIN
         RETURN maximun_representable_value(intype);
      END,
```

```
FUNCTION GetMinValue(intype InputType)
   RETURNS InputType
   LANGUAGE SQL
   BEGIN
      RETURN minimum_representable_value(intype);
   END
)

--    The following family of functions for the real and
--    complex elementary functions handle all possible
--    combinations of input data types. The returns data type,
--    including its precision, is dependent upon the input
--    data type.

CREATE FUNCTION Abs(x SMALLINT)
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END

CREATE FUNCTION Abs(x INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END

CREATE FUNCTION Abs(x NUMERIC(p,s))
   RETURNS NUMERIC(p,s)
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END

CREATE FUNCTION Abs(x DECIMAL(p,s))
   RETURNS DECIMAL(p,s)
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END
```

```
CREATE FUNCTION Abs(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END

CREATE FUNCTION Abs(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END

CREATE FUNCTION Abs(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      IF x >= 0
         THEN RETURN x;
         ELSE RETURN -x;
      END IF;
   END

CREATE FUNCTION Ceiling(x DOUBLE PRECISION)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F01';
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      IF Abs(x) > GetMaxValue(INTEGER)
         THEN SIGNAL InvalidInput;
      END IF;
      IF x = TruncateToInteger(x)
         THEN RETURN x;
      END IF;
      IF x >= 0
         THEN RETURN TruncateToInteger(x) + 1;
         ELSE RETURN -TruncateToInteger(-x);
      END IF;
   END
```

```
CREATE FUNCTION Exponent(x DOUBLE PRECISION)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE y               DOUBLE PRECISION;
      DECLARE I               SMALLINT DEFAULT 0;
      DECLARE MaxInt          INTEGER DEFAULT GetMaxValue(INTEGER);
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      IF x = 0
         THEN RETURN 0;
      END IF;
      SET y = Abs(x);
      IF y > MaxInt
         THEN
            WHILE y > MaxInt
               DO
                  SET y = y/10;
                  SET I = I + 1;
               END WHILE;
            RETURN I;
      END IF;
      IF y > 1
         THEN
            SET y = y*Power(10,GetDecPrec(INTEGER));
            WHILE y > MaxInt
               DO
                  SET y = y/10;
                  SET I = I + 1;
               END WHILE;
            RETURN I - GetDecPrec(INTEGER);
      END IF;
      IF y < 1
         THEN
            WHILE y < 0.1
               DO
                  SET y = 10 * y;
                  SET I = I + 1;
               END WHILE;
            SET y = y*Power(10,GetDecPrec(INTEGER));
            -- Note: See definition of Mantissa(x)
            RETURN -(I + GetDecPrec(INTEGER));
      END IF;
   END
```

```
CREATE FUNCTION Floor(x DOUBLE PRECISION)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F02';
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      IF Abs(x) > GetMaxValue(INTEGER)
         THEN SIGNAL InvalidInput;
      END IF;
      IF x = TruncateToInteger(x)
         THEN RETURN x;
      END IF;
      IF x >= 0
         THEN RETURN TruncateToInteger(x);
         ELSE RETURN -TruncateToInteger(-x) - 1;
      END IF;
   END

CREATE FUNCTION Invert(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE RealZeroDivide  EXCEPTION FOR SQLSTATE '22012';
      IF x = 0
         THEN SIGNAL RealZeroDivide;
      END IF;
      RETURN 1/x;
   END

CREATE FUNCTION Invert(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE RealZeroDivide  EXCEPTION FOR SQLSTATE '22012';
      IF x = 0
         THEN SIGNAL RealZeroDivide;
      END IF;
      RETURN 1/x;
   END

CREATE FUNCTION Invert(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE RealZeroDivide  EXCEPTION FOR SQLSTATE '22012';
      IF x = 0
         THEN SIGNAL RealZeroDivide;
      END IF;
      RETURN 1/x;
   END
```

```
CREATE FUNCTION Mantissa(x DOUBLE PRECISION)
RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE y               DOUBLE PRECISION;
      DECLARE I               SMALLINT DEFAULT 0;
      DECLARE MaxInt          INTEGER DEFAULT GetMaxValue(INTEGER);
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      IF x = 0
         THEN RETURN 0;
      END IF;
      SET y = Abs(x);
      IF y > MaxInt
         THEN
            WHILE y > MaxInt
               DO
                  SET y = y/10;
                  SET I = I + 1;
               END WHILE;
            RETURN TruncateToInteger(y);
      END IF;
      IF y > 1
         THEN
            SET y = y*Power(10,GetDecPrec(INTEGER));
            WHILE y > MaxInt
               DO
                  SET y = y/10;
                  SET I = I + 1;
               END WHILE;
            RETURN TruncateToInteger(y);
      END IF;
      IF y < 1
         THEN
            WHILE y < 0.1
               DO
                  SET y = 10 * y;
                  SET I = I + 1;
               END WHILE;
            SET y = y*Power(10,GetDecPrec(INTEGER));
            RETURN TruncateToInteger(y);
      END IF;
   END

CREATE FUNCTION Mod(m SMALLINT, n SMALLINT)
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR 'H4F03';
      IF m IS NULL OR n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n <= 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN m - n*Floor(m/n);
   END
```

```
CREATE FUNCTION Mod(m INTEGER, n INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR 'H4F03';
      IF m IS NULL OR n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n <= 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN m - n*Floor(m/n);
   END

CREATE FUNCTION Minus(x SMALLINT)
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END

CREATE FUNCTION Minus(x INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END

CREATE FUNCTION Minus(x NUMERIC(p,s))
   RETURNS NUMERIC(p,s)
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END

CREATE FUNCTION Minus(x DECIMAL(p,s))
   RETURNS DECIMAL(p,s)
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END

CREATE FUNCTION Minus(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END

CREATE FUNCTION Minus(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END
```

```
CREATE FUNCTION Minus(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      RETURN -x;
   END

CREATE FUNCTION Plus(x SMALLINT)
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,

CREATE FUNCTION Plus(x INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,

CREATE FUNCTION Plus(x NUMERIC(p, s))
   RETURNS NUMERIC(p, s)
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,

CREATE FUNCTION Plus(x DECIMAL(p, s))
   RETURNS DECIMAL(p, s)
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,

CREATE FUNCTION Plus(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,

CREATE FUNCTION Plus(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,

CREATE FUNCTION Plus(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      RETURN x;
   END,
```

```
CREATE FUNCTION Power(x INTEGER, n INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE y               INTEGER DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F04';
      IF x IS NULL OR n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x <> 0
         THEN RETURN 1;
      END IF;
      WHILE I <= n
         DO
            SET y = y * x;
            SET I = I + 1;
         END WHILE;
      RETURN y;
   END

CREATE FUNCTION Power(x REAL, n INTEGER)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE y               INTEGER DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F04';
      IF x IS NULL OR n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x <> 0
         THEN RETURN 1;
      END IF;
      WHILE I <= n
         DO
            SET y = y * x;
            SET I = I + 1;
         END WHILE;
      RETURN y;
   END
```

```
CREATE FUNCTION Power(x FLOAT(p), n INTEGER)
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE y               INTEGER DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F04';
      IF x IS NULL OR n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x <> 0
         THEN RETURN 1;
      END IF;
      WHILE I <= n
         DO
            SET y = y * x;
            SET I = I + 1;
         END WHILE;
      RETURN y;
   END

CREATE FUNCTION Power(x DOUBLE PRECISION, n INTEGER)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE y               INTEGER DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F04';
      IF x IS NULL OR n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n = 0 and x <> 0
         THEN RETURN 1;
      END IF;
      WHILE I <= n
         DO
            SET y = y * x;
            SET I = I + 1;
         END WHILE;
      RETURN y;
   END
```

```
CREATE FUNCTION Sign(x DOUBLE PRECISION)
   RETURNS SMALLINT
   LANGUAGE SQL
   BEGIN
      IF x IS NULL
         THEN RETURN NULL;
      END IF;
      IF x = 0
         THEN RETURN 0;
      ELSE IF x > 0
         THEN RETURN +1;
         ELSE RETURN -1;
      END IF;
   END

   -- The following functions calculate factorials,
   -- permutations, and combinations when
   -- the input parameters are non-negative integers
   -- satisfying certain input restrictions.

CREATE FUNCTION Fact(n INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE fact            INTEGER DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 2;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F06';
      IF n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      WHILE I <= n
         DO
            SET fact = fact * I;
            SET I = I + 1;
         END WHILE;
      RETURN fact;
   END
```

```
CREATE FUNCTION FactX(n INTEGER)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE fact            REAL DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 2;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F06';
      IF n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      WHILE I <= n
         DO
            SET fact = fact * I;
            SET I = I + 1;
         END WHILE;
      RETURN fact;
   END

CREATE FUNCTION FactXX(n INTEGER)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE fact            DOUBLE PRECISION DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 2;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F06';
      IF n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      WHILE I <= n
         DO
            SET fact = fact * I;
            SET I = I + 1;
         END WHILE;
      RETURN fact;
   END

CREATE FUNCTION Stirling(n INTEGER)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F06';
      IF n IS NULL
         THEN RETURN NULL;
      END IF;
      IF n < 0
         THEN SIGNAL InvalidInput;
      END IF;
      IF n < 8
         THEN RETURN Fact(n);
      END IF;
      RETURN Sqrt(2*Radian(CAST(180 AS DOUBLE PRECISION))*n) *
         Power(n/Exp(CAST(1 AS DOUBLE PRECISION)), n);
   END
```

```
CREATE FUNCTION Perm(n INTEGER, r INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE perm            INTEGER DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F07';
      IF n IS NULL OR r IS NULL
         THEN RETURN NULL;
      END IF;
      IF n<0 OR r<0 OR r>n
         THEN SIGNAL InvalidInput;
      END IF;
      WHILE I <= r
         DO
            SET perm = perm*(n-r+I);
            SET I = I + 1;
         END WHILE;
      RETURN perm;
   END

CREATE FUNCTION PermX(n INTEGER, r INTEGER)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE perm            REAL DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F07';
      IF n IS NULL OR r IS NULL
         THEN RETURN NULL;
      END IF;
      IF n<0 OR r<0 OR r>n
         THEN SIGNAL InvalidInput;
      END IF;
      WHILE I <= r
         DO
            SET perm = perm*(n-r+I);
            SET I = I + 1;
         END WHILE;
      RETURN perm;
   END
```

```
CREATE FUNCTION PermXX(n INTEGER, r INTEGER)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE perm            DOUBLE PRECISION DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F07';
      IF n IS NULL OR r IS NULL
         THEN RETURN NULL;
      END IF;
      IF n<0 OR r<0 OR r>n
         THEN SIGNAL InvalidInput;
      END IF;
      WHILE I <= r
         DO
            SET perm = perm*(n-r+I);
            SET I = I + 1;
         END WHILE;
      RETURN perm;
   END
```

```
CREATE FUNCTION Comb(n INTEGER, r INTEGER)
   RETURNS INTEGER
   LANGUAGE SQL
   BEGIN
      DECLARE comb1           INTEGER;
      DECLARE comb2           INTEGER;
      DECLARE I               INTEGER DEFAULT 0;
      DECLARE J               INTEGER DEFAULT 1;
      DECLARE K               INTEGER DEFAULT r;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F08';
      IF n IS NULL OR r IS NULL
         THEN RETURN NULL;
      END IF;
      IF n<0 OR r<0 OR r>n
         THEN SIGNAL InvalidInput;
      END IF;
      IF n-r < r
         THEN SET K = n-r;
      END IF;
      DECLARE TEMPORARY
         TABLE PascalTriangle(n_value INTEGER, r_value INTEGER,
            Comb INTEGER);
      WHILE I <= n-K
         DO
            INSERT INTO PascalTriangle VALUES (I, 0, 1);
            SET I = I + 1;
         END WHILE;
      WHILE J <= K
         DO
            INSERT INTO PascalTriangle VALUES (J, J, 1);
            SET J = J + 1;
         END WHILE;
      SET I = 1;
      SET J = 1;
      WHILE I <= n-K
         DO
            WHILE J <= K
               DO
                  SET comb1 = (SELECT Comb FROM PascalTriangle
                     WHERE n_value = I+J-1 AND r_value = J-1);
                  SET comb2 = (SELECT Comb FROM PascalTriangle
                     WHERE n_value = I+J-1 AND r_value = J);
                  INSERT INTO PascalTriangle
                     VALUES (I+J, J, comb1+comb2);
                  SET J = J + 1;
               END WHILE;
            SET I = I + 1;
         END WHILE;
      RETURN (SELECT Comb FROM PascalTriangle
         WHERE n_value = n AND r_value = K);
   END
```

```
CREATE FUNCTION CombX(n INTEGER, r INTEGER)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE comb            REAL DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE K               INTEGER DEFAULT r;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F08';
      IF n IS NULL OR r IS NULL
         THEN RETURN NULL;
      END IF;
      IF n<0 OR r<0 OR r>n
         THEN SIGNAL InvalidInput;
      END IF;
      IF n-r < r
         THEN SET K = n-r;
      END IF;
      WHILE I <= K
         DO
            SET comb = comb*(n-K+I)/I;
            SET I = I + 1;
         END WHILE;
      RETURN comb;
   END

CREATE FUNCTION CombXX(n INTEGER, r INTEGER)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE comb            DOUBLE PRECISION DEFAULT 1;
      DECLARE I               INTEGER DEFAULT 1;
      DECLARE K               INTEGER DEFAULT r;
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F08';
      IF n IS NULL OR r IS NULL
         THEN RETURN NULL;
      END IF;
      IF n<0 OR r<0 OR r>n
         THEN SIGNAL InvalidInput;
      END IF;
      IF n-r < r
         THEN SET K = n-r;
      END IF;
      WHILE I <= K
         DO
            SET comb = comb*(n-K+I)/I;
            SET I = I + 1;
         END WHILE;
      RETURN comb;
   END

CREATE FUNCTION Sqrt(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F05';
      IF x < 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealSqrt(x, GetPrec(REAL));
   END
```

```
CREATE FUNCTION Sqrt(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F05';
      IF x < 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealSqrt(x, GetPrec(FLOAT(p)));
   END

CREATE FUNCTION Sqrt(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F05';
      IF x < 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealSqrt(x, GetPrec(DOUBLE PRECISION));
   END

--    The following family of functions for Exp and Log handle
--    all possible combinations of input data types. The
--    returns data type, including its precision, is dependent
--    upon the input data type.

CREATE FUNCTION Arg(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      RETURN ComplexArg(Complex(x,0), GetPrec(REAL));
   END

CREATE FUNCTION Arg(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      RETURN ComplexArg(Complex(x,0), GetPrec(FLOAT(p)));
   END

CREATE FUNCTION Arg(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      RETURN ComplexArg(Complex(x,0), GetPrec(DOUBLE PRECISION));
   END

CREATE FUNCTION Arg(z COMPLEX)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      RETURN ComplexArg(z, GetPrec(REAL));
   END
```

```
CREATE FUNCTION Arg(z COMPLEX_DOUBLE)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      RETURN ComplexArg(z, GetPrec(DOUBLE PRECISION));
   END

CREATE FUNCTION Arg(z COMPLEX(BaseType))
   RETURNS BaseType
   LANGUAGE SQL
   BEGIN
      RETURN ComplexArg(z, GetPrec(BaseType));
   END

CREATE FUNCTION Exp(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      RETURN RealExp(x, GetPrec(REAL));
   END

CREATE FUNCTION Exp(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      RETURN RealExp(x, GetPrec(FLOAT(p)));
   END

CREATE FUNCTION Exp(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      RETURN RealExp(x, GetPrec(DOUBLE PRECISION));
   END

CREATE FUNCTION Exp(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      RETURN ComplexExp(z, GetPrec(REAL));
   END

CREATE FUNCTION Exp(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      RETURN ComplexExp(z, GetPrec(DOUBLE PRECISION));
   END

CREATE FUNCTION Exp(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN ComplexExp(z, GetPrec(BaseType));
   END
```

```
CREATE FUNCTION Log(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE NegativeReal    EXCEPTION FOR SQLSTATE 'H4F09';
      IF x <= 0
         THEN SIGNAL NegativeReal;
      END IF;
      RETURN RealLog(x, GetPrec(REAL));
   END

CREATE FUNCTION Log(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE NegativeReal    EXCEPTION FOR SQLSTATE 'H4F09';
      IF x <= 0
         THEN SIGNAL NegativeReal;
      END IF;
      RETURN RealLog(x, GetPrec(FLOAT(p)));
   END

CREATE FUNCTION Log(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE NegativeReal    EXCEPTION FOR SQLSTATE 'H4F09';
      IF x <= 0
         THEN SIGNAL NegativeReal;
      END IF;
      RETURN RealLog(x, GetPrec(DOUBLE PRECISION));
   END

CREATE FUNCTION Log(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F09';
      IF z = Complex(0,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN COMPLEX(RealLog(Abs(z), GetPrec(REAL)),
         ComplexArg(z, GetPrec(REAL)));
   END

CREATE FUNCTION Log(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F09';
      IF z = Complex(0,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN COMPLEX(RealLog(Abs(z),GetPrec(DOUBLE PRECISION)),
         ComplexArg(z, GetPrec(DOUBLE PRECISION)));
   END
```

```
CREATE FUNCTION Log(z COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   -- Note: This is not a legal SQL clause at present.
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F09';
      IF z = Complex(0,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN COMPLEX(RealLog(Abs(z), GetPrec(BaseType)),
            ComplexArg(z, GetPrec(BaseType)));
   END

--    The following trigonometric functions are all defined in
--    terms of Exp, Log and Sqrt. In each case, a family of six
--    functions is defined in order to accommodate differing
--    precisions of the returns value for: REAL, FLOAT,
--    DOUBLE_PRECISION, COMPLEX, COMPLEX_DOUBLE, and
--    COMPLEX(BaseType).

CREATE FUNCTION Radian(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE pi              REAL DEFAULT Arg(Complex(-1,0));
      RETURN pi*x/180;
   END

CREATE FUNCTION Radian(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE pi              FLOAT(p) DEFAULT Arg(Complex(-1,0));
      RETURN pi*x/180;
   END

CREATE FUNCTION Radian(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE pi              DOUBLE_PRECISION
         DEFAULT Arg(Complex(-1,0));
      RETURN pi*x/180;
   END

CREATE FUNCTION Sin(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      RETURN RealPart((Exp(i*x)-Exp(-i*x))/(2*i));
   END

CREATE FUNCTION Sin(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      RETURN RealPart((Exp(i*x)-Exp(-i*x))/(2*i));
   END
```

```
CREATE FUNCTION Sin(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      RETURN RealPart((Exp(i*x)-Exp(-i*x))/(2*i));
   END

CREATE FUNCTION Sin(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      RETURN (Exp(i*z)-Exp(-i*z))/(2*i);
   END

CREATE FUNCTION Sin(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      RETURN (Exp(i*z)-Exp(-i*z))/(2*i);
   END

CREATE FUNCTION Sin(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      RETURN (Exp(i*z)-Exp(-i*z))/(2*i);
   END

CREATE FUNCTION Cos(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      RETURN RealPart(Exp(i*x)+Exp(-i*x))/2;
   END

CREATE FUNCTION Cos(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      RETURN RealPart(Exp(i*x)+Exp(-i*x))/2;
   END

CREATE FUNCTION Cos(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      RETURN RealPart(Exp(i*x)+Exp(-i*x))/2;
      END
```

```
CREATE FUNCTION Cos(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      RETURN (Exp(i*z)+Exp(-i*z))/2;
   END

CREATE FUNCTION Cos(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      RETURN (Exp(i*z)+Exp(-i*z))/2;
   END

CREATE FUNCTION Cos(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      RETURN (Exp(i*z)+Exp(-i*z))/2;
   END

CREATE FUNCTION Tan(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F10';
      IF Cos(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sin(x)/Cos(x);
   END

CREATE FUNCTION Tan(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F10';
      IF Cos(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sin(x)/Cos(x);
   END

CREATE FUNCTION Tan(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F10';
      IF Cos(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sin(x)/Cos(x);
   END
```

```
CREATE FUNCTION Tan(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F10';
      IF Cos(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sin(z)/Cos(z);
   END

CREATE FUNCTION Tan(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F10';
      IF Cos(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sin(z)/Cos(z);
   END

CREATE FUNCTION Tan(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F10';
      IF Cos(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sin(z)/Cos(z);
   END

CREATE FUNCTION Sec(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F11';
      IF Cos(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cos(x));
   END

CREATE FUNCTION Sec(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F11';
      IF Cos(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cos(x));
   END
```

```
CREATE FUNCTION Sec(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F11';
      IF Cos(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cos(x));
   END

CREATE FUNCTION Sec(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F11';
      IF Cos(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cos(z));
   END

CREATE FUNCTION Sec(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F11';
      IF Cos(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cos(z));
   END

CREATE FUNCTION Sec(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F11';
      IF Cos(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cos(z));
   END

CREATE FUNCTION Csc(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F12';
      IF Sin(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sin(x));
   END
```

```
CREATE FUNCTION Csc(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F12';
      IF Sin(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sin(x));
   END

CREATE FUNCTION Csc(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F12';
      IF Sin(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sin(x));
   END

CREATE FUNCTION Csc(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F12';
      IF Sin(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sin(z));
   END

CREATE FUNCTION Csc(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F12';
      IF Sin(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sin(z));
   END

CREATE FUNCTION Csc(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F12';
      IF Sin(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sin(z));
   END
```

```
CREATE FUNCTION Cot(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F13';
      IF Sin(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cos(x)/Sin(x);
   END

CREATE FUNCTION Cot(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F13';
      IF Sin(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cos(x)/Sin(x);
   END

CREATE FUNCTION Cot(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F13';
      IF Sin(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cos(x)/Sin(x);
   END

CREATE FUNCTION Cot(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F13';
      IF Sin(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cos(z)/Sin(z);
   END

CREATE FUNCTION Cot(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F13';
      IF Sin(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cos(z)/Sin(z);
   END
```

```
CREATE FUNCTION Cot(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F13';
      IF Sin(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cos(z)/Sin(z);
   END

CREATE FUNCTION ArcSin(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F14';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(i*z + Sqrt(1 - z*z)));
   END

CREATE FUNCTION ArcSin(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F14';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(i*z + Sqrt(1 - z*z)));
   END

CREATE FUNCTION ArcSin(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F14';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(i*z + Sqrt(1 - z*z)));
   END

CREATE FUNCTION ArcSin(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      RETURN -i*Log(i*z + Sqrt(1 - z*z));
   END
```

```
CREATE FUNCTION ArcSin(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      RETURN -i*Log(i*z + Sqrt(1 - z*z));
   END

CREATE FUNCTION ArcSin(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      RETURN -i*Log(i*z + Sqrt(1 - z*z));
   END

CREATE FUNCTION ArcCos(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F15';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION ArcCos(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F15';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION ArcCos(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F15';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(z + Sqrt(z*z - 1)));
   END
```

```
CREATE FUNCTION ArcCos(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F15';
      RETURN RealPart(-i*Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION ArcCos(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F15';
      RETURN RealPart(-i*Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION ArcCos(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F15';
      RETURN RealPart(-i*Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION ArcTan(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart((i/2)*Log((i + z)/(i - z)));
   END

CREATE FUNCTION ArcTan(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      RETURN RealPart((i/2)*Log((i + z)/(i - z)));
   END

CREATE FUNCTION ArcTan(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      RETURN RealPart((i/2)*Log((i + z)/(i - z)));
   END
```

```
CREATE FUNCTION ArcTan(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F16';
      IF z = i OR z = -i
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (i/2)*Log((i + z)/(i - z));
   END

CREATE FUNCTION ArcTan(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F16';
      IF z = i OR z = -i
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (i/2)*Log((i + z)/(i - z));
   END

CREATE FUNCTION ArcTan(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F16';
      IF z = i OR z = -i
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (i/2)*Log((i + z)/(i - z));
   END

CREATE FUNCTION ArcSec(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F17';
      IF Abs(x) < 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END
```

```
CREATE FUNCTION ArcSec(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F17';
      IF Abs(x) < 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END

CREATE FUNCTION ArcSec(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F17';
      IF Abs(x) < 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END

CREATE FUNCTION ArcSec(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F17';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END

CREATE FUNCTION ArcSec(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F17';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END
```

```
CREATE FUNCTION ArcSec(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F17';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END

CREATE FUNCTION ArcCsc(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F18';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(1 - Invert(z*z))));
   END

CREATE FUNCTION ArcCsc(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F18';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(1 - Invert(z*z))));
   END

CREATE FUNCTION ArcCsc(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F18';
      IF Abs(x) > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(-i*Log(Invert(z) + Sqrt(1 - Invert(z*z))));
   END
```

```
CREATE FUNCTION ArcCsc(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F18';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN -i*Log(Invert(z) + Sqrt(1 - Invert(z*z));
   END

CREATE FUNCTION ArcCsc(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F18';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN -i*Log(Invert(z) + Sqrt(1 - Invert(z*z));
   END
```

****Editor's Note 4-035****

Illegal Syntax:

```
CREATE FUNCTION Coth(z COMPLEX(BaseType))
      RETURNS COMPLEX(BaseType)
```

```
CREATE FUNCTION ArcCsc(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F18';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN -i*Log(Invert(z) + Sqrt(1 - Invert(z*z));
   END

CREATE FUNCTION ArcCot(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart((-i/2)*Log(i*z - 1)/(i*z + 1)));
   END
```

```
CREATE FUNCTION ArcCot(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(FLOAT(p)) DEFAULT Complex(0,1);
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      RETURN RealPart((-i/2)*Log(i*z - 1)/(i*z + 1)));
   END

CREATE FUNCTION ArcCot(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      RETURN RealPart((-i/2)*Log(i*z - 1)/(i*z + 1)));
   END

CREATE FUNCTION ArcCot(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F19';
      IF z = i OR z = -i
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (-i/2)*Log(i*z - 1)/(i*z + 1));
   END

CREATE FUNCTION ArcCot(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX_DOUBLE DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F19';
      IF z = i OR z = -i
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (-i/2)*Log(i*z - 1)/(i*z + 1));
   END

CREATE FUNCTION ArcCot(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE i               COMPLEX(BaseType) DEFAULT Complex(0,1);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F19';
      IF z = i OR z = -i
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (-i/2)*Log(i*z - 1)/(i*z + 1));
   END
```

```
--    The following hyperbolic functions are all defined in
--    terms of Exp, Log and Sqrt. In each case, a family of
--    six functions is defined in order to accommodate differing
--    precisions of the returns value for: REAL, FLOAT,
--    DOUBLE_PRECISION, COMPLEX, COMPLEX_DOUBLE,
--    COMPLEX(BaseType).

CREATE FUNCTION Sinh(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart((Exp(z) - Exp(-z))/2);
   END

CREATE FUNCTION Sinh(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart((Exp(z) - Exp(-z))/2);
   END

CREATE FUNCTION Sinh(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart((Exp(z) - Exp(-z))/2);
   END

CREATE FUNCTION Sinh(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      RETURN (Exp(z) - Exp(-z))/2;
   END

CREATE FUNCTION Sinh(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      RETURN (Exp(z) - Exp(-z))/2;
   END

CREATE FUNCTION Sinh(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN (Exp(z) - Exp(-z))/2;
   END

CREATE FUNCTION Cosh(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart(Exp(z) + Exp(-z))/2;
   END
```

```
CREATE FUNCTION Cosh(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart(Exp(z) + Exp(-z))/2;
   END

CREATE FUNCTION Cosh(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      RETURN RealPart(Exp(z) + Exp(-z))/2;
   END

CREATE FUNCTION Cosh(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      RETURN (Exp(z) + Exp(-z))/2;
   END

CREATE FUNCTION Cosh(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      RETURN (Exp(z) + Exp(-z))/2;
   END

CREATE FUNCTION Cosh(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN (Exp(z) + Exp(-z))/2;
   END

CREATE FUNCTION Tanh(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F20';
      IF Cosh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sinh(x)/Cosh(x);
   END

CREATE FUNCTION Tanh(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F20';
      IF Cosh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sinh(x)/Cosh(x);
   END
```

```
CREATE FUNCTION Tanh(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
   DECLARE InvalidInput EXCEPTION FOR SQLSTATE 'H4F20';
      IF Cosh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sinh(x)/Cosh(x);
   END

CREATE FUNCTION Tanh(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F20';
      IF Cosh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sinh(z)/Cosh(z);
   END

CREATE FUNCTION Tanh(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F20';
      IF Cosh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sinh(z)/Cosh(z);
   END

CREATE FUNCTION Tanh(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F20';
      IF Cosh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Sinh(z)/Cosh(z);
   END

CREATE FUNCTION Sech(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F21';
      IF Cosh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cosh(x));
   END
```

```
CREATE FUNCTION Sech(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F21';
      IF Cosh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cosh(x));
   END

CREATE FUNCTION  Sech(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F21';
      IF Cosh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cosh(x));
   END

CREATE FUNCTION Sech(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F21';
      IF Cosh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cosh(z));
   END

CREATE FUNCTION Sech(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F21';
      IF Cosh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cosh(z));
   END

CREATE FUNCTION Sech(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F21';
      IF Cosh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Cosh(z));
   END
```

```
CREATE FUNCTION Csch(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F22';
      IF Sinh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sinh(x));
   END

CREATE FUNCTION Csch(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F22';
      IF Sinh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sinh(x));
   END

CREATE FUNCTION Csch(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F22';
      IF Sinh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN  Invert(Sinh(x));
   END

CREATE FUNCTION Csch(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F22';
      IF Sinh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sinh(z));
   END

CREATE FUNCTION Csch(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F22';
      IF Sinh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sinh(z));
   END
```

```
CREATE FUNCTION Csch(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F22';
      IF Sinh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Invert(Sinh(z));
   END

CREATE FUNCTION Coth(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F23';
      IF Sinh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cosh(x)/Sinh(x);
   END

CREATE FUNCTION Coth(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F23';
      IF Sinh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cosh(x)/Sinh(x);
   END

CREATE FUNCTION Coth(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F23';
      IF Sinh(x) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cosh(x)/Sinh(x);
   END

CREATE FUNCTION Coth(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F23';
      IF Sinh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cosh(z)/Sinh(z);
   END
```

```
CREATE FUNCTION Coth(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F23';
      IF Sinh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cosh(z)/Sinh(z);
   END

CREATE FUNCTION Coth(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F23';
      IF Sinh(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Cosh(z)/Sinh(z);
   END

CREATE FUNCTION InvSinh(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      RETURN RealPart(Log(z + Sqrt(z*z + 1)));
   END

CREATE FUNCTION InvSinh(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      RETURN RealPart(Log(z + Sqrt(z*z + 1)));
   END

CREATE FUNCTION InvSinh(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      RETURN RealPart(Log(z + Sqrt(z*z + 1)));
   END

CREATE FUNCTION InvSinh(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      RETURN Log(z + Sqrt(z*z + 1));
   END

CREATE FUNCTION InvSinh(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      RETURN Log(z + Sqrt(z*z + 1));
   END
```

```
CREATE FUNCTION InvSinh(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN Log(z + Sqrt(z*z + 1));
   END

CREATE FUNCTION InvCosh(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F24';
      IF x < 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION InvCosh(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F24';
      IF x < 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION InvCosh(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F24';
      IF x < 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(z + Sqrt(z*z - 1)));
   END

CREATE FUNCTION InvCosh(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      RETURN Log(z + Sqrt(z*z - 1));
   END

CREATE FUNCTION InvCosh(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      RETURN Log(z + Sqrt(z*z - 1));
   END
```

```
CREATE FUNCTION InvCosh(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      RETURN Log(z + Sqrt(z*z - 1));
   END

CREATE FUNCTION InvTanh(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F25';
      IF Abs(x) >= 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log((1+z)/(1-z))/2);
   END

CREATE FUNCTION InvTanh(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F25';
      IF Abs(x) >= 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log((1+z)/(1-z))/2);
   END

CREATE FUNCTION InvTanh(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F25';
      IF Abs(x) >= 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log((1+z)/(1-z))/2);
   END

CREATE FUNCTION InvTanh(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F25';
      IF z = Complex(1,0) OR z = Complex(-1,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log((1+z)/(1-z))/2);
   END
```

```
CREATE FUNCTION InvTanh(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F25';
      IF z = Complex(1,0) OR z = Complex(-1,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log((1+z)/(1-z))/2);
   END

CREATE FUNCTION InvTanh(z COMPLEX(BaseType))
   RETURNS COMPLEX(BaseType)
   -- Note: This is not a legal SQL clause at present.
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F25';
      IF z = Complex(1,0) OR z = Complex(-1,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log((1+z)/(1-z))/2);
   END

CREATE FUNCTION InvSech(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F26';
      IF x <= 0 OR x > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END

CREATE FUNCTION InvSech(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F26';
      IF x <= 0 OR x > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END

CREATE FUNCTION InvSech(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F26';
      IF x <= 0 OR x > 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(Invert(z) + Sqrt(Invert(z*z) - 1)));
   END
```

```
CREATE FUNCTION InvSech(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F26';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Log(Invert(z) + Sqrt(Invert(z*z) - 1));
   END

CREATE FUNCTION InvSech(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F26';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Log(Invert(z) + Sqrt(Invert(z*z) - 1));
   END

CREATE FUNCTION InvSech(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F26';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Log(Invert(z) + Sqrt(Invert(z*z) - 1));
   END

CREATE FUNCTION InvCsch(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F27';
      IF x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(Invert(z) + Sqrt(Invert(z*z) + 1)));
   END

CREATE FUNCTION InvCsch(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F27';
      IF x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(Invert(z) + Sqrt(Invert(z*z) + 1)));
   END
```

```
CREATE FUNCTION InvCsch(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F27';
      IF x = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart(Log(Invert(z) + Sqrt(Invert(z*z) + 1)));
   END

CREATE FUNCTION InvCsch(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F27';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Log(Invert(z) + Sqrt(Invert(z*z) + 1));
   END

CREATE FUNCTION InvCsch(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F27';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (Log(Invert(z) + Sqrt(Invert(z*z) + 1));
   END

CREATE FUNCTION InvCsch(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F27';
      IF Abs(z) = 0
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN Log(Invert(z) + Sqrt(Invert(z*z) + 1));
   END

CREATE FUNCTION InvCoth(x REAL)
   RETURNS REAL
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F28';
      IF Abs(x) <= 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart((1/2)*Log((z+1)/(z-1)));
   END
```

```
CREATE FUNCTION InvCoth(x FLOAT(p))
   RETURNS FLOAT(p)
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX(FLOAT(p)) DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F28';
      IF Abs(x) <= 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart((1/2)*Log((z+1)/(z-1)));
   END

CREATE FUNCTION InvCoth(x DOUBLE PRECISION)
   RETURNS DOUBLE PRECISION
   LANGUAGE SQL
   BEGIN
      DECLARE z               COMPLEX_DOUBLE DEFAULT Complex(x,0);
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F28';
      IF Abs(x) <= 1
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN RealPart((1/2)*Log((z+1)/(z-1)));
   END

CREATE FUNCTION InvCoth(z COMPLEX)
   RETURNS COMPLEX
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F28';
      IF z = Complex(1,0) OR z = Complex(-1,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (1/2)*Log((z+1)/(z-1));
   END

CREATE FUNCTION InvCoth(z COMPLEX_DOUBLE)
   RETURNS COMPLEX_DOUBLE
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F28';
      IF z = Complex(1,0) OR z = Complex(-1,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (1/2)*Log((z+1)/(z-1));
   END
```

```
CREATE FUNCTION InvCoth(z COMPLEX(BaseType))
   -- Note: This is not a legal SQL clause at present.
   RETURNS COMPLEX(BaseType)
   LANGUAGE SQL
   BEGIN
      DECLARE InvalidInput    EXCEPTION FOR SQLSTATE 'H4F28';
      IF z = Complex(1,0) OR z = Complex(-1,0)
         THEN SIGNAL InvalidInput;
      END IF;
      RETURN (1/2)*Log((z+1)/(z-1));
   END
```

****Editor's Note 4-008****

The SQL3 specification now supports syntax for specifying the overloading of SQL symbolic operators. Now that thiscapability is available, it will be used to specify new operators "EXP" and "**", with equal precedence relative to each other and with precedence to precede negation among the numeric operators, and defined as follows:

x EXP y = Exp(y*Log(x))

x ** n = Power(x,n)

Maybe SQL/MM shouldn't define two alternative ways to do not quite the same thing. Maybe only one of these operators should be defined, and that one should use the Exp/Log definition.

7.6 Conformance

7.6.1 Elementary functions conformance

An implementation may claim support for the SQL/MM elementary functions as follows:

The implementation shall support the SQL/MM elementary numeric functions specified in Subclause 7.4.1, "Elementary numeric functions", with the semantics specified in the SQLMM_NUMERICS schema.

The implementation shall support the SQL/MM combinatorial functions specified in Subclause 7.4.2, "Combinatorial functions", with the semantics specified in the SQLMM_NUMERICS schema.

An implementation may claim support for these functions if it effectively produces the specified result, without necessarily employing the specified algorithm.

This conformance option is not further subdivided. If an implementation supports both elementary numeric and combinatorial functions, then it may claim conformance to: SQL/MM elementary functions.

7.6.2 Transcendental function conformance

An implementation may claim conformance to the SQL/MM specifications for transcendental functions in several different ways. The minimum requirements for support are as follows:

The implementation shall support "SQL/MM elementary functions" as specified above in Subclause 7.7.1, "Elementary functions conformance".

The implementation shall support the SQL/MM exponential, trigonometric, and hyperbolic functions specified in Subclause 7.4.3, "Transcendental functions", for all real input variables, with the semantics specified in the SQL/MM_NUMERICS schema.

An implementation may claim support for these functions if it effectively produces the specified result, without necessarily employing the specified algorithm.

In addition, an implementation may claim support for the complex-valued transcendental functions, or may support modification or enhancement of the SQL/MM_NUMERICS schema definition. Implementors may choose to support any combination of the following pre-defined conformance alternatives:

a) SQL/MM real transcendental functions.

The minimal support for real-valued SQL/MM transcendental functions as specified above.

b) SQL/MM complex transcendental functions

Support for both real-valued and complex-valued SQL/MM transcendental functions, including support for at least one of the complex number conformance alternatives specified in Subclause 5.7, "Complex number conformance", and support for the SQL/MM exponential, trigonometric, and hyperbolic functions specified in Subclause 7.4.3, "Transcendental functions", for all real or complex input variables, with the semantics specified in the SQLMM_NUMERICS schema.

7.6.3 SQLMM_NUMERICS schema conformance

An implementation may claim support for modification or enhancement of the SQLMM_NUMERICS schema definition. Implementors may choose to support any combination of the following pre-defined conformance alternatives:

a) SQLMM_NUMERICS metadata

Support for the SQLMM_NUMERICS schema as if the <schema> syntax had been processed by SQL. Inclusion of all schema descriptor information in the INFORMATION_SCHEMA.

b) SQLMM_NUMERICS schema

Support for compilation and processing of the complete <schema definition> for SQLMM_NUMERICS as specified in Subclause 7.5, "Numerics Schema", with an arbitrary authorization instead of "_SYSTEM" authorization, and with implementation-supplied routines for the implementation-dependent routines in the SQLMM_NUMERICS schema definition.

7.7 Status Codes

The character string value returned in an SQLSTATE parameter comprises a 2-character class value followed by a 3-character subclass value. Table 4 specifies the class value for each condition and the subclass value or values for each class value.

The "Category" column has the following meanings: "S" means that the class value given corresponds to successful completion and is a completion condition; "W" means that the class value given corresponds to a successful completion but with a warning and is a completion condition; "N" means that the class value corresponds to a no-data situation and is a completion condition; "X" means that the class value given corresponds to an exception condition.

Table 4 — SQLSTATE class and subclass values

Category	Condition	Class	Subcondition	Subclass
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - ceiling	F01
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - floor	F02
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - modulo	F03
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - power	F04
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - square root	F05
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - factorial	F06
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - permutations	F07
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - combinations	F08
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - logarithm	F09
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - tangent	F10
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - secant	F11
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - cosecant	F12
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - cotangent	F13
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse sine	F14
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse cosine	F15
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse tangent	F16
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse secant	F17
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse cosecant	F18
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse cotangent	F19
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - hyperbolic tangent	F20

Category	Condition	Class	Subcondition	Subclass
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - hyperbolic secant	F21
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - hyperbolic cosecant	F22
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - hyperbolic cotangent	F23
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse hyperbolic cosine	F24
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse hyperbolic tangent	F25
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse hyperbolic secant	F26
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse hyperbolic cosecant	F27
X	SQL/MM General Purpose Facilities exception	H4	invalid input value - inverse hyperbolic cotangent	F28

Index

Index entries appearing in **boldface** indicated a page where the word, phrase, attribute, routine, or type was described; index entries appearing in *italics* indicates a page where the attribute, routine, or type was declared; and index entries appearing in roman type indicate a page where the word, phrase, attribute, routine, or type was used.

—A—

Abs, **7**, 8, *11*, 12, 13, 16, **20**, *23*, **28**, 30, *39*, *40*, 41, 42, 43, 56, 57, 64, 65, 67, 68, 69, 70, 80, 82, 83, 84
Add, **6**, *10*, 15, 16, **19**, *21*, 24
addition, 18, 20, 26, 30, 86
additive group, **18**
algebraic function, 27
angle, **5**, 6, 7, 8, 16
Angle, **6**, 8, *9*, 16
ANGLE, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16
ArcCos, **33**, *65*, *66*
ArcCot, **33**, *70*, *71*
ArcCsc, **33**, *69*, *70*
ArcSec, **33**, *67*, *68*, *69*
ArcSin, **33**, *64*, *65*
ArcTan, **33**, *66*, *67*
Arg, 30, 31, **32**, *54*, *55*, 57
associative, 18
Authorization
_SYSTEM, 26, *36*, 86

—C—

Ceiling, **28**, *40*
Comb, **30**, *52*
CombX, **30**, *53*
CombXX, **30**, *53*
commutative, 18
Complex, **19**, *20*, 21, 22, 23, 30, 31, 32, 54, 56, 57, 58, 59, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 79, 80, 81, 82, 83, 84, 85, 86
COMPLEX, 18, 19, *20*, 21, 22, 23, 24, 26, 32, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85
Complex Number, 20, 26
COMPLEX_DOUBLE, 19, *24*, 26, 32, 37, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 78, 79, 80, 81, 82, 83, 84
ComplexArg, *37*, 54, 55, 56, 57
ComplexExp, *37*, 55
Conjugate, **19**, **20**, *23*, 35
Cos, **32**, 33, 35, *58*, *59*, 60, 61, 63, 64
Cosh, **33**, 34, *72*, *73*, 74, 75, 77, 78
Cot, **33**, *63*, *64*
Coth, **34**, *77*, *78*
Csc, **33**, *61*, *62*
Csch, **34**, *76*, *77*

—D—

degree, 5, 6, 7
Degree, **7**, *12*, 13
distributive, 18
Divide, **19**, *22*, 24

—E—

Editor's Note
4-008, **85**
4-011, **38**
4-014, **8**
4-015, **8**
4-016, **9**
4-017, **14**
4-018, **15**
4-024, **24**
4-025, **25**
4-033, **19**
4-034, **36**
4-035, **70**
4-036, **9**
4-037, **20**
Exp, 29, 30, 31, **32**, 35, 49, 54, *55*, 57, 58, 59, 72, 73, 85
Exponent, **28**, *41*

—F—

Fact, **29**, 30, 48, 49
FactX, **29**, *49*
FactXX, **29**, *49*
field, 18, 19
Floor, **28**, *42*, 43, 44
Fraction, **7**, *13*

—G—

GetBaseType, *38*
GetBinPrec, *38*
GetDecPrec, *38*, 41, 43
GetMaxValue, *38*, 40, 41, 42, 43
GetMinValue, *39*
GetType, *38*

—I—

identity, 18, 20
ImagPart, **19**, 20, 21, 22, *23*, 24
InvCosh, **34**, *79*, *80*
InvCoth, **34**, *83*, *84*, 85
InvCsch, **34**, *82*, *83*
inverse, 18, 19, 28, 31, 32, 33, 34, 88, 89
Invert, **19**, *22*, **28**, *42*, 60, 61, 62, 67, 68, 69, 70, 74, 75, 76, 77, 81, 82, 83
InvSech, **34**, *81*, *82*
InvSinh, **34**, *78*, *79*
InvTanh, **34**, *80*, *81*
IsEqual, **6**, *10*, 14, 15, **19**, *21*, 24, 25
IsLessThan, **6**, *10*, 14, 15

—L—

Log, 31, **32**, 33, 34, 35, 54, *56*, *57*, 64, 65, 66, 67, 68, 69, 70, 71, 72, 78, 79, 80, 81, 82, 83, 84, 85

—M—

Mantissa, 28, 41, *43*
Minus, **6**, *10*, 11, 15, 16, **19**, *21*, 24, **28**, *44*, *45*
Minute, **7**, *12*, 13
Mod, **28**, *43*, *44*
Multiply, **6**, *11*, 15, 16, **19**, *22*, 24

—N—

NUMBER, 38

—P—

PascalTriangle, *52*
periodic function, 27, 30
Perm, **29**, *50*
PermX, **30**, *50*
PermXX, **30**, *51*
pi, 5, 8, 9, 12, 13
Plus, **6**, *10*, 15, 16, **19**, *21*, 24, **28**, *45*
Power, 28, **29**, 41, 43, *46*, *47*, 49, 85

—R—

radian, **5**, 6, 8, 9, 10, 11, 12, 13, 14
Radian, **7**, 8, 9, 12, 13, 16, 29, **32**, 49, *57*
rational function, 27
RealExp, *36*, 55
RealLog, *37*, 56, 57
RealPart, **19**, 20, 21, 22, 23, 24, 57, 58, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 78, 79, 80, 81, 82, 83, 84
RealSqrt, *37*, 53, 54
RealToComplex, **20**, *23*, 24

—S—

Sec, **33**, *60*, *61*
Sech, **34**, *74*, *75*
Second, **7**, *12*, 13
Sign, **7**, 8, *12*, 13, 16, 28, **29**, *48*
Sin, **32**, 33, 35, *57*, *58*, 59, 60, 61, 62, 63, 64
Sinh, **33**, 34, *72*, 73, 74, 76, 77, 78
SQL_CHARACTER, *36*
SQLMM_NUMERICS, 23, 27, *36*, 85, 86
SQLSTATE
H4F01, 40
H4F02, 42
H4F03, 43, 44
H4F04, 46, 47
H4F05, 53, 54
H4F06, 48, 49
H4F07, 50, 51
H4F08, 52, 53
H4F09, 56, 57
H4F10, 59, 60
H4F11, 60, 61
H4F12, 61, 62
H4F13, 63, 64
H4F14, 64
H4F15, 65, 66
H4F16, 67
H4F17, 67, 68, 69
H4F18, 69, 70
H4F19, 71
H4F20, 73, 74
H4F21, 74, 75
H4F22, 76, 77
H4F23, 77, 78
H4F24, 79
H4F25, 80, 81
H4F26, 81, 82
H4F27, 82, 83
H4F28, 83, 84, 85
Sqrt, 23, **29**, 33, 34, 49, *53*, *54*, 57, 64, 65, 66, 67, 68, 69, 70, 72, 78, 79, 80, 81, 82, 83
Stirling, **29**, *49*
Subtract, **6**, *11*, 15, 16, **19**, *21*, 24

—T—

Tan, **32**, *59*, *60*
Tanh, **33**, *73*, *74*
ToDoubleAngle, **7**, *14*
ToFloat20Angle, **7**, *14*
ToFloat47Angle, **7**, *14*
ToRealAngle, **7**, *13*, 14
ToVarChar, **7**, *13*, 14, **20**, *24*
TruncateToInteger, *36*, 40, 42, 43