Using Data Tables in the GNU MathProg Modeling Language

Andrew Makhorin <mao@gnu.org>
Heinrich Schuchardt <heinrich.schuchardt@gmx.de>

November, 2009

Abstract

This is a supplement to the document "Modeling Language GNU MathProg". It describes the table statement, which allows reading data from external tables into model objects such as sets and parameters as well as writing data from the model to external tables.

Table statement

```
table name alias IN driver arg ... arg :
      set <- [ fld , ... , fld ] , par ~ fld , ... , par ~ fld ;

table name alias domain OUT driver arg ... arg :
      expr ~ fld , ... , expr ~ fld ;
```

Where: *name* is a symbolic name of the table;
alias is an optional string literal which specifies the alias of the table;
domain is an indexing expression which specifies the subscript domain of the (output) table;
`IN` is the keyword which means reading data from the input table;
`OUT` is the keyword which means writing data to the output table;
driver is a symbolic expression which specifies the name of the driver used to access the table. (For details see Section "Table drivers" below.)

arg is an optional symbolic expression which is an argument passed to the table driver. This symbolic expression must not contain dummy indices specified in the domain;
set is the name of an optional simple set called control set. It can be omitted along with the delimiter '`<-`';
fld is the field name. Within square brackets at least one field should be specified. The field name following parameter name or expression is optional and can be omitted along with the delimiter '`~`', in which case the name of corresponding model object is used as the field name;
par is the symbolic name of a model parameter;
expr is a numeric or symbolic expression.

Examples

```
table data IN "CSV" "data.csv":
      s <- [FROM,TO], d~DISTANCE, c~COST;
table result{(f,t) in s} OUT "CSV" "result.csv":
      f~FROM, t~TO, x[f,t]~FLOW;
```

The table statement allows reading data from a table into model objects such as sets and (non-scalar) parameters as well as writing data from the model to a table.

Table structure

The *data table* is an (unordered) set of *records*, where each record consists of the same number of *fields*, and each field is provided with a unique symbolic name called the *field name*. For example:

	First field ↓	Second field ↓	. . .	Last field ↓
Table header →	FROM	TO	DISTANCE	COST
First record →	Seattle	New-York	2.5	0.12
Second record →	Seattle	Chicago	1.7	0.08
	Seattle	Topeka	1.8	0.09
. . .	San-Diego	New-York	2.5	0.15
	San-Diego	Chicago	1.8	0.10
Last record →	San-Diego	Topeka	1.4	0.07

Reading data from input table

The input table statement causes reading data from the specified table record by record.

Once a next record has been read, numeric or symbolic values of fields, whose names are enclosed in square brackets in the table statement, are gathered into n-tuple, and if the control set is specified in the table statement, this n-tuple is added to it. Besides, a numeric or symbolic value of each field associated with a model parameter is assigned to the parameter member identified by subscripts, which are components of the n-tuple just read.

For example, the following input table statement:

```
table data IN "...": s <- [FROM,TO], d~DISTANCE, c~COST;
```

causes reading values of four fields named `FROM`, `TO`, `DISTANCE`, and `COST` from each record of the specified table. Values of fields `FROM` and `TO` give a pair (f, t), which is added to the control set `s`. The value of field `DISTANCE` is assigned to parameter member `d`$[f, t]$, and the value of field `COST` is assigned to parameter member `c`$[f, t]$.

Note that the input table may contain extra fields whose names are not specified in the table statement, in which case values of these fields on reading the table are ignored.

Writing data to output table

The output table statement causes writing data to the specified table. Note that some drivers (namely, CSV and xBASE) destroy the output table before writing data, i.e. delete all its existing records.

Each n-tuple in the specified domain set generates one record written to the output table. Values of fields are numeric or symbolic values of corresponding expressions specified in the table statement. These expressions are evaluated for each n-tuple in the domain set and, thus, may include dummy indices introduced in the corresponding indexing expression.

For example, the following output table statement:

```
table result{(f,t) in s} OUT "...": f~FROM, t~TO, x[f,t]~FLOW;
```

causes writing records, by one record for each pair (f, t) in set `s`, to the output table, where each record consists of three fields named `FROM`, `TO`, and `FLOW`. The values written to fields `FROM` and `TO` are current values of dummy indices `f` and `t`, and the value written to field `FLOW` is a value of member `x`$[f, t]$ of corresponding subscripted parameter or variable.

Table drivers

The *table driver* is a program module which provides transmitting data between MathProg model objects and data tables.

Currently the GLPK package has four table drivers:

- built-in CSV table driver;
- built-in xBASE table driver;
- ODBC table driver;
- MySQL table driver.

CSV table driver

The CSV table driver assumes that the data table is represented in the form of a plain text file in the CSV (comma-separated values) file format as described below.

To choose the CSV table driver its name in the table statement should be specified as `"CSV"`, and the only argument should specify the name of a plain text file containing the table. For example:

```
table data IN "CSV" "data.csv": ... ;
```

The filename suffix may be arbitrary, however, it is recommended to use the suffix '`.csv`'.

On reading input tables the CSV table driver provides an implicit field named `RECNO`, which contains the current record number. This field can be specified in the input table statement as if there were the actual field having the name `RECNO` in the CSV file. For example:

```
table list IN "CSV" "list.csv": num <- [RECNO], ... ;
```

CSV format[1]

The CSV (comma-separated values) format is a plain text file format defined as follows.

1. Each record is located on a separate line, delimited by a line break. For example:

```
aaa,bbb,ccc\n
xxx,yyy,zzz\n
```

where '`\n`' means the control character LF (0x0A).

[1] This material is based on the RFC document 4180.

2. The last record in the file may or may not have an ending line break. For example:

```
aaa,bbb,ccc\n
xxx,yyy,zzz
```

3. There should be a header line appearing as the first line of the file in the same format as normal record lines. This header should contain names corresponding to the fields in the file. The number of field names in the header line should be the same as the number of fields in the records of the file. For example:

```
name1,name2,name3\n
aaa,bbb,ccc\n
xxx,yyy,zzz\n
```

4. Within the header and each record there may be one or more fields separated by commas. Each line should contain the same number of fields throughout the file. Spaces are considered as part of a field and therefore not ignored. The last field in the record should not be followed by a comma. For example:

```
aaa,bbb,ccc\n
```

5. Fields may or may not be enclosed in double quotes. For example:

```
"aaa","bbb","ccc"\n
zzz,yyy,xxx\n
```

6. If a field is enclosed in double quotes, each double quote which is part of the field should be coded twice. For example:

```
"aaa","b""bb","ccc"\n
```

The following is a complete example of the data table in CSV format:

```
FROM,TO,DISTANCE,COST
Seattle,New-York,2.5,0.12
Seattle,Chicago,1.7,0.08
Seattle,Topeka,1.8,0.09
San-Diego,New-York,2.5,0.15
San-Diego,Chicago,1.8,0.10
San-Diego,Topeka,1.4,0.07
```

xBASE table driver

The xBASE table driver assumes that the data table is stored in the .dbf file format.

To choose the xBASE table driver its name in the table statement should be specified as `"xBASE"`, and the first argument should specify the name of a .dbf file containing the table. For the output table there should be the second argument defining the table format in the form `"FF...F"`, where `F` is either `C(`n`)`, which specifies a character field of length n, or `N(`n`[,`p`])`, which specifies a numeric field of length n and precision p (by default p is 0).

The following is a simple example which illustrates creating and reading a .dbf file:

```
table tab1{i in 1..10} OUT "xBASE" "foo.dbf"
   "N(5)N(10,4)C(1)C(10)": 2*i+1 ~ B, Uniform(-20,+20) ~ A,
   "?" ~ FOO, "[" & i & "]" ~ C;
set S, dimen 4;
table tab2 IN "xBASE" "foo.dbf": S <- [B, C, RECNO, A];
display S;
end;
```

ODBC table driver

The ODBC table driver allows connecting to SQL databases using an implementation of the ODBC interface based on the Call Level Interface (CLI).[2]

Debian GNU/Linux. Under Debian GNU/Linux the ODBC table driver uses the iODBC package,[3] which should be installed before building the GLPK package. The installation can be effected with the following command:

```
sudo apt-get install libiodbc2-dev
```

Note that on configuring the GLPK package to enable using the iODBC library the option '`--enable-odbc`' should be passed to the configure script.

The individual databases must be entered for systemwide usage in `/etc/odbc.ini` and `/etc/odbcinst.ini`. Database connections to be used by a single user are specified by files in the home directory (`.odbc.ini` and `.odbcinst.ini`).

[2]The corresponding software standard is defined in ISO/IEC 9075-3:2003.

[3]See `<http://www.iodbc.org/>`.

Microsoft Windows. Under Microsoft Windows the ODBC table driver uses the Microsoft ODBC library. To enable this feature the symbol:

```
#define ODBC_DLNAME "odbc32.dll"
```

should be defined in the GLPK configuration file '`config.h`'.

Data sources can be created via the Administrative Tools from the Control Panel.

To choose the ODBC table driver its name in the table statement should be specified as `'ODBC'` or `'iODBC'`.

The argument list is specified as follows.

The first argument is the connection string passed to the ODBC library, for example:

`'DSN=glpk;UID=user;PWD=password'`, or

`'DRIVER=MySQL;DATABASE=glpkdb;UID=user;PWD=password'`.

Different parts of the string are separated by semicolons. Each part consists of a pair *fieldname* and *value* separated by the equal sign. Allowable fieldnames depend on the ODBC library. Typically the following fieldnames are allowed:

`DATABASE`	database;
`DRIVER`	ODBC driver;
`DSN`	name of a data source;
`FILEDSN`	name of a file data source;
`PWD`	user password;
`SERVER`	database;
`UID`	user name.

The second argument and all following are considered to be SQL statements

SQL statements may be spread over multiple arguments. If the last character of an argument is a semicolon this indicates the end of a SQL statement.

The arguments of a SQL statement are concatenated separated by space. The eventual trailing semicolon will be removed.

All but the last SQL statement will be executed directly.

For IN-table the last SQL statement can be a SELECT command starting with the capitalized letters `'SELECT '`. If the string does not start with `'SELECT '` it is considered to be a table name and a SELECT statement is automatically generated.

For OUT-table the last SQL statement can contain one or multiple question marks. If it contains a question mark it is considered a template for the write routine. Otherwise the string is considered a table name and an INSERT template is automatically generated.

The writing routine uses the template with the question marks and replaces the first question mark by the first output parameter, the second question mark by the second output parameter and so forth. Then the SQL command is issued.

The following is an example of the output table statement:

```
table ta { l in LOCATIONS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'INSERT INTO result 'VALUES ( 4, ?, ? )' :
   l ~ LOC, quantity[l] ~ QUAN;
```

Alternatively it could be written as follows:

```
table ta { l in LOCATIONS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'result' :
   l ~ LOC, quantity[l] ~ QUAN, 4 ~ ID;
```

Using templates with '?' supports not only INSERT, but also UPDATE, DELETE, etc. For example:

```
table ta { l in LOCATIONS } OUT
   'ODBC'
   'DSN=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'UPDATE result SET DATE = ' & date & ' WHERE ID = 4;'
   'UPDATE result SET QUAN = ? WHERE LOC = ? AND ID = 4' :
   quantity[l], l;
```

MySQL table driver

The MySQL table driver allows connecting to MySQL databases.

Debian GNU/Linux. Under Debian GNU/Linux the MySQL table driver uses the MySQL package,[4] which should be installed before build-

[4]For download development files see <http://dev.mysql.com/downloads/mysql/>.

ing the GLPK package. The installation can be effected with the following command:

```
sudo apt-get install libmysqlclient15-dev
```

Note that on configuring the GLPK package to enable using the MySQL library the option '`--enable-mysql`' should be passed to the configure script.

Microsoft Windows. Under Microsoft Windows the MySQL table driver also uses the MySQL library. To enable this feature the symbol:

```
#define MYSQL_DLNAME "libmysql.dll"
```

should be defined in the GLPK configuration file '`config.h`'.

To choose the MySQL table driver its name in the table statement should be specified as `'MySQL'`.

The argument list is specified as follows.

The first argument specifies how to connect the data base in the DSN style, for example:

`'Database=glpk;UID=glpk;PWD=gnu'`.

Different parts of the string are separated by semicolons. Each part consists of a pair *fieldname* and *value* separated by the equal sign. The following fieldnames are allowed:

`Server`	server running the database (defaulting to localhost);
`Database`	name of the database;
`UID`	user name;
`PWD`	user password;
`Port`	port used by the server (defaulting to 3306).

The second argument and all following are considered to be SQL statements

SQL statements may be spread over multiple arguments. If the last character of an argument is a semicolon this indicates the end of a SQL statement.

The arguments of a SQL statement are concatenated separated by space. The eventual trailing semicolon will be removed.

All but the last SQL statement will be executed directly.

For IN-table the last SQL statement can be a SELECT command starting with the capitalized letters `'SELECT '`. If the string does not start with `'SELECT '` it is considered to be a table name and a SELECT statement is automatically generated.

For OUT-table the last SQL statement can contain one or multiple question marks. If it contains a question mark it is considered a template for the write routine. Otherwise the string is considered a table name and an INSERT template is automatically generated.

The writing routine uses the template with the question marks and replaces the first question mark by the first output parameter, the second question mark by the second output parameter and so forth. Then the SQL command is issued.

The following is an example of the output table statement:

```
table ta { l in LOCATIONS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'INSERT INTO result VALUES ( 4, ?, ? )' :
   l ~ LOC, quantity[l] ~ QUAN;
```

Alternatively it could be written as follows:

```
table ta { l in LOCATIONS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'DROP TABLE IF EXISTS result;'
   'CREATE TABLE result ( ID INT, LOC VARCHAR(255), QUAN DOUBLE );'
   'result' :
   l ~ LOC, quantity[l] ~ QUAN, 4 ~ ID;
```

Using templates with '?' supports not only INSERT, but also UPDATE, DELETE, etc. For example:

```
table ta { l in LOCATIONS } OUT
   'MySQL'
   'Database=glpkdb;UID=glpkuser;PWD=glpkpassword'
   'UPDATE result SET DATE = ' & date & ' WHERE ID = 4;'
   'UPDATE result SET QUAN = ? WHERE LOC = ? AND ID = 4' :
   quantity[l], l;
```