

# DSN Parsing Package

Georg Bauhaus\*

Date: Mon, 31 Jan 2005 20:40

## 1 Data Source Name Parts

The *DSN* package and children provide a parser that selects components from a Data Source Name (DSN). A data source name is a string value that states how to connect to a database. A DSN is made from a few parts. In its standard form, the first part of a DSN specifies a database system (brand), the second part names a user, the third part specifies a server, and the fourth part is a database name.

The packages assume DSN syntax conventions similar to the conventions used for PEAR::DB, a PHP database abstraction layer. Details about acceptable data source names are spelled out in the grammar, see section 1.1. For example, some parts of a DSN can be left out, some parts can be optionally added.

The parser can handle some syntax errors in DSNs. It first follows the grammar rules, including error rules if no derivation can be found. Then it performs a few additional checks. The checks help detect spurious input, to some extent. [TODO: Write the predicates, easy enough, but not there yet. Try for example, `db2 : //usr : 4!@tcp@user`, which is clearly wrong. The parser does not accept this string and says, “No”. But with the checks, it could report two @s. Likewise, the presence of an @ indicates where the server part can be at all.]

The rest of this section is about some pitfalls of DSN construction and can be skipped.

(When seen from the perspective of a parser, a DSN might be fine. But seen from the perspective of the application that uses the parser’s results a syntactically valid DSN is not necessarily sufficient. Its parts might lead to quite unwanted behavior if used “blindly”. On the other hand a DSN can have syntax errors that stop the application when the syntax errors are no more than typos. These properties of DSNs are relevant when DSNs are input to a program at runtime.

The typical construction of environment variables in shell scripts provides an example of a DSN that is syntactically correct, and plausible. Yet the person who has written the DSN string might not have wanted the result. A shell script can combine a DSN from several variables. Consider a variable in the position of the server name, as in

```
FRUIT_DB=mysql://$SERVER/fruits.
```

Let `$SERVER` have the value `frank@home`, which is not a server part string according to the DSN grammar, but a user part followed by ‘@’ and a server part. The result of shell variable expansion is

---

\*send eMail to [bauhaus@futureapps.de](mailto:bauhaus@futureapps.de). This is for ProjectVersion: 0b.3

```
FRUIT_DB=mysql://frank@home/fruits.
```

This is a correct and plausible DSN. It might be harmless to use this DSN, even if the intention was not to connect the program to the `fruits` database running at a server home as user `frank`. Or it might not be harmless at all. In general, when parsing succeeds this is not yet an indication of a valid DSN. There are false positives. (You can check the pieces found by the parser using the functions from the *DSN* package described in a later section.)

As an example of the difficulty of finding spurious input, assume that in some DSN a server part shell variable is given. A query string-like shell variable has been appended, for passing additional parameters, like so:

```
LOCAL_DEFAULT_DB=pgsql://$SERVER$QS.
```

If the `'?` starting the query string part has been accidentally omitted when assigning to `$QS`, it is not easy for the DSN parser to know what to do with the text resulting from shell variable expansion. It cannot know whether an `'@'` appearing in the query string part is in fact from the query string part stored in `$QS`. Rather, it will interpret the `'@'` as introducing a server part. (Whether or not `'@'` is a valid character in a query string is subject to similar considerations. Someone might just have placed it in `$QS` by accident, or in an attempt to fool the program, or as part of a clever solution.)

On the other hand, if `'?` is present the parser can decide that an `'@'` following the `'?` is probably wrong. Likewise, a missing slash, `'/'`, can lead to some speculation about the possible intent of the rest of the text.

The parser doesn't fully solve the problem of finding spurious input (of course?). For the reasons outlined, it is recommended that programmers use a suitable set of predicates to further check the pieces of input provided by the parser. Still the parser tries to be helpful and reports when an input could be correct provided some symbol were added. [Actually, inserting missing symbols is a TODO item.)]

## 1.1 The Grammar of DSNs

The grammar has *DSN* as the start symbol. There are two rules with *DSN* on the left, the first produces strings in the traditional DSN form, the second produces strings in variant syntax. (Rules dealing with errors have been left out here, they can be found in the spec of package *DSN*.) The  $\perp$  sign stands for the end of input.

DSN	→	DB Traditional
DSN	→	DBSpeak Variant
DB	→	“access”   “dbase”   “db2”   “ifx”   “msql”   “mssql”
DB	→	“mysql”   “navision”
DB	→	“oci”   “pgsql”   “solid”   “sqlite”   “sybase”
Traditional	→	$\perp$
Traditional	→	‘:’ ‘/’ ‘/’ DBTail
Traditional	→	‘:’ ‘/’ ‘/’ Login QS
Traditional	→	‘:’ ‘/’ ‘/’ Login DBTail
Login	→	Server
Login	→	User ‘@’ Server
User	→	NAME
User	→	NAME ‘:’ PASSWORD
Server	→	Protocol ‘+’ HostPort
Server	→	HostPort

Figure 1: Basic Model of Use.

```
begin
    result := Parsers.parse(some_dsn_string);
exception
    when Parsers.syntax_error =>
        ... handle
end;
```

HostPort	→	Host
HostPort	→	Host ':' Port
Host	→	IP   HOSTNAME
DBTail	→	'/' Database QS
Database	→	NOTQMARKS
Port	→	DECIMAL_DIGITS
QS	→	'?' PARAMETERS
QS	→	⊥
DBSpeak	→	"odbc" '(' DB ')'
Variant	→	⊥
Variant	→	':' '/' '/' ".....TODO....."

Symbols in all upper case that don't appear in the grammar have their usual meaning. (The comments in the spec of package *DSN* give a more precise description.)

## 1.2 Using the Parser

The file *INSTALL.txt* describes how to build the library.

When you are ready to use the library, write  
with *DSN.Messages*, *DSN.Parsers*;  
at the start of your compilation unit.

The function *parse* from package *DSN.Parsers* reads a string and parses it assuming DSN syntax. It produces an object of type *Data\_Source\_Descriptor* containing the parse results. Figure 1 shows a summary of use. When *parse* succeeds, the programmer can investigate the result using any of the functions described in the next section.

If *parse* fails, the exception *syntax\_error* is raised. You can further investigate the situation if you like. The *DSN* package provides the generic function *diagnose* that finds errors in a DSN. Like *parse*, it will analyse the input string. Unlike *parse* it does not raise an exception. Instead, when the *diagnose* instance cannot parse the string at some point, every failure is reported through the generic formal procedure *report*. This allows a library user to keep track of what has happened during parsing. See the demo program for an example.

(It is still possible to use the results of *diagnose* when it can find a derivation. *diagnose* then sends either the message code *Spotless*, or the message code *Syntax\_Error*. In the first case the contents of the the variable passed for the *scratch* parameter should be fine—until the *report* procedure is done. In the second case the contents of the variable passed for *scratch* can be analysed inside the *report* procedure. However, when the *report* callback is done the parser continues, performing backtracking. This will likely overwrite fields or clear fields in its *Data\_Source\_Descriptor*. Therefore, the variable passed for *scratch* will subsequently be in a state not consistent with parser message and parser findings.)

### 1.2.1 The Interface of *Data\_Source\_Descriptor*

*d* stands for an object of type *Data\_Source\_Descriptor*. *d* has been declared in the client program, and has been assigned the value of the *parse* function. For example,

```
d: constant Data_Source_Descriptor := parse(some_dsn_string);
```

When no exception has been raised, the following functions deliver all parts of the input DSN string.

*database\_system\_name(d)*

Returns a database system name, like "pgsql". The grammar rules for *DB* list the known values. There is an alternative function of the same name that returns a value of enumeration type *Database*. This type consists of one value for each known database system.

*dialect(d)*

an empty string or, if variant syntax is used, the substring following the database system name in parenthesis

*user\_name(d)*

a possibly empty string

*password(d)*

a possibly empty string

*connection\_protocol(d)*

A possibly empty string. There is an alternative function of the same name that returns an enumeration value of type *Protocol* (or the *Unknown* protocol). This type consists of one value for each known protocol.

*host\_name(d)*

if not an empty string, a HOSTNAME string, or an IP number (as a string) as explained in the grammar.

*port(d)*

The port number specified in the DSN or else 0 (assuming a database system doesn't listen on port 0.) (The value returned can be greater than the highest port number available on some system, the parser doesn't check.)

*database\_name(d)*

Usually this is a file system path or a simple name. The string can be empty.

*additional\_parameters(d)*

The part after the first question mark, if any. This substring is usually in the form of a query string as used in passing CGI parameters using URLs. Currently the parser just returns this part of the DSN. It is not parsed as a query string.

Figure 2: A valid DSN fed to the demo program.

```
DSNT> mysql://joe@somehost:3306/parts
Yes.
- system name: mysql
- user: joe
- password:
- protocol:
- server: somehost
- port: 3306
- database: parts
- parameters:
DSNT>
```

Figure 3: A DSN fed to the demo program, this one has a mistake.

```
DSNT> mysql://tco+localhost/parts
9: not a known protocol, skipping "tco"
1: input is not valid, but could be corrected
Maybe.
DSNT>
```

### 1.2.2 The dsnt Demo Program

`dsnt` is a simple interactive application that reads DSN strings from current input and prints any findings of the parser on current output. (There is a short introduction in the units “`dsnt.adb`”, and “`dsnt-io.adb`”.) The program text can be used as an example of how to construct an application using the parser provided by the DSN packages.

A sample run is presented in Figure 2. The user has entered a DSN string, the program prints the parser’s findings. If the input DSN isn’t correct, the parser will report errors it can find. This is displayed in Figure 3.

These figures correspond to the two main ways of using the parser subprograms in your programs. The first display basically corresponds to calling *parse* on the user’s input, and then calling the functions for querying the returned *Data\_Source\_Descriptor* (the lines starting with a ‘-’). The second display shows what might be written to a log file, using an instance of the *diagnose* procedure (the lines starting with a number and colon).

### 1.3 Configuration Parameters

The package *DSN.Config* declares some fixed capacity constraints. They should affect only the parser as they have an effect on the maximum size of some parser buffers. [TODO: Say what is obsolescent (string indices?), fix the details and explain.]

The variable *verbose* in *DSN.Config* can be set to *True*. This will result in trace output on *current\_error* showing each completed derivation. A default printing procedure from package *DSN.Support* is used for this. You can set this procedure to your own by setting a pointer variable in the package.