



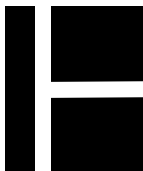
More ALS Prolog Tools



Copyright (c) 1998-99 Applied Logic Systems, Inc.

Restricted Rights Legend

When the Licensee is the U.S. Government or a duly authorized agency thereof, use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subdivision (b)(3)(II) of the Rights in Technical Data and Computer Software clause at 52.277.7013, dated Nov. 9, 1984.



Applied Logic Systems, Inc.

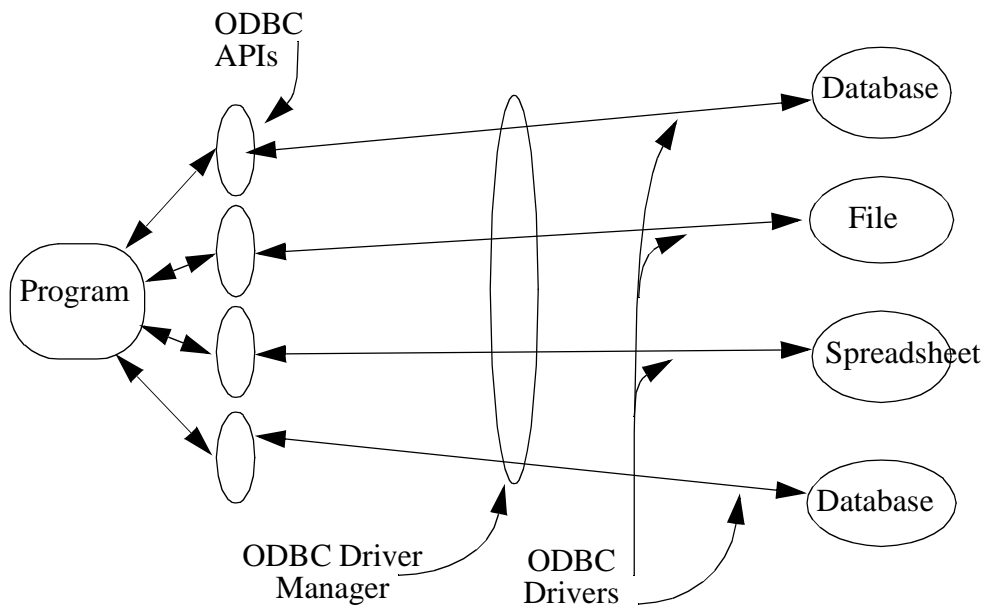
PO Box 400175
Cambridge, MA 02140 USA
Email: info@als.com
WWW: <http://www.als.com>



1 Using ODBC¹

ODBC (Open DataBase Connectivity) is a standardized approach to accessing and manipulating external data from programs. It provides an API (Application Programming Interface) based on the SQL data query language. ODBC is principally oriented towards relational databases, though it can be utilized with data sources as diverse as flat files, relational databases managed by systems such as ORACLE, Postgres, Access, and SYBASE, and spreadsheets created by systems such as EXCEL.

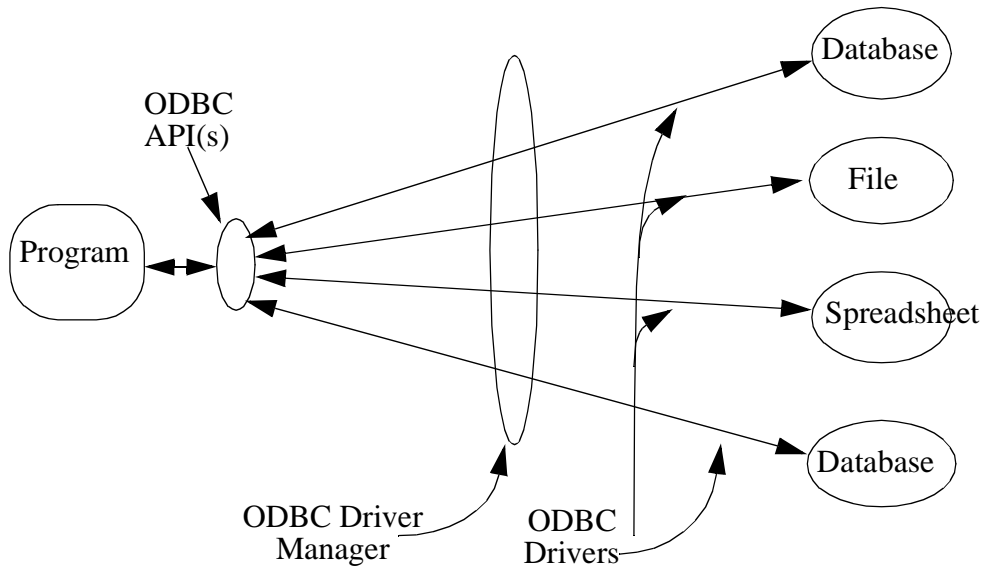
The ODBC standard *specifies* the API a program may use. This API is *realized* by a combination of ODBC drivers and ODBC driver managers, as suggested in this diagram:



Note that in principle, the various ODBC APIs should all be identical, and so the

1. ALS Prolog ODBC Interface was developed by Chuck Houpt.

diagram could just as well have been drawn this way:



There is normally just one ODBC Driver Manager resident in each computing environment. As its name indicates, it *manages* the various ODBC drivers for connecting to diverse data sources. Typically, there is one ODBC driver for each class of data source. Thus, there will be an ODBC driver for ORACLE databases, a different ODBC driver for SYBASE databases, a different driver for EXCEL spreadsheets, etc., etc. Each of these drivers is capable of providing access to all of the specific data sources in that class. Thus, the ORACLE ODBC driver provides access to all ORACLE-managed databases, while the EXCEL ODBC driver provides access to all EXCEL spreadsheets.

So, to use ODBC, you must obtain (if necessary), both an ODBC Driver Manager, together with ODBC Drivers for the various classes of data you wish to access.

The ALS Prolog ODBC Interface makes the function calls in the ODBC API available to ALS Prolog programs. The interface is implemented as a *shared prolog library* (a “*.psl” file) which can be dynamically loaded whenever your program needs to utilize ODBC. The direct interface between ALS Prolog and ODBC is rather ugly and difficult to use, and so several files are included in the interface to make using the ODBC interface considerably simpler. These files define prolog

predicates making the use of the underlying raw ODBC interfaces, but which are more oriented towards the ordinary Prolog point of view. Here is a list of the files together with brief descriptions of their roles:

odbcintf.psl	The shared prolog library providing linkage to the ODBC library functions.
odbc.pro	The “prolog-visible” side of the the odbcintf.psl linkage; the “raw” interface; very ugly.
prolog_odbc.pro	Predicates providing access to the basic odbc.pro-level predicates, but with appropriate type conversions and error handling support.
meta_odbc.pro	An abstraction layer above the prolog_odbc.pro-level predicates. Most programs using ODBC will want to work at this level, only dipping into the prolog_odbc.pro-level as needed.

There are many books describing SQL; see, for example, C.J. Date, *A Guide to DB2*, Addison-Wesley, 1984, the crisp introduction on pp. 210-239 of J.D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I*, Computer Science Press, 1988, or the web-based tutorials at

<http://www.willcam.com/sql/>

http://www.soc.staffs.ac.uk/~cmntrk/4gl/99/sql_tut.htm

(also at http://www.vetmed.auburn.edu/~campbj3/sql_tut.htm)

ODBC is discussed at length in K.Geiger, *Inside ODBC*, Microsoft Press, 1995, and also in the ODBC.hlp on-line help reference contained in the Microsoft SDK for ODBC.

2 An Example: SQL_Shell.

In addition to the files listed in the previous section, the distribution contains another file, `sql_shell.pro`, which implements a Prolog-ODBC program providing interactive SQL access to data sources via whatever ODBC drivers are resident where the program runs. Note that the ALS Prolog project `sql_shell.ppj` contains all of the four prolog (*.pro) files discussed above. The shared library, `odbcintf.psl`, must be separately loaded. The file 'odbcintf.psl' should normally reside in the directory

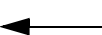
`alsdir/shared {or alsdir\shared or :alsdir:shared }`

in the ALS Prolog installation directory. Use `consult('odbcintf.psl')` to load it.

Here is a sample of the execution of `sql_shell`:

ALS Prolog (Byte) Version 3.1.2 [mswinnt]

Copyright (c) 1987-99 Applied Logic Systems, Inc.

?- **consult('odbcintf.psl')**.  *Use consult/1, not load_slib/1.*

Attempting to consult odbcintf.psl...

... consulted C:\PROGRA~1\ALSPRO~1\alsdir\shared\odbcintf.psl

yes.

Load the project 'sql_shell.ppj' (or consult all of the files in the project).

?- Attempting to consult sql_shell.pro...

Attempting to consult odbc...

... consulted D:\odbc\mswinnt\odbc.obp

Attempting to consult prolog_odbc...

... consulted D:\odbc\mswinnt\prolog_odbc.obp

Attempting to consult meta_odbc...

... consulted D:\odbc\mswinnt\meta_odbc.obp

... consulted D:\odbc\sql_shell.pro

Attempting to consult odbc.pro...


... consulted D:\odbc\odbc.pro

Attempting to consult prolog_odbc.pro...

... consulted D:\odbc\prolog_odbc.pro

Attempting to consult meta_odbc.pro...



... consulted D:\odbc\meta_odbc.pro

?- sql_init.  *Low-level initialization. Defined in meta_odbc.pro*

yes.

?- sql_shell. *One of several ways of starting this example; defined in sql_shell.pro. This way of starting causes the ODBC driver manager to prompt the user for a data source to which to connect. (The data source must have been previously specified to the ODBC driver manager.) The output below is the ODBC Driver Manager's response when the user selected the sample Access database "Northwind.mdb".*

Connected to DSN=NorthwindDB;DBQ=C:\Program Files\Microsoft Access\Office\Samples\Northwind.mdb;DriverId=25;FIL=MS Access;MaxBufferSize=512;PageTimeout=5;PWD=logical;UID=admin;

 *Sample program(sql_shell.pro) prompt*
 *User input SQL statement*

SQL> SELECT * from customers

['ALFKI','Alfreds Futterkiste','Maria Anders','Sales Representative',

```
'Obere Str. 57','Berlin','12209','Germany','030-0074321','030-0076545']
['ANATR','Ana Trujillo Emparedados y helados','Ana Trujillo','Owner',
'Avda. de la Constituci\363n 2222','M\351xico D.F.','05021','Mexico',
'(5) 555-4729','(5) 555-3745']
```

```
.....
['WOLZA','Wolski Zajazd','Zbyszek Piestrzeniewicz','Owner',
'ul. Filtrowa 68','Warszawa','01-012','Poland','(26) 642-7012',
'(26) 642-7012']
```

```
SQL> ^D Control-D exits from this sample program.
```

yes.

?- sql_shell. *Start sql_shell again, and connect to the :economics.mdb Access datasource.*

Connected to DSN=Economics;DBQ=D:\databases\odbc\economics.mdb;Driver-Id=25;FIL=MS Access;MaxBufferSize=512;PageTimeout=5;UID=admin;

Sample program(sql_shell.pro) prompt

User input SQL statement

```
SQL> SELECT * from cpi
[1.0,'All items (1967=100)',100.0,164.5,165.0,1.7,0.3,0.1,0.1,0.2]
[2.0,'Food and beverages',16.408,163.8,163.7,2.2,-0.1,0.4,0.2,-0.2]
.....
[90.0,'Energy commodities',2.72,83.9,86.4,-5.4,3.0,0.0,-0.5,3.5]
[91.0,'Services less energy services',54.316,194.0,194.7,2.8,0.4,0.2,0.2,0.3]
```

SQL> **^D** *Control-D exits from this sample program.*

yes.

3 Accessing Catalog Information.

Catalog information is easily obtained. Consider this predicate defined by meta_odbc (DS for 'DataSource')

```
ds_tables(DS, TL)
:-
    sql_open_connection(, '', '', C),
    sql_tables(C, TL).
```

We obtain information on the tables in datasource 'Economics' as follows:

```
?- ds_tables('Economics',X).

X=[
['D:\\databases\\odbc\\economics','','MSysACEs','SYSTEM TABLE'],
['D:\\databases\\odbc\\economics','','MSysIMEXColumns','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','MSysIMEXSpecs','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','MSysModules','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','MSysModules2','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','MSysObjects','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','MSysQueries','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','MSysRelationships','SYSTEM
TABLE'],
['D:\\databases\\odbc\\economics','','Cpi','TABLE'],
['D:\\databases\\odbc\\economics','','finished_goods','TABLE']]
yes.
```

Information about the columns of a table is also easily obtained. Here is the definition:

```
ds_table_cols(DS, Table, ColsList)
```

```
:-
  sql_open_connection('Economics', '', '', C),
  sql_columns(C, Table, ColsList0),
  cvrt_to_tms(ColsList0, c, ColsList).
```

The basic access predicate, `sql_columns/3`, returns information as a list of 13-element sublists. To make access to the elements of the individual row information simple, the predicate `ds_cols` converts these lists to terms with functor 'c'. Then the individual elements can easily be accessed using `arg/3`. In particular, the column name appears in position arg 4, and the column type appears in position arg 6.

```
?- ds_table_cols('Economics', cpi, X).
X=[
  c('D:\\databases\\odbc\\economics',"",cpi,'ID',4.0,
    'COUNTER',10.0,4.0,0.0,10.0,1.0,"",0.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'Expenditure category',12.0,
    'TEXT',255.0,255.0,0.0,10.0,1.0,"",1.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'RellmpDec-
    99',8.0,'DOUBLE', 15.0,8.0,0.0,10.0,1.0,"",2.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'UadjIdx2-99',8.0,'DOUBLE',
    15.0,8.0,0.0,10.0,1.0,"",3.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'UadjIdx3-99',8.0,'DOUBLE',
    15.0,8.0,0.0,10.0,1.0,"",4.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'UadjPctChg3-98to3-99',8.0,
    'DOUBLE',15.0,8.0,0.0,10.0,1.0,"",5.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'UadjPctChg2-99to3-99',8.0,
    'DOUBLE',15.0,8.0,0.0,10.0,1.0,"",6.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'SeasAdjPctDec-Jan',8.0,
    'DOUBLE',15.0,8.0,0.0,10.0,1.0,"",7.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'SeasAdjPctJan-Feb',8.0,
    'DOUBLE',15.0,8.0,0.0,10.0,1.0,"",8.0),
  c('D:\\databases\\odbc\\economics',"",cpi,'SeasAdjPctFeb-Mar',8.0,
    'DOUBLE',15.0,8.0,0.0,10.0,1.0,"",9.0)]
```

Finally, meta_odbc includes two calls for obtaining all the information the ODBC driver manager knows about ODBC drivers, and about datasources, as follows:

```
?- sql_all_drivers(DriversList).
```

```
DriversList=[
['Microsoft Access Driver (*.mdb)', 'UsageCount=1'],
['Microsoft dBase Driver (*.dbf)', 'UsageCount=1'],
['Microsoft Excel Driver (*.xls)', 'UsageCount=1'],
['Microsoft FoxPro Driver (*.dbf)', 'UsageCount=1'],
['Microsoft Text Driver (*.txt; *.csv)', 'UsageCount = 1'],
['SQL Server', 'DSNConverted=F'],
['PostgreSQL', 'APILevel=1']]
```

```
yes.
```

```
?- sql_all_data_sources(SourcesList).
```

```
SourcesList=[
['dBASE Files', 'Microsoft dBase Driver (*.dbf)'],
['Excel Files', 'Microsoft Excel Driver (*.xls)'],
['FoxPro Files', 'Microsoft FoxPro Driver (*.dbf)'],
['Text Files', 'Microsoft Text Driver (*.txt; *.csv)'],
['MS Access 97 Database', 'Microsoft Access Driver (*.mdb)'],
['Test Contacts', 'Microsoft Access Driver (*.mdb)'],
['Economics', 'Microsoft Access Driver (*.mdb)']]
```

```
yes.
```

4 Directly Accessing Tables from Prolog.

Consider the following example which illustrates a method for making an ODBC data source table appear as a set of facts in Prolog. The datasource 'Economics' contains a table cpi holding current Consumer Price Index information.

```
cpi_row_view(Row) :-
    sql_open_connection('Economics', '', '', C),
    printf('Connected to: %s\n', ['Economics']),
    sql_open_statement(C,
        'select "Expenditure Category",
            "RelImpDec-99",
            "UadjPctChg3-98to3-99",
            "SeasAdjPctJan-Feb"
            from cpi', S),
    sql_execute_statement(S),
    get_a_row(S, Row).

get_a_row(S, Row)
:-
    sql_fetch_row(S, Row).
get_a_row(S, Row)
:-
    get_a_row(S, Row).
```

Executing this, we get:

```
?- cpi_row_view(X).
Connected to: Economics
X=['All items (1967=100)',100.0,1.7,0.1] ;
X=['Food and beverages',16.408,2.2,0.2] ;
X=['Food',15.422,2.3,0.1] ;
X=['Food at home',9.691,2.0,0.1] ;
.....etc.....
```

Next, suppose we desire a simple 4-place predicate corresponding to cpi_row_view/1. It can be defined as follows:

```

simple_cpi_view(Cat, RelImp, UPCMarch99, SeasPctFeb)
:-
    sql_open_connection('Economics', '', '', C),
    sql_open_statement(C,
        'select "Expenditure Category", "RelImpDec-
99", "UadjPctChg3-98to3-99", "SeasAdjPctJan-Feb" from cpi',
        S),
    sql_execute_statement(S),
    get_a_row(S, [Cat, RelImp, UPCMarch99, SeasPctFeb]).

```

This predicate was called “simple” because it does not pass conditions on its arguments down to the database retrieval mechanism. That is, if any of its arguments are instantiated, this information is not utilized until the call to `get_a_row/2`, long after the data has been retrieved from the database and passed back into the application. To force information about instantiated variables down to the database retrieval mechanism, one would add a `WHERE` clause in the SQL `SELECT` statement. This begins to get quite complicated to carry out by hand. In Chapter 6 (*Generating Predicate Interfaces*) we introduce meta-level predicates which automatically create such definitions.

Finally, assume that another table with the following columns has been added to the Economics database as follows:

datatable1:

```

text1      - text
value1     - number

```

Consider the following two code samples which insert and retrieve values from this table:

```

do_insert1(Text, Num) :-
    sql_open_connection('Economics', '', '', C),
    printf('Connected to: %s\n', ['Economics']),
    sprintf(atom(SQLCmd),
        'INSERT INTO datatable1(text1, value1) VALUES
(\'%t\', %t)',
        [Text, Num] ),
    sql_open_statement(C, SQLCmd, S),

```

```
sql_execute_statement(S).
```

```
do_retrieve1(Rows) :-  
    sql_open_connection('Economics', '', '', C),  
    printf('Connected to: %s\n', ['Economics']),  
    sql_open_statement(C, 'SELECT ALL * FROM datatable1', S),  
    sql_execute_statement(S),  
    sql_fetch_all_rows(S, Rows).
```

Here are some sample executions:

```
?- do_retrieve1(X).
```

```
Connected to: Economics
```

```
X=[]
```

```
yes.
```

```
?-do_insert1('Some kind of thing', 23.456).
```

```
Connected to: Economics
```

```
yes.
```

```
?- do_insert1('And another', 6767676).
```

```
Connected to: Economics
```

```
yes.
```

```
?-do_retrieve1(X).
```

```
Connected to: Economics
```

```
X=[[ 'Some kind of thing',23.0],
```

```
    [ 'And another',6767676.0]]
```

yes.

5 Predicates Defined in meta_odbc.pro

Here is a schematic example of using the meta-ODBC interfa

```
sql_init.  
sql_open_connection('sdk21-Access32', '', '', C),  
sql_open_statement(C, 'select * from customer', S),  
sql_execute_statement(S),  
sql_fetch_row(S, R),  
sql_close_statement(S),  
sql_close_connection(C),  
sql_shutdown.  
R = ['bob', '203 Main St.', ...]
```

The predicates in meta_odbc.pro are all defined in the module odbc, and are exported for use in other modules which "use odbc."

```
sql_init/0  
sql_init  
sql_init
```

Allocate a global environment for ODBC. Defined by:

```
sql_init :- sql_init(_).
```

```
sql_init/1  
sql_init(Env)  
sql_init(-)
```

Allocate and return a global environment for ODBC. Calls sql_alloc_env(Environment) to set this global, obtains a handle Environment to it, and stores this in the prolog global variable "_odbc_environment":

```
set_odbc_environment(Environment)  
get_odbc_environment(Environment)
```

If there is an existing non-zero value for Environment in

```
get_odbc_environment(Environment),
```

sql_init/1 does nothing.

```
sql_shutdown/0
sql_shutdown
sql_shutdown
```

Deallocates the currently allocated ODBC global environment.

There are several ODBC approaches to connecting to a datasource.

```
sql_open_connection/4
sql_open_connection(DataSource, User, Password, Connection)
sql_open_connection(+, +, +, -)
```

Opens a connection to a DataSource, supplying User and Password as appropriate; returns an ODBC connection. Example:

```
sql_open_connection('Economics', '', '', C),

sql_open_connection/3
sql_open_connection(ConString, OutString, Connection)
sql_open_connection(+, -, -)
```

- open a connection to a data source

```
sql_open_connection/2
sql_open_connection(Connection, ConnectionString)
sql_open_connection(-, -)
```

Open a connection to a data source; ODBC Driver Manager prompts user for connection information.

```
sql_close_connection/1
sql_close_connection(Connection)
sql_close_connection(+)
```

Deallocate an SQL/ODBC connection.

```
sql_commit  
sql_commit(Connection)  
sql_commit(+)
```

Submit a statement 'SQL_COMMIT' to a data source.

```
sql_open_statement/3  
sql_open_statement(Connection, SQLQuery, StatementTerm)  
sql_open_statement(+, +, -)
```

Allocate a Prolog-level statement data structure, and appropriately prepare the data structure for SQLQuery. `StatementTerm` is a Prolog term of the form

```
statement(StatementHandle, ColumnInfoList, ParamInfoList)
```

where `StatementHandle` refers to the allocated ODBC Statement entity, and `ColumnInfoList` and `ParamInfoList` are Prolog lists of terms containing information corresponding to the columns and parameters occurring in SQLQuery.

```
sql_close_statement/1  
sql_close_statement(StatementTerm)  
sql_close_statement(+)
```

Deallocate an SQL statement data structure. `StatementTerm` must be a term returned by a call to

```
sql_open_statement/3
```

and is of the form

```
statement(StatementHandle, ColumnInfoList, ParamInfoList)
```

Also deallocates the associated column info structures.

```
sql_execute_statement/1
```

```
sql_execute_statement(StatementTerm)
sql_execute_statement(+)
```

Executes an SQL/ODBC statement term. StatementTerm must be a term returned by a call to

```
sql_open_statement/3
```

and is of the form

```
statement(StatementHandle, ColumnInfoList, ParamInfoList).
```

```
sql_fetch_row/2
sql_fetch_row(StatementTerm, Row)
sql_fetch_row(+, -)
```

Fetches a row of returned data from a StatementTerm. Row will be a Prolog list of the values.

```
sql_fetch_all_rows/2.
sql_fetch_all_rows(StatementTerm, RowList).
sql_fetch_all_rows(+, -).
```

Returns the list of a rows which can be fetched from StatementTerm

```
sql_tables/2.
sql_tables(Connection, TablesList)
sql_tables(+, -)
```

Returns information about all the tables contained in the DataSource attached to Connection.

```
sql_columns/3.
sql_columns(Connection, Table, ColsList)
sql_columns(Connection, Table, ColsList)
```

Returns information about all the columns of `Table`, which must be a table contained in the `DataSource` attached to `Connection`.

```
sql_all_drivers/1  
sql_all_drivers(DriversList)  
sql_all_drivers(-)
```

Obtain the list, `DriversList`, of descriptions of all ODBC drivers known by the ODBC Driver Manager. The elements of the list are of the form

```
[Desc, Attrib],
```

where `Desc` and `Attrib` are both atoms.

```
sql_all_data_sources/1  
sql_all_data_sources(SourcesList)  
sql_all_data_sources(-)
```

Obtains the list, `SourcesList`, of descriptions of all data sources known by the ODBC Driver Manager. The elements of the list are of the form

```
[Name, Desc],
```

where `Name` and `Desc` are both atoms.

6 Generating Predicate Interfaces

The creation of the access and insertion routines for individual predicates described in Chapter 4 (*Directly Accessing Tables from Prolog.*), including WHERE clauses to pass variable instantiation information to the database retrieval mechanism, can be encapsulated in a meta-level routine. One provides a specification, `Spec`, describing the desired interface, and then utilizes the call

```
defSQLview(Spec)
```

to cause the code to be created and asserted. `defSQLview/1` is implemented as a metapredicate (`module_closure`), so that the generated code is asserted in whatever module the call is made to `defSQLview/1`. Consider the predicate `simple_cpi_view/4` created in Chapter 4 (*Directly Accessing Tables from Prolog.*). Suppose that we desire a related predicate `my_cpi/4` with four arguments, but which passes variable instantiation to the database mechanism, together with a predicate `insert_my_cpi/4` which inserts new rows in the table `cpi`. The appropriate `Spec` to use is:

```
Spec = [
    ds = 'Economics',
    table = 'cpi',
    cols_list = [
        'Expenditure category', 'RelImpDec-99',
        'UadjPctChg3-98to3-99', 'SeasAdjPctJan-Feb'],
    pred=my_cpi
]
```

Thus, at the prolog command line:

```
?- Spec = [
    ds = 'Economics',
    table = 'cpi',
    cols_list = [
        'Expenditure category', 'RelImpDec-99',
        'UadjPctChg3-98to3-99', 'SeasAdjPctJan-Feb'],
    pred=my_cpi
],
```

```
defSQLview(Spec).
yes.
```

Then:

```
?- listing(my_cpi/_).

% user:my_cpi/4
my_cpi(_A,_B,_C,_D) :-
    sql_open_connection(Economics,,,_E),
    odbc:
        where_conds([_A,_B,_C,_D],
            [Expenditure category,RelImpDec-99,UadjPctChg3-
98to3-99,
                SeasAdjPctJan-Feb],
            [TEXT,DOUBLE,DOUBLE,DOUBLE],
            SELECT Expenditure category,RelImpDec-
99,UadjPctChg3-98to3-99,SeasAdjPctJan-Feb FROM cpi WHERE %t
            ,
                _F),
    sql_open_statement(_E,_F,_G),
    sql_execute_statement(_G), !,
    get_a_row(_G,[_A,_B,_C,_D]).

?- listing(insert_my_cpi/_).

% user:insert_my_cpi/4
insert_my_cpi(_A,_B,_C,_D) :-
    sql_open_connection(Economics,,,_E),
    sprintf(atom(_F),
        INSERT INTO cpi(Expenditure category,RelImpDec-
99,UadjPctChg3-98to3-99,SeasAdjPctJan-Feb) VALUES ('%t',%t,
%t, %t),
        [_A,_B,_C,_D]),
    sql_open_statement(_E,_F,_G), !,
    sql_execute_statement(_G).
```

```

?- listing(simple_my_cpi/_).

% user:simple_my_cpi/4
simple_my_cpi(_A,_B,_C,_D) :-
    sql_open_connection(Economics,,,_E),
    sql_open_statement(_E,
        SELECT ALL Expenditure category,RelImpDec-
        99,UadjPctChg3-98to3-99,SeasAdjPctJan-Feb FROM cpi,
        _F),
    sql_execute_statement(_F), !,
    get_a_row(_F,[_A,_B,_C,_D]).

```

Note that it is important that in the Spec, the table and column names are type exactly as they appear in the datasource; no conversion or normalization is performed by defSQLview.

Finally, calls to defSQLview can appear in files which are consulted. There are two provisions you must make in order to do this. First, meta_odbc must be loaded before the file is consulted, and the initialization goal sql_init must have been run. Secondly, if your call to defSQLview occurs within any module other than module user, you must guarantee that the module “uses” the module odbc. Here is an example:

```

module zap.
use odbc.
:- defSQLview([
    ds = 'Economics',
    table = 'cpi',
    cols_list = [
        'Expenditure category','RelImpDec-99',
        'UadjPctChg3-98to3-99','SeasAdjPctJan-Feb'],
    pred=my_cpi
    ] ).

endmod.

```

Assuming that meta_odbc has been loaded and the sql_init has been run,

consulting the file containing the code above will have the following effect:

```
?- [odbc_samples].
?- Attempting to consult D:\databases\odbc\odbc_samples.pro...
... consulted D:\databases\odbc\odbc_samples.pro
?- listing(zap:_/_).

% zap:my_cpi/4
my_cpi(_A,_B,_C,_D) :-
    sql_open_connection(Economics,,,_E),
    odbc:
        where_conds([_A,_B,_C,_D],
            [Expenditure category,RelImpDec-99,
             UadjPctChg3-98to3-99,
             SeasAdjPctJan-Feb],
            [TEXT,DOUBLE,DOUBLE,DOUBLE],
            SELECT Expenditure category,RelImpDec-
99,UadjPctChg3-98to3-99,SeasAdjPctJan-Feb FROM cpi WHERE %t
        ,
            _F),
    sql_open_statement(_E,_F,_G),
    sql_execute_statement(_G), !,
    get_a_row(_G,[_A,_B,_C,_D]).

% zap:simple_my_cpi/4
simple_my_cpi(_A,_B,_C,_D) :-
    sql_open_connection(Economics,,,_E),
    sql_open_statement(_E,
        SELECT ALL Expenditure category,RelImpDec-
99,UadjPctChg3-98to3-99,SeasAdjPctJan-Feb FROM cpi,
        _F),
    sql_execute_statement(_F), !,
    get_a_row(_F,[_A,_B,_C,_D]).

% zap:insert_my_cpi/4
```

```
insert_my_cpi(_A,_B,_C,_D) :-  
    sql_open_connection(Economics,,,_E),  
    sprintf(atom(_F),  
        INSERT INTO cpi(Expenditure category,RelImpDec-  
99,UadjPctChg3-98to3-99,SeasAdjPctJan-Feb) VALUES ('%t',%t,  
%t, %t),  
        [_A,_B,_C,_D]),  
    sql_open_statement(_E,_F,_G), !,  
    sql_execute_statement(_G).
```