CLSQL Users' Guide

Kevin M. Rosenberg
Maintainer of CLSQL

Pierre R. Mai
Author of Original MaiSQL Code

***CLSQL* Users' Guide**
by Kevin M. Rosenberg and Pierre R. Mai

$Date: 2002/03/23 15:17:43 $

$Id: bookinfo.sgml,v 1.3 2002/03/23 15:17:43 kevin Exp $

Table of Contents

Preface

This guide provides reference to the features of *CLSQL*. The first chapter provides an introduction to *CLSQL* and installation instructions. Following that chapter is the reference section for all user accessible symbols of *CLSQL* with examples of usage, followed by the reference section for all accessible symbols of the database back-end interface. At the end there you will find 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

CLSQL is written by Kevin M. Rosenberg and based substantially on Pierre R. Mai's excellent *MaiSQL* package. The main changes from *MaiSQL* are:

- port from the CMUCL FFI to *UFFI*.
- new AllegroCL ODBC interface back-end.
- compatibility layer for CMUCL specific code.
- much improved robustness for the MySQL back-end.
- improved system loading.
- improved packages and symbol export.
- transaction support.

Prerequisites

Defsystem

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.med-info.com/) as a *Foreign Function Interface* (*FFI*) to support multiple ANSI Common Lisp implementations.

You can download *UFFI* from its FTP *site* (ftp://ftp.med-info.com/pub/uffi/). There are zip files for Microsoft Windows systems and gzipped tar files for other systems.

UFFI

CLSQL's postgresql-socket interface uses Pierre Mai's md5 (ftp://clsql.b9.com/) module. If you plan to use this interface please download the md5 module from ftp://clsql.b9.com

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 v4.2 on Debian Linux and Microsoft Windows XP.
- CMUCL 18d on Debian Linux, FreeBSD 4.5, and Solaris 2.8.
- SBCL 0.7.14 on Debian Linux.
- SCL 1.1 on Debian Linux.
- OpenMCL 0.13 on Debian Linux PowerPC.

Supported SQL Implementation

Currently, *CLSQL* supports the following databases:

- MySQL v3.23.51.
- PostgreSQL v7.2 with both direct API and TCP socket connections.
- Allegro's ODBC interface (AODBC) using iODBC ODBC manager.

Installation

Ensure Defsystem is loaded

Simply load the file `defsystem.lisp`.

```
(load "defsystem.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.

Makefiles for Microsoft Windows and GNU/Solaris systems are supplied to build the libraries. Since many Microsoft Windows users don't have access to a compiler, the `DLL` and `LIB` files for Microsoft Windows are supplied with the distribution.

To build the libraries on a GNU or Solaris, use the shell and change to the root directory of *CLSQL*. You may need to edit the file `interfaces/mysql/Makefile` to specify the location of your MySQL installation. The default Makefiles are setup for shared library linking on Linux. If you are using FreeBSD or Solaris, you will need to change the linker setting as instructed in the Makefile. Then, you can give the command

```
make libs
```

in the root directory of *CLSQL* to build the libraries `interfaces/mysql/clsql-mysql.so` and `interfaces/clsql-uffi/clsql-uffi.so`.

Load *UFFI*

Unzip or untar the *UFFI* distribution which creates a directory for the *UFFI* files. Add that directory to Defsystem'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:oos '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 Defsystem'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:oos '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 Defsystem'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.

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

Run test suite

After loading *CLSQL*, you can execute the test program in the directory `./test-suite`. The test file, `tester-clsql` has instructions for creating a `test.config`. After creating that file, simple load the test file with Lisp and the tests should automatically execute.

I. CLSQL

This part gives a reference to all 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*.

CLSQL-CONDITION

Name

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

Condition Type

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.

CLSQL-ERROR

Name

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

Condition Type

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.

CLSQL-SIMPLE-ERROR

Name

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

Condition Type

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.

CLSQL-WARNING

Name

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

Condition Type

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.

CLSQL-SIMPLE-WARNING

Name

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

Condition Type

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.

CLSQL-INVALID-SPEC-ERROR

Name

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

Condition Type

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.

CLSQL-CONNECT-ERROR

Name

CLSQL-CONNECT-ERROR — condition representing errors during connection

Condition Type

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.

CLSQL-SQL-ERROR

Name

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

Condition Type

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.

CLSQL-EXISTS-CONDITION

Name

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

Condition Type

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.

CLSQL-EXISTS-WARNING

Name

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

Condition Type

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.

CLSQL-EXISTS-ERROR

Name

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

Condition Type

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.

CLSQL-CLOSED-ERROR

Name

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

Condition Type

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.

DEFAULT-DATABASE-TYPE

Name

`*DEFAULT-DATABASE-TYPE*` — The default database type to use

Variable

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.

INITIALIZED-DATABASE-TYPES

Name

`*INITIALIZED-DATABASE-TYPES*` — List of all initialized database types

Variable

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.

INITIALIZE-DATABASE-TYPE

Name

`INITIALIZE-DATABASE-TYPE` — Initializes a database type

Function

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 implementation (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.

CONNECT-IF-EXISTS

Name

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

Variable

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.

CONNECTED-DATABASES

Name

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

Function

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.

DEFAULT-DATABASE

Name

DEFAULT-DATABASE — The default database object to use

Variable

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.

DATABASE

Name

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

Class

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*.

CLOSED-DATABASE

Name

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

Class

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 `disconnect`. 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`.

DATABASE-NAME

Name

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

Generic Function

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.

FIND-DATABASE

Name

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

Function

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.

CONNECT

Name

`CONNECT` — create a connection to a database

Function

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 databas
>>
>> 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.

DISCONNECT

Name

DISCONNECT — close a database connection

Function

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.

DISCONNECT-POOLED

Name

`DISCONNECT-POOLED` — closes all pooled database connections

Function

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.

DATABASE-NAME-FROM-SPEC

Name

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

Generic Function

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.

EXECUTE-COMMAND

Name

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

Function

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-DATABASE
>>  #<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.

QUERY

Name

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

Function

Syntax

`query` *`query-expression`* `&key` *`database types => 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*`.

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.

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.

MAP-QUERY

Name

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

Function

Syntax

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

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*.

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.

DO-QUERY

Name

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

Macro

Syntax

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

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*.

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.

LOOP-FOR-AS-TUPLES

Name

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

Loop Clause

Compatibility

> **Caution**
>
> loop-for-as-tuples only works with CMUCL.

Syntax

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

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.

II. CLSQL-SYS

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

DATABASE-INITIALIZE-DATABASE-TYPE

Name

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

Generic Function

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

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

```
(mk:load-system :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.

AODBC

Libraries

The AODBC back-end requires access to the ODBC interface of AllegroCL.

Initialization

Use

```
(mk:load-system :clsql-aodbc)
```

to load the MySQL 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.

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

```
(mk:load-system :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

```
(mk:load-system :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.

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.