

# DSN Parsing Package

Georg Bauhaus\*

Date: Wed, 15 Jun 2005 13:20<sup>†</sup>

## 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, much like a URL, 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 names a server, and the fourth part is a database name.

The packages assume DSN syntax conventions similar to the conventions used in 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. If no derivation can be found, error rules will be employed. Then it performs a few additional checks. The checks help detect spurious input, to some extent.

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 no necessarily fine. If used "blindly", DSN parts might lead to quite unwanted behavior when setting up a database connection. 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

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

---

\*the author can be reached via eMail, bauhaus@futureapps.de

<sup>†</sup>ProjectVersion: 0b.4

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 parameters part has been accidentally omitted when assigning to `$QS`, it is not easy for the DSN parser to know what to do with the text resulting from shell variable expansion. It cannot know whether an `@` appearing in the parameters part is in fact from the parameters part stored in `$QS`. Rather, it will interpret the `@` as introducing a server part. (Whether or not `@` is a valid character in additional parameters 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 `?` might have been misplaced and issue a warning. 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, as always, 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 of a more traditional DSN form (string start with a URL-like prefix). The second *DSN* rule produces strings in alternative (“new”) syntax which allows specifying both a database access method (like ODBC) and a brand (like DB2). (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 ConSpec
DSN	→	AccessMethod '(' DB ')' ConSpec
DB	→	“access”   “dbase”   “db2”   “ifx”   “mimer”   “monetdb”
DB	→	“msql”   “mssql”   “mysql”   “navision”
DB	→	“oci”   “pgsql”   “solid”   “sqlite”   “sybase”
AccessMethod	→	“adbc”   “jdbc”   “odbc”
ConSpec	→	$\perp$
ConSpec	→	':' '/' '/' DBTail
ConSpec	→	':' '/' '/' Login QS
ConSpec	→	':' '/' '/' Login DBTail
Login	→	Server
Login	→	User '@' Server
Login	→	User '@'
User	→	NAME

Figure 1: Basic Model of Use.

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

User	→	NAME ':' PASSWORD
Server	→	HostPort
Server	→	Protocol '+' HostPort
Server	→	Protocol '(' HostPort ')'
Server	→	Protocol '(' BAL_NOTQMARKS_NOTATSIGNS ')'
HostPort	→	Host
HostPort	→	Host ':' Port
Host	→	IP   HOSTNAME
Protocol	→	"tcp"   "unix"
DBTail	→	'/' Database QS
Database	→	NOTQMARKS
Port	→	DECIMAL_DIGITS
QS	→	'?' PARAMETERS
QS	→	⊥

Symbols in all upper case that don't appear on the left side of a rule have their usual meaning. (The comments in the spec of package *DSN* give a more precise description.) Character case is ignored when matching literals denoting DB and Protocol.

## 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

way programs can keep track of what has happened during parsing. See the `dsnt` demo program for an example.

Notice that the same error may be reported more than once, due to exhaustive parsing.

(It is still possible to use the results of *diagnose* when it can find a derivation possibly choosing error rules. *diagnose* then sends either the message code *Spotless*, or the message code *Syntax\_Error*. In the first case the contents of 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 messages and parser findings.)

### 1.2.1 Internationalization

A *diagnose* instance can print messages in any language enumerated in type *Language*, which is declared in *DSN.Messages*. By default, English is used.

Typically, choosing a language requires passing the same *Language* value when instantiating two subprograms: 1. The *diagnose* template, which has a generic formal parameter *lang*. (A *diagnose* instance constructs message text during execution by combining phrase constants from the *DSN.Messages* package. The message text will be in language *lang*.) 2. Your *report* procedure can instantiate *diagnosis.in.words* for fetching localized messages from the *DSN.Messages* package. (In *DSN.Messages*, each *report.code* is associated with a message text, one for each supported language and code.) See the `dsnt` demo program for an example (*IO.eval* subprogram).

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

Let *d* stand for an object of type *Data\_Source\_Descriptor*. *d* has been declared in the library client program, and has been assigned the result value of a *parse* function call. 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. Note that some parts might be empty. Some functions may return a default value in case the corresponding part of the input string is empty.

***database\_system\_name(d)*** 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 includes one value for each known database system.

***database\_access\_method(d)*** an empty string or, if corresponding syntax is used, the substring before the database system name in parenthesis. There is an alternative function of the same name that returns a value of enumeration type *Access\_Method*. This type includes one value for each known access method. For a

list of known values, see the corresponding grammar rule and the type in the *DSN* spec.

*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 includes one value for each known protocol.

*connection\_via(d)* A possibly empty string. This is the part in parenthesis after the connection protocol, provided this part does not just name a host and an optional port. When this part does name only a host, possibly followed by a port, then these parts will be available through calls of the query functions for host and port.

*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 in URLs. Currently the parser just returns this part of the DSN. It is not parsed as a query string.

### 1.2.3 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 both as an example of how to construct an application using the parser provided by the DSN packages, and as a utility program with which to test your data source names before you feed them to production programs.

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 any errors that it has found. This is displayed in Figure 3.

Figure 2 and Figure 3 correspond to the two main ways of using the parser sub-programs in your programs. The first display corresponds to calling *parse* with some DSN as input, and then calling the functions for retrieving the parts of the resulting *Data\_Source\_Descriptor* (results shown on 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).

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

```
DSNT> mysql://joe@somehost:3306/parts
Yes.
- access method:
-   system name: mysql
-     user: joe
-   password:
-   protocol:
-     via:
-   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>
```

### 1.3 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. These limits might be removed in a future version.

The variable *verbose* in *DSN.Config* can be set to *True*. By default, 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. If you want to supply your own tracing procedure, you can do so by having *DSN.Support.trace* point to your procedure.