

Database
Management
System **LINTER**®

Version 5.9

Stored Procedure Functions

Relational Expert Systems



Table of Contents

Introduction	5
Definitions	6
Linter Vocabulary.....	6
Text Conventions.....	6
Names	7
Comments	8
Data Types	9
Data Type Compatibility	10
Constants	11
Numeric Constants.....	11
Integer.....	11
Floating Point.....	11
String Literals	11
Date Constants.....	12
Logical Constants	12
Cursor Constants.....	12
NULL Values	12
Variable Declaration	12
Predefined Trigger Variables	13
General Syntax of Stored Procedures	13
Header.....	13
Declarations Section	14
Declaration of Variables.....	15
Declaration of Exceptions.....	15
Code Section	15
Exception Handler Section.....	16
Statements	17
Assignment.....	17
IF	17
CASE	18
WHILE.....	18
GOTO	19

Stored Procedure Calls	19
RETURN	20
EXCEPTION	20
Exception Transfer.....	20
OPEN Cursor.....	21
FETCH (Data from Cursor)	21
CLOSE (Cursor)	22
EXECUTE (Non-cursor Query)	22
COMMIT (Transaction) and ROLLBACK.....	23
Queries.....	23
Precompiled Queries.....	23
Dynamic Queries.....	24
Expressions	25
Operands	25
Operators	26
Arithmetic Expressions.....	26
String Expressions.....	26
Logical Expressions	26
Date Expressions	27
Value Assignments	28
Conditional Expressions	29
Standard Functions.....	30
String Functions.....	30
LEN.....	30
TRIM.....	30
SUBSTR.....	30
STRPOS.....	31
DATE Functions.....	32
DAY.....	32
MONTH.....	32
YEAR	32
HOUR	32
MINUTE	33
SECOND	33
TICKS	33
MAKE_DATE.....	33
Type Conversion Functions.....	34
ITOA.....	34
FOTA	34
NTOA.....	35
DOTA.....	35
ATOD.....	35

TOSMALLINT.....	36
TOINTEGER.....	37
TOREAL.....	37
TONUMERIC.....	37
Cursor Functions.....	37
OUTOFCURSOR.....	37
ROWCOUNT.....	38
ERRCODE.....	38
Miscellaneous Functions.....	38
MAX.....	38

Introduction

This document explains the syntax and semantics of the Linter procedural language that is used to design stored procedures and triggers for a database. The procedural language is based on the SQL-92/PSM (Persistent Stored Modules) standard.

Definitions

Linter Vocabulary

Module	A set of SQL statements that may be executed by a compilation unit, i.e., called by an application program. Each module is made up of one or more procedures.
Procedure	The component(s) of a module. Each procedure contains a set of parameter definitions and a single SQL statement.
Stored procedure	An instance of a procedure that is incorporated in the data dictionary and called on the basis of specified, variable conditions.
Trigger	An instance of a stored procedure that is called when one or more specified attributes in a database are updated.

Text Conventions

The text conventions used in this document follow this Guide's standard conventions, described on p.xii, and add:

KEYWORDS	Which are shown in capitalized, SANS SERIF.
-----------------	---

Stored procedure scripts and the text conventions in this document may reflect the following:

Whitespace	Lexical units, lexemes, in stored procedure or trigger scripts may be separated by whitespace (any number of spaces or tabs) and new line, line feed, characters, all of which are ignored.
Case sensitivity	Keywords and identifiers in stored procedures are not case sensitive. The names of procedures called from another procedure are the only exception. Such called names are subject to Linter's general rule that if a name identifier is used without quotes, the name is not case sensitive; otherwise, the name must be used exactly as presented in the called procedure.

Names

Names or identifiers are used for naming formal parameters, local variables, exceptions, and labels for executable statements. Names are strings consisting of any alphanumeric and graphical characters. The following characters are excluded:

+ - * / ^ () \ = # < > . , ; [] { } " \$:

Names may start with any character but a digit, and may contain up to 3000 characters, of which only the first 30 characters are significant

Comments

A comment is the substring of a line following the double slash, //, characters.

Data Types

INTEGER (INT)	signed integers ranging from -2,147,483,648 to +2,147,483,647.
SMALLINT	signed integers ranging from -32,768 to +32767.
REAL	single-precision floating point numbers ranging from -1 .0E+38 to +1 .0E+38 with 6 significant digit precision.
DOUBLE (NUMERIC)	double-precision floating-point numbers ranging from is -1 .0E+38 to +1 .0E+38 with 15 significant digit precision.
CHAR	syntax: <size>. Alpha, numeric, graphic, and control characters in a string with a maximum length of 4,000 characters.
BOOL	syntax: (<true false>). Defines a logical data type.
CURSOR	syntax: (<name><scalar type>[, ...]). Defines a data structure containing table columns representing a relational database.
DATE	a data type restricted to date and time information. Formats: <ul style="list-style-type: none">• US : MM/DD/YYYY: [HH: [MI: [SS: [TT]]]• European: DD.MM.YYYY: [HH: [MI: [SS: [TT]]] See the Date Constants section, below, for detail.
TYPEOF	syntax: (<name of object>). Defines a data type to be the same type as the data type of the object named as an argument to TYPEOF. E.g., TYPEOF (RESULT) would define the data type to be the same type as the data type returned by a procedure. Any variable name may be used as <name of object>.

Data Type Compatibility

String expressions, dates, and logical data are compatible only with data of the same type. Numeric expressions are compatible with numeric types that can be converted into each other without loss of data. For example, the data type `INTEGER` is compatible with both `SMALLINT` and `INTEGER`, but `SMALLINT` is compatible only with `SMALLINT`.

Constants

Numeric Constants

Integer

Integer constants are strings of digits with an optional plus or minus sign preceding the digits.

Floating Point

The constants for floating point numbers have the same syntax as integers, with the difference that a decimal point may be positioned between digits to separate the integer and mantissa. If the decimal point is located at the end of a number, the fractional part of the number is assumed to be zero. An integer is always required; i.e., for values < 1 , a zero must be placed before the decimal point.

Examples of floating point numbers:

```
0
+123
-456
3.1415
-0.33
777.
```

String Literals

A string constant, or literal, is a sequence of characters surrounded by double quotes, as used in the C programming language. Literals may contain printable characters and special escape-sequences, such as:

Examples of escape characters in string literals:

```
\n  new line
\r  carriage return
\t  horizontal tab
\x<code>  hexadecimal number
\ <char to be escaped>
```

The backslash may be used to escape the special operation of some characters such as the quote and backslash itself.

Examples of string literals:

```
"abce"
"New line\n"
"\x7 the Bell"
"\ "c:\linter\ " "
```

Date Constants

Date and time data is presented in one of the following formats:

US : MM/DD/YYYY: [HH: [MI: [SS: [TT]]]]

European: DD.MM.YYYY: [HH: [MI: [SS: [TT]]]]

where:

DD	Day	1-31
MM	Month	1-12
YYYY	Year	1-9999
HH	Hours	0-23
MI	Minutes	0-59
SS	Seconds	0-59
TT	Tics	0-99

 00/00/0000:00:00:00:00 is a valid date.

Examples of date constants:

12/25/1999 25.12.2000 8/21/2001:20:53 21.8.2001:9:10:55

Logical Constants

The BOOL data type constants are TRUE and FALSE.

Cursor Constants

There are no constants for the cursor data type.

NULL Values

In stored procedures, scalar objects can have a null value, indicating the absence of any data. A null value may be assigned to a variable of any type, and may be compared with the current value of another variable. Variables can obtain a null value as the result of explicit assignment, or as a default value (see below).

An attempt to perform an operation (arithmetic, logical, or others except assignment and comparison) with null values will generate a NULLDATA error (see below).

The NULL constant is used to assign a null value.

Variable Declaration

Variable declaration is similar to the declaration of a group of parameters:

```
VAR<list of names> <type> [DEFAULT <list of initializers>];
<name 1 > ... [<name N> <type> [DEFAULT ]
```

Predefined Trigger Variables

Inside the trigger's body, special predefined variables may be used. The variables obtain the values of a processed record, both before and after execution of the trigger function. In the trigger's code, these values are presented as a CURSOR data type; therefore, the following syntax should be used to access a field of the processed record:

<name of value>.<name of field>

<name of field> is the name of a table field.

<name of value> is given in the OLD [AS] and NEW [AS] options of the CREATE TRIGGER statement. Alternatively, <name of value> may be used for the OLD and NEW values of the record if OLD [AS] and NEW [AS] are not given

The trigger's function can assign new values to the fields of a NEW variable or the variable referred in the NEW AS clause. In this case, the variable will be used to create a value for the processed record resulting from execution of the function to which the trigger is bound.

The OLD and NEW values are not defined for all triggers. For example, only NEW is defined in the trigger for INSERT, while only OLD is defined in the trigger for DELETE. No values are defined in triggers for the whole STATEMENT.

Example of predefined trigger variables:

```
if old.personid <> new.personid then
execute direct "select personid from person where
personid="+itoa(new.personid) + ";//"
    if errcode() = 2 then
        execute direct "insert into journal
values('AUTO','UPDATE'," +itoa(old.personid) + ",
sysdate, 'update to bad personid -IGNORED');//"
        return false; //
    endif
endif
```

General Syntax of Stored Procedures

A stored procedure has the following syntax:

```
<header>
[<declaration>]
<statements>
[<exception handlers>]
END
```

Header

The header of a stored procedure has the following syntax:

```
PROCEDURE<name>(<list of parameters>) [RESULT<type>]
[FOR DEBUG]
```

The list of parameters may be empty or consists of one or more groups of parameters. A semicolon separates the groups of parameters. The group of parameters has the following syntax:

```
<modifier> <list of names><type> [DEFAULT <list of initializers>]
```

modifier is one of the following words:

IN	Passed to the procedure (input parameters)
OUT	Returned from the procedure (output parameters)
INOUT	Passed to and from the procedure (input/output parameters).

<list of names> contains a single parameter name, or a group of parameter names separated by commas.

<list of initializers> includes expressions separated by commas. Expressions defining the initial values of parameters must allow for their evaluation at compile time. When writing the procedure you may use constants, parameters previously defined in the procedure, and variables for which default values will be used.

The number of expressions in the <list of initializers> may be less than the number of parameters in the group. Remaining parameters are initialized the same way as when the DEFAULT clause is not used, namely, with NULL values. The construct without DEFAULT is equivalent to the explicit form: DEFAULT NULL.

All procedures return a value. The RESULT option defines the type of this value. If this option is not included, the default value is NULL.

If the FOR DEBUG option is used, the procedure will be compiled with debug data. Without this option the stored procedures debugger cannot be used to debug the procedure.

The DEFAULT option is used to assign a value to all parameters appearing in the declaration section of the procedure that do not appear in the list of parameters. If left in the end of the list when the called list of parameters is shorter than the list declared in the procedure, or in the beginning (in the latter case, all skipped parameters are replaced by commas: ,,).).

Example of stored procedure header

```
procedure retcur(in name char(20) default "AUTO";
out success bool)
result cursor (i int,
               a char(20),
               s smallint,
               d date,
               n numeric,
               r real,
) for debug;
```

Declarations Section

Declarations have the following syntax:

```
DECLARE
declaration 1>
...
declaration n>
```

where <declaration> is a declaration of local variables or exceptions.

Declaration of Variables

The syntax for declaring variables is similar to that of a header's list of parameters:

```
VAR<list of names> <type> [DEFAULT<list of initializers>];
```

Declaration of Exceptions

Declarations of exceptions have the following syntax:

```
EXCEPTION<name> FOR<type of exception>;
```

The following exceptions are recognized in stored procedures:

DIVZERO	Divide by zero.
UNDEFPROC	Call for an undefined procedure.
BADPARAM	Invalid procedure parameter.
BADRETVAl	Invalid type of returned value.
NULLDATA	Attempt to evaluate an expression with no data.
BADCURSOR	Disagreement between cursor's structure and number and/or types of query's response (when opening or returning the cursor).
CURNOTOPEN	Attempt to execute a FETCH or CLOSE for an unopened cursor.
<number>	Completion code of Linter's operation.
CUSTOM	User's exception.

Names of user's exceptions are local in scope within each procedure. When such exceptions are declared, they are automatically assigned internal codes. In order to pass such an exception to the call environment, RESIGNAL, and to correctly handle it there, the exception code must be the same in both the call environment and within the procedure. For this purpose, you can explicitly assign a code (an integer) to the user's exception:

```
Custom<code>
```

User's exception codes should never be the same as the return codes used in Linter. Return codes are stored as negative numbers.

Example of the declaration section

```
declare
  var a,b typeof (result);
  var I int default 0;
  exception badcur for badcursor;
  exception notab for 2202;
```

Code Section

The code section of a stored procedure has the following syntax:

```
CODE <statements>
```

The syntax of statements is described in Search, Statements.

Exception Handler Section

The exception handler section has the following syntax:

```
EXCEPTIONS
  WHEN<list exception names>
    THEN <statements>
    ...
  [ WHEN OTHERS
    THEN <statements>
    ... ]
```

The list of exception names consists of one or more names declared in the declaration section. The list is separated by commas.

When one of the above exceptions has occurred during execution of the procedure, the control flow goes to the specific statements following WHEN. The statements will be executed until the next WHEN is encountered with the procedure returning (it works the same way as RETURN see below). In order to continue the procedure, the GOTO statement should be used (see below).

If the WHEN OTHERS clause is not used, all exceptions with no handler code defined, but with the exception declarations given in the declaration section, will be automatically passed to the upper level (RESIGNAL of nested calls. All, but performance critical, exceptions not declared in the section of exception declarations are to be ignored. The following exceptions are considered critical: OTHERS, UNDEFPROC, BADPARAM, BADRETVAl, BADCURSOR, and CURNOTOPEN. When an exception of this kind occurs and there is no routine for handling it, RESIGNAL is automatically performed.

Example of the exception handler section

```
exceptions
  when notab then
    close first_cursor;
    goto notab_recovery;
  when badcur then
    success := false;
  when others then
    success := false;
    resignal;
```

Statements

Any executable statement has the following generalized syntax:

```
[label:] <statement body>
```

The label is a name followed by a colon.

The <statement body> has a syntax specific to its functionality.

Assignment

```
<expression>;
```

The <execution> of this statement is equivalent to evaluation of its expression. Commonly, this statement is used to obtain its side effect when evaluating the expression, namely to assign the values and to call standard functions. The syntax of expressions in stored procedures is explained in the following sections.

Examples of assignments

```
i := 0;
sum := eif[c.i < 100] sum + 10 else sum + 100;
```

IF

The IF statement is used to change control flow when executing a stored procedure or trigger.

The IF statement has the following syntax:

```
IF <condition_expression_1 > THEN
    <statement_1 >
[ ELSEIF <condition_expression_2> THEN
    <statements_2>
]...
[ ELSE
    <statement_n>
]
ENDIF
```

The IF statement may have none or any number of ELSEIF branches followed by none or a single ELSE branch. During the statement's execution, <condition_expression_1 > is evaluated producing a logical type result. If the result's value is TRUE <statement_1> will be evaluated with control flow switching to the statement following ENDIF. If the result's value is FALSE the same sequence of actions is followed for the expression and statement after each ELSEIF, if any. If there are no ELSEIF branches, or all values of their expressions are FALSE, then <statement_n> of the ELSE branch will be executed. If there is no ELSE branch, control is passed to the statement following ENDIF.

Example of the IF statement

```
if n > -1 and n < 0.5 then
    call processing(1);
elseif n >= 0.5 and n < 2 then
    call processing(2);
elseif n >= 2 and n < 10 then
```

```

        call processing(3);
    else
        call processing(4);
    endif

```

CASE

The CASE statement can be used when multiple control flow branches are needed. Functionally, this statement is equivalent to the IF statement with the appropriate number of ELSEIF conditions but with a simpler and more obvious syntax.

The CASE statement has the following syntax:

```

CASE <expression_1 >
    WHEN <constant_1> [, ...]> THEN
        <statements_1 >
    WHEN <constant_2> [, ...]> THEN
        <statements_2>
    ...
    [WHEN OTHERS THEN
        <statement>
    ]
ENDCASE

```

Data types of constants and CASE expressions must be agreeable.

The value of the CASE expression is compared, sequentially, with constants in the WHEN options. As soon as the value of the CASE expression equates to a constant in one of the WHEN options:

- all statements following that option, up to the next WHEN statement, or ENDCASE, are evaluated;
- the CASE statement exits.

If the value of the CASE expression is not equal to any of the constants in the statement, the WHEN OTHERS branch is executed. If there is no WHEN OTHERS branch, control flow jumps to the statement following ENDCASE.

Example of the CASE statement

```

case rel_name
    when "$$$SYSRL", "$$$ATTRI", "$$$USR" then
        definit := "System dictionary table";
    when "$$$PROC", "$$$PRCD" then
        definit := "System stored procedures table";
    when others then
        definit := "Unknown table class";
endcase

```

WHILE

The WHILE statement allows the conditional repetition of specific parts of a program. It has the following syntax:

```

WHILE <expression> LOOP
    <repeated statements>
ENDLOOP

```

The WHILE <expression> must be a logical data type. If its value is TRUE the repeated statements preceding ENDLOOP will be executed. After each execution, the WHILE <expression> is again evaluated. While, so long as, the value is TRUE, the <repeated statements> are executed. When the value becomes FALSE, control flow jumps to the statement following ENDLOOP. To avoid an infinite loop, the WHILE <expression> must be capable of returning a FALSE value as the result of executing the <repeated statements>.

Example of the WHILE statement (a "classic" case of searching through all answers in a selection):

```
while not outofcursor(curs) loop
  //handling an answer
  sum := sum + curs.i + curs.j;
  execute direct "update tabl set s = " + itoa(sum) +
    "where current of \"CURS\";";
  fetch curs;
endloop
```

GOTO

The GOTO statement is used to change flow control while executing a program. It has the following syntax:

```
GOTO <label name>;
```

Execution of this statement causes control flow to jump to the program statement preceded by the label identified by <label name>. <label name> does not include a terminating colon.

Example of the GOTO statement

```
goto tab_recovery;
```

Stored Procedure Calls

Syntax

```
CALL <name> (<list of parameters>) [INTO <variable name>];
```

This statement executes another stored procedure or itself. Any number of recursive calls are allowed.

The <list of parameters> includes expressions or variable names separated by commas. The list may be empty. The <list of parameters> must be the same in number and sequence as the parameters of the called procedure.

If the default value(s) of one or several parameter(s) are to be passed, the actual parameter(s) can be omitted, but the comma separator(s) must be included. It is not necessary to use a comma at the end of the list. All omitted parameters will always obtain their values by default, even if the list is empty.

If the IN modifier has been used with the parameters of the called procedure, any expression may be used as a parameter in CALL. Compatibility of data types is checked at run time, see below.

If either the OUT or INOUT modifier has been used with the parameters of the called procedure, expressions are not allowed as parameters in CALL. In this case, only variable names can be used to hold the output values. If the variable name is omitted, no output value will be generated.

If the INTO clause is used in a CALL statement, the returned value of the procedure will be assigned to the specified variable. Such usage is not allowed for procedures returning a cursor. In order to pass a cursor into a calling procedure, a special construct exists; see RETURN, below.

Reference to the called procedure is verified when the procedure is executed; the procedure is searched by name. If the procedure is not found, the UNDEFPROC exception is generated. Data types and number of parameters and data types of the returned values are also checked at this time. If an error occurs, the BADPARAM or BADRETVL exceptions are generated.

Example of the CALL statement

```
call myproc("auto", ,1 ,aa) into bb;
```

RETURN

Syntax

```
RETURN [<value>];
```

<value> is an expression, or the name of a cursor variable if the procedure returns a cursor. When a value is not provided, the procedure returns NULL. Executing this statement causes the procedure to exit. On exiting, control flow returns to the calling procedure.

Example of the RETURN statement

```
return sum * a;
```

EXCEPTION

Syntax

```
SIGNAL <exception name>;
```

This statement calls an exception by its specific name. The name must have been declared in the declaration section.

Example of the EXCEPTION statement

```
Signal badparam; // send a signal indicating invalid  
                // value of an input parameter
```

Exception Transfer

Syntax

```
RESIGNAL;
```

This statement can be used only in the exception handlers' section. Its actions include the exiting the procedure (with NULL as the returned value), and transfer of the exception to the calling procedure or as a response to a user's query.

Except for critical exceptions (see "Exception Handler Section"), exceptions occurring in a procedure are not transferred unless they explicitly call RESIGNAL.

OPEN Cursor

The OPEN statement associates a cursor variable with the procedural language object that will fill in the cursor's fields during program execution. A data retrieve query or a procedure returning the cursor may be used as such objects.

Syntax

```
OPEN <name of cursor variable> [AS "<cursor name>"] FOR<query>;
```

```
OPEN <name of cursor variable> [AS "<cursor name>"]
FOR CALL <name of procedure> (<list of parameters>);
```

Cursor is a fetch consisting of the set of values of a specific structure, such as a field's collection. The field values can be accessed using the cursor variable. Navigation within the fetch is implemented by the FETCH statement (see below).

The <query> syntax is explained below, in sections "QUERIES". Only the SELECT query is a valid method for opening a cursor.

The second form of the OPEN statement calls a procedure that returns a cursor. The syntax is similar to that of the CALL statement. The difference is that the INTO option is not allowed in an OPEN statement.

When the AS option is specified, a named cursor will be opened. Cursors should be named if queries with WHERE CURRENT conditions are to be used. The cursor's name must be surrounded with quotes and must be valid according to Linter's naming conventions.

After the cursor is opened the values of the first result immediately become available via the cursor variable. See the FETCH statement, below.

If the number, and/or types of cursor's fields do not agree with the structure of the cursor variable, the BADCURSOR exception occurs during the execution of OPEN.

Two examples of the OPEN statement:

```
open a for direct "select * from auto where make = ' ' + make +
' '";
open b as "cursor_b" for call retcur("auto");
```

FETCH (Data from Cursor)

Syntax

```
FETCH <name of cursor variable> [<direction>];
```

This statement fetches the next result from the cursor as specified in <direction>. The result is accessed via the fields of the cursor variable. The fields are referred as follows:

```
<name of cursor variable>.<name of field>
```

The <direction> option has a default value of NEXT. The possible values for <direction> are:

NEXT	Next result
PREVIOUS	Previous result
FIRST	First result
LAST	Last result

ABSOLUTE	Result from index evaluated with an expression.
<expression>	Must evaluate to INTEGER or SMALLINT.
RELATIVE <expression>	Result from index offset from current position by a number evaluated with any expression. The offset value must be an INTEGER.

In order to create a looped implementation of fetch operations, a group of FETCH and WHILE statements is used with a call to the OUTFETCH function as a conditional expression within the WHILE statement. In addition to OUTFETCH, the ROWCOUNT function is provided to obtain a count of results. The ERRCODE function is used to capture the error code when the error is not intercepted by an exception.

Attempting to apply FETCH to an unopened cursor will generate a CURNOTOPEN exception.

Three examples of the FETCH statement:

```
fetch a;
fetch b previous;
fetch c relative offset * size - 1;
```

CLOSE (Cursor)

The CLOSE statement is used to close operations with the opened cursor and to release all resources used in working with the cursor.

Syntax

```
CLOSE <name of cursor variable>;
```

Using CLOSE is optional. When programs exit, all opened cursors are automatically closed except for cursors returned from cursor-type procedures. It is important that such cursors, those returned from cursor-type procedures, NOT be closed explicitly.

Attempting to close an unopened cursor generates a CURNOTOPEN exception.

EXECUTE (Non-cursor Query)

The non-cursor query is used for implementing direct access to a database.

Syntax

```
EXECUTE <query>;
```

The cursor mechanism only allows you to use SELECT queries when working with a database. The EXECUTE statement permits execution of any query (including SELECT) using a dedicated channel. The channel is automatically opened when executing this statement. Channel closing depends on transaction processing; see COMMIT, below.

Linter exit code exceptions are generated by EXECUTE in the same way they are generated by the OPEN, FETCH and CLOSE statements. The syntax of queries is shown in the QUERIES section, below.

Two examples of the EXECUTE statement:

```
execute direct "update tabl set s = " + itoa(sum) +
"where current of \"CURS\";";
```

```
execute direct "create table test(i int);";
```

COMMIT (Transaction) and ROLLBACK

The COMMIT and ROLLBACK statements control acceptance or cancellation of updates to the database when performing a current transaction.

Syntax

```
COMMIT [RELEASE];    // saving changes  
ROLLBACK [RELEASE]; // rollback
```

Every update in the database is made using queries provided by the EXECUTE statement. All queries from the EXECUTE statement in the stored procedure are transmitted via a channel that opens automatically. This channel is independent of the user's application channel; the channel used for execution of the user's Linter connection.

The procedure can roll back or save all the updates made during execution of the statements of its body or it may simply exit. If exiting, it is up to the application to decide how to implement the transaction. The application can issue the COMMIT or ROLLBACK commands using its channel which will impact all the updates made by both the procedure itself and all its affiliated procedures.

Thus, the procedure may partially roll back or save all the updates it has made or the updates may be used as part of a larger transaction.

If these statements specify the RELEASE option, the internal cursor for which the statements are executed immediately closes. Otherwise, it remains open which permits fast implementation of the update.

Queries

There are two types of queries for stored procedures:

- 1) precompiled –parsed during compilation;
- 2) dynamic created and executed in run time.



In the current release of Linter, only dynamic queries are supported. Precompiled queries will be implemented in the next release.

Precompiled Queries

Syntax

```
<query string> [<list of parameters>[, ...] ]  
where:
```

<query string> is a literal; a string in quotes;

<list of parameters> is a comma-separated list of expressions.

During compilation, the query string is precompiled and stored as procedure code. During run time, any evaluated parameters are attached to the precompiled query.

Dynamic Queries

Syntax

DIRECT <char type expression>;

When executing a dynamic query, the char type expression is evaluated and interpreted as an SQL query.

If, while implementing the query, an error is encountered, an exception is issued with an appropriate code.

Expressions

An expression is a combination of procedural language objects called operands. Local variables, values of passed parameters, fields of a structure, standard functions, and other, simpler expressions may be used as operands. Operands are grouped into expressions using arithmetic, logical, and relational operators.

When evaluating an expression, the order of operations with operands is determined based on the precedence of the operators. Precedence can be changed using parentheses.

For the sake of clarity, expressions can be surrounded by curly brackets, { and }, when they are embedded in other expressions.

A sequence of expressions separated by semicolons may be considered as a single expression. In such sequences, each subexpression is evaluated in order from left to right. The result of the last subexpression's evaluation is the result for the whole expression.

A special syntax is used for evaluating conditional expressions. These are not to be confused with conditional statements! In a conditional expression, value is established by the value of one of two subexpressions depending on the value of the third logical data type subexpression. This third, logical expression is the condition.

Expressions are categorized based on their data types, which are the same as the data type of the evaluation's result. The types of expressions match the types processed in the stored procedure, with the exception of cursors. Cursor variables may only assign cursor variables each other.

If the expression evaluation is accompanied by an attempt to divide by zero, the OTHERS exception is issued. The result of a divide by zero evaluation will be zero.

Operands

Local variables, parameters and functions are included in expressions by their names.

The following syntax is used to name cursor fields:

`<name of cursor>.<name of field>`

The following syntax is used to call a function:

`<name of function>([<list of parameters>])`

The list of parameters is a possibly empty, comma-separated list of expressions. Parameters are passed into the function in the order of their occurrence in the list. One or more of the last parameters in the list may be omitted. If omitted, they will be assigned default values; however, one or more parameters in the middle of the list may not be omitted when calling standard functions. All parameters of standard functions must be input.

Agreement of parameter data types of standard functions is checked at the compile time. During run time, the validity of parameter's values is checked. When an invalid value is passed to a standard function, the BAD PAR AM exception will be called.

Every expression may be considered as an operand when it is used with a specific operator in a compound expression.

Operators

All operators for all types of expressions are explained below. The operators are listed in descending order of precedence.

Arithmetic Expressions

The unary operators, '-', or '+', have the highest precedence.

The '-' operator followed by a numerical operand is considered to be a unary minus, meaning that the operand's sign should change.

The '+' operator is a unary plus, and does not affect the value of the number. The following binary operators have the second level of precedence:

- * Multiplication.
- / Division.
- \ Integer division, the remainder is dropped.
- ^ Power operator; raises the left operand to the power of the second operand.

When the divisor is equal to zero, the OTHERS exception is issued and the quotient is set to zero.

The following binary operators have the lowest level of precedence:

- + (addition);
- (subtraction).

String Expressions

In string expressions, there are just two operators. They have the same precedence:

- + (concatenate strings);
- (concatenate strings without preserving spaces).

Logical Expressions

The NOT, logical negation, operator and relational operators have the highest precedence among the logical operators.

The not operator is unary and has the following syntax:

NOT<logical_expression>

The following relational operators are supported:

- = equal to.
- <> not equal.

- > greater than.
- < less than.
- >= greater than or equal.
- <= less than or equal.

All relational operators are binary. They are positioned between two expressions that are to be compared. The expressions must have compatible data types. All types of expressions may be compared. Logical expressions may only be compared for equality or non-equality. An expression of any data type can also be compared for equality or non-equality with NULL. Date comparison is performed exactly, down to the level of time ticks.

The AND operator is the boolean. A binary operator, it is positioned between two logical expressions. During evaluation, the second expression is evaluated only if the first one is true.

Date Expressions

The following operators are used with DATE data types or expressions:

- \$ Evaluates the difference between two dates, in days, returning an integer. Both operands must be DATE data type. The evaluation of the date difference does not include hours. If the first date is less than the second one, the difference will be presented as a negative number. This operator has the highest priority.
- + Adds a given number of days to the date.
- Subtracts a given number of days from the date.

In the addition/subtraction operations with DATE, the left expression must be of the DATE data type. The second operand, the number of days, must be one of the numerical data types.

Value Assignments

The following methods are used to assign values to procedural language objects:

- 1) declaring local variables using a parameter with the DEFAULT value;
- 2) automatically to the input parameters of functions and procedures when they are called;
- 3) cursor fields are assigned values when executing FETCH and assign statements;
- 4) Automatically to predefined variables in triggers;
- 5) executing assign statements.

The ASSIGN operator assigns a value to a variable and has the following syntax:

```
<name_of_variable>:=<expression>
```

When executing the assignment, the value of <expression> is evaluated first. Then, the result of evaluation is assigned to the specified variable.

Since the assignment is understood as an expression having a value of its own, the right-side <expression>, it can be used as part of a compound expression, as in the C programming language. For example, the following assignment is valid:

```
a := b := c := o
```

The right side of the assignment may be NULL. Therefore, the following assignment is also valid:

```
i := s := d := NULL
```

even if i, s, and d are of different data types. This is true because a NULL value is compatible with all data types.

Conditional Expressions

A conditional expression has the following syntax:

```
EIF '[' <logical_expression> ']' <expression1> [ELSE <expression2>]
```

In the definition above, the square brackets following EIF are themselves required as a part of the expressions.

The <logical expression> is evaluated first, then, if the result of its evaluation is TRUE, <expression1> is evaluated. If <logical expression> is FALSE, <expression2>, if given, is evaluated.

When the ELSE <expression2> option is not used, the result of the conditional expression is NULL if the value of the <logical expression> is FALSE.

Standard Functions

This section describes the standard functions that may be used in expressions.

In descriptions of standard functions, the term numerical type is used as a generic name for one of the following data types: INT (integer), SMALLINT, REAL, or NUMERIC, (or DOUBLE).

String Functions

String functions are used in operations with string expressions. All string functions except the LEN function return a char type result.

LEN

Syntax

```
LEN(<string>);
```

where <string> is a char data type expression.

This function evaluates the length of a string and returns the numeric value of its length, in characters. If the string has a NULL value, NULL will be returned.

Example

```
str:=Time_table';
lenstr:=len(str);
```

TRIM

Syntax

```
TRIM (<string>);
```

where <string> is a char data type expression. TRIM removes the string's leading and trailing spaces.

Example

```
Str:='      1.      The name of column      \n      ';
Str:=trim(str);      //str='1.      The name of column      \n'
```

SUBSTR

Syntax

```
SUBSTR(<string>,<beg_pos>,<length>);
```

where

<string> char expression;

<beg_pos> integer with value ≥ 1 ;

<length> integer with value ≥ 0 .

SUBSTR returns a substring of <string>, beginning with the character at the <beg_pos> position and having the length of <length>.

If <length> is greater than the remaining string length, those remaining characters will be returned. If <string> has a NULL value, a NULL value will be returned.

If <string> is empty, an empty string will be returned. The return from NULL string and empty string are unaffected by any other parameters.

If <beg_pos> or <length> is improperly specified, an empty string will be returned.

Examples

```

1 str:="d.60-k.51";
  str:=substr(str,3,2); //60

2 str:="format:3B-####.#";
  str:= substr(str,8,len(str)); // 3B-####.#

3 str:=NULL;
  str:= substr(str,5,200); // NULL

4 str:=" ";
  str:= substr(str,5,200); // " "

5 str:="d.60-k.51";
  str:= substr(str,0,2); // " "
  str:= substr(str,-3,2); // " "
  str:= substr(str,2,-7); // " "
```

STRPOS

Syntax

STRPOS(<string>,<substring> [, <from_the_right>]);

where

<string> char expression;

<substring> char expression;

<from_the_right> TRUE | FALSE

STRPOS searches for the first occurrence of <substring> in <string>, starting from the left if the <from_the_right> option is set as FALSE. The search starts from the right if the <from_the_right> option is given with a value of TRUE. When STRPOS is first called, <from_the_right> is set to FALSE. An existing TRUE or FALSE setting persists until changed.

STRPOS returns the position, starting from 1 in the original string, where the specified substring begins.

If the substring is not found, the function returns zero. An attempt to search a NULL substring will generate an error during procedure compilation.

Examples

```

str:="Example of search of a substring\n";
pos:=strpos(str,search"); // 12
pos:=strpos(str,"\n",TRUE); // 32
pos:=strpos(str,""); // 0
pos:=strpos(str,"se",FALSE); // 12
```

```
pos:=strpos(str," s",TRUE); // 23
pos:=strpos(str,"Elamples"); // 0
```

DATE Functions

DAY

Syntax

```
DAY(<date>);
```

where <date> is a date data type expression.

DAY returns the day number in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;
num_day:=day(cur_date); // 17
```

MONTH

Syntax

```
MONTH (<date>);
```

where <date> is a date data type expression.

MONTH returns the month number in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;
num_month:=month(cur_date); // 11
```

YEAR

Syntax

```
YEAR(<date>);
```

where <date> is a date data type expression.

YEAR returns the year number in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;
num_year:=year(cur_date); // 1999
```

HOUR

Syntax

```
HOUR(<date>);
```

where <date> is a date data type expression.

HOUR returns the hour number in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;  
num_hour:=hour(cur_date);    // 18
```

MINUTE**Syntax**

MINUTE(<date>);

where <date> is a date data type expression.

MINUTE returns the minute number in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;  
num_minute:=minute(cur_date);    // 25
```

SECOND**Syntax**

SECOND(<date>);

where <date> is a date data type expression.

SECOND returns the number of seconds in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;  
num_second:=second(cur_date);    // 47
```

TICKS**Syntax**

TICKS(<date>);

where <date> is a date data type expression.

TICKS returns the tick number in the date.

Example

```
cur_date:=17.11.1999:18:25:47.88;  
num_ticks:=ticks(cur_date);    // 88
```

MAKE_DATE**Syntax**

MAKE_DATE(<day>,<month>,<year>,<hour>,<min>,<sec>,<tic>);

where all the expressions are numeric data types.

The MAKE_DATE function returns the value of the DATE data type for the specified day, month, year, hour, minute, second, and ticks. Any number of time parameters from the right can be omitted. Such omitted values are set to zero. If month or day values are invalid, the function returns 0.0.0:0:0:0.0 as the date.

Type Conversion Functions

A single parameter of one data type is passed to type conversion functions, which return a value of a different type.

ITOA

Syntax

ITOA<number>

where <number> is a numeric data type expression.

ITOA returns a string of the signed <number>. The plus, +, sign is omitted. If <number> is not an integer, the fractional digits are omitted.

Examples

```
str_num:=itoe(0); // '0'  
str_num:=itoe(NULL); // 'NULL'  
str_num:=itoe(123); // '123'  
str_num:=itoe(-27); // '-27'  
str_num:=itoe(3.1415); // '3'  
str_num :=itoe(77.); // '77'
```

FOTA

Syntax

FOTA<number>

where <number> is a numeric data type expression.

FOTA returns a string of the signed <number>. <number> is interpreted as a real data type. Numeric conversion is based on the following rules:

- 1) The plus, +, sign is omitted for positive numbers;
- 2) The total number of decimal digits in the resulting string, including the integral, fractional digits, and the decimal point, will not exceed 7;
- 3) If the number's value goes beyond the conversion range, all insignificant zeros will be truncated.

Examples

```
Str_num:=ftoe(5); // '5'  
Str_num:=ftoe(NULL); // 'NULL'  
str_num:=ftoe(09999.348); // '9999.35'  
str_num:=ftoe(0.0001); // '0.0001'  
str_num:=ftoe(-27.1239); // '-27.1239'  
Str_num:=ftoe(3.1415); // '3.1415'  
Str_num:=ftoe(77.); // '77'  
Str_num:=ftoe(89.56); // '89.56'  
Str_num :=ftoe(89.5600); // '89.56'
```

NTOA

Syntax

NTOA(<number>[,<size>[,<precision>]]);

where the arguments are all numeric data type expressions and

<number> is interpreted as a real number;

<size>total length of the result string including all digits, the sign, and the decimal point;

<precision> specifies the number of fractional digits.

NTOA returns a string of the signed <number> with a specific precision. By default, the string's size is 30 and precision is 10. This function operates under the same rules of conversion, except the range of values, as the FTOA function.

DOTA

Syntax

DTOA(<date>[,<ptime>]);

where

<date> date data type expression;

<ptime> TRUE | FALSE DTOA returns the date as a string:

If <ptime>is not specified or is set to FALSE, the returned string does not include time.

If <ptime> is set to TRUE, the returned string includes all the time parameters.

Examples

```
cur_date:=18.11.1999:14:27:48.89;
str_date:=dtoa(cur_dat); // '18.11.1999:14:27:48.89'
str_date:=dtoa(cur_dat, FALSE); // '18.11.1999'
str_date:=dtoa(cur_dat,TRUE); // '1 8.11.1999:14:27:48.89'
```

ATOD

Syntax

ATOD(<string>);

where <string> is a char data type expression.

ATOD returns a value of the date data type as a result of converting <string> which must be in the DD.MM.YYYY[.HH[:MI[:SS[:TI]]]] format . If <string> has an invalid format, 0.0.0:0:0:0.0 is returned.

Examples

```
str_date:="18.11.1999:14:27:48.89";
cur_date:=atod(str_dat); // 18.11.1999:14:27:48.89

str_date:="18.11.1999:14:27:48";
cur_date:=atod(str_dat); // 18.11.1999:14:27:48.0
```

```

str_date:="18.11.1999";
cur_date:=atod(str_dat); // 18.11.1999:0:0:0.0

str_date:="18.11";
cur_date:=atod(str_dat); // 0.0.0:0:0:0.0

str_date:="18.11";
cur_date:=atod(str_dat); // 0.0.0:0:0:0.0
str_date:=" ";
cur_date:=atod(str_dat); // 0.0.0:0:0:0.0

str_date:="18.15.1999:14:27:48.89";
cur_date:=atod(str_dat); // 0.0.0:0:0:0.0

```

TOSMALLINT

Syntax

TOSMALLINT(<numeric> | <char>);

where <value> is any numeric data type expression.

TOSMALLINT returns a smallint data type value as a result of converting its argument, using the following rules:

- 1) Any fractional digits are omitted;
- 2) The function exits when the first non-digit character is found in the char string;
- 3) If the <value> is outside the smallint range (-32,767 to +32,767), the remainder resulting from the division of <value> by 65,536, including the sign bit, is returned, i.e., <value> = mod 65536.

Examples

```

sml_int:=tosmallint("148"); // 148
sml_int:=tosmallint(148); // 148
sml_int:=tosmallint("-34"); // -34
sml_int:=tosmallint(-34); // -34
sml_int:=tosmallint(+34); // 34
sml_int:=tosmallint(146 - 56 / 8+10); // 151
sml_int:=tosmallint("65535"); // -1
sml_int:=tosmallint(65536); // 0
sml_int:=tosmallint(70000); // 4464
sml_int:=tosmallint(-70000); // -4464
sml_int:=tosmallint(6fs65); // 6
sml_int:=tosmallint(65.9); // 65
sml_int:=tosmallint(65535*2+1000); // 998

```

The following three functions, TOINTEGER, TOREAL, and TONUMERIC, convert the function's argument to the function's data type. If a string that is not a valid number is passed to the function, the value returned is zero.

TOINTEGER

Syntax

TOINTEGER(<numeric> | <char>);

where the argument is any data type expression that can be converted to an integer.

TOINTEGER returns an integer data type value as a result of converting its argument

TOREAL

Syntax

TOREAL(<numeric> | <char>);

where the argument is any data type expression capable of being converted to a real number.

TOREAL returns a real data type value as a result of converting its argument.

TONUMERIC

Syntax

TONUMERIC(<numeric> | <char>);

where the argument is any data type expression capable of being converted to the numeric data type.

TONUMERIC returns a numeric data type value as a result of converting its argument.

Cursor Functions

OUTOFCURSOR

Syntax

OUTOFCURSOR(<cursor>);

where <cursor> is a cursor variable.

OUTOFCURSOR returns the logical value TRUE, if an attempt is made to fetch a row beyond the cursor, otherwise, FALSE.

Fetching beyond the cursor occurs when:

- 1) the FETCH NEXT statement is performed when the cursor is positioned at the last row;
- 2) the FETCH PREVIOUS statement is performed while the cursor is positioning at the first row; or
- 3) as the result of the FETCH ABSOLUTE or FETCH RELATIVE statements, a row with a non-existent number is called.

Example

Use of OUTOFCURSOR function where curs is the cursor variable:

```

// Typical sequence of statements to retrieve all rows
open curs for ... ; // open cursor
// fetch curs last // to retrieve in reverse order
while not outofcursor(curs) loop
...
// processing results
... fetch curs;
// fetch curs previous; // to fetch in reverse order.
endloop

```

ROWCOUNT

Syntax

ROWCOUNT[(<cursor>)];

where <cursor> is a cursor variable.

ROWCOUNT returns an integer specifying Linter's exit code for <cursor>. If no cursor variable is included, the exit code for the last EXECUTE statement is returned.

ERRCODE

Syntax

ERRCODE(<cursor>);

where <cursor> is a cursor variable.

ERRCODE returns an integer specifying the e cursor.

Example

```

... execute 'select * from AUTO where personid=345';
IF ERRCODE() <> 0 THEN return (TRUE)
ENDIF
... EXCEPTIONS
WHEN DIVZERO THEN
// in this case, it makes no sense to use ERRCODE
...
WHEN BADPARAM, BADRETVAL, UNDEFPROC THEN
print ('Error in declaration or procedure:' errcode( )
// here it makes sense to use errcode only for codes not
// specifically declared as exceptions in EXCEPTIONS
WHEN OTHERS THEN
    case errcode ( )
...

```

Miscellaneous Functions

MAX

Syntax

MAX(<number1 >,<number2>);

where arguments are numeric data type expressions.

MAX returns the larger of two values.

Examples

```
max_num:=max(5,10); // 10
max_num:=max(5,8.6); // 8
max_num:=max(-5,-10); // -5
max_num:=max(tointeger("345"),toreal(567)); // 567
max_num:=max(5,(10,max(4,9))); // 10
```