

Processing data from CSV and JSON files with OpTeX

Petr Olšák

Introduction

When we want to do something with CSV or JSON files in L^AT_EX, we can load an appropriate package. In ConT_EXt, we can use relevant commands directly. What can we do in OpT_EX?

OpT_EX [1] implements this feature by a set of OpT_EX tricks [2]. They are grouped in the section “Databases”. Short macro definitions are here. Users can copy them to their macro files and modify them to meet their specific requirements. Moreover, many OpT_EX tricks are set as *autoloads*. This means that if you don’t need to modify the presented macro code, then you can simply use the user-level macro (`\csvread`, `\dbloop`, `\jsonread` for example) and the corresponding bundle of macro code is loaded automatically at its first use. You need to do nothing more than use this command. No packages need to be declared in the preamble of the document.

This article shows the options for using CSV and JSON files with OpT_EX, i. e. by the relevant OpT_EX tricks.

Processing CSV files

CSV files are simple databases. They collect data in rows and columns of a table. The columns have their names declared in the first row. The data items in each row are optionally surrounded by special characters (typically ") and they are separated by another character, traditionally comma but semicolon, tab, etc., are also used. A CSV file might look like this:

```
id, "FirstName", "LastName", "Address"
013, "Petr", "Olšák", "Prague"
142, "Ferdinand", "Mravenec", "U lesa"
```

We can read a CSV file by the command

```
\csvread {<filename>.csv}
```

Once this is done, the macro

```
\dbitem(<row>, <column>)
```

expands to the corresponding data from the file `<filename>.csv`. The macro `\colname <name>` can be used instead of the numeric `<row>`. The number of rows and columns are saved in `\dbmaxrow` and `\dbmaxcol` macros. The column name can be retrieved with the macro `\dbhead(<column>)`.

Suppose that the CSV file shown above is saved in the `names.csv` file. Then we can do:

```
\csvread {names.csv}
% reads CSV file and saves data to memory
\dbitem(1,2)
% returns an item from 1-st row, 2-nd column,
% i. e. Petr
\dbitem(1, \colname Address) % returns Prague
\dbhead(3) % returns LastName
\edef\foo{\dbitem(2,1)} % does \def\foo{142}
\edef\address{\dbitem(2, \colname Address)}
% does \def\address{U lesa}

\fornum 1.. \dbmaxrow \do
{\dbitem(#1, \colname LastName) }
% prints all data in the column Lastname

\fornum 1.. \dbmaxrow \do
{\def\e##1{\dbitem(#1, \colname##1)}
Here is \e{LastName}, \e{Address}, ... \par}
% processes the whole database row by row
```

If a CSV file uses a separator other than comma, or a surrounding character other than ", you can declare `\csvdecl<o><c><s>` before `\csvread`. The opening character is `<o>`, the closing character is `<c>` and the separator is `<s>`. By default `\csvdecl"`, is assumed.

You can read more CSV files and work with more databases in a single document when optional database name `<dbname>` is used. The syntax is:

```
\csvread <dbname>{<filename>.csv}
\dbitem <dbname>(<row>, <column>)
\dbhead <dbname>(<column>)
\dbmaxrow<dbname>, \dbmaxcol<dbname>
```

The `\dbmaxrow<dbname>`, `\dbmaxcol<dbname>` are single control sequences. If we did, for example, `\csvread xx{file.csv}` then the `\dbmaxrowxx` and `\dbmaxcolxx` macros are defined. They include the number of rows and columns in the `xx` database.

If we want to loop over database rows more comfortably than via `\fornum` (shown above), we can use the `\dbloop` macro with syntax

```
\dbloop <dbname>(<assign list>) {<text>}
```

where `<dbname>` is the database name, `<assign list>` is a comma-separated list where local macros for selected columns are declared, and `<text>` will be repeated for each row. For example:

```
\csvread {names.csv}
\dbloop (\fname=FirstName,
        \lname=LastName,
        \address=Address)
{%
  \lname\ \fname\ lives at
  the address: \address \par
}
```

Users must decide in what catcode regime the CSV file is to be read. It depends on the application. The data in the file can be strings that have nothing to do with T_EX. Then we can deactivate special T_EX characters used in data, for example:

```
{\catcode'\%=12 \catcode'\$=12 \catcode'\&=12
 \csvread {file.json}}
```

The `\csvread` can be used in the group because it defines all output macros globally. If you want to read the CSV file completely in verbatim mode, declare `\let\csvreadsetup=_setverb`.

On the other hand, the CSV file may include T_EX chunks, for example, math formulas. Then we can keep the catcodes unchanged during `\csvread`. The `\dbitem` macro expands (in `\edef` context) to the real data where possible macros are kept unexpanded.

Modifying and saving data in databases

Once a database is read by `\csvread`, we can make modifications of the data by

```
\dbinsert <dbname>({<data1>}{<data2>}{<data3>})
 % inserts a new row to the database <dbname>
\dbitemset <dbname>(2,3){<data>}
 % replaces item(2,3) with <data>
\dbheadset <dbname>(3){<head>}
 % replaces header of the 3. column by <head>
```

Also, you needn't read the database from CSV file with `\csvread`; you can create it from scratch with `\dbcreate <dbname>({<head1>}{<head2>}..{<head n>})` and then use `\dbinsert` repeatedly.

If you want to add a new column to an existing database, then the following code can be used. We suppose in this example that the database name is `xx`.

```
\xdef\dbmaxcolxx{\the\numexpr\dbmaxcolxx+1}
\dbheadset xx(\dbmaxcolxx){New-head}
```

The following macros show database data in various ways:

```
\dbshow <dbname>()
 % expands the whole database in format
 % {<data1>}{<data2>}{<data3>}
\dbshow <dbname>("",)
 % expands the whole database in format
 % "data1","data2","data3"
\dbshowrow <dbname>(3)("",)
 % expands row 3 only, in format
 % "data1","data2","data3"
\dbshowhead <dbname>("",)
 % expands headers in format
 % "head1","head2","head3"
\wterm{\dbshow <dbname>("",)}
 % prints database <dbname> on the terminal
```

And this macro writes database data to a CSV file:

```
\dbwrite <dbname>("",){<filename>.csv}
 % writes database <dbname> to <filename>.csv
```

Searching and sorting in databases

We want to search a row in the database (loaded by `\csvread`, for example) by a value in the given column. We can search it sequentially, but a better idea is to first prepare a hash table for the given column (especially if we want to do more searching in the same column). This can be done with `\dbhash(<column>)`. Then the row number with `<data>` in the given column is expanded by the macro `\dbwhere(<column>){<data>}`. Suppose our example above. Then:

```
\csvread {names.csv}
\dbhash (\colname LastName)
 % hash table for searching in LastName
 % column prepared
\dbitem (\dbwhere(\colname LastName){Olšák},
 \colname FirstName)
 % returns Petr
```

Databases with given names can be searched too:

```
\dbhash <dbname>(<column>)
 % prepares the hash table for <dbname>
\dbwhere <dbname>(<column>){<data>}
 % returns row where <data> in <column>
```

`\dbwhere` returns zero if a hash table for a given column isn't prepared by `\dbhash` or the `<data>` is not found.

You can prepare hash tables for more columns. For example

```
\for num 1..\dbmaxcol \do {\dbhash(#1)}
```

prepares hash tables for all columns in the database.

If there are more equal data items in the column used for searching then the row number with the first one is returned by `\dbwhere`. But `\dbhash` prepares a set of macros `\dbrow:<dbname>(<column>){<data>}` which include the comma-separated list of row numbers where `<data>` is in the given `<column>`s. We can use them together with the `\dbloop` macro (introduced in the first section) because of an additional feature of `\dbloop`: If the `\dblist:<dbname>` macro is defined as a comma-separated list of row numbers then `\dbloop` executes `<assign list>` and `<text>` for these rows in the given order. Otherwise, it processes all rows in the database from the first to the last row.

We show this feature in the next example, where we use our testing file `names.csv` with additional lines. Now we have not only Olšák Petr, but also Olšák Miroslav and Olšák Radek in the file:

```
id, "FirstName", "LastName", "Address"
013, "Petr", "Olšák", "Prague"
142, "Ferdinand", "Mravenec", "U lesa"
014, "Miroslav", "Olšák", "Cambridge"
015, "Radek", "Olšák", "Prague"
```

Then we can try:

```
\csvread {names.csv}
\dbhash (\colname LastName)
% The \dbrow:(3)<data> macros are created
\sxdef{dblist:}%
{\trycs{dbrow:(\colname LastName)Olšák}{}}
\dbloop (\fn=FirstName) {\fn,}
% expands to comma-separated list
% Petr,Miroslav,Radek,
```

OpTeX is able to run a sorting algorithm internally. Primarily, it is used for sorting index entries. But the database entries can be sorted with the same algorithm too. We can use:

```
\dbsort <dbname>(<column>)
```

This macro prepares a comma-separated list of row numbers saved in the macro `\dblist:<dbname>`. These numbers are ordered with respect to the ordered items in the given `<column>`. Thus we can use `\dbloop` just after `\dbsort` and the rows are executed in sorted order. Example:

```
\csvread {names.csv}
\dbsort (\colname LastName)
% data sorted alphabetically by LastName
\dbloop (\fn=FirstName,\ln=LastName,\ad=Address)
{\ln\ \fn\ lives at the address: \ad \par}
% prints data sorted by LastName
```

The macro `\dbsort` sorts all rows by default, but if `\db:sortlist` is defined as a comma-separated list of row numbers, it sorts only these rows. This means that we can do a search, then sort all matched rows, then print them:

```
\csvread {names.csv}
\dbhash (\colname LastName)
\sxdef{db:sortlist}%
{\trycs{dbrow:(\colname LastName)Olšák}{}}
% what we want to sort
\dbsort (\colname FirstName)
\dbloop (\fn=FirstName) {\fn,}
% expands to comma-separated list
% Miroslav,Petr,Radek,
```

The first names are printed in alphabetical order in this example. More options for working with simple databases can be found at the OpTeX tricks [2] web page.

JSON files and trees

A JSON file allows describing the tree data structure in human-readable form. To summarize the syntax,

each `{...}` is a node, its children are listed in it and they have identifiers; each `[...]` is also a node, but its children have no identifiers; each `"name":"value"` is a leaf with an identifier; and a `"value"` (without `"name":`) is a leaf without an identifier. For further examples in this article, we use this `bib.json` file:

```
{
  "title": "Big Book",
  "isbn": "123456789",
  "authors": [
    {
      "firstname": "John",
      "lastname": "Doe"
    },
    {
      "firstname": "Patrik",
      "lastname": "Black"
    }
  ]
}
```

We can read the data from a JSON file and save it to a `\macro` with

```
\jsonread [root-id]\macro {<filename>.json}
```

This command saves data to `\macro` in the tree-structure format declared in OpTeX trick 0153 [3]. All three data items are included in a single `\macro`. Depth-first processing over the tree can be done simply by executing `\macro`. For example, the data from the file `bib.json` is saved to `\macro` in the following structure:

```
\inode[root-id] {%
  \enode[title] {Big Book}%
  \enode[isbn] {123456789}%
  \inode[authors] {%
    \inode[] {%
      \enode[firstname] {John}%
      \enode[lastname] {Doe}}%
    \inode[] {%
      \enode[firstname] {Patrik}%
      \enode[lastname] {Black}}}%
}
```

The listing above was printed by the following code:

```
\jsonread[root-id]\macro{bib.json}
\wterm{\showtree\macro}
```

The `\showtree` macro used here prints the tree structure with correct indentation.

You can see that the tree-structure format is based on two sub-macros:

```
\inode[<id>] {<nodes>} % internal node
\enode[<id>] {<data>} % external node (leaf)
```

The `\inode` macro expands to

```
\startinode <nodes> \stopinode
```

and the `\enode` macro expands to

```
\execenode{<data>}
```

Thus a macro programmer can declare macros `\startinode`, `\stopinode` and `\execenode#1`.

OpTeX trick 0155 [4] shows how to read data for invoices from a JSON file and print the invoices including the product list, prices and calculation of the summary price. The data is read by executing `\macro` as above, where a `\startinode`, `\stopinode` and `\execenode` are appropriately declared.

All child nodes of a selected node can be saved to a macro; let's call it `\cmacro`. Then the command `\execedges\cmacro{<text>}` does the following for each node in `\cmacro`: a) executes its subtree; then b) executes `<text>`. It was used in the example with invoices. The node `items_sold` includes one or more nodes with products. They are executed by `\execedges`, i. e. data from their subtree is read and printed to the invoice table.

Next we show another approach. The tree-structure can be converted by a set of macros which include the data of external nodes. This can be done via `\treedata[<root-id>]\macro`. For example:

```
\jsonread\macro{bib.json} % read JSON to \macro
\tracingtreedata % information on the terminal
\treedata[bib]\macro % does the conversion
```

The following message is printed on the terminal:

```
\treedata:
  \sdef {bib/title}{Big Book}
  \sdef {bib/isbn}{123456789}
  \sdef {bib/authors[1]/firstname}{John}
  \sdef {bib/authors[1]/lastname}{Doe}
  \sdef {bib/authors[2]/firstname}{Patrik}
  \sdef {bib/authors[2]/lastname}{Black}
```

We can see that macros `\bib/title`, `\bib/isbn`, etc., are defined by `\treedata` and user can use them for example by `\cs{bib/title}`, `\cs{bib/isbn}`, `\cs{bib/authors[1]/lastname}`, etc.

Suppose that we have more bibliography entries like the above in our JSON file in the format

```
[ { entry }, { entry }, ..., { entry } ]
```

Now, `\treedata` prints the following information on the terminal:

```
\treedata:
  \sdef {bib[1]/title}{Big Book}
  \sdef {bib[1]/isbn}{123456789}
  \sdef {bib[1]/authors[1]/firstname}{John}
  \sdef {bib[1]/authors[1]/lastname}{Doe}
  \sdef {bib[1]/authors[2]/firstname}{Patrik}
  \sdef {bib[1]/authors[2]/lastname}{Black}

  \sdef {bib[2]/title}{Nice Article}
  \sdef {bib[2]/isbn}{987654321}
  \sdef {bib[2]/authors[1]/firstname}{Helen}
  \sdef {bib[2]/authors[1]/lastname}{Green}
  ...
```

All nodes without identifiers are numbered (locally started from one). The number of numbered entries is saved in macros `\relevant/path[]`. For example `\cs{bib[1]/authors[]}` expands to 2 (two authors in the first entry) and `\cs{bib[]}` expands to the number of bibliography entries in the JSON file. You can use `\fornum 1..\cs{bib[]} \do{...}`.

Users can declare more node identifiers to be numbered in the control sequence names. For example, we might need to support a single author having more than one `firstname` value. Then declaring `\def\reptreenames{firstname}` before invoking `\treedata` causes the `firstname` nodes to be locally numbered too.

We can do more manipulation with trees as shown in OpTeX tricks. For example, the macro

```
\treenodes[<id>]\macro
```

converts the tree to a set of macros, each node is a single macro and they are linked together from parents to children and back. Then users can merge more trees into a single one, connect a tree as a subtree of another one, remove subtrees, change the order of edges of a given node, add new edges, replace data of selected leaves, run tree-processing algorithms, etc. Then the `\treenodes` format can be converted back to a single `\macro` tree-structure with `\inodes` and `\enodes` with

```
\edef\macro{\treefromnodes{<root-node>}}
```

References

1. OpTeX. petr.olsak.net/optex/
2. OpTeX — tips, tricks, howto. <https://petr.olsak.net/optex/optex-tricks.html>
3. OpTeX trick 0153: Working with tree data structures. <https://petr.olsak.net/optex/optex-tricks.html#showtree>
4. OpTeX trick 0155: Example of using the tree data structure. <https://petr.olsak.net/optex/optex-tricks.html#usetree>

◊ Petr Olšák
Czech Technical University
in Prague
<https://petr.olsak.net>