CLSQL Users' Guide

Kevin M. Rosenberg
Pierre R. Mai
onShore Development, Inc.

CLSQL Users' Guide

by Kevin M. Rosenberg, Pierre R. Mai, and onShore Development, Inc.

Table of Contents

Preface

This guide provides reference to the features of *CLSQL*. The first chapter provides an introduction to *CLSQL* and installation instructions. The reference sections document all user accessible symbols with examples of usage. There is a glossary of commonly used terms with their definitions.

Chapter 1. Introduction

Purpose

CLSQL is a Common Lisp interface to SQL databases. A number of Common Lisp implementations and SQL databases are supported. The general structure of *CLSQL* is based on the CommonSQL package by Xanalys.

History

The *CLSQL* project was started by Kevin M. Rosenberg in 2001 to support SQL access on multiple Common Lisp implementations using the *UFFI* library. The initial code was based substantially on Pierre R. Mai's excellent *MaiSQL* package. In late 2003, the UncommonSQL library was orphaned by its author, onShore Development, Inc. In April 2004, Marcus Pearce ported the UncommonSQL library to *CLSQL*. The UncommonSQL library provides a CommonSQL-compatible API for *CLSQL*.

The main changes from *MaiSQL* and UncommonSQL are:

- Port from the CMUCL FFI to *UFFI* which provide compatibility with the major Common Lisp implementations.

- Optimized loading of integer and floating-point fields.

- Additional database backends: ODBC, AODBC, and SQLite.

- A compatibility layer for CMUCL specific code.

- Much improved robustness for the MySQL back-end along with version 4 client library support.

- Improved library loading and installation documentation.

- Improved packages and symbol export.

- Pooled connections.

- Integrated transaction support for the classic *MaiSQL* iteration macros.

Prerequisites

ASDF

CLSQL uses ASDF to compile and load its components. ASDF is included in the *CCLAN* [http://cclan.sourceforge.net] collection.

UFFI

CLSQL uses *UFFI* [http://uffi.b9.com/] as a *Foreign Function Interface* (FFI) to support multiple ANSI Common Lisp implementations.

MD5

CLSQL's postgresql-socket interface uses Pierre Mai's md5 [ftp://clsql.b9.com/] module.

Supported Common Lisp Implementation

The implementations that support *CLSQL* is governed by the supported implementations of *UFFI*. The following implementations are supported:

- AllegroCL v6.2 and 7.0b on Debian Linux x86 & x86_64 & PowerPC, FreeBSD 4.5, and Microsoft Windows XP.

- Lispworks v4.3 on Debian Linux and Microsoft Windows XP.

- CMUCL 18e on Debian Linux, FreeBSD 4.5, and Solaris 2.8.

- SBCL 0.8.5 on Debian Linux.

- SCL 1.1.1 on Debian Linux.

- OpenMCL 0.14 on Debian Linux PowerPC.

Supported SQL Implementation

Currently, *CLSQL* supports the following databases:

- MySQL v3.23.51 and v4.0.18.

- PostgreSQL v7.4 with both direct API and TCP socket connections.

- SQLite.

- Direct ODBC interface.

- Oracle OCI.

- Allegro's DB interface (AODBC).

Installation

Ensure ASDF is loaded

Simply load the file `asdf.lisp`.

```
(load "asdf.lisp")
```

Build C helper libraries

CLSQL uses functions that require 64-bit integer parameters and return values. The *FFI* in most *CLSQL* implementations do not support 64-bit integers. Thus, C helper libraries are required to break these 64-bit integers into two compatible 32-bit integers. The helper libraries reside in the directories `uffi`

and `db-mysql`.

Microsoft Windows

Files named `Makefile.msvc` are supplied for building the libraries under Microsoft Windows. Since Microsoft Windows does not come with that compiler, compiled DLL and LIB library files are supplied with *CLSQL*.

UNIX

Files named `Makefile` are supplied for building the libraries under UNIX. Loading the `.asd` files automatically invokes make when necessary. So, manual building of the helper libraries is not necessary on most UNIX systems. However, the location of the MySQL library files and include files may need to adjusted in `db-mysql/Makefile` on non-Debian systems.

Add *UFFI* path

Unzip or untar the *UFFI* distribution which creates a directory for the *UFFI* files. Add that directory to ASDF's `asdf:*central-registry*`. You can do that by pushing the pathname of the directory onto this variable. The following example code assumes the *UFFI* files reside in the `/usr/share/lisp/uffi/` directory.

```
(push #P"/usr/share/lisp/uffi/" asdf:*central-registry*)
```

Add MD5 path

If you plan to use the clsql-postgresql-socket interface, you must load the md5 module. Unzip or untar the cl-md5 distribution, which creates a directory for the cl-md5 files. Add that directory to ASDF's `asdf:*central-registry*`. You can do that by pushing the pathname of the directory onto this variable. The following example code assumes the cl-md5 files reside in the `/usr/share/lisp/cl-md5/` directory.

```
(push #P"/usr/share/lisp/cl-md5/" asdf:*central-registry*)
```

Add *CLSQL* path and load module

Unzip or untar the *CLSQL* distribution which creates a directory for the *CLSQL* files. Add that directory to ASDF's `asdf:*central-registry*`. You can do that by pushing the pathname of the directory onto this variable. The following example code assumes the *CLSQL* files reside in the `/usr/share/lisp/clsql/` directory. You need to load the clsql system.

```
(push #P"/usr/share/lisp/clsql/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op 'clsql)                    ; main CLSQL package
```

Run test suite (optional)

The test suite can be executed using the ASDF test-op operator. If *CLSQL* has not been loaded with asdf:load-op, the asdf:test-op operator will automatically load *CLSQL*. A configuration file named

`.clsql-test.config` must be created in your home directory. There are instructures on the format of that file in the `tests/README`. After creating `.clsql-test.config`, you can run the test suite with ASDF:

```
(asdf:operate 'asdf:test-op 'clsql)
```

Chapter 2. CommonSQL Tutorial

Based on the UncommonSQL Tutorial

Introduction

The goal of this tutorial is to guide a new developer thru the process of creating a set of *CLSQL* classes providing a Object-Oriented interface to persistent data stored in an SQL database. We will assume that the reader is familiar with how SQL works, how relations (tables) should be structured, and has created at least one SQL application previously. We will also assume a minor level of experience with Common Lisp.

CLSQL provides two different interfaces to SQL databases, a Functional interface, and an Object-Oriented interface. The Functional interface consists of a special syntax for embedded SQL expressions in Lisp, and provides lisp functions for SQL operations like SELECT and UPDATE. The object-oriented interface provides a way for mapping Common Lisp Objects System (CLOS) objects into databases and includes functions for inserting new objects, querying objects, and removing objects. Most applications will use a combination of the two.

CLSQL is based on the CommonSQL package from Xanalys, so the documentation that Xanalys makes available online is useful for *CLSQL* as well. It is suggested that developers new to *CLSQL* read their documentation as well, as any differences between CommonSQL and *CLSQL* are minor. Xanalys makes the following documents available:

- *Xanalys Lispworks User Guide - The CommonSQL Package* [http://www.lispworks.com/reference/lw43/LWUG/html/lwuser-167.htm]

- *Xanalys Lispworks Reference Manual - The SQL Package* [http://www.lispworks.com/reference/lw43/LWRM/html/lwref-383.htm]

- *CommonSQL Tutorial by Nick Levine* [http://www.ravenbrook.com/doc/2002/09/13/common-sql/]

Data Modeling with *CLSQL*

Before we can create, query and manipulate *CLSQL* objects, we need to define our data model as noted by Philip Greenspun [1]

When data modeling, you are telling the relational database management system (RDBMS) the following:

- What elements of the data you will store.

- How large each element can be.

- What kind of information each element can contain.

- What elements may be left blank.

- Which elements are constrained to a fixed range.

- Whether and how various tables are to be linked.

---

[1] *Philip Greenspun's "SQL For Web Nerds" - Data Modeling* [http://www.arsdigita.com/books/sql/data-modeling.html]

With SQL database one would do this by defining a set of relations, or tables, followed by a set of queries for joining the tables together in order to construct complex records. However, with *CLSQL* we do this by defining a set of CLOS classes, specifying how they will be turned into tables, and how they can be joined to one another via relations between their attributes. The SQL tables, as well as the queries for joining them together are created for us automatically, saving us from dealing with some of the tedium of SQL.

Let us start with a simple example of two SQL tables, and the relations between them.

```
CREATE TABLE EMPLOYEE ( emplid     NOT NULL number(38),
                        first_name NOT NULL varchar2(30),
                        last_name  NOT NULL varchar2(30),
                        email               varchar2(100),
                        companyid  NOT NULL number(38),
                        managerid           number(38))

CREATE TABLE COMPANY ( companyid   NOT NULL number(38),
                       name        NOT NULL varchar2(100),
                       presidentid NOT NULL number(38))
```

This is of course the canonical SQL tutorial example, "The Org Chart".

In *CLSQL*, we would have two "view classes" (a fancy word for a class mapped into a database). They would be defined as follows:

```
(clsql:def-view-class employee ()
  ((emplid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :emplid)
   (first-name
    :accessor first-name
    :type (string 30)
    :initarg :first-name)
   (last-name
    :accessor last-name
    :type (string 30)
    :initarg :last-name)
   (email
    :accessor employee-email
    :type (string 100)
    :nulls-ok t
    :initarg :email)
   (companyid
    :type integer
    :initarg :companyid)
   (managerid
    :type integer
    :nulls-ok t
    :initarg :managerid))
  (:base-table employee))

(clsql:def-view-class company ()
  ((companyid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :companyid)
   (name
```

```
   :type (string 100)
   :initarg :name)
  (presidentid
   :type integer
   :initarg :presidentid))
(:base-table company))
```

The `DEF-VIEW-CLASS` macro is just like the normal CLOS `DEFCLASS` macro, except that it handles several slot options that `DEFCLASS` doesn't. These slot options have to do with the mapping of the slot into the database. We only use a few of the slot options in the above example, but there are several others.

- :column - The name of the SQL column this slot is stored in. Defaults to the slot name. If the slot name is not a valid SQL identifier, it is escaped, so foo-bar becomes foo_bar.

- :db-kind - The kind of database mapping which is performed for this slot. :base indicates the slot maps to an ordinary column of the database view. :key indicates that this slot corresponds to part of the unique keys for this view, :join indicates a join slot representing a relation to another view and :virtual indicates that this slot is an ordinary CLOS slot. Defaults to :base.

- :db-reader - If a string, then when reading values from the database, the string will be used for a format string, with the only value being the value from the database. The resulting string will be used as the slot value. If a function then it will take one argument, the value from the database, and return the value that should be put into the slot.

- :db-writer - If a string, then when reading values from the slot for the database, the string will be used for a format string, with the only value being the value of the slot. The resulting string will be used as the column value in the database. If a function then it will take one argument, the value of the slot, and return the value that should be put into the database.

- :column- - A string which will be used as the type specifier for this slots column definition in the database.

- :void-value - The Lisp value to return if the field is `NULL`. The default is NIL.

- :db-info - A join specification.

In our example each table as a primary key attribute, which is required to be unique. We indicate that a slot is part of the primary key (*CLSQL* supports multi-field primary keys) by specifying the :db-kind key slot option.

The SQL type of a slot when it is mapped into the database is determined by the :type slot option. The argument for the :type option is a Common Lisp datatype. The *CLSQL* framework will determine the appropriate mapping depending on the database system the table is being created in. If we really wanted to determine what SQL type was used for a slot, we could specify a :db-type option like "NUMBER(38)" and we would be guaranteed that the slot would be stored in the database as a NUMBER(38). This is not recomended because it could makes your view class unportable across database systems.

`DEF-VIEW-CLASS` also supports some class options, like :base-table. The :base-table option specifies what the table name for the view class will be when it is mapped into the database.

Class Relations

In an SQL only application, the EMPLOYEE and COMPANY tables can be queried to determine things like, "Who is Vladamir's manager?", "What company does Josef work for?", and "What employees work

for Widgets Inc.". This is done by joining tables with an SQL query.

Who works for Widgets Inc.?

```
SELECT first_name, last_name FROM employee, company
        WHERE employee.companyid = company.companyid
                AND company.company_name = "Widgets Inc."
```

Who is Vladamir's manager?

```
SELECT managerid FROM employee
        WHERE employee.first_name = "Vladamir"
                AND employee.last_name = "Lenin"
```

What company does Josef work for?

```
SELECT company_name FROM company, employee
        WHERE employee.first_name = "Josef"
                AND employee.last-name = "Stalin"
                AND employee.companyid = company.companyid
```

With *CLSQL* however we do not need to write out such queries because our view classes can maintain the relations between employees and companies, and employees to their managers for us. We can then access these relations like we would any other attribute of an employee or company object. In order to do this we define some join slots for our view classes.

What company does an employee work for? If we add the following slot definition to the employee class we can then ask for it's COMPANY slot and get the appropriate result.

```
;; In the employee slot list
(company
  :accessor employee-company
  :db-kind :join
  :db-info (:join-class company
            :home-key companyid
            :foreign-key companyid
            :set nil))
```

Who are the employees of a given company? And who is the president of it? We add the following slot definition to the company view class and we can then ask for it's EMPLOYEES slot and get the right result.

```
;; In the company slot list
(employees
  :reader company-employees
  :db-kind :join
  :db-info (:join-class employee
            :home-key companyid
            :foreign-key companyid
            :set t))

 (president
  :reader president
```

```
                :db-kind :join
                :db-info (:join-class employee
                          :home-key presidentid
                          :foreign-key emplid
                          :set nil))
```

And lastly, to define the relation between an employee and their manager:

```
            ;; In the employee slot list
          (manager
           :accessor employee-manager
           :db-kind :join
           :db-info (:join-class employee
                     :home-key managerid
                     :foreign-key emplid
                     :set nil))
```

CLSQL join slots can represent one-to-one, one-to-many, and many-to-many relations. Above we only have one-to-one and one-to-many relations, later we will explain how to model many-to-many relations. First, let's go over the slot definitions and the available options.

In order for a slot to be a join, we must specify that it's :db-kind :join, as opposed to :base or :key. Once we do that, we still need to tell *CLSQL* how to create the join statements for the relation. This is what the :db-info option does. It is a list of keywords and values. The available keywords are:

- :join-class - The view class to which we want to join. It can be another view class, or the same view class as our object.

- :home-key - The slot(s) in the immediate object whose value will be compared to the foreign-key slot(s) in the join-class in order to join the two tables. It can be a single slot-name, or it can be a list of slot names.

- :foreign-key - The slot(s) in the join-class which will be compared to the value(s) of the home-key.

- :set - A boolean which if false, indicates that this is a one-to-one relation, only one object will be returned. If true, than this is a one-to-many relation, a list of objects will be returned when we ask for this slots value.

There are other :join-info options available in *CLSQL*, but we will save those till we get to the many-to-many relation examples.

Object Creation

Now that we have our model laid out, we should create some object. Let us assume that we have a database connect set up already. We first need to create our tables in the database:

Note: the file examples/clsql-tutorial.lisp contains view class definitions which you can load into your list at this point in order to play along at home.

```
(clsql:create-view-from-class 'employee)
(clsql:create-view-from-class 'company)
```

Then we will create our objects. We create them just like you would any other CLOS object:

```
(defvar company1 (make-instance 'company
                                :companyid 1
                                :presidentid 1
                                :name "Widgets Inc."))

(defvar employee1 (make-instance 'employee
                                 :emplid 1
                                 :first-name "Vladamir"
                                 :last-name "Lenin"
                                 :email "lenin@soviet.org"
                                 :companyid 1))

(defvar employee2 (make-instance 'employee
                                 :emplid 2
                                 :first-name "Josef"
                                 :last-name "Stalin"
                                 :email "stalin@soviet.org"
                                 :companyid 1
                                 :managerid 1))
```

In order to insert an objects into the database we use the UPDATE-RECORDS-FROM-INSTANCE func-
tion as follows:

```
(clsql:update-records-from-instance employee1)
(clsql:update-records-from-instance employee2)
(clsql:update-records-from-instance company1)
```

After you make any changes to an object, you have to specifically tell *CLSQL* to update the SQL data-
base. The UPDATE-RECORDS-FROM-INSTANCE method will write all of the changes you have made
to the object into the database.

Since *CLSQL* objects are just normal CLOS objects, we can manipulate their slots just like any other ob-
ject. For instance, let's say that Lenin changes his email because he was getting too much spam from the
German Socialists.

```
;; Print Lenin's current email address, change it and save it to the
;; database.  Get a new object representing Lenin from the database
;; and print the email

;; This lets us use the functional CLSQL interface with [] syntax
(clsql:locally-enable-sql-reader-syntax)

(format t "The email address of ~A ~A is ~A"
        (first-name employee1)
        (last-name employee1)
        (employee-email employee1))

(setf (employee-email employee1) "lenin-nospam@soviets.org")

;; Update the database
(clsql:update-records-from-instance employee1)

(let ((new-lenin (car (clsql:select 'employee
                        :where [= [slot-value 'employee 'emplid] 1]))))
     (format t "His new email is ~A"
          (employee-email new-lenin)))
```

Everything except for the last `LET` expression is already familiar to us by now. To understand the call to `CLSQL:SELECT` we need to discuss the Functional SQL interface and it's integration with the Object Oriented interface of *CLSQL*.

Finding Objects

Now that we have our objects in the database, how do we get them out when we need to work with them? *CLSQL* provides a functional interface to SQL, which consists of a special Lisp reader macro and some functions. The special syntax allows us to embed SQL in lisp expressions, and lisp expressions in SQL, with ease.

Once we have turned on the syntax with the expression:

```
(clsql:locally-enable-sql-reader-syntax)
```

We can start entering fragments of SQL into our lisp reader. We will get back objects which represent the lisp expressions. These objects will later be compiled into SQL expressions that are optimized for the database backed we are connected to. This means that we have a database independent SQL syntax. Here are some examples:

```
;; an attribute or table name
[foo] => #<CLSQL-SYS::SQL-IDENT-ATTRIBUTE FOO>

;; a attribute identifier with table qualifier
[foo bar] => #<CLSQL-SYS::SQL-IDENT-ATTRIBUTE FOO.BAR>

;; a attribute identifier with table qualifier
[= "Lenin" [first_name]] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP ('Lenin' = FIRST_NAME)>

[< [emplid] 3] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP (EMPLID < 3)>

[and [< [emplid] 2] [= [first_name] "Lenin"]] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP ((EMPLID < 2) AND
                                     (FIRST_NAME = 'Lenin'))>


;; If we want to reference a slot in an object we can us the
;;  SLOT-VALUE sql extension
[= [slot-value 'employee 'emplid] 1] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP (EMPLOYEE.EMPLID = 1)>

[= [slot-value 'employee 'emplid]
   [slot-value 'company 'presidentid]] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP (EMPLOYEE.EMPLID = COMPANY.PRESIDENTID)>
```

The `SLOT-VALUE` operator is important because it let's us query objects in a way that is robust to any changes in the object->table mapping, like column name changes, or table name changes. So when you are querying objects, be sure to use the `SLOT-VALUE` SQL extension.

Since we can now formulate SQL relational expression which can be used as qualifiers, like we put after the WHERE keyword in SQL statements, we can start querying our objects. *CLSQL* provides a function SELECT which can return use complete objects from the database which conform to a qualifier, can be sorted, and various other SQL operations.

The first argument to SELECT is a class name. it also has a set of keyword arguments which are covered in the documentation. For now we will concern ourselves only with the :where keyword. Select returns a list of objects, or nil if it can't find any. It's important to remember that it always returns a list, so even if you are expecting only one result, you should remember to extract it from the list you get from SE-LECT.

```
;; all employees
(clsql:select 'employee)
;; all companies
(clsql:select 'company)

;; employees named Lenin
(clsql:select 'employee :where [= [slot-value 'employee 'last-name]
                                 "Lenin"])

(clsql:select 'company :where [= [slot-value 'company 'name]
                                "Widgets Inc."])

;; Employees of Widget's Inc.
(clsql:select 'employee
            :where [and [= [slot-value 'employee 'companyid]
                          [slot-value 'company 'companyid]]
                      [= [slot-value 'company 'name]
                        "Widgets Inc."]])

;; Same thing, except that we are using the employee
;; relation in the company view class to do the join for us,
;; saving us the work of writing out the SQL!
(company-employees company1)

;; President of Widgets Inc.
(president company1)

;; Manager of Josef Stalin
(employee-manager employee2)
```

Deleting Objects

Now that we know how to create objects in our database, manipulate them and query them (including using our predefined relations to save us the trouble writing alot of SQL) we should learn how to clean up after ourself. It's quite simple really. The function DELETE-INSTANCE-RECORDS will remove an object from the database. However, when we remove an object we are responsible for making sure that the database is left in a correct state.

For example, if we remove a company record, we need to either remove all of it's employees or we need to move them to another company. Likewise if we remove an employee, we should make sure to update any other employees who had them as a manager.

Conclusion

There are many nooks and crannies to *CLSQL*, some of which are covered in the Xanalys documents we refered to earlier, some are not. The best documentation at this time is still the source code for *CLSQL* itself and the inline documentation for its various functions.

Connection and Initialisation

Name

DATABASE -- The super-type of all *CLSQL* databases

DATABASE

Class Precedence List

database, standard-object, t

Description

This class is the superclass of all *CLSQL* databases. The different database back-ends derive subclasses of this class to implement their databases. No instances of this class are ever created by *CLSQL*.

Class details

```
(defclass DATABASE ()(...))
```

Slots

Name

CONNECT-IF-EXISTS -- Default value for the `if-exists` parameter of `connect`.

CONNECT-IF-EXISTS

Value Type

A valid argument to the `if-exists` parameter of `connect`, i.e. one of :new, :warn-new, :error, :warn-old, :old.

Initial Value

:error

Description

The value of this variable is used in calls to `connect` as the default value of the `if-exists` parameter. See `connect` for the semantics of the valid values for this variable.

Examples

None.

Affected By

None.

See Also

`connect`

Notes

None.

Name

DEFAULT-DATABASE -- The default database object to use

DEFAULT-DATABASE

Value Type

Any object of type database, or nil to indicate no default database.

Initial Value

nil

Description

Any function or macro in *CLSQL* that operates on a database uses the value of this variable as the default value for it's `database` parameter.

The value of this parameter is changed by calls to `connect`, which sets *default-database* to the database object it returns. It is also changed by calls to `disconnect`, when the database object being disconnected is the same as the value of *default-database*. In this case `disconnect` sets *default-database* to the first database that remains in the list of active databases as returned by `connected-databases`, or nil if no further active databases exist.

The user may change *default-database* at any time to a valid value of his choice.

Caution

If the value of *default-database* is nil, then all calls to *CLSQL* functions on databases must provide a suitable `database` parameter, or an error will be signalled.

Examples

```
(connected-databases)
=> NIL
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48385F55}>
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {483868FD}>
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql :if-exist
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48387265}>
*default-database*
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48387265}>
(disconnect)
=> T
*default-database*
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {483868FD}>
(disconnect)
=> T
*default-database*
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48385F55}>
(disconnect)
=> T
*default-database*
```

```
=> NIL
(connected-databases)
=> NIL
```

Affected By

```
connect
disconnect
```

See Also

```
connected-databases
```

Notes

Note

This variable is intended to facilitate working with *CLSQL* in an interactive fashion at the top-level loop, and because of this, `connect` and `disconnect` provide some fairly complex behaviour to keep *default-database* set to useful values. Programmatic use of *CLSQL* should never depend on the value of *default-database* and should provide correct database objects via the `database` parameter to functions called.

Name

DEFAULT-DATABASE-TYPE -- The default database type to use

DEFAULT-DATABASE-TYPE

Value Type

Any keyword representing a valid database back-end of *CLSQL*, or nil.

Initial Value

nil

Description

The value of this variable is used in calls to `initialize-database-type` and `connect` as the default value of the `database-type` parameter.

Caution

If the value of this variable is nil, then all calls to `initialize-database-type` or `connect` will have to specify the `database-type` to use, or a general-purpose error will be signalled.

Examples

```
(setf *default-database-type* :mysql)
=> :mysql
(initialize-database-type)
=> t
```

Affected By

None.

See Also

None.

Notes

None.

Name

INITIALIZED-DATABASE-TYPES -- List of all initialized database types

INITIALIZED-DATABASE-TYPES

Value Type

A list of all initialized database types, each of which represented by it's corresponding keyword.

Initial Value

nil

Description

This variable is updated whenever `initialize-database-type` is called for a database type which hasn't already been initialized before, as determined by this variable. In that case the keyword representing the database type is pushed onto the list stored in *INITIALIZED-DATABASE-TYPES*.

Caution

Attempts to modify the value of this variable will result in undefined behaviour.

Examples

```
(setf *default-database-type* :mysql)
=> :mysql
(initialize-database-type)
=> t
*initialized-database-types*
=> (:MYSQL)
```

Affected By

`initialize-database-type`

See Also

None.

Notes

Direct access to this variable is primarily provided because of compatibility with Harlequin's Common SQL.

Name

CONNECT --

CONNECT

Syntax

```
(CONNECT CONNECTION-SPEC &KEY (IF-EXISTS *CONNECT-IF-EXISTS*) (MAKE-DEFAULT
```

Arguments and Values

Description

Connects to a database of the supplied DATABASE-TYPE which defaults to *DEFAULT-DATABASE-TYPE*, using the type-specific connection specification CONNECTION-SPEC. The value of IF-EXISTS, which defaults to *CONNECT-IF-EXISTS*, determines what happens if a connection to the database specified by CONNECTION-SPEC is already established. A value of :new means create a new connection. A value of :warn-new means warn the user and create a new connect. A value of :warn-old means warn the user and use the old connection. A value of :error means fail, notifying the user. A value of :old means return the old connection. MAKE-DEFAULT is t by default which means that *DEFAULT-DATABASE* is set to the new connection, otherwise *DEFAULT-DATABASE* is not changed. If POOL is t the connection will be taken from the general pool, if POOL is a CONN-POOL object the connection will be taken from this pool.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

CONNECTED-DATABASES --

CONNECTED-DATABASES

Syntax

```
(CONNECTED-DATABASES) [function] =>
```

Arguments and Values

Description

Returns the list of active database objects.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

CREATE-DATABASE --

CREATE-DATABASE

Syntax

```
(CREATE-DATABASE CONNECTION-SPEC &KEY DATABASE-TYPE) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DATABASE-NAME --

DATABASE-NAME

Syntax

```
(DATABASE-NAME (OBJ DATABASE)) [reader] =>
```

Arguments and Values

Description

"Returns the name of DATABASE."

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DATABASE-TYPE --

DATABASE-TYPE

Syntax

```
(DATABASE-TYPE (OBJ DATABASE)) [reader] =>
```

Arguments and Values

Description

"Returns the type of DATABASE."

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DESTROY-DATABASE --

DESTROY-DATABASE

Syntax

```
(DESTROY-DATABASE CONNECTION-SPEC &KEY DATABASE-TYPE) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DISCONNECT --

DISCONNECT

Syntax

```
(DISCONNECT &KEY (DATABASE *DEFAULT-DATABASE*) (ERROR NIL)) [function] =>
```

Arguments and Values

Description

Closes the connection to DATABASE and resets *DEFAULT-DATABASE* if that database was disconnected. If DATABASE is a database instance, this object is closed. If DATABASE is a string, then a connected database whose name matches DATABASE is sought in the list of connected databases. If no matching database is found and ERROR and DATABASE are both non-nil an error is signaled, otherwise nil is returned. If the database is from a pool it will be released to this pool.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DISCONNECT-POOLED --

DISCONNECT-POOLED

Syntax

```
(DISCONNECT-POOLED &OPTIONAL CLEAR) [function] =>
```

Arguments and Values

Description

Disconnects all connections in the pool.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

FIND-DATABASE --

FIND-DATABASE

Syntax

```
(FIND-DATABASE DATABASE &KEY (ERRORP T) (DB-TYPE NIL)) [function] =>
```

Arguments and Values

Description

Returns the connected databases of type DB-TYPE whose names match the string DATABASE. If DATABASE is a database object, it is returned. If DB-TYPE is nil all databases matching the string DATABASE are considered. If no matching databases are found and ERRORP is nil then nil is returned. If ERRORP is nil and one or more matching databases are found, then the most recently connected database is returned as a first value and the number of matching databases is returned as a second value. If no, or more than one, matching databases are found and ERRORP is true, an error is signalled.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

INITIALIZE-DATABASE-TYPE --

INITIALIZE-DATABASE-TYPE

Syntax

```
(INITIALIZE-DATABASE-TYPE &KEY (DATABASE-TYPE *DEFAULT-DATABASE-TYPE*)) [fu
```

Arguments and Values

Description

Initializes the supplied DATABASE-TYPE, if it is not already initialized, as indicated by *INITIALIZED-DATABASE-TYPES* and returns DATABASE-TYPE. *DEFAULT-DATABASE-TYPE* is set to DATABASE-TYPE and, if DATABASE-TYPE has not been initialised, it is added to *INITIALIZED-DATABASE-TYPES*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-DATABASES --

LIST-DATABASES

Syntax

```
(LIST-DATABASES CONNECTION-SPEC &KEY DATABASE-TYPE) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

PROBE-DATABASE --

PROBE-DATABASE

Syntax

```
(PROBE-DATABASE CONNECTION-SPEC &KEY DATABASE-TYPE) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

RECONNECT --

RECONNECT

Syntax

```
(RECONNECT &KEY (DATABASE *DEFAULT-DATABASE*) (ERROR NIL) (FORCE T)) [funct
```

Arguments and Values

Description

Reconnects DATABASE which defaults to *DEFAULT-DATABASE* to the underlying database management system. On success, t is returned and the variable *DEFAULT-DATABASE* is set to the newly reconnected database. If DATABASE is a database instance, this object is closed. If DATABASE is a string, then a connected database whose name matches DATABASE is sought in the list of connected databases. If no matching database is found and ERROR and DATABASE are both non-nil an error is signaled, otherwise nil is returned. When the current database connection cannot be closed, if FORCE is non-nil, as it is by default, the connection is closed and errors are suppressed. If force is nil and the database connection cannot be closed, an error is signalled.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

STATUS --

STATUS

Syntax

```
(STATUS &OPTIONAL FULL) [function] =>
```

Arguments and Values

Description

Prints information about the currently connected databases to *STANDARD-OUTPUT*. The argument FULL is nil by default and a value of t means that more detailed information about each database is printed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

TRUNCATE-DATABASE --

TRUNCATE-DATABASE

Syntax

```
(TRUNCATE-DATABASE &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

WITH-DATABASE --

WITH-DATABASE

Syntax

```
(WITH-DATABASE DB-VAR CONNECTION-SPEC &REST CONNECT-ARGS &BODY BODY) [macro
```

Arguments and Values

Description

Evaluate the body in an environment, where `db-var' is bound to the database connection given by `connection-spec' and `connect-args'. The connection is automatically closed or released to the pool on exit from the body.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

WITH-DEFAULT-DATABASE --

WITH-DEFAULT-DATABASE

Syntax

```
(WITH-DEFAULT-DATABASE DATABASE &REST BODY) [macro] =>
```

Arguments and Values

Description

Perform BODY with DATABASE bound as *default-database*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

The Symbolic SQL Syntax

Name

DISABLE-SQL-READER-SYNTAX --

DISABLE-SQL-READER-SYNTAX

Syntax

```
(DISABLE-SQL-READER-SYNTAX) [macro] =>
```

Arguments and Values

Description

Turns off the SQL reader syntax setting the syntax state such that if the syntax is subsequently enabled, RESTORE-SQL-READER-SYNTAX-STATE will disable it again.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

ENABLE-SQL-READER-SYNTAX --

ENABLE-SQL-READER-SYNTAX

Syntax

```
(ENABLE-SQL-READER-SYNTAX) [macro] =>
```

Arguments and Values

Description

Turns on the SQL reader syntax setting the syntax state such that if the syntax is subsequently disabled, RESTORE-SQL-READER-SYNTAX-STATE will enable it again.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LOCALLY-DISABLE-SQL-READER-SYNTAX --

LOCALLY-DISABLE-SQL-READER-SYNTAX

Syntax

```
(LOCALLY-DISABLE-SQL-READER-SYNTAX) [macro] =>
```

Arguments and Values

Description

Turns off the SQL reader syntax without changing the syntax state such that RESTORE-SQL-READER-SYNTAX-STATE will re-establish the current syntax state.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LOCALLY-ENABLE-SQL-READER-SYNTAX --

LOCALLY-ENABLE-SQL-READER-SYNTAX

Syntax

```
(LOCALLY-ENABLE-SQL-READER-SYNTAX) [macro] =>
```

Arguments and Values

Description

Turns on the SQL reader syntax without changing the syntax state such that RESTORE-SQL-READER-SYNTAX-STATE will re-establish the current syntax state.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

RESTORE-SQL-READER-SYNTAX-STATE --

RESTORE-SQL-READER-SYNTAX-STATE

Syntax

```
(RESTORE-SQL-READER-SYNTAX-STATE) [macro] =>
```

Arguments and Values

Description

Enables the SQL reader syntax if ENABLE-SQL-READER-SYNTAX has been called more recently than DISABLE-SQL-READER-SYNTAX and otherwise disables the SQL reader syntax. By default, the SQL reader syntax is disabled.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SQL --

SQL

Syntax

```
(SQL &REST ARGS) [function] =>
```

Arguments and Values

Description

Returns an SQL string generated from the SQL expressions ARGS. The expressions are translated into SQL strings and then concatenated with a single space delimiting each expression.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SQL-EXPRESSION --

SQL-EXPRESSION

Syntax

```
(SQL-EXPRESSION &KEY STRING TABLE ALIAS ATTRIBUTE TYPE) [function] =>
```

Arguments and Values

Description

Returns an SQL expression constructed from the supplied arguments which may be combined as follows: ATTRIBUTE and TYPE; ATTRIBUTE; ALIAS or TABLE and ATTRIBUTE and TYPE; ALIAS or TABLE and ATTRIBUTE; TABLE, ATTRIBUTE and TYPE; TABLE and ATTRIBUTE; TABLE and ALIAS; TABLE; and STRING.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SQL-OPERATION --

SQL-OPERATION

Syntax

```
(SQL-OPERATION OPERATION &REST REST) [function] =>
```

Arguments and Values

Description

Returns an SQL expression constructed from the supplied SQL operator or function OPERATION and its arguments REST. If OPERATION is passed the symbol FUNCTION then the first value in REST is taken to be a valid SQL function and the remaining values in REST its arguments.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SQL-OPERATOR --

SQL-OPERATOR

Syntax

```
(SQL-OPERATOR OPERATION) [function] =>
```

Arguments and Values

Description

Returns the Lisp symbol corresponding to the SQL operation represented by the symbol OPERATION.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Functional Data Definition Language (FDDL)

Name

CREATE-TABLE --

CREATE-TABLE

Syntax

        (CREATE-TABLE NAME DESCRIPTION &KEY (DATABASE *DEFAULT-DATABASE*) (CONSTRAI

Arguments and Values

Description

Creates a table called NAME, which may be a string, symbol or SQL table identifier, in DATABASE which defaults to *DEFAULT-DATABASE*. DESCRIPTION is a list whose elements are lists containing the attribute names, types, and other constraints such as not-null or primary-key for each column in the table. CONSTRAINTS is a string representing an SQL table constraint expression or a list of such strings. With MySQL databases, if TRANSACTIONS is t an InnoDB table is created which supports transactions.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DESCRIBE-TABLE --

DESCRIBE-TABLE

Syntax

```
(DESCRIBE-TABLE TABLE &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Describes a table, returns a list of name/type for columns in table

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DROP-TABLE --

DROP-TABLE

Syntax

```
(DROP-TABLE NAME &KEY (IF-DOES-NOT-EXIST :ERROR) (DATABASE *DEFAULT-DATABAS
```

Arguments and Values

Description

Drops the table called NAME from DATABASE which defaults to *DEFAULT-DATABASE*. If the
table does not exist and IF-DOES-NOT-EXIST is :ignore then DROP-TABLE returns nil whereas an er-
ror is signalled if IF-DOES-NOT-EXIST is :error.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-TABLES --

LIST-TABLES

Syntax

```
(LIST-TABLES &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Returns a list of strings representing table names in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only tables owned by users are
listed. If OWNER is a string denoting a user name, only tables owned by OWNER are listed. If OWN-
ER is :all then all tables are listed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

TABLE-EXISTS-P --

TABLE-EXISTS-P

Syntax

```
(TABLE-EXISTS-P NAME &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [funct
```

Arguments and Values

Description

Tests for the existence of an SQL table called NAME in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only tables owned by users are
examined. If OWNER is a string denoting a user name, only tables owned by OWNER are examined. If
OWNER is :all then all tables are examined.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

CREATE-VIEW --

CREATE-VIEW

Syntax

```
(CREATE-VIEW NAME &KEY AS COLUMN-LIST (WITH-CHECK-OPTION NIL) (DATABASE *DE
```

Arguments and Values

Description

Creates a view called NAME in DATABASE which defaults to *DEFAULT-DATABASE*. The view is created using the query AS and the columns of the view may be specified using the COLUMN-LIST parameter. The WITH-CHECK-OPTION is nil by default but if it has a non-nil value, then all insert/ update commands on the view are checked to ensure that the new data satisfy the query AS.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DROP-VIEW --

DROP-VIEW

Syntax

```
(DROP-VIEW NAME &KEY (IF-DOES-NOT-EXIST :ERROR) (DATABASE *DEFAULT-DATABASE
```

Arguments and Values

Description

Drops the view called NAME from DATABASE which defaults to *DEFAULT-DATABASE*. If the view does not exist and IF-DOES-NOT-EXIST is :ignore then DROP-VIEW returns nil whereas an error is signalled if IF-DOES-NOT-EXIST is :error.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-VIEWS --

LIST-VIEWS

Syntax

```
(LIST-VIEWS &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Returns a list of strings representing view names in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only views owned by users are
listed. If OWNER is a string denoting a user name, only views owned by OWNER are listed. If OWN-
ER is :all then all views are listed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

VIEW-EXISTS-P --

VIEW-EXISTS-P

Syntax

```
(VIEW-EXISTS-P NAME &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [functi
```

Arguments and Values

Description

Tests for the existence of an SQL view called NAME in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only views owned by users are
examined. If OWNER is a string denoting a user name, only views owned by OWNER are examined. If
OWNER is :all then all views are examined.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

CREATE-INDEX --

CREATE-INDEX

Syntax

```
(CREATE-INDEX NAME &KEY ON (UNIQUE NIL) ATTRIBUTES (DATABASE *DEFAULT-DATAB
```

Arguments and Values

Description

Creates an index called NAME on the table specified by ON in DATABASE which default to
DEFAULT-DATABASE. The table attributes to use in constructing the index NAME are specified by
ATTRIBUTES. The UNIQUE argument is nil by default but if it has a non-nil value then the indexed at-
tributes must have unique values.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DROP-INDEX --

DROP-INDEX

Syntax

```
(DROP-INDEX NAME &KEY (IF-DOES-NOT-EXIST :ERROR) (ON NIL) (DATABASE *DEFAUL
```

Arguments and Values

Description

Drops the index called NAME in DATABASE which defaults to *DEFAULT-DATABASE*. If the index does not exist and IF-DOES-NOT-EXIST is :ignore then DROP-INDEX returns nil whereas an error is signalled if IF-DOES-NOT-EXIST is :error. The argument ON allows the optional specification of a table to drop the index from.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

INDEX-EXISTS-P --

INDEX-EXISTS-P

Syntax

```
(INDEX-EXISTS-P NAME &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [funct
```

Arguments and Values

Description

Tests for the existence of an SQL index called NAME in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only indexes owned by users
are examined. If OWNER is a string denoting a user name, only indexes owned by OWNER are ex-
amined. If OWNER is :all then all indexes are examined.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-INDEXES --

LIST-INDEXES

Syntax

```
(LIST-INDEXES &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Returns a list of strings representing index names in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only indexes owned by users
are listed. If OWNER is a string denoting a user name, only indexes owned by OWNER are listed. If
OWNER is :all then all indexes are listed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-TABLE-INDEXES --

LIST-TABLE-INDEXES

Syntax

```
(LIST-TABLE-INDEXES TABLE &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [
```

Arguments and Values

Description

Returns a list of strings representing index names on the table specified by TABLE in DATABASE which defaults to *DEFAULT-DATABASE*. OWNER is nil by default which means that only indexes owned by users are listed. If OWNER is a string denoting a user name, only indexes owned by OWNER are listed. If OWNER is :all then all indexes are listed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

ATTRIBUTE-TYPE --

ATTRIBUTE-TYPE

Syntax

```
(ATTRIBUTE-TYPE ATTRIBUTE TABLE &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABAS
```

Arguments and Values

Description

Returns a string representing the field type of the supplied attribute ATTRIBUTE in the table specified
by TABLE in DATABASE which defaults to *DEFAULT-DATABASE*. OWNER is nil by default
which means that the attribute specified by ATTRIBUTE, if it exists, must be user owned else nil is re-
turned. If OWNER is a string denoting a user name, the attribute, if it exists, must be owned by OWN-
ER else nil is returned, whereas if OWNER is :all then the attribute, if it exists, will be returned regard-
less of its owner.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-ATTRIBUTE-TYPES --

LIST-ATTRIBUTE-TYPES

Syntax

```
(LIST-ATTRIBUTE-TYPES TABLE &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*))
```

Arguments and Values

Description

Returns a list containing information about the SQL types of each of the attributes in the table specified by TABLE in DATABASE which has a default value of *DEFAULT-DATABASE*. OWNER is nil by default which means that only attributes owned by users are listed. If OWNER is a string denoting a user name, only attributes owned by OWNER are listed. If OWNER is :all then all attributes are listed. The elements of the returned list are lists where the first element is the name of the attribute, the second element is its SQL type, the third is the type precision, the fourth is the scale of the attribute and the fifth is 1 if the attribute accepts null values and otherwise 0.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-ATTRIBUTES --

LIST-ATTRIBUTES

Syntax

```
(LIST-ATTRIBUTES NAME &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [func
```

Arguments and Values

Description

Returns a list of strings representing the attributes of table NAME in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only attributes owned by users
are listed. If OWNER is a string denoting a user name, only attributes owned by OWNER are listed. If
OWNER is :all then all attributes are listed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

CREATE-SEQUENCE --

CREATE-SEQUENCE

Syntax

```
(CREATE-SEQUENCE NAME &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Creates a sequence called NAME in DATABASE which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DROP-SEQUENCE --

DROP-SEQUENCE

Syntax

```
(DROP-SEQUENCE NAME &KEY (IF-DOES-NOT-EXIST :ERROR) (DATABASE *DEFAULT-DATA
```

Arguments and Values

Description

Drops the sequence called NAME from DATABASE which defaults to *DEFAULT-DATABASE*. If the sequence does not exist and IF-DOES-NOT-EXIST is :ignore then DROP-SEQUENCE returns nil whereas an error is signalled if IF-DOES-NOT-EXIST is :error.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-SEQUENCES --

LIST-SEQUENCES

Syntax

```
(LIST-SEQUENCES &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [function]
```

Arguments and Values

Description

Returns a list of strings representing sequence names in DATABASE which defaults to
DEFAULT-DATABASE. OWNER is nil by default which means that only sequences owned by users
are listed. If OWNER is a string denoting a user name, only sequences owned by OWNER are listed. If
OWNER is :all then all sequences are listed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SEQUENCE-EXISTS-P --

SEQUENCE-EXISTS-P

Syntax

```
(SEQUENCE-EXISTS-P NAME &KEY (OWNER NIL) (DATABASE *DEFAULT-DATABASE*)) [fu
```

Arguments and Values

Description

Tests for the existence of an SQL sequence called NAME in DATABASE which defaults to *DEFAULT-DATABASE*. OWNER is nil by default which means that only sequences owned by users are examined. If OWNER is a string denoting a user name, only sequences owned by OWNER are examined. If OWNER is :all then all sequences are examined.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SEQUENCE-LAST --

SEQUENCE-LAST

Syntax

```
(SEQUENCE-LAST NAME &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Return the last value of the sequence called NAME in DATABASE which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SEQUENCE-NEXT --

SEQUENCE-NEXT

Syntax

```
(SEQUENCE-NEXT NAME &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Return the next value in the sequence called NAME in DATABASE which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SET-SEQUENCE-POSITION --

SET-SEQUENCE-POSITION

Syntax

```
(SET-SEQUENCE-POSITION NAME POSITION &KEY (DATABASE *DEFAULT-DATABASE*)) [f
```

Arguments and Values

Description

Explicitly set the the position of the sequence called NAME in DATABASE, which defaults to *DEFAULT-DATABSE*, to POSITION.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Functional Data Manipulation Language (FDML)

Name

CACHE-TABLE-QUERIES-DEFAULT --

CACHE-TABLE-QUERIES-DEFAULT

Value Type

Initial Value

nil

Description

Examples

Affected By

None.

See Also

None.

Notes

None.

Name

BIND-PARAMETER --

BIND-PARAMETER

Syntax

```
(BIND-PARAMETER PREPARED-STMT POSITION VALUE) [function] =>
```

Arguments and Values

Description

Sets the value of a parameter in a prepared statement.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

CACHE-TABLE-QUERIES --

CACHE-TABLE-QUERIES

Syntax

```
(CACHE-TABLE-QUERIES TABLE &KEY (ACTION NIL) (DATABASE *DEFAULT-DATABASE*))
```

Arguments and Values

Description

Controls the caching of attribute type information on the table specified by TABLE in DATABASE which defaults to *DEFAULT-DATABASE*. ACTION specifies the caching behaviour to adopt. If its value is t then attribute type information is cached whereas if its value is nil then attribute type information is not cached. If ACTION is :flush then all existing type information in the cache for TABLE is removed, but caching is still enabled. TABLE may be a string representing a table for which the caching action is to be taken while the caching action is applied to all tables if TABLE is t. Alternativly, when TABLE is :default, the default caching action specified by *CACHE-TABLE-QUERIES-DEFAULT* is applied to all table for which a caching action has not been explicitly set.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DELETE-RECORDS --

DELETE-RECORDS

Syntax

```
(DELETE-RECORDS &KEY (FROM NIL) (WHERE NIL) (DATABASE *DEFAULT-DATABASE*))
```

Arguments and Values

Description

Deletes records satisfying the SQL expression WHERE from the table specified by FROM in DATA-
BASE specifies a database which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DO-QUERY --

DO-QUERY

Syntax

```
(DO-QUERY &KEY (DATABASE '*DEFAULT-DATABASE*) (RESULT-TYPES :AUTO) &REST QU
```

Arguments and Values

Description

Repeatedly executes BODY within a binding of ARGS on the fields of each row selected by the SQL query QUERY-EXPRESSION, which may be a string or a symbolic SQL expression, in DATABASE which defaults to *DEFAULT-DATABASE*. The values returned by the execution of BODY are returned. RESULT-TYPES is a list of symbols which specifies the lisp type for each field returned by QUERY-EXPRESSION. If RESULT-TYPES is nil all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

EXECUTE-COMMAND --

EXECUTE-COMMAND

Syntax

```
(EXECUTE-COMMAND EXPRESSION &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Executes the SQL command EXPRESSION, which may be an SQL expression or a string representing any SQL statement apart from a query, on the supplied DATABASE which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

FOR-EACH-ROW --

FOR-EACH-ROW

Syntax

```
(FOR-EACH-ROW &KEY FROM ORDER-BY WHERE DISTINCT LIMIT &REST FIELDS &BODY BO
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

FREE-PREPARED-SQL --

FREE-PREPARED-SQL

Syntax

```
(FREE-PREPARED-SQL PREPARED-STMT) [function] =>
```

Arguments and Values

Description

Delete the objects associated with a prepared statement.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

INSERT-RECORDS --

INSERT-RECORDS

Syntax

```
(INSERT-RECORDS &KEY (INTO NIL) (ATTRIBUTES NIL) (VALUES NIL) (AV-PAIRS NIL
```

Arguments and Values

Description

Inserts records into the table specified by INTO in DATABASE which defaults to
DEFAULT-DATABASE. There are five ways of specifying the values inserted into each row. In the
first VALUES contains a list of values to insert and ATTRIBUTES, AV-PAIRS and QUERY are nil.
This can be used when values are supplied for all attributes in INTO. In the second, ATTRIBUTES is a
list of column names, VALUES is a corresponding list of values and AV-PAIRS and QUERY are nil. In
the third, ATTRIBUTES, VALUES and QUERY are nil and AV-PAIRS is an alist of (attribute value)
pairs. In the fourth, VALUES, AV-PAIRS and ATTRIBUTES are nil and QUERY is a symbolic SQL
query expression in which the selected columns also exist in INTO. In the fifth method, VALUES and
AV-PAIRS are nil and ATTRIBUTES is a list of column names and QUERY is a symbolic SQL query
expression which returns values for the specified columns.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LOOP-FOR-AS-TUPLES -- Iterate over all the tuples of a query via a loop clause

LOOP-FOR-AS-TUPLES

Compatibility

Caution

`loop-for-as-tuples` only works with CMUCL.

Syntax

```
var [type-spec] being {each | the} {record | records | tuple | tuples} {in | of} q
```

Arguments and Values

`var`	A `d-var-spec`, as defined in the grammar for `loop`-clauses in the ANSI Standard for Common Lisp. This allows for the usual loop-style destructuring.
`type-spec`	An optional `type-spec` either simple or destructured, as defined in the grammar for `loop`-clauses in the ANSI Standard for Common Lisp.
`query`	An sql expression that represents an SQL query which is expected to return a (possibly empty) result set, where each tuple has as many attributes as `function` takes arguments.
`database`	An optional database object. This will default to the value of *default-database*.

Description

This clause is an iteration driver for `loop`, that binds the given variable (possibly destructured) to the consecutive tuples (which are represented as lists of attribute values) in the result set returned by executing the SQL `query` expression on the `database` specified.

Examples

```
(defvar *my-db* (connect '("dent" "newesim" "dent" "dent"))
  "My database"
=> *MY-DB*
(loop with time-graph = (make-hash-table :test #'equal)
      with event-graph = (make-hash-table :test #'equal)
      for (time event) being the tuples of "select time,event from log"
      from *my-db*
      do
      (incf (gethash time time-graph 0))
      (incf (gethash event event-graph 0))
      finally
      (flet ((show-graph (k v) (format t "~40A => ~5D~%" k v)))
        (format t "~&Time-Graph:~%===========~%")
```

```
        (maphash #'show-graph time-graph)
        (format t "~&~%Event-Graph:~%============~%")
        (maphash #'show-graph event-graph))
     (return (values time-graph event-graph)))
>> Time-Graph:
>> ===========
>> D                                         => 53000
>> X                                         =>     3
>> test-me                                   =>  3000
>>
>> Event-Graph:
>> ============
>> CLOS Benchmark entry.                     =>  9000
>> Demo Text...                              =>     3
>> doit-text                                 =>  3000
>> C    Benchmark entry.                     => 12000
>> CLOS Benchmark entry                      => 32000
=> #<EQUAL hash table, 3 entries {48350A1D}>
=> #<EQUAL hash table, 5 entries {48350FCD}>
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

Otherwise, any of the exceptional situations of `loop` applies.

See Also

```
query
map-query
do-query
```

Notes

None.

Name

MAP-QUERY -- Map a function over all the tuples from a query

MAP-QUERY

Syntax

```
map-query output-type-spec function query-expression &key database result-types =>
```

Arguments and Values

output-type-spec	A sequence type specifier or nil.
function	A function designator. *function* takes a single argument which is the atom value for a query single with a single column or is a list of values for a multi-column query.
query-expression	An sql expression that represents an SQL query which is expected to return a (possibly empty) result set.
database	A database object. This will default to the value of *default-database*.
result-types	A field type specifier. The default is NIL. See `query` for the semantics of this argument.
result	If *output-type-spec* is a type specifier other than nil, then a sequence of the type it denotes. Otherwise nil is returned.

Description

Applies *function* to the successive tuples in the result set returned by executing the SQL *query-expression*. If the *output-type-spec* is nil, then the result of each application of *function* is discarded, and `map-query` returns nil. Otherwise the result of each successive application of *function* is collected in a sequence of type *output-type-spec*, where the jths element is the result of applying *function* to the attributes of the jths tuple in the result set. The collected sequence is the result of the call to `map-query`.

If the *output-type-spec* is a subtype of list, the result will be a list.

If the *result-type* is a subtype of vector, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is t; otherwise, an error is signaled.

If RESULT-TYPES is nil all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field.

Examples

```
(map-query 'list #'(lambda (tuple)
```

```
                        (multiple-value-bind (salary name) tuple
                          (declare (ignorable name))
                          (read-from-string salary)))
           "select salary,name from simple where salary > 8000")
=> (10000.0 8000.5)

(map-query '(vector double-float)
           #'(lambda (tuple)
               (multiple-value-bind (salary name) tuple
                 (declare (ignorable name))
                 (let ((*read-default-float-format* 'double-float))
                   (coerce (read-from-string salary) 'double-float))
                   "select salary,name from simple where salary > 8000")))
=> #(10000.0d0 8000.5d0)
(type-of *)
=> (SIMPLE-ARRAY DOUBLE-FLOAT (2))

(let (list)
  (values (map-query nil #'(lambda (tuple)
                             (multiple-value-bind (salary name) tuple
                               (push (cons name (read-from-string salary)) list))
                         "select salary,name from simple where salary > 8000")
          list))
=> NIL
=> (("Hacker, Random J." . 8000.5) ("Mai, Pierre" . 10000.0))
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

An error of type type-error must be signaled if the *output-type-spec* is not a recognizable subtype of list, not a recognizable subtype of vector, and not nil.

An error of type type-error should be signaled if *output-type-spec* specifies the number of elements and the size of the result set is different from that number.

See Also

```
query
do-query
```

Notes

None.

Name

PREPARE-SQL --

PREPARE-SQL

Syntax

```
(PREPARE-SQL SQL-STMT TYPES &KEY (DATABASE *DEFAULT-DATABASE*) (RESULT-TYPE
```

Arguments and Values

Description

Prepares a SQL statement for execution. TYPES contains a list of types corresponding to the input parameters. Returns a prepared-statement object. A type can be :int :double :null (:string n)

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

PRINT-QUERY --

PRINT-QUERY

Syntax

```
(PRINT-QUERY QUERY-EXP &KEY TITLES (FORMATS T) (SIZES T) (STREAM T) (DATABA
```

Arguments and Values

Description

Prints a tabular report of the results returned by the SQL query QUERY-EXP, which may be a symbolic SQL expression or a string, in DATABASE which defaults to *DEFAULT-DATABASE*. The report is printed onto STREAM which has a default value of t which means that *STANDARD-OUTPUT* is used. The TITLE argument, which defaults to nil, allows the specification of a list of strings to use as column titles in the tabular output. SIZES accepts a list of column sizes, one for each column selected by QUERY-EXP, to use in formatting the tabular report. The default value of t means that minimum sizes are computed. FORMATS is a list of format strings to be used for printing each column selected by QUERY-EXP. The default value of FORMATS is t meaning that ~A is used to format all columns or ~VA if column sizes are used.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

QUERY --

QUERY

Syntax

```
(QUERY QUERY-EXPRESSION &KEY DATABASE RESULT-TYPES FLATP FIELD-NAMES) [gene
```

Arguments and Values

Description

Executes the SQL query expression QUERY-EXPRESSION, which may be an SQL expression or a
string, on the supplied DATABASE which defaults to *DEFAULT-DATABASE*. RESULT-TYPES is
a list of symbols which specifies the lisp type for each field returned by QUERY-EXPRESSION. If
RESULT-TYPES is nil all results are returned as strings whereas the default value of :auto means that
the lisp types are automatically computed for each field. FIELD-NAMES is t by default which means
that the second value returned is a list of strings representing the columns selected by QUERY-
EXPRESSION. If FIELD-NAMES is nil, the list of column names is not returned as a second value.
FLATP has a default value of nil which means that the results are returned as a list of lists. If FLATP is t
and only one result is returned for each record selected by QUERY-EXPRESSION, the results are re-
turned as elements of a list.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

RUN-PREPARED-SQL --

RUN-PREPARED-SQL

Syntax

```
(RUN-PREPARED-SQL PREPARED-STMT) [function] =>
```

Arguments and Values

Description

Execute the prepared sql statment. All input parameters must be bound.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SELECT --

SELECT

Syntax

```
(SELECT &REST SELECT-ALL-ARGS) [function] =>
```

Arguments and Values

Description

Executes a query on DATABASE, which has a default value of *DEFAULT-DATABASE*, specified by the SQL expressions supplied using the remaining arguments in SELECT-ALL-ARGS. The SELECT argument can be used to generate queries in both functional and object oriented contexts. In the functional case, the required arguments specify the columns selected by the query and may be symbolic SQL expressions or strings representing attribute identifiers. Type modified identifiers indicate that the values selected from the specified column are converted to the specified lisp type. The keyword arguments ALL, DISTINCT, FROM, GROUP-by, HAVING, ORDER-BY, SET-OPERATION and WHERE are used to specify, using the symbolic SQL syntax, the corresponding components of the SQL query generated by the call to SELECT. RESULT-TYPES is a list of symbols which specifies the lisp type for each field returned by the query. If RESULT-TYPES is nil all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field. FIELD-NAMES is t by default which means that the second value returned is a list of strings representing the columns selected by the query. If FIELD-NAMES is nil, the list of column names is not returned as a second value. In the object oriented case, the required arguments to SELECT are symbols denoting View Classes which specify the database tables to query. In this case, SELECT returns a list of View Class instances whose slots are set from the attribute values of the records in the specified table. Slot-value is a legal operator which can be employed as part of the symbolic SQL syntax used in the WHERE keyword argument to SELECT. REFRESH is nil by default which means that the View Class instances returned are retrieved from a cache if an equivalent call to SELECT has previously been issued. If REFRESH is true, the View Class instances returned are updated as necessary from the database and the generic function INSTANCE-REFRESHED is called to perform any necessary operations on the updated instances. In both object oriented and functional contexts, FLATP has a default value of nil which means that the results are returned as a list of lists. If FLATP is t and only one result is returned for each record selected in the query, the results are returned as elements of a list.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-RECORDS --

UPDATE-RECORDS

Syntax

```
(UPDATE-RECORDS TABLE &KEY (ATTRIBUTES NIL) (VALUES NIL) (AV-PAIRS NIL) (WH
```

Arguments and Values

Description

Updates the attribute values of existing records satsifying the SQL expression WHERE in the table specified by TABLE in DATABASE which defaults to *DEFAULT-DATABASE*. There are three ways of specifying the values to update for each row. In the first, VALUES contains a list of values to use in the update and ATTRIBUTES, AV-PAIRS and QUERY are nil. This can be used when values are supplied for all attributes in TABLE. In the second, ATTRIBUTES is a list of column names, VALUES is a corresponding list of values and AV-PAIRS and QUERY are nil. In the third, ATTRIBUTES, VALUES and QUERY are nil and AV-PAIRS is an alist of (attribute value) pairs.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Transaction Handling

Name

ADD-TRANSACTION-COMMIT-HOOK --

ADD-TRANSACTION-COMMIT-HOOK

Syntax

```
(ADD-TRANSACTION-COMMIT-HOOK DATABASE COMMIT-HOOK) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

ADD-TRANSACTION-ROLLBACK-HOOK --

ADD-TRANSACTION-ROLLBACK-HOOK

Syntax

```
(ADD-TRANSACTION-ROLLBACK-HOOK DATABASE ROLLBACK-HOOK) [function] =>
```

Arguments and Values

Description

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

COMMIT --

COMMIT

Syntax

```
(COMMIT &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

If DATABASE, which defaults to *DEFAULT-DATABASE*, is currently within the scope of a transaction, commits changes made since the transaction began.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

IN-TRANSACTION-P --

IN-TRANSACTION-P

Syntax

```
(IN-TRANSACTION-P &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

A predicate to test whether DATABASE, which defaults to *DEFAULT-DATABASE*, is currently within the scope of a transaction.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

ROLLBACK --

ROLLBACK

Syntax

```
(ROLLBACK &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

If DATABASE, which defaults to *DEFAULT-DATABASE*, is currently within the scope of a transaction, rolls back changes made since the transaction began.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SET-AUTOCOMMIT --

SET-AUTOCOMMIT

Syntax

```
(SET-AUTOCOMMIT VALUE &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Sets autocommit on or off. Returns old value of of autocommit flag.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

START-TRANSACTION --

START-TRANSACTION

Syntax

```
(START-TRANSACTION &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Starts a transaction block on DATABASE which defaults to *DEFAULT-DATABASE* and which continues until ROLLBACK or COMMIT are called.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

WITH-TRANSACTION --

WITH-TRANSACTION

Syntax

```
(WITH-TRANSACTION &KEY (DATABASE '*DEFAULT-DATABASE*) &REST BODY) [macro] =
```

Arguments and Values

Description

Starts a transaction in the database specified by DATABASE, which is *DEFAULT-DATABASE* by default, and executes BODY within that transaction. If BODY aborts or throws, DATABASE is rolled back and otherwise the transaction is committed.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Object Oriented Data Definition Language (OODDL)

Name

STANDARD-DB-OBJECT -- Superclass for all *CLSQL* View Classes.

STANDARD-DB-OBJECT

Class Precedence List

standard-db-object, standard-object, t

Description

This class is the superclass of all *CLSQL* View Classes.

Class details

```
(defclass STANDARD-DB-OBJECT ()(...))
```

Slots

Name

DEFAULT-VARCHAR-LENGTH --

DEFAULT-VARCHAR-LENGTH

Value Type

Initial Value

nil

Description

Examples

Affected By

None.

See Also

None.

Notes

None.

Name

CREATE-VIEW-FROM-CLASS --

CREATE-VIEW-FROM-CLASS

Syntax

```
(CREATE-VIEW-FROM-CLASS VIEW-CLASS-NAME &KEY (DATABASE *DEFAULT-DATABASE*))
```

Arguments and Values

Description

Creates a table as defined by the View Class VIEW-CLASS-NAME in DATABASE which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DEF-VIEW-CLASS --

DEF-VIEW-CLASS

Syntax

```
(DEF-VIEW-CLASS CLASS SUPERS SLOTS &REST CL-OPTIONS) [macro] =>
```

Arguments and Values

Description

Creates a View Class called CLASS whose slots SLOTS can map onto the attributes of a table in a database. If SUPERS is nil then the superclass of CLASS will be STANDARD-DB-OBJECT, otherwise SUPERS is a list of superclasses for CLASS which must include STANDARD-DB-OBJECT or a descendent of this class. The syntax of DEFCLASS is extended through the addition of a class option :base-table which defines the database table onto which the View Class maps and which defaults to CLASS. The DEFCLASS syntax is also extended through additional slot options. The :db-kind slot option specifies the kind of DB mapping which is performed for this slot and defaults to :base which indicates that the slot maps to an ordinary column of the database table. A :db-kind value of :key indicates that this slot is a special kind of :base slot which maps onto a column which is one of the unique keys for the database table, the value :join indicates this slot represents a join onto another View Class which contains View Class objects, and the value :virtual indicates a standard CLOS slot which does not map onto columns of the database table. If a slot is specified with :db-kind :join, the slot option :db-info contains a list which specifies the nature of the join. For slots of :db-kind :base or :key, the :type slot option has a special interpretation such that Lisp types, such as string, integer and float are automatically converted into appropriate SQL types for the column onto which the slot maps. This behaviour may be over-ridden using the :db-type slot option which is a string specifying the vendor-specific database type for this slot's column definition in the database. The :column slot option specifies the name of the SQL column which the slot maps onto, if :db-kind is not :virtual, and defaults to the slot name. The :void-value slot option specifies the value to store if the SQL value is NULL and defaults to NIL. The :db-constraints slot option is a string representing an SQL table constraint expression or a list of such strings.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DROP-VIEW-FROM-CLASS --

DROP-VIEW-FROM-CLASS

Syntax

```
(DROP-VIEW-FROM-CLASS VIEW-CLASS-NAME &KEY (DATABASE *DEFAULT-DATABASE*)) [
```

Arguments and Values

Description

Removes a table defined by the View Class VIEW-CLASS-NAME from DATABASE which defaults to *DEFAULT-DATABASE*.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-CLASSES --

LIST-CLASSES

Syntax

```
(LIST-CLASSES &KEY (TEST #'IDENTITY) (ROOT-CLASS (FIND-CLASS 'STANDARD-DB-O
```

Arguments and Values

Description

Returns a list of all the View Classes which are connected to DATABASE, which defaults to *DEFAULT-DATABASE*, and which descend from the class ROOT-CLASS and which satisfy the function TEST. By default ROOT-CLASS is STANDARD-DB-OBJECT and TEST is IDENTITY.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Object Oriented Data Manipulation Language (OODML)

Name

DB-AUTO-SYNC --

DB-AUTO-SYNC

Value Type

Initial Value

nil

Description

Examples

Affected By

None.

See Also

None.

Notes

None.

Name

DEFAULT-UPDATE-OBJECTS-MAX-LEN --

DEFAULT-UPDATE-OBJECTS-MAX-LEN

Value Type

Initial Value

nil

Description

Examples

Affected By

None.

See Also

None.

Notes

None.

Name

DELETE-INSTANCE-RECORDS --

DELETE-INSTANCE-RECORDS

Syntax

(DELETE-INSTANCE-RECORDS OBJECT) [generic] =>

Arguments and Values

Description

Deletes the records represented by OBJECT in the appropriate table of the database associated with OB-JECT. If OBJECT is not yet associated with a database, an error is signalled.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

INSTANCE-REFRESHED --

INSTANCE-REFRESHED

Syntax

```
(INSTANCE-REFRESHED OBJECT) [generic] =>
```

Arguments and Values

Description

Provides a hook which is called within an object oriented call to SELECT with a non-nil value of RE-FRESH when the View Class instance OBJECT has been updated from the database. A method special-ised on STANDARD-DB-OBJECT is provided which has no effects. Methods specialised on particular View Classes can be used to specify any operations that need to be made on View Classes instances which have been updated in calls to SELECT.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-INSTANCE-FROM-RECORDS --

UPDATE-INSTANCE-FROM-RECORDS

Syntax

```
(UPDATE-INSTANCE-FROM-RECORDS OBJECT &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Updates the slot values of the View Class instance OBJECT using the attribute values of the appropriate table of DATABASE which defaults to the database associated with OBJECT or, if OBJECT is not associated with a database, *DEFAULT-DATABASE*. Join slots are updated but instances of the class on which the join is made are not updated.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-OBJECTS-JOINS --

UPDATE-OBJECTS-JOINS

Syntax

```
(UPDATE-OBJECTS-JOINS OBJECTS &KEY (SLOTS T) (FORCE-P T) CLASS-NAME (MAX-LE
```

Arguments and Values

Description

Updates from the records of the appropriate database tables the join slots specified by SLOTS in the supplied list of View Class instances OBJECTS. SLOTS is t by default which means that all join slots with :retrieval :immediate are updated. CLASS-NAME is used to specify the View Class of all instance in OBJECTS and default to nil which means that the class of the first instance in OBJECTS is used. FORCE-P is t by default which means that all join slots are updated whereas a value of nil means that only unbound join slots are updated. MAX-LEN defaults to *DEFAULT-UPDATE-OBJECTS-MAX-LEN* and when non-nil specifies that UPDATE-OB-JECT-JOINS may issue multiple database queries with a maximum of MAX-LEN instances updated in each query.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-RECORD-FROM-SLOT --

Syntax

```
(UPDATE-RECORD-FROM-SLOT OBJECT SLOT &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Updates the value stored in the column represented by the slot, specified by the CLOS slot name SLOT, of View Class instance OBJECT. DATABASE defaults to *DEFAULT-DATABASE* and specifies the database in which the update is made only if OBJECT is not associated with a database. In this case, a record is created in DATABASE and the attribute represented by SLOT is initialised from the value of the supplied slots with other attributes having default values. Furthermore, OBJECT becomes associated with DATABASE.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-RECORD-FROM-SLOTS --

Syntax

```
(UPDATE-RECORD-FROM-SLOTS OBJECT SLOTS &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Updates the values stored in the columns represented by the slots, specified by the CLOS slot names SLOTS, of View Class instance OBJECT. DATABASE defaults to *DEFAULT-DATABASE* and specifies the database in which the update is made only if OBJECT is not associated with a database. In this case, a record is created in the appropriate table of DATABASE and the attributes represented by SLOTS are initialised from the values of the supplied slots with other attributes having default values. Furthermore, OBJECT becomes associated with DATABASE.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-RECORDS-FROM-INSTANCE --

UPDATE-RECORDS-FROM-INSTANCE

Syntax

```
(UPDATE-RECORDS-FROM-INSTANCE OBJECT &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Using an instance of a View Class, OBJECT, update the table that stores its instance data. DATABASE defaults to *DEFAULT-DATABASE* and specifies the database in which the update is made only if OBJECT is not associated with a database. In this case, a record is created in the appropriate table of DATABASE using values from the slot values of OBJECT, and OBJECT becomes associated with DATABASE.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

UPDATE-SLOT-FROM-RECORD --

UPDATE-SLOT-FROM-RECORD

Syntax

```
(UPDATE-SLOT-FROM-RECORD OBJECT SLOT &KEY DATABASE) [generic] =>
```

Arguments and Values

Description

Updates the slot value, specified by the CLOS slot name SLOT, of the View Class instance OBJECT using the attribute values of the appropriate table of DATABASE which defaults to the database associated with OBJECT or, if OBJECT is not associated with a database, *DEFAULT-DATABASE*. Join slots are updated but instances of the class on which the join is made are not updated.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

SQL I/O Recording

Name

ADD-SQL-STREAM --

ADD-SQL-STREAM

Syntax

```
(ADD-SQL-STREAM STREAM &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE*))
```

Arguments and Values

Description

Adds the supplied stream STREAM (or T for *standard-output*) as a component of the recording broadcast stream for the SQL recording type specified by TYPE on DATABASE which defaults to *DEFAULT-DATABASE*. TYPE must be one of :commands, :results, or :both, defaulting to :commands, depending on whether the stream is to be added for recording SQL commands, results or both.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DELETE-SQL-STREAM --

DELETE-SQL-STREAM

Syntax

```
(DELETE-SQL-STREAM STREAM &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE
```

Arguments and Values

Description

Removes the supplied stream STREAM from the recording broadcast stream for the SQL recording type specified by TYPE on DATABASE which defaults to *DEFAULT-DATABASE*. TYPE must be one of :commands, :results, or :both, defaulting to :commands, depending on whether the stream is to be added for recording SQL commands, results or both.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

LIST-SQL-STREAMS --

LIST-SQL-STREAMS

Syntax

```
(LIST-SQL-STREAMS &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE*)) [fun
```

Arguments and Values

Description

Returns the list of component streams for the broadcast stream recording SQL commands sent to and/or results returned from DATABASE which defaults to *DEFAULT-DATABASE*. TYPE must be one of :commands, :results, or :both, defaulting to :commands, and determines whether the listed streams contain those recording SQL commands, results or both.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SQL-RECORDING-P --

SQL-RECORDING-P

Syntax

```
(SQL-RECORDING-P &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE*)) [func
```

Arguments and Values

Description

Predicate to test whether the SQL recording specified by TYPE is currently enabled for DATABASE which defaults to *DEFAULT-DATABASE*. TYPE may be one of :commands, :results, :both or :either, defaulting to :commands, otherwise nil is returned.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

SQL-STREAM --

SQL-STREAM

Syntax

```
(SQL-STREAM &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE*)) [function]
```

Arguments and Values

Description

Returns the broadcast stream used for recording SQL commands sent to or results returned from DATA-BASE which defaults to *DEFAULT-DATABASE*. TYPE must be one of :commands or :results, defaulting to :commands, and determines whether the stream returned is that used for recording SQL commands or results.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

START-SQL-RECORDING --

START-SQL-RECORDING

Syntax

```
(START-SQL-RECORDING &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE*)) [
```

Arguments and Values

Description

Starts recording of SQL commands sent to and/or results returned from DATABASE which defaults to *DEFAULT-DATABASE*. The SQL is output on one or more broadcast streams, initially just *STANDARD-OUTPUT*, and the functions ADD-SQL-STREAM and DELETE-SQL-STREAM may be used to add or delete command or result recording streams. The default value of TYPE is :commands which means that SQL commands sent to DATABASE are recorded. If TYPE is :results then SQL results returned from DATABASE are recorded. Both commands and results may be recorded by passing TYPE value of :both.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

STOP-SQL-RECORDING --

STOP-SQL-RECORDING

Syntax

```
(STOP-SQL-RECORDING &KEY (TYPE :COMMANDS) (DATABASE *DEFAULT-DATABASE*)) [f
```

Arguments and Values

Description

Stops recording of SQL commands sent to and/or results returned from DATABASE which defaults to *DEFAULT-DATABASE*. The default value of TYPE is :commands which means that SQL commands sent to DATABASE will no longer be recorded. If TYPE is :results then SQL results returned from DATABASE will no longer be recorded. Recording may be stopped for both commands and results by passing TYPE value of :both.

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

CLSQL Condition System

Name

BACKEND-WARNING-BEHAVIOR --

BACKEND-WARNING-BEHAVIOR

Value Type

Initial Value

nil

Description

Action to perform on warning messages from backend. Default is to :warn. May also be set to :error to signal an error or :ignore/nil to silently ignore the warning.

Examples

Affected By

None.

See Also

None.

Notes

None.

Name

SQL-CONDITION -- the super-type of all *CLSQL*-specific conditions

SQL-CONDITION

Class Precedence List

sql-condition, condition, t

Description

This is the super-type of all *CLSQL*-specific conditions defined by *CLSQL*, or any of it's database-specific interfaces. There are no defined initialization arguments nor any accessors.

Name

SQL-ERROR -- the super-type of all *CLSQL*-specific errors

SQL-ERROR

Class Precedence List

sql-error, error, serious-condition, sql-condition, condition, t

Description

This is the super-type of all *CLSQL*-specific conditions that represent errors, as defined by *CLSQL*, or any of it's database-specific interfaces. There are no defined initialization arguments nor any accessors.

Name

SQL-WARNING -- the super-type of all *CLSQL*-specific warnings

SQL-WARNING

Class Precedence List

sql-warning, warning, sql-condition, condition, t

Description

This is the super-type of all *CLSQL*-specific conditions that represent warnings, as defined by *CLSQL*, or any of it's database-specific interfaces. There are no defined initialization arguments nor any accessors.

Name

SQL-DATABASE-WARNING -- Used to warn while accessing a *CLSQL* database.

SQL-DATABASE-WARNING

Class Precedence List

sql-database-warning, sql-warning, warning, sql-condition, condition, t

Description

This condition represents warnings signalled while accessing a database. The following initialization arguments and accessors exist:
Initarg: :database
Accessor: `sql-warning-database`
Description: The database object that was involved in the incident.

Name

SQL-USER-ERROR -- condition representing errors because of invalid parameters from the library user.

SQL-USER-ERROR

Class Precedence List

sql-user-error, sql-error, sql-condition, condition, t

Description

This condition represents errors that occur because the user supplies invalid data to *CLSQL*. This includes errors such as an invalid format connection specification or an error in the syntax for the LOOP macro extensions. The following initialization arguments and accessors exist:
Initarg: :message
Accessor: `sql-user-error-message`
Description: The error message.

Name

SQL-DATABASE-ERROR -- condition representing errors during query or command execution

SQL-DATABASE-ERROR

Class Precedence List

sql-database-error, sql-error, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur while executing SQL statements, either as part of query operations or command execution, either explicitly or implicitly, as caused e.g. by `with-transaction`. The following initialization arguments and accessors exist:

Initarg: :database
Accessor: `sql-database-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

SQL-CONNECTION-ERROR -- condition representing errors during connection

SQL-CONNECTION-ERROR

Class Precedence List

sql-connection-error, sql-database-error, sql-error, sql-condition, condition, t

Description

This condition represents errors that occur while trying to connect to a database. The following initialization arguments and accessors exist:
Initarg: :database-type
Accessor: `sql-connection-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `sql-connection-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :database
Accessor: `sql-database-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-database-error-error`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

SQL-DATABASE-DATA-ERROR -- Used to signal an error with the SQL data passed to a database.

SQL-DATABASE-DATA-ERROR

Class Precedence List

sql-database-data-error, sql-database-error, sql-error, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur while executing SQL statements, specifically as a result of malformed SQL expressions. The following initialization arguments and accessors exist:
Initarg: :expression
Accessor: `sql-database-error-expression`
Description: The SQL expression whose execution caused the error.
Initarg: :database
Accessor: `sql-database-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

SQL-TEMPORARY-ERROR -- Used to signal a temporary error in the database backend.

SQL-TEMPORARY-ERROR

Class Precedence List

sql-database-error, sql-error, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors occurring when the database cannot currently process a valid interaction because, for example, it is still executing another command possibly issued by another user. The following initialization arguments and accessors exist:

Initarg: :database
Accessor: `sql-database-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

SQL-TIMEOUT-ERROR -- condition representing errors when a connection times out.

SQL-TIMEOUT-ERROR

Class Precedence List

sql-connection-error, sql-database-error, sql-error, sql-condition, condition, t

Description

This condition represents errors that occur when the database times out while processing some operation. The following initialization arguments and accessors exist:
Initarg: :database-type
Accessor: `sql-connection-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `sql-connection-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :database
Accessor: `sql-database-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

SQL-FATAL-ERROR -- condition representing a fatal error in a database connection

SQL-FATAL-ERROR

Class Precedence List

sql-connection-error, sql-database-error, sql-error, sql-condition, condition, t

Description

This condition represents errors occurring when the database connection is no longer usable. The following initialization arguments and accessors exist:
Initarg: :database-type
Accessor: `sql-connection-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `sql-connection-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :database
Accessor: `sql-database-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Large Object Support

Name

CREATE-LARGE-OBJECT --

CREATE-LARGE-OBJECT

Syntax

```
(CREATE-LARGE-OBJECT &KEY (DATABASE *DEFAULT-DATABASE*)) [function] =>
```

Arguments and Values

Description

Creates a new large object in the database and returns the object identifier

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

DELETE-LARGE-OBJECT --

DELETE-LARGE-OBJECT

Syntax

```
(DELETE-LARGE-OBJECT OBJECT-ID &KEY (DATABASE *DEFAULT-DATABASE*)) [functio
```

Arguments and Values

Description

Deletes the large object in the database

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

READ-LARGE-OBJECT --

READ-LARGE-OBJECT

Syntax

```
(READ-LARGE-OBJECT OBJECT-ID &KEY (DATABASE *DEFAULT-DATABASE*)) [function]
```

Arguments and Values

Description

Reads the large object content

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

Name

WRITE-LARGE-OBJECT --

WRITE-LARGE-OBJECT

Syntax

```
(WRITE-LARGE-OBJECT OBJECT-ID DATA &KEY (DATABASE *DEFAULT-DATABASE*)) [fun
```

Arguments and Values

Description

Writes data to the large object

Examples

Side Effects

Affected by

Exceptional Situations

See Also

Notes

CLSQL-SYS

This part gives a reference to the symbols exported from the package CLSQL-SYS, which are not exported from CLSQL package.. These symbols are part of the interface for database back-ends, but not part of the normal user-interface of *CLSQL*.

Name

DATABASE-INITIALIZE-DATABASE-TYPE -- Back-end part of `initialize-database-type`.

DATABASE-INITIALIZE-DATABASE-TYPE

Syntax

```
database-initialize-database-type database-type => result
```

Arguments and Values

database-type   A keyword indicating the database type to initialize.

result   Either t if the initialization succeeds or nil if it fails.

Description

This generic function implements the main part of the database type initialization performed by `initialize-database-type`. After `initialize-database-type` has checked that the given database type has not been initialized before, as indicated by *initialized-database-types*, it will call this function with the database type as it's sole parameter. Database back-ends are required to define a method on this generic function which is specialized via an eql-specializer to the keyword representing their database type.

Database back-ends shall indicate successful initialization by returning t from their method, and nil otherwise. Methods for this generic function are allowed to signal errors of type clsql-error or subtypes thereof. They may also signal other types of conditions, if appropriate, but have to document this.

Examples

Side Effects

All necessary side effects to initialize the database instance.

Affected By

None.

Exceptional Situations

Conditions of type clsql-error or other conditions may be signalled, depending on the database back-end.

See Also

```
initialize-database-type
```

initialized-database-types

Notes

None.

Index

Name

Alphabetical Index for package CLSQL -- Clickable index of all symbols

Alphabetical Index for package CLSQL

BACKEND-WARNING-BEHAVIOR
CACHE-TABLE-QUERIES-DEFAULT
CONNECT-IF-EXISTS
DB-AUTO-SYNC
DEFAULT-DATABASE
DEFAULT-DATABASE-TYPE
DEFAULT-UPDATE-OBJECTS-MAX-LEN
DEFAULT-VARCHAR-LENGTH
INITIALIZED-DATABASE-TYPES
ADD-SQL-STREAM
ADD-TRANSACTION-COMMIT-HOOK
ADD-TRANSACTION-ROLLBACK-HOOK
ATTRIBUTE-TYPE
BIND-PARAMETER
CACHE-TABLE-QUERIES
COMMIT
CONNECT
CONNECTED-DATABASES
CREATE-DATABASE
CREATE-INDEX
CREATE-LARGE-OBJECT
CREATE-SEQUENCE
CREATE-TABLE
CREATE-VIEW
CREATE-VIEW-FROM-CLASS
DATABASE
DATABASE-NAME
DATABASE-TYPE
DEF-VIEW-CLASS
DELETE-INSTANCE-RECORDS
DELETE-LARGE-OBJECT
DELETE-RECORDS
DELETE-SQL-STREAM
DESCRIBE-TABLE
DESTROY-DATABASE
DISABLE-SQL-READER-SYNTAX
DISCONNECT
DISCONNECT-POOLED
DO-QUERY
DROP-INDEX
DROP-SEQUENCE
DROP-TABLE
DROP-VIEW
DROP-VIEW-FROM-CLASS
ENABLE-SQL-READER-SYNTAX
EXECUTE-COMMAND
FIND-DATABASE
FOR-EACH-ROW
FREE-PREPARED-SQL
IN-TRANSACTION-P

INSTANCE-REFRESHED
LIST-ATTRIBUTE-TYPES
LIST-ATTRIBUTES
LIST-CLASSES
LIST-DATABASES
LIST-INDEXES
LIST-SEQUENCES
LIST-SQL-STREAMS
LIST-TABLE-INDEXES
LIST-TABLES
LIST-VIEWS
LOCALLY-DISABLE-SQL-READER-SYNTAX
LOCALLY-ENABLE-SQL-READER-SYNTAX
LOOP-FOR-AS-TUPLES
MAP-QUERY
PREPARE-SQL
PROBE-DATABASE
QUERY
READ-LARGE-OBJECT
RECONNECT
RESTORE-SQL-READER-SYNTAX-STATE
ROLLBACK
RUN-PREPARED-SQL
SELECT
SEQUENCE-EXISTS-P
SEQUENCE-LAST
SEQUENCE-NEXT
SET-AUTOCOMMIT
SET-SEQUENCE-POSITION
SQL
SQL-EXPRESSION
SQL-OPERATION
SQL-OPERATOR
SQL-RECORDING-P
SQL-STREAM
START-SQL-RECORDING
START-TRANSACTION
STATUS
STOP-SQL-RECORDING
TABLE-EXISTS-P
TRUNCATE-DATABASE
UPDATE-INSTANCE-FROM-RECORDS
UPDATE-OBJECTS-JOINS
UPDATE-RECORD-FROM-SLOT
UPDATE-RECORD-FROM-SLOTS
UPDATE-RECORDS
UPDATE-RECORDS-FROM-INSTANCE
UPDATE-SLOT-FROM-RECORD
VIEW-EXISTS-P
WITH-DATABASE

Appendix A. Database Back-ends

PostgreSQL

Libraries

The PostgreSQL back-end requires the PostgreSQL C client library (`libpq.so`). The location of this library is specified via *postgresql-so-load-path*, which defaults to `/usr/lib/libpq.so`. Additional flags to ld needed for linking are specified via *postgresql-so-libraries*, which defaults to ("-lcrypt" "-lc").

Initialization

Use

```
(asdf:operate 'adsf:load-op 'clsql-postgresql)
```

to load the PostgreSQL back-end. The database type for the PostgreSQL back-end is :postgresql.

Connection Specification

Syntax of connection-spec

```
(host db user password &optional port options tty)
```

Description of connection-spec

For every parameter in the connection-spec, nil indicates that the PostgreSQL default environment variables (see PostgreSQL documentation) will be used, or if those are unset, the compiled-in defaults of the C client library are used.

host	String representing the hostname or IP address the PostgreSQL server resides on. Use the empty string to indicate a connection to localhost via Unix-Domain sockets instead of TCP/IP.
db	String representing the name of the database on the server to connect to.
user	String representing the user name to use for authentication.
password	String representing the unencrypted password to use for authentication.
port	String representing the port to use for communication with the PostgreSQL server.
options	String representing further runtime options for the PostgreSQL server.
tty	String representing the tty or file to use for debugging messages from the PostgreSQL server.

PostgreSQL Socket

Libraries

The PostgreSQL Socket back-end needs *no* access to the PostgreSQL C client library, since it communicates directly with the PostgreSQL server using the published frontend/backend protocol, version 2.0. This eases installation and makes it possible to dump CMU CL images containing CLSQL and this backend, contrary to backends which require FFI code.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-postgresql-socket)
```

to load the PostgreSQL Socket back-end. The database type for the PostgreSQL Socket back-end is :postgresql-socket.

Connection Specification

Syntax of connection-spec

(*host db user password* &optional *port options tty*)

Description of connection-spec

host   If this is a string, it represents the hostname or IP address the PostgreSQL server resides on. In this case communication with the server proceeds via a TCP connection to the given host and port.

If this is a pathname, then it is assumed to name the directory that contains the server's Unix-Domain sockets. The full name to the socket is then constructed from this and the port number passed, and communication will proceed via a connection to this unix-domain socket.

db   String representing the name of the database on the server to connect to.

user   String representing the user name to use for authentication.

password String representing the unencrypted password to use for authentication. This can be the empty string if no password is required for authentication.

port   Integer representing the port to use for communication with the PostgreSQL server. This defaults to 5432.

options   String representing further runtime options for the PostgreSQL server.

tty   String representing the tty or file to use for debugging messages from the PostgreSQL server.

MySQL

Libraries

The MySQL back-end requires the MySQL C client library (`libmysqlclient.so`). The location of this library is specified via *mysql-so-load-path*, which defaults to `/usr/lib/libmysqlclient.so`. Additional flags to ld needed for linking are specified via *mysql-so-libraries*, which defaults to ("-lc").

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-mysql)
```

to load the MySQL back-end. The database type for the MySQL back-end is :mysql.

Connection Specification

Syntax of connection-spec

(*host db user password*)

Description of connection-spec

host	String representing the hostname or IP address the MySQL server resides on, or nil to indicate the localhost.
db	String representing the name of the database on the server to connect to.
user	String representing the user name to use for authentication, or nil to use the current Unix user ID.
password	String representing the unencrypted password to use for authentication, or nil if the authentication record has an empty password field.

ODBC

Libraries

The ODBC back-end requires access to an ODBC driver manager as well as ODBC drivers for the underlying database server. *CLSQL* has been tested with unixODBC ODBC Driver Manager as well as Microsoft's ODBC manager. These driver managers have been tested with the *psqlODBC* [http://odbc.postgresql.org] driver for PostgreSQL and the *MyODBC* [http://www.mysql.com/products/connector/odbc/] driver for MySQL.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-odbc)
```

to load the ODBC back-end. The database type for the ODBC back-end is :odbc.

Connection Specification

Syntax of connection-spec

(*dsn user password*)

Description of connection-spec

dsn   String representing the ODBC data source name.

user   String representing the user name to use for authentication.

password   String representing the unencrypted password to use for authentication.

AODBC

Libraries

The AODBC back-end requires access to the ODBC interface of AllegroCL named DBI. This interface is not available in the trial version of AllegroCL

Initialization

Use

```
(require 'aodbc-v2)
(asdf:operate 'asdf:load-op 'clsql-aodbc)
```

to load the AODBC back-end. The database type for the AODBC back-end is :aodbc.

Connection Specification

Syntax of connection-spec

(*dsn user password*)

Description of connection-spec

dsn   String representing the ODBC data source name.

user   String representing the user name to use for authentication.

password   String representing the unencrypted password to use for authentication.

SQLite

Libraries

The SQLite back-end requires the SQLite shared library file. Its default file name is /usr/lib/libsqlite.so.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-sqlite)
```

to load the SQLite back-end. The database type for the SQLite back-end is :sqlite.

Connection Specification

Syntax of connection-spec

(*filename*)

Description of connection-spec

filename   String representing the filename of the SQLite database file.

Oracle

Libraries

The Oracle back-end requires the Oracle OCI client library. (libclntsh.so). The location of this library is specified relative to the ORACLE_HOME value in the operating system environment. *CLSQL* has tested sucessfully using the client library from Oracle 9i and Oracle 10g server installations as well as Oracle's 10g Instant Client library.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-oracle)
```

to load the Oracle back-end. The database type for the Oracle back-end is :oracle.

Connection Specification

Syntax of connection-spec

```
(global-name user password)
```

Description of connection-spec

global-name   String representing the global name of the Orace database. This is looked up through the tnsnames.ora file.

user   String representing the user name to use for authentication.

password   String representing the password to use for authentication..

Glossary

Note

This glossary is still very thinly populated, and not all references in the main text have been properly linked and coordinated with this glossary. This will hopefully change in future revisions.

Active database	See Database Object.
Connection	See Database Object.
Closed Database	An object of type closed-database. This is in contrast to the terms connection, database, active database or database object which don't include objects which are closed database.
database	See Database Object.
Foreign Function Interface (FFI)	An interface from Common Lisp to a external library which contains compiled functions written in other programming languages, typically C.
Database Object	An object of type database.
Field Types Specifier	A value that specifies the type of each field in a query.
Structured Query Language (SQL)	An ANSI standard language for storing and retrieving data in a relational database.
SQL Expression	Either a string containing a valid SQL statement, or an object of type sql-expression.

Note

This has not been implemented yet, so only strings are valid SQL expressions for the moment.