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 on Debian Linux, 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.
- 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.

Load *UFFI*

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*)
(asdf:operate 'asdf:load-op :uffi)
```

Load MD5 module

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*)
(asdf:operate 'asdf:load-op :md5)
```

Load *CLSQL* modules

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, at a minimum, the main clsql system and at least one interface system. The below example show loading all *CLSQL* systems.

```
(push #P"/usr/share/lisp/clsql/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op 'clsql-base)              ; base CLSQL package
(asdf:operate 'asdf:load-op 'clsql-mysql)             ; MySQL interface
(asdf:operate 'asdf:load-op 'clsql-postgresql)        ; PostgreSQL interface
(asdf:operate 'asdf:load-op 'clsql-postgresql-socket) ; Socket PGSQL interface
(asdf:operate 'asdf:load-op 'clsql-odbc)              ; ODBC interface
(asdf:operate 'asdf:load-op 'clsql-sqlite)            ; SQLite interface
(asdf:operate 'asdf:load-op 'clsql-aodbc)             ; Allegro ODBC interface
(asdf:operate 'asdf:load-op 'clsql)                   ; main CLSQL package
```

Run test suite

After loading *CLSQL*, you can execute the test suite. 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)
   (managerid
    :type integer
    :nulls-ok t))
  (: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))
  (: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 employee1 (make-instance 'employee
                                 :emplid 1
                                 :first-name "Vladamir"
```

```
                                :last-name "Lenin"
                                :email "lenin@soviet.org"))

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


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

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

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

Now we can set up some of the relations between employees and companies, and their managers. The `ADD-TO-RELATION` method provides us with an easy way of doing that. It will update both the relation slot, as well as the home-key and foreign-key slots in both objects in the relation.

```
;; Lenin manages Stalin (for now)
(clsql:add-to-relation employee2 'manager employee1)

;; Lenin and Stalin both work for Widgets Inc.
(clsql:add-to-relation company1 'employees employee1)
(clsql:add-to-relation company1 'employees employee2)

;; Lenin is president of Widgets Inc.
(clsql:add-to-relation company1 'president employee1)
```

After you make any changes to an object, you have to specifically tell *CLSQL* to update the SQL database. 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 object. 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 SELECT.

```
;; 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.

CLSQL

This part gives a reference to the symbols exported from the package CLSQL-SYS, which are also re-exported from the package CLSQL. These symbols constitute the normal user-interface of *CLSQL*. Currently, the symbols of the CommonSQL-API are not documented here.

Name

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

CLSQL-CONDITION

Class Precedence List

clsql-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

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

CLSQL-ERROR

Class Precedence List

clsql-error, error, serious-condition, clsql-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

CLSQL-SIMPLE-ERROR -- Unspecific simple *CLSQL* errors

CLSQL-SIMPLE-ERROR

Class Precedence List

clsql-simple-error, simple-condition, clsql-error, error, serious-condition, clsql-condition, condition, t

Description

This condition is used in all instances of errors, where there exists no *CLSQL*-specific condition that is more specific. The valid initialization arguments and accessors are the same as for simple-condition.

Name

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

CLSQL-WARNING

Class Precedence List

clsql-warning, warning, clsql-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

CLSQL-SIMPLE-WARNING -- Unspecific simple *CLSQL* warnings

CLSQL-SIMPLE-WARNING

Class Precedence List

clsql-simple-warning, simple-condition, clsql-warning, warning, clsql-condition, condition, t

Description

This condition is used in all instances of warnings, where there exists no *CLSQL*-specific condition that is more specific. The valid initialization arguments and accessors are the same as for simple-condition.

Name

CLSQL-INVALID-SPEC-ERROR -- condition representing errors because of invalid connection specifications

CLSQL-INVALID-SPEC-ERROR

Class Precedence List

clsql-invalid-spec-error, clsql-error, error, serious-condition, clsql-condition, condition, t

Description

This condition represents errors that occur because the user supplies an invalid connection specification to either `database-name-from-spec` or `connect`. The following initialization arguments and accessors exist:
Initarg: :connection-spec
Accessor: `clsql-invalid-spec-error-connection-spec`
Description: The invalid connection specification used.
Initarg: :database-type
Accessor: `clsql-invalid-spec-error-database-type`
Description: The Database type used in the attempt.
Initarg: :template
Accessor: `clsql-invalid-spec-error-template`
Description: An argument describing the template that a valid connection specification must match for this database type.

Name

CLSQL-CONNECT-ERROR -- condition representing errors during connection

CLSQL-CONNECT-ERROR

Class Precedence List

clsql-connect-error, clsql-error, error, serious-condition, clsql-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: `clsql-connect-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `clsql-connect-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :errno
Accessor: `clsql-connect-error-errno`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :error
Accessor: `clsql-connect-error-error`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

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

CLSQL-SQL-ERROR

Class Precedence List

clsql-sql-error, clsql-error, error, serious-condition, clsql-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: `clsql-sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :expression
Accessor: `clsql-sql-error-expression`
Description: The SQL expression whose execution caused the error.
Initarg: :errno
Accessor: `clsql-sql-error-errno`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :error
Accessor: `clsql-sql-error-error`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Name

CLSQL-EXISTS-CONDITION -- condition indicating situations arising because of existing connections

CLSQL-EXISTS-CONDITION

Class Precedence List

clsql-exists-condition, clsql-condition, condition, t

Description

This condition is the super-type of all conditions which represents problems that occur during calls to `connect`, if a connection to the database exists already. Depending on the value of *`if-exists`* to the call of `connect`, either a warning, an error or no condition at all is signalled. If a warning or error is signalled, either clsql-exists-warning or clsql-exists-error is signalled, which are subtypes of clsql-exists-condition and clsql-warning or clsql-error. clsql-exists-condition is never signalled itself.

The following initialization arguments and accessors exist:
Initarg: :old-db
Accessor: `clsql-exists-condition-old-db`
Description: The database object that represents the existing connection. This slot is always filled.
Initarg: :new-db
Accessor: `clsql-exists-condition-new-db`
Description: The database object that will be used and returned by this call to connect, if execution continues normally. This can be either nil, indicating that a new database object is to be created on continuation, or a database object representing the newly created continuation, or the same database object as old-db, indicating that the existing database object will be reused. This slot is always filled and defaults to nil.

Name

CLSQL-EXISTS-WARNING -- condition representing warnings arising because of existing connections

CLSQL-EXISTS-WARNING

Class Precedence List

clsql-exists-warning, clsql-exists-condition, clsql-warning, warning, clsql-condition, condition, t

Description

This condition is a subtype of clsql-exists-condition, and is signalled during calls to `connect` when there is an existing connection, and *`if-exists`* is either :warn-new or :warn-old. In the former case, new-db will be the newly created database object, in the latter case it will be the existing old database object.

The initialization arguments and accessors are the same as for clsql-exists-condition.

Name

CLSQL-EXISTS-ERROR -- condition representing errors arising because of existing connections

CLSQL-EXISTS-ERROR

Class Precedence List

clsql-exists-error, clsql-exists-condition, clsql-error, error, serious-condition, clsql-condition, condition, t

Description

This condition is a subtype of clsql-exists-condition, and is signalled during calls to `connect` when there is an existing connection, and *`if-exists`* is :error. In this case, new-db will be nil, indicating that the database object to be returned by `connect` depends on user action in continuing from this correctable error.

The initialization arguments and accessors are the same as for clsql-exists-condition.

Name

CLSQL-CLOSED-ERROR -- condition representing errors because the database has already been closed

CLSQL-CLOSED-ERROR

Class Precedence List

clsql-closed-error, clsql-error, error, serious-condition, clsql-condition, condition, t

Description

This condition represents errors that occur because the user invokes an operation on the given database object, although the database is invalid because `disconnect` has already been called on this database object.

Functions which signal this error when called with a closed database will usually provide a continue restart, that will just return nil from the function.

The following initialization arguments and accessors exist:
Initarg: :database
Accessor: `clsql-closed-error-database`
Description: The database object that was involved in the incident.

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

INITIALIZE-DATABASE-TYPE -- Initializes a database type

INITIALIZE-DATABASE-TYPE

Syntax

`initialize-database-type &key` *`database-type`* `=> result`

Arguments and Values

`database-type`	The database type to initialize, i.e. a keyword symbol denoting a known database back-end. Defaults to the value of *default-database-type*.
result	Either nil if the initialization attempt fails, or t otherwise.

Description

If the back-end specified by *`database-type`* has not already been initialized, as seen from *initialized-database-types*, an attempt is made to initialize the database. If this attempt succeeds, or the back-end has already been initialized, the function returns t, and places the keyword denoting the database type onto the list stored in *initialized-database-types*, if not already present.

If initialization fails, the function returns nil, and/or signals an error of type clsql-error. The kind of action taken depends on the back-end and the cause of the problem.

Examples

```
*initialized-database-types*
=> NIL
(setf *default-database-type* :mysql)
=> :MYSQL
(initialize-database-type)
>> Compiling LAMBDA (#:G897 #:G898 #:G901 #:G902):
>> Compiling Top-Level Form:
>>
=> T
*initialized-database-types*
=> (:MYSQL)
(initialize-database-type)
=> T
*initialized-database-types*
=> (:MYSQL)
```

Side Effects

The database back-end corresponding to the database type specified is initialized, unless it has already been initialized. This can involve any number of other side effects, as determined by the back-end im-

plementation (like e.g. loading of foreign code, calling of foreign code, networking operations, etc.). If initialization is attempted and succeeds, the *database-type* is pushed onto the list stored in *initialized-database-types*.

Affected by

default-database-type
initialized-database-types

Exceptional Situations

If an error is encountered during the initialization attempt, the back-end may signal errors of kind clsql-error.

See Also

None.

Notes

None.

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

CONNECTED-DATABASES -- Return the list of active database objects.

CONNECTED-DATABASES

Syntax

```
connected-databases => databases
```

Arguments and Values

databases The list of active database objects.

Description

This function returns the list of active database objects, i.e. all those database objects created by calls to `connect`, which have not been closed by calling `disconnect` on them.

Caution

The consequences of modifying the list returned by `connected-databases` are undefined.

Examples

```
(connected-databases)
=> NIL
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {4830BC65}>
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {4830C5AD}>
(connected-databases)
=> (#<CLSQL-MYSQL:MYSQL-DATABASE {4830C5AD}>
    #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {4830BC65}>)
(disconnect)
=> T
(connected-databases)
=> (#<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {4830BC65}>)
(disconnect)
=> T
(connected-databases)
=> NIL
```

Side Effects

None.

Affected By

```
connect
disconnect
```

Exceptional Situations

None.

See Also

None.

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-exists :new)
=> #<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

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*.

Name

CLOSED-DATABASE -- The class representing all closed *CLSQL* databases

CLOSED-DATABASE

Class Precedence List

closed-database, standard-object, t

Description

CLSQL database instances are changed to this class via `change-class` after they are closed via `dis-connect`. All functions and generic functions that take database objects as arguments will signal errors of type clsql-closed-error when they are called on instances of closed-database, with the exception of `database-name`, which will continue to work as for instances of database.

Name

DATABASE-NAME -- Get the name of a database object

DATABASE-NAME

Syntax

`database-name` *`database`* `=> name`

Arguments and Values

`database`	A database object, either of type database or of type closed-database.
name	A string describing the identity of the database to which this database object is connected to.

Description

This function returns the database name of the given database. The database name is a string which somehow describes the identity of the database to which this database object is or has been connected. The database name of a database object is determined at `connect` time, when a call to `database-name-from-spec` derives the database name from the connection specification passed to `connect` in the *`connection-spec`* parameter.

The database name is used via `find-database` in `connect` to determine whether database connections to the specified database exist already.

Usually the database name string will include indications of the host, database name, user, or port that where used during the connection attempt. The only important thing is that this string shall try to identify the database at the other end of the connection. Connection specifications parts like passwords and credentials shall not be used as part of the database name.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48391DCD}>
(database-name *default-database*)
=> "dent/newesim/dent"

(database-name-from-spec '(nil "template1" "dent" nil) :postgresql)
=> "/template1/dent"
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-name *default-database*)
=> "/template1/dent"

(database-name-from-spec '("www.pmsf.de" "template1" "dent" nil) :postgresql)
=> "www.pmsf.de/template1/dent"
```

Side Effects

None.

Affected By

`database-name-from-spec`

Exceptional Situations

Will signal an error if the object passed as the *`database`* parameter is neither of type database nor of type closed-database.

See Also

`connect`
`find-database`

Notes

None.

Name

FIND-DATABASE -- Locate a database object through it's name.

FIND-DATABASE

Syntax

`find-database` *`database`* `&optional` *`errorp`* `=> result`

Arguments and Values

`database`	A database object or a string, denoting a database name.
`errorp`	A generalized boolean. Defaults to t.
result	Either a database object, or, if *`errorp`* is nil, possibly nil.

Description

`find-database` locates an active database object given the specification in *`database`*. If *`database`* is an object of type database, `find-database` returns this. Otherwise it will search the active databases as indicated by the list returned by `connected-databases` for a database whose name (as returned by `database-name` is equal as per `string=` to the string passed as *`database`*. If it succeeds, it returns the first database found.

If it fails to find a matching database, it will signal an error of type clsql-error if *`errorp`* is true. If *`errorp`* is nil, it will return nil instead.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48391DCD}>
(database-name *default-database*)
=> "dent/newesim/dent"

(database-name-from-spec '(nil "template1" "dent" nil) :postgresql)
=> "/template1/dent"
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-name *default-database*)
=> "/template1/dent"

(database-name-from-spec '("www.pmsf.de" "template1" "dent" nil) :postgresql)
=> "www.pmsf.de/template1/dent"

(find-database "dent/newesim/dent")
=> #<CLSQL-MYSQL:MYSQL-DATABASE {484E91C5}>
(find-database "/template1/dent")
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
```

```
(find-database "www.pmsf.de/template1/dent" nil)
=> NIL
(find-database **)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
```

Side Effects

None.

Affected By

`connected-databases`

Exceptional Situations

Will signal an error of type clsql-error if no matching database can be found, and *`errorp`* is true. Will signal an error if the value of *`database`* is neither an object of type database nor a string.

See Also

`database-name`
`database-name-from-spec`

Notes

None.

Name

CONNECT -- create a connection to a database

CONNECT

Syntax

`connect` *`connection-spec`* `&key` *`if-exists database-type pool`* `=>` `database`

Arguments and Values

`connection-spec`	A connection specification
`if-exists`	This indicates the action to take if a connection to the same database exists already. See below for the legal values and actions. It defaults to the value of *connect-if-exists*.
`database-type`	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
`pool`	A boolean flag. If T, acquire connection from a pool of open connections. If the pool is empty, a new connection is created. The default is NIL.
database	The database object representing the connection.

Description

This function takes a connection specification and a database type and creates a connection to the database specified by those. The type and structure of the connection specification depend on the database type.

The parameter *`if-exists`* specifies what to do if a connection to the database specified exists already, which is checked by calling `find-database` on the database name returned by `database-name-from-spec` when called with the *`connection-spec`* and *`database-type`* parameters. The possible values of *`if-exists`* are:

:new	Go ahead and create a new connection.
:warn-new	This is just like :new, but also signals a warning of type clsql-exists-warning, indicating the old and newly created databases.
:error	This will cause `connect` to signal a correctable error of type clsql-exists-error. The user may choose to proceed, either by indicating that a new connection shall be created, via the restart create-new, or by indicating that the existing connection shall be used, via the restart use-old.
:old	This will cause `connect` to use an old connection if one exists.
:warn-old	This is just like :old, but also signals a warning of type clsql-exists-warning, indicating the old database used, via the slots old-db and new-db

The database name of the returned database object will be the same under `string=` as that which would be returned by a call to `database-name-from-spec` with the given *`connection-spec`* and *`database-type`* parameters.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48036F6D}>
(database-name *)
=> "dent/newesim/dent"

(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
>> In call to CONNECT:
>>   There is an existing connection #<CLSQL-MYSQL:MYSQL-DATABASE {48036F6D}> to d
>>
>> Restarts:
>>   0: [CREATE-NEW] Create a new connection.
>>   1: [USE-OLD   ] Use the existing connection.
>>   2: [ABORT     ] Return to Top-Level.
>>
>> Debug  (type H for help)
>>
>> (CONNECT ("dent" "newesim" "dent" "dent") :IF-EXISTS NIL :DATABASE-TYPE ...)
>> Source:
>> ; File: /prj/CLSQL/sql/sql.cl
>> (RESTART-CASE (ERROR 'CLSQL-EXISTS-ERROR :OLD-DB OLD-DB)
>>               (CREATE-NEW NIL :REPORT "Create a new connection."
>>                (SETQ RESULT #))
>>               (USE-OLD NIL :REPORT "Use the existing connection."
>>                (SETQ RESULT OLD-DB)))
>> 0] 0
=> #<CLSQL-MYSQL:MYSQL-DATABASE {480451F5}>
```

Side Effects

A database connection is established, and the resultant database object is registered, so as to appear in the list returned by `connected-databases`.

Affected by

default-database-type
connect-if-exists

Exceptional Situations

If the connection specification is not syntactically or semantically correct for the given database type, an error of type clsql-invalid-spec-error is signalled. If during the connection attempt an error is detected (e.g. because of permission problems, network trouble or any other cause), an error of type clsql-connect-error is signalled.

If a connection to the database specified by *`connection-spec`* exists already, conditions are signalled according to the *`if-exists`* parameter, as described above.

See Also

```
connected-databases
disconnect
```

Notes

None.

Name

DISCONNECT -- close a database connection

DISCONNECT

Syntax

`disconnect &key` *`database pool`* `=> t`

Arguments and Values

`pool` A boolean flag indicating whether to put the database into a pool of opened databases. If T, rather than terminating the database connection, the connection is left open and the connection is placed into a pool of connections. Subsequent calls to `connect` can then reuse this connection. The default is NIL.

`database` The database to disconnect, which defaults to the database indicated by *default-database*.

Description

This function takes a database object as returned by `connect`, and closes the connection. The class of the object passed is changed to closed-database after the disconnection succeeds, thereby preventing further use of the object as an argument to *CLSQL* functions, with the exception of `database-name`. If the user does pass a closed database object to any other *CLSQL* function, an error of type clsql-closed-error is signalled.

Examples

```
(disconnect :database (find-database "dent/newesim/dent"))
=> T
```

Side Effects

The database connection is closed, and the database object is removed from the list of connected databases as returned by `connected-databases`.

The class of the database object is changed to closed-database.

If the database object passed is the same under `eq` as the value of *default-database*, then *default-database* is set to the first remaining database from `connected-databases` or to nil if no further active database exists.

Affected by

default-database

Exceptional Situations

If during the disconnection attempt an error is detected (e.g. because of network trouble or any other cause), an error of type clsql-error might be signalled.

See Also

`connect`
`closed-database`

Notes

None.

Name

DISCONNECT-POOLED -- closes all pooled database connections

DISCONNECT-POOLED

Syntax

```
disconnect-pool => t
```

Description

This function disconnects all database connections that have been placed into the pool. Connections are placed in the pool by calling `disconnection`.

Examples

```
(disconnect-pool)
=> T
```

Side Effects

Database connections will be closed and entries in the pool are removed.

Affected by

```
disconnect
```

Exceptional Situations

If during the disconnection attempt an error is detected (e.g. because of network trouble or any other cause), an error of type clsql-error might be signalled.

See Also

```
connect
closed-database
```

Notes

None.

Name

CREATE-DATABASE -- create a database

CREATE-DATABASE

Syntax

`create-database` *`connection-spec`* `&key` *`database-type`* `=> success`

Arguments and Values

`connection-spec`	A connection specification
`database-type`	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
`success`	A boolean flag. If T, a new database was successfully created.

Description

This function creates a database in the database system specified by *`database-type`*.

Examples

```
(create-database '("localhost" "new" "dent" "dent") :database-type :mysql)
=> T

(create-database '("localhost" "new" "dent" "badpasswd") :database-type :mysql)
=>
Error: While trying to access database localhost/new/dent
  using database-type MYSQL:
  Error database-create failed: mysqladmin: connect to server at 'localhost' faile
error: 'Access denied for user: 'root@localhost' (Using password: YES)'
  has occurred.
  [condition type: CLSQL-ACCESS-ERROR]
```

Side Effects

A database will be created on the filesystem of the host.

Exceptional Situations

An exception will be thrown if the database system does not allow new databases to be created or if database creation fails.

Notes

This function may invoke the operating systems functions. Thus, some database systems may require the administration functions to be available in the current PATH. At this time, the :mysql backend requires `mysqladmin` and the :postgresql backend requires `createdb`.

Name

DESTROY-DATABASE -- destroys a database

DESTROY-DATABASE

Syntax

`destroy-database` *`connection-spec`* `&key` *`database-type`* `=> success`

Arguments and Values

`connection-spec`	A connection specification
`database-type`	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
`success`	A boolean flag. If T, the database was successfully destroyed.

Description

This function destroy a database in the database system specified by *`database-type`*.

Examples

```
(destroy-database '("localhost" "new" "dent" "dent") :database-type :postgresql)
=> T

(destroy-database '("localhost" "new" "dent" "dent") :database-type :postgresql)
=>
Error: While trying to access database localhost/test2/root
  using database-type POSTGRESQL:
  Error database-destory failed: dropdb: database removal failed: ERROR:  database
  has occurred.
  [condition type: CLSQL-ACCESS-ERROR]
```

Side Effects

A database will be removed from the filesystem of the host.

Exceptional Situations

An exception will be thrown if the database system does not allow databases to be removed, the database does not exist, or if database removal fails.

Notes

This function may invoke the operating systems functions. Thus, some database systems may require the

administration functions to be available in the current PATH. At this time, the :mysql backend requires `mysqladmin` and the :postgresql backend requires `dropdb`.

Name

PROBE-DATABASE -- tests for existence of a database

PROBE-DATABASE

Syntax

`probe-database` *`connection-spec`* `&key` *`database-type`* `=> success`

Arguments and Values

`connection-spec`	A connection specification
`database-type`	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
`success`	A boolean flag. If T, the database exists in the database system.

Description

This function tests for the existence of a database in the database system specified by *`database-type`*.

Examples

```
(probe-database '("localhost" "new" "dent" "dent") :database-type :postgresql)
=> T
```

Side Effects

None

Exceptional Situations

An exception maybe thrown if the database system does not receive administrator-level authentication since function may need to read the administrative database of the database system.

Notes

None.

Name

DATABASE-NAME-FROM-SPEC -- Return the database name string corresponding to the given connection specification.

DATABASE-NAME-FROM-SPEC

Syntax

```
database-name-from-spec connection-spec database-type => name
```

Arguments and Values

connection-spec	A connection specification, whose structure and interpretation are dependent on the *database-type*.
database-type	A database type specifier, i.e. a keyword.
name	A string denoting a database name.

Description

This generic function takes a connection specification and a database type and returns the database name of the database object that would be created had `connect` been called with the given connection specification and database types.

This function is useful in determining a database name from the connection specification, since the way the connection specification is converted into a database name is dependent on the database type.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48391DCD}>
(database-name *default-database*)
=> "dent/newesim/dent"

(database-name-from-spec '(nil "template1" "dent" nil) :postgresql)
=> "/template1/dent"
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-name *default-database*)
=> "/template1/dent"

(database-name-from-spec '("www.pmsf.de" "template1" "dent" nil) :postgresql)
=> "www.pmsf.de/template1/dent"

(find-database "dent/newesim/dent")
=> #<CLSQL-MYSQL:MYSQL-DATABASE {484E91C5}>
(find-database "/template1/dent")
```

```
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(find-database "www.pmsf.de/template1/dent" nil)
=> NIL
(find-database **)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
```

Side Effects

None.

Affected by

None.

Exceptional Situations

If the value of *connection-spec* is not a valid connection specification for the given database type, an error of type clsql-invalid-spec-error might be signalled.

See Also

connect

Notes

None.

Name

EXECUTE-COMMAND -- Execute an SQL command which returns no values.

EXECUTE-COMMAND

Syntax

`execute-command` *`sql-expression`* `&key` *`database`* `=> t`

Arguments and Values

`sql-expression`	An sql expression that represents an SQL statement which will return no values.
`database`	A database object. This will default to the value of *default-database*.

Description

This will execute the command given by *`sql-expression`* in the *`database`* specified. If the execution succeeds it will return t, otherwise an error of type clsql-sql-error will be signalled.

Examples

```
(execute-command "create table eventlog (time char(30),event char(70))")
=> T

(execute-command "create table eventlog (time char(30),event char(70))")
>>
>> While accessing database #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {480B2B6D}>
>>   with expression "create table eventlog (time char(30),event char(70))":
>>   Error NIL: ERROR:  amcreate: eventlog relation already exists
>>   has occurred.
>>
>> Restarts:
>>   0: [ABORT] Return to Top-Level.
>>
>> Debug  (type H for help)
>>
>> (CLSQL-POSTGRESQL::|(PCL::FAST-METHOD DATABASE-EXECUTE-COMMAND (T POSTGRESQL-DA
>>  #<unused-arg>
>>  #<unused-arg>
>>  #<unavailable-arg>
>>  #<unavailable-arg>)
>> Source: (ERROR 'CLSQL-SQL-ERROR :DATABASE DATABASE :EXPRESSION ...)
>> 0] 0

(execute-command "drop table eventlog")
=> T
```

Side Effects

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

Affected by

None.

Exceptional Situations

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

See Also

`query`

Notes

None.

Name

QUERY -- Execute an SQL query and return the tuples as a list

QUERY

Syntax

`query` *`query-expression`* `&key` *`database result-types field-names`* `=> result`

Arguments and Values

`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. The purpose of this argument is cause *CLSQL* to import SQL numeric fields into numeric Lisp objects rather than strings. This reduces the cost of allocating a temporary string and the *CLSQL* users' inconvenience of converting number strings into number objects. A value of :auto causes *CLSQL* to automatically convert SQL fields into a numeric format where applicable. The default value of NIL causes all fields to be returned as strings regardless of the SQL type. Otherwise a list is expected which has a element for each field that specifies the conversion. If the list is shorter than the number of fields, the a value of t is assumed for the field. If the list is longer than the number of fields, the extra elements are ignored. :int Field is imported as a signed integer, from 8-bits to 64-bits depending upon the field type. :double Field is imported as a double-float number. t Field is imported as a string.
`field-names` result	A list representing the result set obtained. For each tuple in the result set, there is an element in this list, which is itself a list of all the attribute values in the tuple.

Description

This will execute the query given by *`query-expression`* in the *`database`* specified. If the execution succeeds it will return the result set returned by the database, otherwise an error of type clsql-sql-error will be signalled.

Examples

```
(execute-command "create table simple (name char(50), salary numeric(10,2))")
=> T
```

```
(execute-command "insert into simple values ('Mai, Pierre',10000)")
=> T
(execute-command "insert into simple values ('Hacker, Random J.',8000.50)")
=> T
(query "select * from simple")
=> (("Mai, Pierre" "10000.00") ("Hacker, Random J." "8000.50"))
(query "select salary from simple")
=> (("10000.00") ("8000.50"))
(query "select salary from simple where salary > 10000")
=> NIL
(query "select salary,name from simple where salary > 10000")
=> NIL
(query "select salary,name from simple where salary > 9000")
=> (("10000.00" "Mai, Pierre"))
(query "select salary,name from simple where salary > 8000")
=> (("10000.00" "Mai, Pierre") ("8000.50" "Hacker, Random J."))

;; MySQL-specific:
(query "show tables")
=> (("demo") ("log") ("newlog") ("simple") ("spacetrial"))
```

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 clsql-sql-error is signalled.

See Also

`execute-command`

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`* must take as many arguments as are attributes in the result set returned by executing the SQL *`query-expression`*.
`query-expression`	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`	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 attributes of 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.

Examples

```
(map-query 'list #'(lambda (salary name)
                     (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 (salary name)
               (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 (salary name)
                             (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 clsql-sql-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

DO-QUERY -- Iterate over all the tuples of a query

DO-QUERY

Syntax

`do-query ((&rest` *`args`*`) `*`query-expression`*` &key `*`database result-types`*`) &body `*`body`*` =>`

Arguments and Values

`args`	A list of variable names.
`query-expression`	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`	A database object. This will default to *default-database*.
`result-types`	A field type specifier. The default is NIL. See `query` for the semantics of this argument.
`body`	A body of Lisp code, like in a `destructuring-bind` form.

Description

Executes the *`body`* of code repeatedly with the variable names in *`args`* bound to the attributes of each tuple in the result set returned by executing the SQL *`query-expression`* on the *`database`* specified.

The body of code is executed in a block named nil which may be returned from prematurely via `return` or `return-from`. In this case the result of evaluating the `do-query` form will be the one supplied to `return` or `return-from`. Otherwise the result will be nil.

The body of code appears also is if wrapped in a `destructuring-bind` form, thus allowing declarations at the start of the body, especially those pertaining to the bindings of the variables named in *`args`*.

Examples

```
(do-query ((salary name) "select salary,name from simple")
  (format t "~30A gets $~2,5$~%" name (read-from-string salary)))
>> Mai, Pierre                    gets $10000.00
>> Hacker, Random J.              gets $08000.50
=> NIL

(do-query ((salary name) "select salary,name from simple")
  (return (cons salary name)))
=> ("10000.00" . "Mai, Pierre")
```

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 clsql-sql-error is signalled.

If the number of variable names in *args* and the number of attributes in the tuples in the result set don't match up, an error is signalled.

See Also

`query`
`map-query`

Notes

None.

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 clsql-sql-error is signalled.

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

See Also

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

Notes

None.

CLSQL-BASE

This part gives a reference to the symbols exported from the package CLSQL-BASE, 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.

Appendix A. Database Back-ends

PostgreSQL

Libraries

The PostgreSQL back-end needs access to 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 needs access to 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 access to 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.

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.