# OpenCReports 0.5 Manual

**Zoltán Böszörményi**

**OpenCReports 0.5 Manual**
by Zoltán Böszörményi

# Table of Contents

# Chapter 1. Introduction and concepts

## The predecessor: RLIB

The idea to write OpenCReports[1] started with my getting acqauinted with RLIB[2] in 2005 and working with it (and on it) for a very long time, with the original implementors finally losing interest in developing RLIB further. This was around 2018. Even the original documentation site for RLIB was retired. But thanks to the Internet Archive, it can still be read[3]

To overcome some of the shortcomings seen in RLIB, its *ideas* were used for a completely new implementation with high level of compatibility to the original.

RLIB is a report generator library, so is OpenCReports. In this documentation, a lot of references contain comparisons to RLIB.

The name OpenCReports came from the fact that it's implemented in the C programming language and in an open way and using a free software license.

## Concepts

### What is a report generator?

A report generator uses a tabular data source (for example an array of data with labels for the column names, or an SQL query) and a description file that specifies how to display the data. The input data is transformed into various output formats, some for human viewing, some for further machine processing. Such output formats may be PDF, HTML, XML, plain text or CSV.

### XML based report description

The XML file format is widely used. It can describe structured data in a hierarchy with names for its sections or "nodes".

OpenCReports uses an RLIB-compatible report description with extensions. See XML description and the RLIB documentation[4]

### Comprehensive API for report creation

The Low level C API allows creating a report purely via program code. The High level C API allows loading an XML report description that contains all details about the report including database access. Mixing the high and low level APIs allows a balance anywhere between the two extremes, e.g. loading the report description but passing database access details via program code. RLIB's API and report description allowed neither extremes, it relied on the report description to provide the layout look, with the data access added from programming code.

### Strict expression parser

OpenCReports uses a Flex/Bison based expression parser. The expression grammar doesn't allow buggy expressions. See the Expressions chapter.

RLIB's expression parser was a custom implementation which was slightly fragile and forgiving. For example, it allowed unclosed parentheses at the end of an expression.

### Expression optimization

OpenCReports does some expression optimization to reduce runtime cost of computing expression values. For example, in `a * 2/3` the part `2/3` is two constants in a division. This is precomputed into a single constant as an optimization.

RLIB didn't contain automatic optimizations. It relied on manual optimizations, like using Report variables.

### Report variables

OpenCReports supports using standard report variables for calculating sums, minimum, maximum and average values or custom defined ones. See Report variables

### Extensive and extensible set of functions

OpenCReports has many operators and functions to be used in expressions. See Operators and functions in the Expressions chapter.

In fact, OpenCReports has all the operators and functions that RLIB also had, with many additions.

Custom functions can also be added to a report by programming. Custom functions may override stock functions.

### UTF-8 string handling

OpenCReports exclusively uses UTF-8 for strings. Input data must be in UTF-8 and output formats also use UTF-8.

RLIB could have been built with or without UTF-8 suppport. The build that disabled UTF-8 support relied on single byte character set encodings and conversion between them. That wasn't always reliable. Since that time, multibyte character encodings have won.

### High precision numeric data type

For historic record, RLIB was designed for the US and slower machines. It used a fixed point numeric representation. For the US, with its strong currency and prices expressed in low numbers, this was an acceptable design decision. But for countries, whose currencies are a few orders of magnitude weaker and conversely, the prices are similarly higher, the fixed point numeric value range was easily overflown, leading to wrong data in the report output.

Another potential problem with the fixed point numeric representation was that converting numbers from the input data to this internal representation always and unconditionally rounded down. This can be demonstrated with a carefully constructed (small) data set that would add up exactly to 100% both on paper and with using the IEEE-754 `double` data type, it would only add up to 99.99% with RLIB's internal numeric representation.

To avoid these kind of problems, OpenCReports uses GNU MPFR floating point values with 256-bit precision by default. This allows storing very large and very small numbers. E.g. this allows computations even with late stage hyperinflation[5] prices. See Numeric constants in the Expressions chapter and the Numeric behavior related functions part in the Low level C API chapter.

### Datetime and interval data types

OpenCReports differentiates between timestamp and time interval data types, with the latter allowing adding or subtracting a custom time period to and from timestamp data. See Datetime constants in the Expressions chapter.

## Automatic input data conversion

For maximum portability, databases provide their data in strings. They also provide metadata that describes the data type of every column in the data record.

RLIB relied on explicit data conversion functions, like `val()` and `stodt()`.

OpenCReports discovers the columns' data type and applies the conversion automatically. For RLIB compatibility, the conversion functions pass through values as is if they are already converted. For example, the `val()` function accepts an operand with both string and numeric data types. In the former case the conversion to numeric is performed, but if the operand is already numeric, no further conversion takes place. This is user friendly in two aspects: the conversion function is not necessary when implementing a new report, and an already existing RLIB report (that contains such conversion functions for the input data) may be used as is.

## Versatile field alignment and multi-row fields

In the report output, fields may have a fixed width in which they are displayed. Some field values are longer that the field width. When displaying them in a single row, fields may be left-, right- or center-aligned. With the PDF output, this means pixel-perfect alignment with the parts of the field value that don't fit masked off visually. RLIB aligned the data actually truncated to fit into the field width.

Fields longer than the designated width may be wrapped either at word or character boundaries. This way, they become multi-row fields. RLIB called them "memo" fields and the XML report description in OpenCReports also uses the same naming for the flags dealing with this detail. Multi-row fields are configurable regading word or character wrapping, or they may be limited to print only a certain amount of lines. Memo fields can break over column or page boundaries.

OpenCReports have some differences from RLIB regards to memo fields.

Thanks to Pango rendering, when using character wrapping, hyphenation is done.

The second extension over RLIB is that `justified` alignment can also be used for multi-line fields. This is also thanks to Pango. When `justified` alignment is used, all lines but the last are justified. Therefore, `justified` alignment equivalent to `left` alignment for single line fields.

## Report breaks

OpenCReports supports report breaks defined on arbitrary expressions. Breaks occur when the expression value changes, when stepping from one data set row to the next. See Report breaks and Breaks

## Multi-column reports

OpenCReports, just like RLIB, supports both single-column and multi-column report layout.

If a report is "narrow", so more columns would fit on the same page, the report may be set to use multiple columns on the same page.

## PDF output format, fonts, layout details

By using the excellent drawing possibilities of Cairo or later, OpenCReports supports generating the report in PDF format. RLIB relied on an internal PDF generator in earlier versions called RPDF, and used libHaru in later versions.

In RLIB, sizing of other details are a mix of units, making it harder to design the report layout:

- field widths and report heights are in number of characters
- width of lines are in points (1/72th inch)
- width of gaps between columns of a multi-column report is in inches

OpenCReports faithfully reimplements these for compatibility.

However, there are problems with field widths calculated in number of characters. Widths using a 12 point font is not the same as widths using a 20 point font. Also, font width and height are usually not identical. Some fonts are lean, with their width smaller than their height. RLIB expected that fonts are using the same character width and height. For this reason, RLIB only supported a small variety of monospace fonts.

Also, character widths for proportional fonts differ. Field width expressed in number of characters cannot be exact when using proportional fonts. For this reason, RLIB didn't support proportional fonts at all.

OpenCReports extended upon these ideas, so individual fields in a text line may use custom font settings while keeping the field width calculation identical to RLIB, which ends with the font parameters set for the text line. In OpenCReports, The custom field font settings (font name and size) don't influence the field width. Using this method, OpenCReports may use proportional fonts even with using the RLIB compatible size settings.

On top of that, OpenCReports also has a new, consistent size calculation method where everything is measured in points.

### Extensive set of unit tests

The unit tests ensure that OpenCReports' features keep working when adding new features or fixes. Units tests exercise many aspects of the high and low level API, report description handling, runtime behavior and output generation.

### Standard Linux dependencies

OpenCReports uses LibXML2[6], utf8proc[7], MPFR[8], libpaper[9], libcsv[10], yajl[11], Cairo[12], Pango[13], librsvg2[14], gdk-pixbuf2[15], PostgreSQL[16], MariaDB[17] and unixODBC[18].

For running the unit tests, Ghostscript[19] and `compare` from Imagemagick[20] are also needed.

`docbook2pdf` and `docbook2html` from Docbook[21] is used to generate the documentation.

## OpenCReports planned features

### More output formats

XML, CSV, plaintext and JSON are not currently supported as output formats.

### Graph and chart support in HTML and PDF output

Currently Gantt chart and various graph types (like barchart, pie chart and their various subtypes) are not supported.

### Visual editor for report XML descriptions

There are other report generators on the market with nice GUIs to create the report visually.

## Notes

1. https://github.com/zboszor/OpenCReports
2. https://github.com/SICOM/rlib
3. https://web.archive.org/web/20131116192438/http://newrlib.sicom.com/~rlib/index.php/Mai
4. https://web.archive.org/web/20131116192438/http://newrlib.sicom.com/~rlib/index.php/Mai
5. https://en.wikipedia.org/wiki/Hyperinflation
6. https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home
7. https://juliastrings.github.io/utf8proc/
8. https://www.mpfr.org
9. http://packages.qa.debian.org/libp/libpaper.html
10. https://github.com/rgamble/libcsv
11. http://lloyd.github.com/yajl/
12. https://www.cairographics.org
13. https://pango.gnome.org
14. https://wiki.gnome.org/Projects/LibRsvg
15. https://gitlab.gnome.org/GNOME/gdk-pixbuf
16. https://www.postgresql.org
17. https://mariadb.com
18. http://www.unixodbc.org
19. https://www.ghostscript.com
20. https://imagemagick.org
21. http://sources.redhat.com/docbook-tools/

# Chapter 2. Report XML description

## XML description structure

OpenCReports[1] uses an RLIB[2]-compatible report description with extensions.

The report XML description, like all XML files start with declaring that it's an XML file and the optional declaration of the Document Type Definition that the XML file can be checked against. For an OpenCReports report description, the first two lines are:

```
<?xml version="1.0"?>
<!DOCTYPE OpenCReport SYSTEM "opencreport.dtd">
```

The DTD file `opencreport.dtd` can be found in the sources of OpenCReports here[3].

After the XML header lines, a fully specified report description looks like this:

```
<OpenCReport>
    <Datasources>
        ...
    </Datasources>
    <Queries>
        ...
    </Queries>
    <Part>
        <pr>
            <pd>
                <Report>
                    ...
                </Report>
            </pd>
        </pr>
    </Part>
</OpenCReport>
```

or like this:

```
<OpenCReport>
    <Datasources>
        ...
    </Datasources>
    <Queries>
        ...
    </Queries>
    <Report>
        ...
    </Report>
</OpenCReport>
```

The XML sections Datasources and Queries are optional in the XML description. They can be substituted by program code using the datasource and query related calls in the Low level C API, similarly to RLIB.

For RLIB compatibility, a report description may start with either `<Part>` or `<Report>` as the toplevel node. In this case, there's no other way to add datasources and queries, but through program code, like the Low level C API.

Since XML files are hierarchical with a single toplevel node with child nodes, multiple `<Report>`s were only possible if using `<Part>` as the toplevel node with all the nodes having to be spelled out between `<Part>` and `<Report>`. With `<OpenCReport>` being the toplevel node, multiple `<Report>` child nodes can be used without the parent `<Part>` node.

### Notes about XML syntax and attributes

Most (if not all) XML attributes in the report description file are handled with the expression parser (see Expressions), with fallback to literal strings if the the location of expression wouldn't allow identifier references at that location.

For example, the datasource name may be declared this way:

```
<Datasource name="mysource" ... />
```

where the name is a literal string. Or this way:

```
<Datasource name="'mysource'" ... />
```

where the name is a string expression. (Note the extra quotes.) Or even this way:

```
<Datasource name="&quot;mysource&quot;" ... />
```

where name is a string expression which uses the " double quote character which must be "escaped" according to XML rules. This last variant is less readable than the first two. Nevertheless, it is usable and it works, so machine generated XML files may also be used.

The first form is a regular XML string value. Since expression parsing would find that `mysource` is an identifier which may be a query column name and this is not a valid place for a query reference, the non-parsed string value is used.

The second form is a single quoted OpenCReports string constant. The parsed string constant's value is used.

The third form is a double quoted OpenCReports string constant, but in XML the double quote character must be substituted with `&quot;` because they are reserved for quoting the attribute values. The parsed string constant's value is used. (This substitution is called "string escaping" and various other formats require some kind of substutition for reserved characters.)

To make the XML easier to read, the second form is recommended because it still allows embedding the single quote character inside a string (see XML description) in case e.g. a strong password contains this. For security-by-obscurity, the third form may be used because it is harder to read. For all special characters that should be escaped in XML, see Simplified XML Escaping[4].

## OpenCReport element

The toplevel `<OpenCReport>` element controls some global settings and serves as the topmost XML element for child elements.

### Size unit attribute

The `size_unit` attribute specifies report behaviour for size related settings:

```
<OpenCReport size_unit="rlib">
<OpenCReport size_unit="points">
```

Default is `rlib` which is the legacy RLIB behavior, with the known inconsistency in size related settings:

- width for `field` and `literal` (see Output node) are in number of characters. This is influenced by the font size set in either Part font size, Report font size or Line font size.
- height for Part column is measured in number of characters influenced by Part font size

- height for Report is measured in number of characters influenced by Part font size and Report font size
- width of horizontal lines and the optional border width around reports are specified in points
- gaps between columns of multi-column reports is measured in inches

Note that the default setting (as in RLIB) only ever expected monospace fonts that have the same width for every character. It also expected that the character height is identical to the character width. The latter detail is false for many monospace fonts, i.e. their height is usually greater than their width.

With proportional fonts (where the width of characters depend on their image, i.e. an "i" is thinner than an "m") width of text fields cannot reliably be set in "number of characters" because it's not an exact value. There is a workaround for this in `OpenCReports` but it's an over-engineered solution and isn't available in RLIB so it's not backward compatible either. But it's there if someone wants to port reports from RLIB and only make superficial changes. See Text element width.

When `size_unit` is set to `points`, all size related settings in the report are in points, a.k.a. 1/72th inch. Straightforward and consistent.

### No query show NoData

This controls whether NoData node is shown when the report's Query attribute is set to a non-existing query name.

```
<OpenCReport noquery_show_nodata="yes">
<OpenCReport noquery_show_nodata="no">
```

Default is `true` when `OpenCReport` is the toplevel node, `false` when either `<Part>` or `<Report>` is the toplevel node for RLIB compatibility. See NoData node.

### Report height after last

It controls whether report height is applied after the last `<Report>` in the `<pd>` node.

```
<OpenCReport report_height_after_last="yes">
<OpenCReport report_height_after_last="no">
```

Default is `false`.

## Datasources

Datasources in OpenCReports are either database connections, or accessors (mini-drivers) for data files in certain formats.

Datasource descriptions are in the following format:

```
<Datasources>
    <Datasource name="mysource" type="..." ... />
</Datasources>
```

A report may have multiple datasources, i.e. the description may list multiple `<Datasource>` lines.

Datasources must have unique names in a report and their type may be: `mariadb` (or `mysql`), `postgresql`, `odbc`, `csv`, `json`, `xml` or `array`.

## MariaDB (MySQL) database connection

A MariaDB database connection may be declared in three ways. Either by using the database host and port, the database name, user name and password directly:

```
<Datasource
    name="mysource" type="mariadb"
    host="..." port="..."
    dbname="..." user="..." password="..." />
```

or alternatively, instead of the host and port, specifying the UNIX Domain Socket file for a local connection if it's not in the standard location:

```
<Datasource
    name="mysource" type="mariadb"
    unix_socket="..."
    dbname="..." user="..." password="..." />
```

or moving these details out to an external configuration file in an INI file format:

```
<Datasource
    name="mysource" type="mariadb"
    optionfile="myconn.cnf" group="myconn" />
```

In the second case, the configuration file `myconn.cnf` would contain something like this:

```
[myconn]
!include /etc/my.cnf
database=mydb
user=myuser
#password=
#host=
#port=
#unix_socket=
```

Please note that the INI group name `[myconn]` matches `group="myconn"` in the above datasource declaration.

The database name and user name are mandatory. The user password is optional, depending on the database security authentication setup.

The database host and port, or the socket file location are all optional. Without these, a local connection is attempted using the default settings. If the host name is specified but the port isn't, the remote host is used on the default port (as known by the local MariaDB database client library).

## PostgreSQL database connection

A PostgreSQL database connection may be declared in three ways. Either by using the database host and port, the database name, user name and password directly:

```
<Datasource
    name="mysource" type="postgresql"
    host="..." port="..."
    dbname="..." user="..." password="..." />
```

or alternatively, instead of the host and port, specifying the UNIX Domain Socket file for a local connection if it's not in the standard location:

```
<Datasource
    name="mysource" type="postgresql"
```

```
    unix_socket="..."
    dbname="..." user="..." password="..." />
```

or using a so called connection string:

```
<Datasource
    name="mysource" type="postgresql"
    connstr="..." />
```

For the connection string format, see the PostgreSQL documentation[5].

The database name and user name are mandatory. The user password is optional, depending on the database security authentication setup.

The database host and port, or the socket file location are all optional. Without these, a local connection is attempted using the default settings. If the host name is specified but the port isn't, the remote host is used on the default port (as known by the local PostgreSQL database client library).

## ODBC database connection

The above described MariaDB and PostgreSQL database connection types are using their respective client libraries. There is a more generic way, i.e. ODBC. ODBC was invented by Microsoft in the 1990s for Windows. See Microsoft Open Database Connectivity (ODBC)[6] In their solution, there's an abstract client library and individual database drivers adhere to the APIs offered by ODBC toplevel library. Since then, UNIX and UNIX-like systems also gained their ODBC client libraries in two different implementations, both of which are supported by OpenCReports: unixODBC[7] and iODBC[8].

An ODBC database setup is done a differently. There are two system-wide configuration files. The first one is `odbcinst.ini` that lists the database drivers installed into the system. The second one is `odbc.ini` which references the first one and lists pre-defined database connections. These database connections are named. In ODBC speak, these are called Data Source Names or DSNs. The DSNs specify the low level connection parameters, like the database host and port, and optionally the user name and password, too.

Thus, an ODBC database connection may be declared in two ways. The first way is by using the DSN name, and optionally the user name and password:

```
<Datasource
    name="mysource" type="odbc"
    dbname="..." user="..." password="..." />
```

In this case, the `dbname` attribute is not the low level database name, but the ODBC abstract DSN name.

There's also a way to use the so called connection string which contain the same connection information:

```
<Datasource
    name="mysource" type="odbc"
    connstr="..." />
```

For the connection string format, see the public examples[9].

## CSV file datasource

CSV ("Comma Separated Values") is a simple tabulated file format. Every line must have the same number of columns, for which the values are separated by commas. The first line in the file contains the column names.

A CSV file datasource is declared very simply:

```
<Datasource name="mysource" type="'csv'" />
```

In this case, the actual CSV file is not declared, only that a "query" using a CSV file will be listed later under `<Queries>`.

This datasource is less smart than others, in that is cannot specify data type for columns. Every column is assumed to be a string, regardless if the values themselves are quoted or not in the file. Data conversion functions must be used, like `val()`, `stodt()` or `interval()`.

## JSON file datasource

Similarly to CSV, the JSON file datasource is also declared very simply:

```
<Datasource name="mysource" type="'json'" />
```

In this case, the actual JSON file is not declared, only that a "query" using a JSON file will be listed later under `<Queries>`.

A JSON file datasource is expected in this format:

```
{
    "columns": ["colname1", ... ],
    "coltypes": ["type", ... ],
    "rows": [
     { "colname1": value1, ...  },
        ...
    ]
}
```

The JSON file is expected to list the column names in a string array called `columns`.

The column types are optionally listed in the string array called `coltypes`. If they are listed, the `coltypes` array must have the same number of strings as the `columns` array. The types may be `string`, `number` or `datatime`. If the column type array is missing, then all data values are assumed to be strings and data conversion functions must be used, like `val()`, `stodt()` or `interval()`.

The rows are listed in an array called `rows` and column data values for each row are in a collection with data names from the `columns` and data types from the `coltypes` arrays.

## XML file datasource

Similarly to CSV and JSON, the XML file datasource is also declared very simply:

```
<Datasource name="mysource" type="'xml'" />
```

In this case, the actual XML file is not declared, only that a "query" using an XML file will be listed later under `<Queries>`.

An XML file datasource is expected in this format:

```
<?xml version="1.0"?>
<data>
    <rows>
        <row>
            <col>value</col>
            ...
        </row>
        ...
    </rows>
    <fields>
        <field>column1</field>
        ...
```

```
    </fields>
    <coltypes>
        <col>type1</col>
        ...
    </coltypes>
</data>
```

The XML section names `<data>`, `<rows>` and `<fields>` are the same as they were in RLIB for its XML data source. The order of `<rows>` and `<fields>` is not important. But the order of field names in `<fields>` must match the column value order in each `<row>`.

The section <coltypes> is new in OpenCReports and is optional. If it's present, then it must list the data types in the same order as the section `<fields>`. The types may be `string`, `number` or `datatime`. If this section is not present, all values are assumed to be strings and data conversion functions must be used, like `val()`, `stodt()` or `interval()`.

### Array datasource

Arrays are global in-memory structures in the application that should be accessible to the OpenCReports library. For example, when using the C programming language, global non-static symbols are visible to libraries if the application is compiled with `-rdynamic`.

Similarly to file based datasources, the array datasource is declared very simply:

```
<Datasource name="mysource" type="'array'" />
```

In this case, the actual array is not declared, only that a "query" using an array will be listed later under `<Queries>`.

A C array is declared in this format:

```
const char *array[ROWS + 1][COLUMNS] = {
    { "column1", ... },
    { "value1",  ... },
    ...
};
```

The array is declared as a two-dimensional array of C strings. The first row of the array is the column names, `[ROWS + 1]` in the array declaration accounts for the title row.

All rows have the same number of columns. Column values may be NULL, in which case they will be treated the same as SQL NULLs in SQL query results.

Optionally, a column types array is declared separately:

```
#include <opencreport.h>

const enum ocrpt_result_type coltypes[COLUMNS] = {
    ...
};
```

If this array is present, it must have the same number of `COLUMNS` as the matching data array. The `enum ocrpt_result_type` usable in data array type declaration are `OCRPT_RESULT_STRING`, `OCRPT_RESULT_NUMBER` and `OCRPT_RESULT_DATETIME`.

## Queries

Queries in OpenCReports are SQL queries for database connections, or data files files in certain formats. The queries are declared like this:

```
<Queries>
```

```
    <Query name="..." ... />
    ...
</Queries>
```

## SQL queries for SQL datasources

SQL queries for MariaDB, PostgreSQL and ODBC datasources may be declared two ways, either as the XML value for `<Query>`:

```
<Query
    name="myquery"
    datasource="mysource">
SELECT * FROM some_table
</Query>
```

or as the `value` attribute:

```
<Query
    name="myquery"
    datasource="mysource"
    value="SELECT * FROM some_table" />
```

Note, that the XML attribute `datasource="..."` must match a previously declared datasource.

The SQL query can be any `SELECT` statement.

### Special note for SQL datasources

The database client libraries for MariaDB, PostgreSQL and ODBC return all the query rows from the database server at once by default. As such, it is possible that a long query result doesn't fit into the computer memory.

The report needs to traverse the query result twice to pre-compute "delayed" values (see precalculated and Precalculated variables), so it needs to be able to rewind the data set once it was read to the end.

The alternative API in MariaDB to load the rows one by one doesn't allow rewinding, so it's not usable for the report's purposes.

It is only PostgreSQL that allows using an *SQL cursor* as a standalone entity, i.e. outside SQL procedures as defined by the SQL standard. This PostgreSQL extension to the standard allows saving memory in such a way that it allows processing very long queries. Behind the scenes, a `WITH HOLD` cursor is used and 1024 rows are loaded in one go from the server.

## Queries for file based datasources

Queries for CSV, XML and JSON datasources may be declared two ways. Either as the XML value for `<Query>`:

```
<Query
    name="myquery"
    datasource="mysource"
>xmldata.xml</Query>
```

or as the `value` attribute:

```
<Query
    name="myquery"
    datasource="mysource"
    value="'xmldata.xml'" />
```

Notes:

- The XML attribute `datasource="..."` must match a previously declared datasource.
- It is recommended that the `value="..."` form is used, since it's not ensured that whitespace before or after the file name is trimmed in the first variant if the XML is "beautified". The file name that the OpenCReports library receives must be correct in order to use it.
- The declaration must specify a file in the correct format for the datasource type.
- The optional type declaration for columns in the XML and JSON file formats, or for CSV, the complete lack of it can be supplemented with a memory array using the optional `coltypes="..."` attribute. For details, see the Array queries.

## Queries for array based datasources

Queries for array datasources may be declared two ways. Either as the XML value for `<Query>`:

```
<Query
    name="myquery"
    datasource="mysource"
    coltypes="'coltypes'"
    rows="30"
    cols="6"
>array</Query>
```

or as the `value` attribute:

```
<Query
    name="myquery"
    datasource="mysource"
    value="'array'"
    coltypes="'coltypes'"
    rows="30"
    cols="6" />
```

Notes:

- The XML attribute `datasource="..."` must match a previously declared datasource.
- It is recommended that the `value="..."` form is used, since it's not ensured that whitespace before or after the symbol name is trimmed in the first variant if the XML is "beautified". The symbol name that the OpenCReports library receives must be correct in order to use it. The array name must match the correct global symbol name. It will be looked up via `dlsym()`.
- Similarly to the array symbol name, the `coltypes="..."` array name must match the correct global symbol name. It will be looked up via `dlsym()`.
- The value for `cols` must match the second dimension of the data array.
- The value for `rows` must be one less than the first dimension of the C data array. (The title row is not counted in the XML description.)

Failing to fulfill the above may cause crashes or wrong data to be used in the report.

## Follower queries

Multiple queries may be declared. But by default, only the first one is actually used by a report.

Secondary queries may be associated with the primary query as so called "follower" queries. Any query may be a follower to the primary query.

There are two kinds of follower queries:

- regular, or basic follower queries, and
- so called N:1 (N-to-one) followers in RLIB speak.

### Regular follower queries

A follower query is run along the primary query and their rows are laid out side by side. The first row of the follower query is assigned the first row of the primary query. The second row of the follower query is assigned to the second row of the primary query, and so on. The number of rows of the complete data set is determined by the primary query. If the follower query runs out of rows before the primary, the columns values will be presented as SQL NULLs.

This is similar to using `LEFT OUTER JOIN` and using `ROWNUM` in Oracle or the `row_number()` SQL function in PostgreSQL as the matching value between the primary query and the follower query.

A regular follower query is declared by adding the `follower_for="..."` attribute. The value for `follower_for="..."` is the `name` of a previously declared query. Example:

```
<Query
    name="myquery1"
    datasource="mysource1"
    value="'SELECT * FROM table1'" />

<Query
    name="myquery2"
    datasource="mysource2"
    value="'SELECT * FROM table1'"
    follower_for="myquery1" />
```

In this example, two queries of two different datasources are used. This is one of the advantages of using follower queries, i.e. data from different databases may be used. Nowadays, with foreign queries implemented in e.g. PostgreSQL, its use case is more limited.

### N:1 follower queries

An N:1 follower query, for all intents and purposes, is the right side query in a `LEFT OUTER JOIN` query, with the primary query on the left side. Rows of the main query and the follower query are matched according to the boolean expression specified with the `follower_expr="..."` attribute. Example:

```
<Query
    name="myquery1"
    datasource="mysource1"
    value="'SELECT * FROM table1'" />

<Query
    name="myquery2"
    datasource="mysource2"
    value="'SELECT * FROM table1'"
    follower_for="myquery1"
    follower_expr="myquery1.id = myquery2.id" />
```

## Report parts

A report XML may contain multiple actual reports. Such a `<Part>` may be under the toplevel `<OpenCReport>` node, in which case multiple report parts may exist in the same XML, or it may be the toplevel node of the XML. In the latter case, only a single `<Part>` may exist in the XML.

```
<Part>
    <pr>
        <pd>
            <Report>
                ...
            </Report>
            ...
        </pd>
        ...
    </pr>
    ...
</Part>
```

A report `<Part>` may consist multiple reports, arranged in

- rows (`<pr>`),
- columns in rows (`<pd>`), and
- reports (`<Report>`) arranged vertically in a column.

The rows and columns in rows do not form a grid as rows are independent of each other. E.g. one row may contain two columns, the next one may contain three, the next one may contain one. It is completely freeform.

This allows very complex report layouts. One possible application of such a complex layout is printed forms.

### Part attributes

#### Font name

The font name attribute specifies the font for the report part's global scope. It can be overridden by child nodes for their scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Part font_name="Arial">
<Part fontName="Arial">
```

If both forms are specified, `font_name` is used.

Default font name is `Courier`.

#### Font size

The font size attribute specifies the font size for the report part's global scope. It can be overridden by child nodes for their scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Part font_size="10">
```

```
<Part fontSize="10">
```

If both forms are specified, `font_size` is used.

Default font size is 12.

### Size unit

Shortcut for the Size unit attribute in `<OpenCReport>` when `<Part>` is the toplevel node.

```
<Part size_unit="default">
<Part size_unit="rlib">
<Part size_unit="points">
```

When `<OpenCReport>` is the toplevel node in the XML, this attribute for `<Part>` is ignored.

### No query show NoData attribute

Shortcut for No query show NoData attribute in `<OpenCReport>` when `<Part>` is the toplevel node.

```
noquery_show_nodata="yes"
noquery_show_nodata="no"
```

See default in No query show NoData attribute See also NoData node.

### Report height after last attribute

Shortcut for Report height after last attribute in `<OpenCReport>` when `<Part>` is the toplevel node.

```
report_height_after_last="yes"
report_height_after_last="no"
```

See default in Report height after last attribute See also NoData node.

### Orientation

Page orientation for the whole `<Part>`.

```
<Part orientation="portrait">
<Part orientation="landscape">
```

Default is portrait orientation.

### Margin settings

Margin settings for the page for the whole `<Part>`. Individual settings exist for the top, bottom, left and right margins of the page. Every setting exist in two forms: the RLIB compatible "lowerCamelCase" variant and the all lowercase with underscore. The lowecase-with-underscore variants are the preferred ones.

```
<Part top_margin="0.2">
```

```
<Part topMargin="0.2">
<Part bottom_margin="0.2">
<Part bottomMargin="0.2">
<Part left_margin="0.2">
<Part leftMargin="0.2">
<Part right_margin="0.2">
<Part rightMargin="0.2">
```

When `size_unit="rlib"` is in effect (the default case) the margin unit is inches. The margin unit is points (1/72th inches) when `size_unit="points"` is in effect.

Default values for the top, bottom, left and right margins are all 0.2, regardless of the unit.

Note that `rightMargin` didn't exist in RLIB.

### Paper type

Paper type (implicitly: page size) for the whole `<Part>`. It exists in two forms: the RLIB compatible "lowerCamelCase" variant and the all lowercase with underscore. The lowecase-with-underscore variant is preferred.

```
<Part paper_type="A4">
<Part paperType="A4">
```

Default value is the current system paper type that libpaper[10] uses. E.g. if the system is set to US English, the default paper type is implicitly `letter`. In most of Europe, the default paper type is `A4`

The paper type can be specified in either lower case or upper case, both are accepted.

### Iterations

The same `<Part>` may be executed multiple times.

```
<Part iterations="3">
```

Default value is 1.

Note that `<Part>`s and every iteration of one starts on a new page.

### Suppress

Report parts may be suppressed.

```
<Part suppress="yes">
<Part suppress="val(m.suppress_part)">
```

Default value is `false`, i.e. no suppression.

The expression for `suppress` must be a constant expression. An environment variable (since it can't - or shouldn't - change during the report execution) is considered constant. See Expressions.

### Suppress page header on the first page

The <PageHeader> section (see Page header below) for <Part> may be suppressed on the first page.

```
<Part suppressPageHeaderFirstPage="yes">
```

Default value is no.

Note that this suppression applies only to the first page of the complete result (e.g. PDF) and not to the first page of an iteration, which may fall on a later page of the result.

## Part subsections

As described in Report parts, a <Part> may contain one or more report rows (<pr>) which in turn may contain one or more columns (<pr>). See Part row and Part column.

### Page header

This is the description of the page header. It contains an Output node.

```
<Part>
    <PageHeader>
        <Output>
            ...
        </Output>
    </PageHeader>
</Part>
```

### Page footer

This is the description of the page footer. It contains an Output node.

```
<Part>
    <PageFooter>
        <Output>
            ...
        </Output>
    </PageFooter>
</Part>
```

### Part row

See Part row.

```
<Part>
    <pr>
        <pd>
            ...
        </pd>
    </pr>
</Part>
```

*Part row subsections*

*Part column*

See Part column.

# Part row

A part row (`<pr>`) may contain one or more Part columns (`<pd>`) which are layed out side by side horizontally. The longest running column will control the height of the row. The next row will be continuous from that vertical page position.

## Part row attributes

### Layout

The `layout` attribute exists mainly for RLIB compatibility, it's ignored. It accepts two values: `flow` and `fixed`.

```
<pr layout="flow">
<pr layout="fixed">
```

### New page

The `newpage` attribute controls whether the part row starts from the point where the previous row in the same part ended, or it should start on a new page.

```
<pr newpage="yes">
```

Default value is `no`

### Suppress

Report rows may be suppressed.

```
<pr suppress="yes">
<pr suppress="val(m.suppress_row)">
```

Default value is `false`, i.e. no suppression.

The expression for `suppress` must be a constant expression. An environment variable (since it can't - or shouldn't - change during the report execution) is considered constant. See Expressions.

# Part column

A part column (`<pd>`) may contain one or more Reports (`<Report>`) which are layed out vertically in this column continuously.

Such a report may be inlined:

```
<Part>
    <pr>
        <pd>
            <Report>
                ...
            </Report>
        </pd>
    </pr>
</Part>
```

A report may also be loaded from a separate file. For details, see Loaded report.

```
<Part>
    <pr>
        <pd>
            <load ... />
        </pd>
    </pr>
</Part>
```

## Part column attributes

### Width

Width of the part column.

```
<pd width="60">
```

When Size unit attribute is set to `rlib` (the default), the column width is measured in characters, which is controlled by Part font size. Note, that the font width and height may differ, depending on the font face controlled by Part font name. Width is computed from the font width.

When Size unit attribute is set to `points`, width is measured in points.

Columns without explicitly specified width are dynamically sized according to Paper type, Margin settings and columns with explicitly specified width.

Columns that exceed the total page width (according to Paper type and Margin settings) will be shown partially, or won't be shown at all.

### Height

Height of the part column.

```
<pd height="120">
```

When Size unit attribute is set to `rlib` (the default), the column height is measured in characters, which is controlled by Part font size. Note, that the font width and height may differ, depending on the font face controlled by Part font name. Column height is computed from the font height.

When Size unit attribute is set to `points`, height is measured in points.

Report details in this part column are layed out up to the specified height. See Report height for further explanation.

Note, that OpenCReports allows fixed height columns to break over page boundaries. This is a deviation from RLIB.

### Border width

Border width around the part column. It is measured in points.

```
<pd border_width="2">
```

When set, a rectangle around the part column will be drawn. The width of outline of the rectangle is measured in points.

This is a deviation from RLIB where the width of the outline was a fixed 0.1 points and an inner margin (all of top, bottom, left and right) inside the column border was used.

If the column breaks over a page boundary, the border will be drawn the column parts on every page.

### Border color

When `border_width` is set, this color is used to draw the border rectangle.

```
<pd border_color="'blue'">
```

See Color specification.

### Detail columns

Inner `<Report>`s are layed out in one or more columns.

```
<pd detail_columns="3">
```

Default value is 1.

### Column padding

When `detail_columns` is set to higher than 1, inner padding between the detail columns may be specified.

```
<pd column_pad="0.2">
```

Default value is 0, i.e. no padding.

The unit of padding is inches if Size unit attribute is set to `rlib` (default), points otherwise.

### Suppress

Report columns may be suppressed.

```
<pd suppress="yes">
<pd suppress="val(m.suppress_column)">
```

Default value is `false`, i.e. no suppression.

The expression for `suppress` must be a constant expression. An environment variable (since it can't - or shouldn't - change during the report execution) is considered constant. See Expressions.

# Report

This section may occur in a wider context or standalone in an XML description file.

Example XML skeleton structure with `<OpenCReport>` as the toplevel node:

```
<OpenCReport>
    <Part>
        <pr>
            <pd>
                <Report>
                    ...
                </Report>
            </pd>
        </pr>
    </Part>
</OpenCReport>
```

Example XML skeleton structure with `<Part>` as the toplevel node:

```
<Part>
    <pr>
        <pd>
            <Report>
                ...
            </Report>
        </pd>
    </pr>
</Part>
```

Example XML skeleton structure with a standalone `<Report>` node:

```
<Report>
    ...
</Report>
```

When `<Report>` is the toplevel node, parent nodes for `<Part>`, `<pr>` and `<pd>` are implicitly created. Subsections and many attributes specific to these parent nodes can be used as shortcuts in the `<Report>` node.

## Report attributes

### Font name

The font name attribute specifies the font for the report's scope. It can be overridden by child nodes for their scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Report font_name="Arial">
<Report fontName="Arial">
```

If both forms are specified, `font_name` is used.

Default font name is what's set for Part font name, or `Courier` if both are unset.

### Font size

The font size attribute specifies the font size for the report's scope. It can be overridden by child nodes for their scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Report font_size="10">
<Report fontSize="10">
```

If both forms are specified, `font_size` is used.

Default font name is what's set for Part font size, or `12` if both are unset.

### Size unit

Shortcut for the Size unit attribute in `<OpenCReport>` when `<Report>` is the toplevel node.

```
<Report size_unit="default">
<Report size_unit="rlib">
<Report size_unit="points">
```

When `<OpenCReport>` or `<Part>` is the toplevel node in the report XML description, this attribute for `<Report>` is ignored.

### No query show NoData attribute

Shortcut for No query show NoData attribute in `<OpenCReport>` when `<Report>` is the toplevel node.

```
<Report noquery_show_nodata="yes">
<Report noquery_show_nodata="no>
```

See default in No query show NoData attribute See also NoData node.

### Report height after last attribute

Shortcut for Report height after last attribute in `<OpenCReport>` when `<Part>` is the toplevel node.

```
<Report report_height_after_last="yes">
<Report report_height_after_last="no">
```

See default in Report height after last attribute See also NoData node.

### Orientation

Shortcut for Part page orientation for the implicitly created parent `<Part>` node when `<Report>` is standalone.

```
<Report orientation="portrait">
<Report orientation="landscape">
```

Default is portrait orientation.

This setting for `<Report>` is ignored when there is a parent `<Part>` node in the XML description.

### Margin settings

Shortcuts for Margin settings for the implicitly created parent `<Part>` node. Individual settings exist for the top, bottom, left and right margins of the page. Every setting exist in two forms: the RLIB compatible "lowerCamelCase" variant and the all lowercase with underscore. The lowecase-with-underscore variants are the preferred ones.

```
<Report top_margin="0.2">
<Report topMargin="0.2">
<Report bottom_margin="0.2">
<Report bottomMargin="0.2">
<Report left_margin="0.2">
<Report leftMargin="0.2">
<Report right_margin="0.2">
<Report rightMargin="0.2">
```

When `size_unit="rlib"` is in effect (the default case) the margin unit is inches. The margin unit is points (1/72th inches) when `size_unit="points"` is in effect.

Default values for the top, bottom, left and right margins are all 0.2, regardless of the unit.

Note that `rightMargin` didn't exist in RLIB.

These settings for `<Report>` are ignored when there is a parent `<Part>` node in the XML description.

### Paper type

Shortcut for Paper type for the implicitly created parent `<Part>` node. It exists in two forms: the RLIB compatible "lowerCamelCase" variant and the all lowercase with underscore. The lowecase-with-underscore variant is preferred.

```
<Part paper_type="A4">
<Part paperType="A4">
```

Default value is the current system paper type that libpaper[11] uses. E.g. if the system is set to US English, the default paper type is implicitly `letter`. In most of Europe, the default paper type is `A4`

The paper type can be specified in either lower case or upper case, both are accepted.

This setting for `<Report>` is ignored when there is a parent `<Part>` node in the XML description.

### Height

Height of the report.

```
<Report height="120">
```

When Size unit attribute is set to `rlib` (the default), the report height is measured in characters, which is controlled by Report font size. Note, that the font width and height may differ, depending on the font face controlled by Report font name. Report height is computed from the font height.

When Size unit attribute is set to `points`, height is measured in points.

Report details are layed out up to the specified height. If the report would run longer than the specified height, it gets truncated. When the report fits in the specified height, the next report's starts with the remaining height added as padding before it.

If the <Report> is the last one in the <pd> node, then the report may or may not be padded with the remaining height, depending on the Report height after last attribute.

If height is unset for the parent <pd> node, it is expanded with this vertical padding.

In case height is set for both the parent <pd> node and the <Report> nodes in it, the height value for <pd> is applied first. It would limit the displayed rows in whichever <Report> node is terminated by it. Subsequent <Report> nodes would no be displayed in that <pd> node.

Note, that OpenCReports allows fixed-height reports to break over page boundaries. This is a deviation from RLIB.

### Iterations

The same <Report> may be executed multiple times.

```
<Report iterations="3">
```

Default value is 1.

### Suppress

Reports may be suppressed.

```
<Report suppress="yes">
<Report suppress="val(m.suppress_report)">
```

Default value is `false`, i.e. no suppression.

The expression for `suppress` must be a constant expression. An environment variable (since it can't - or shouldn't - change during the report execution) is considered constant. See Expressions.

### Suppress page header on the first page

Shortcut for Suppress page header on the first page in the implicitly created parent <Part> node when <Report> is the toplevel node.

```
<Report suppressPageHeaderFirstPage="yes">
```

Default value is `no`.

This setting for <Report> is ignored when there is a parent <Part> node in the XML description.

### Query

Set the primary query name for <Report> from the list of Queries.

```
<Report query="query1">
```

Default value is unset, i.e. use the first query from the list of Queries.

### Field header priority

Set the field header priority for the report versus break (header and footer) priority. This setting selects which report detail is encompassing the other.

```
<Report field_header_priority="low">
```

Default value is `high`. In this mode, the field header is printed on the top of every page of the report and break headers and footers are encompassed by it. The default is chosen for RLIB compatibility.

When this setting is `low`, field headers are handled with lower priority compared to break headers and footers. In this mode, a break header is followed by the field header, then data rows (field details), followed by the break footer. This brings the field header closer to the field details.

### Border width

This is a shortcut for Part column border width for the implicitly created `<pd>` node when `<Report>` is standalone. It is measured in points.

```
<Report border_width="2">
```

When set, a rectangle around the part column (in this case, around the single report in the part column) will be drawn. The width of outline of the rectangle is measured in points.

This is a deviation from RLIB where the width of the outline was a fixed 0.1 points and an inner margin (all of top, bottom, left and right) inside the column border was used. Also, this shortcut didn't exist in RLIB, the `<pd>` node had to be present.

If the report (column) breaks over a page boundary, the border will be drawn the column parts on every page.

This setting for `<Report>` is ignored when there is a parent `<pd>` node.

### Border color

This is a shortcut for Part column border color for the implicitly created `<pd>` node when `<Report>` is standalone. When `border_width` is set, this color is used to draw the border rectangle.

```
<Report border_color="'blue'">
```

See Color specification.

### Detail columns

Shortcut for Detail columns in the implicitly created parent `<pd>` when `<Parent>` is the toplevel node.

```
<Report detail_columns="3">
```

Default value is 1.

This setting for `<Parent>` is ignored when there is a parent `<pd>` node in the report XML description.

### Column padding

Shortcut for Column padding in the implicitly created parent `<pd>` node when `<Parent>` is the toplevel node.

```
<Report column_pad="0.2">
```

Default value is 0, i.e. no padding.

The unit of padding is inches if Size unit attribute is set to `rlib` (default), points otherwise.

## Report subsections

### Page header

This is a shortcut for Page header in the implicitly created `<Part>` node when `<Report>` is standalone. It contains an Output node.

```
<Report>
    <PageHeader>
        <Output>
            ...
        </Output>
    </PageHeader>
</Part>
```

This subsection for `<Report>` is ignored when there is a page footer section defined for the `<Part>` node, either in the `<Part>` node itself or in a previous child `<Report>` node for the same `<Part>`.

### Page footer

This is a shortcut for Page footer in the implicitly created `<Part>` node when `<Report>` is standalone. It contains an Output node.

```
<Report>
    <PageFooter>
        <Output>
            ...
        </Output>
    </PageFooter>
</Part>
```

This subsection for `<Report>` is ignored when there is a page footer section defined for the `<Part>` node, either in the `<Part>` node itself or in a previous child `<Report>` node for the same `<Part>`.

### Report header

This is the description of the report header that is emitted at the start of the report. It contains an Output node.

```
<Report>
    <ReportHeader>
        <Output>
            ...
        </Output>
    </ReportHeader>
```

```
</Part>
```

### Report footer

This is the description of the report footer that is emitted at the end of the report. It contains an Output node.

```
<Report>
    <ReportFooter>
        <Output>
            ...
        </Output>
    </ReportFooter>
</Part>
```

### Variables

This section describes the Variables in the report.

```
<Report>
    <Variables>
        <Variable ... />
        ...
    </Variables>
</Part>
```

### Breaks

This section describes the Breaks in the report.

```
<Report>
    <Breaks>
        <Break ... >
            ...
        </Break>
        ...
    </Breaks>
</Part>
```

### Detail

This section describes the tabular details of the report. There are two subsections in this node, both contain an Output node.

```
<Report>
    <Detail>
        <FieldHeaders>
            <Output>
                ...
            </Output>
        </FieldHeaders>
        <FieldDetails>
            <Output>
                ...
            </Output>
        </FieldDetails>
    </Detail>
```

```
</Part>
```

<FieldHeaders> is used to describe the header for data rows.

<FieldDetails> is used to show data that is derived from the current data row produced by the report query.

### Alternate output for no data

This section describes the alternate output of the report when the query has no data rows, or there is no such query name defined that's set in Report query name. It contains an Output node.

This section may be declared in two ways. One way is to spell out the <Alternate> node:

```
<Report>
    <Alternate>
        <NoData>
            <Output>
                ...
            </Output>
        </NoData>
    </Alternate>
</Part>
```

The other way is without the <Alternate> node:

```
<Report>
    <NoData>
        <Output>
            ...
        </Output>
    </NoData>
</Part>
```

When the Report query name does not exist in in the global list of Queries and the No query show NoData attribute is set, then the <NoData> section is not displayed.

## Loaded report

It is like an inline report, but the Report is loaded from a different file.

### Loaded Report attributes

#### File name

```
<load name="report1.xml" />
```

### Query

The `query` attribute overrides the same attribute of `<Report>`. This way, the report in the separate file can be reused for a different data set.

```
<load query="query1" />
```

When the No query show NoData attribute is set and the Report query name is set to a non-existing name in the global list of Queries, the `<NoData>` section is not displayed.

### Iterations

The `iterations` attribute overrides the same attribute of `<Report>`.

```
<load iterations="5" />
```

## Variables

This is the parent node for individual `<Variable>` nodes that describe each variable.

```
<Variables>
    <Variable ... />
    ...
</Variables>
```

## Variable

This node describes one `<Variable>` node. It has no children nodes, only attributes.

```
<Variable ... />
```

### Variable attributes

#### Name

The name of the variable. It must be unique in the list of variables for the parent `<Report>` node.

```
<Variable name="var1" />
```

#### Value

The "value" of the variable, or rather, the expression from which the value is computed. Variables' values are computed for every data row produced by the report

query. The expression may therefore reference field names of queries that are declared in the XML description or in programming code.

```
<Variable value="q1.field1 + 2 * q2.field2" />
```

### Type

The type of the variable. Several variable types exist:

```
<Variable value="q1.field" type="expression"/>

<Variable value="q1.field" type="count"/>
<Variable value="q1.field" type="countall"/>

<Variable value="q1.field" type="sum"/>

<Variable value="q1.field" type="average"/>
<Variable value="q1.field" type="averageall"/>

<Variable value="q1.field" type="highest"/>
<Variable value="q1.field" type="lowest"/>

<Variable value="q1.field" type="custom"/>
```

Default type is `expression`, this is just a shortcut for the value expression that saves both typing and time to generate the report.

The `count` and `countall` variable types count the number of expression results for the data set. The former leaves out NULL values, the latter includes them. This is equivalent to COUNT(field) and COUNT(*) in SQL.

The `sum` variable type sums the non-NULL values of the expression results for the data set.

The `average` and `averageall` variable types are combinations of `sum` and either `count` or `countall`. They take the value computed for each data row, add them together, and divide by the number of values. The result of `average` and `averageall` may differ if there is NULL data in the result set.

The `highest` and `lowest` variable types return the highest and the lowest values for the data set, respectively.

All of the above pre-defined variables types work on numeric data.

The `custom` variable type allow arbitrary user variables if the predefined types are not enough, for example, when the base type needs to be something else then a number. See below.

### Custom variable attributes

These attributes below define a custom variable. A base expression, up to two intermediary expressions and one result expression may be defined, together with the expression type.

```
<Variable
    baseexpr="..."
    intermedexpr="..."
    intermed2expr="..."
    resultexpr="..."
    basetype="..."
    type="custom"/>
```

`baseexpr`, `intermedexpr`, `intermed2expr` and `resultexpr` are Expressions.

Iterative or recursive variables can use Expression self reference.

Possible values for `basetype` are `number`, `string` or `datetime`.

It's the user's responsibility to use expressions valid for the base type. Failing that, the result value will be an apppropriate error message.

Note that the `baseexpr` attribute is an alias for `value`.

### Reset on break

A variable may reset on break boundaries to the `baseexpr` value. See Break node and Report breaks

```
<Variable resetonbreak="break1" />
```

Default is unset, i.e. no reset on a break.

### Precalculate (delayed)

A variable may work two ways. One is to generate an immediate value that is either derived from the current row, or derived from the current and previous rows. Say, for a variable of the `average` type, the value of the variable for the 5th row is the average value of the base expression of the first 5 rows.

The other way is Precalculated variables, that are derived from either the last data row or all rows, and supply that value for every data row. The attribute is accepted under two names:

```
<Variable precalculate="yes" />
<Variable delayed="yes" />
```

Default is `no`.

`precalculate="yes"` (or `delayed="yes"` may be combined with `resetonbreak="..."`. In this case, a precalculated value is computed for the break periods separately.

Precalculated variables may be used to show values in `<ReportHeader>`, `<ReportFooter>`, `<FieldHeaders>`, `<FieldDetails>`, `<BreakHeader>` and `<BreakFooter>` but not in `<PageHeader>` and `<PageFooter>`. The reason for this is that the former headers, footers and details are in `<Report>` scope with a query. On the other hand, the latter two are in `<Part>` scope that don't know about queries.

## Breaks

This is the parent node for individual `<Break>` nodes that describe each break. See Report breaks

```
<Breaks>
    <Break ... />
    ...
</Break>
```

## Break

This node describes one <Break>.

```
<Break ... >
    <BreakHeader>
        <Output>
            ...
        </Output>
    </BreakHeader>
    <BreakFooter>
        <Output>
            ...
        </Output>
    </BreakFooter>
    <BreakFields>
        <BreakField />
        ...
    </BreakFields>
</Break>
```

### Break attributes

#### Name

The name of the break. It must be unique in the list of breaks for the parent <Report> node.

```
<Break name="break1" />
```

#### Newpage

Accepted as part of RLIB compatibility. Not implemented.

```
<Break newpage="yes" />
```

#### Header on new page

Accepted as part of RLIB compatibility. Not implemented.

```
<Break headernewpage="yes" />
```

#### Suppress blank break contents

Accepted as part of RLIB compatibility. Not implemented.

```
<Break suppressblank="yes" />
```

## Break subsections

### BreakHeader

The break header is emitted before the new data row if there's a change for the values in the set of break fields. It contains an Output node child node.

```
<BreakHeader>
    <Output>
        ...
    </Output>
</BreakHeader>
```

### BreakFooter

The break header is emitted after the previous data row if there's a change for the values in the set of break fields. It contains an Output node child node.

```
<BreakHeader>
    <Output>
        ...
    </Output>
</BreakHeader>
```

### BreakFields

The break fields node contains one or more BreakField children nodes.

```
<BreakFields>
    <BreakField ... />
    ...
</BreakFields>
```

#### *BreakField*

The break field node only has one attribute and contains no child nodes.

```
<BreakField value="..."/>
```

The sole attribute in `<BreakField>` is `<value>` where the expression watched for changes is declared. See Expressions.

There must be at least one `<BreakField>` node listed. When mode than one break fields are listed, then all of them are watched for changes.

The order in which they are listed matters for two reasons.

The primary reason is that break fields are hierarchical. The break fields listed earlier are higher in the hierarchy. If a break field earlier in the list triggers, all subsequent break fields also trigger implicitly.

The second reason is a consequence of the previous one: emitting the BreakHeaders occur in the order of the list. For logical reasons, BreakFooters are in reverse order.

# Output

The `<Output>` node is used by many previously mentioned sections. This is the generic node that described how details are displayed in reports.

## Output attributes

### Suppress

Output nodes may be suppressed as a whole.

```
<Output suppress="yes">
<Output suppress="val(m.suppress_output)">
```

If it's in a report's `<FieldDetails>` node, the suppression may depend on query data. See Detail node and Expressions.

```
<Output suppress="query1.suppress_row">
```

Default value is `false`, i.e. no suppression.

## Output subsections

### Line

A line containing text elements of varying widths. See Line node.

```
<Output>
    <Line>
        ...
    </Line>
</Output
```

### HorizontalLine

A horizontal line. See HorizontalLine node.

```
<Output>
    <HorizontalLine ... />
</Output
```

### Image

An image. See Image node.

```
<Output>
    <Image ... />
</Output
```

### Image end

Terminator for a previous image. See ImageEnd node.

```
<Output>
    <ImageEnd/>
</Output
```

## Line

A line containing text elements of varying widths.

```
<Output>
    <Line>
        ...
    </Line>
</Output
```

### Line attributes

#### Font name

The font name attribute specifies the font for the line's scope. It can be overridden by child nodes for their scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Line font_name="Arial">
<Line fontName="Arial">
```

If both forms are specified, `font_name` is used.

Default font name is what's set (in decreasing priority) in Report font name or Part font name. If none of them is set, it's `Courier`.

#### Font size

The font size attribute specifies the font size for the line's scope. It can be overridden by child nodes for their scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Line font_size="10">
<Line fontSize="10">
```

If both forms are specified, `font_size` is used.

Default font name is what's set (in decreasing priority) in Report font size or Part font size. If none of them is set, it's `12`.

#### Bold font

Whether the line elements use bold font.

```
<Line bold="yes">
```

Default is `false`.

### Italic font

Whether the line elements use italic font. It is accepted in two forms:

```
<Line italic="yes">
<Line italics="yes">
```

Default is `false`.

### Suppress

Text lines may be suppressed. If the parent `<Output>` node is in `<FieldDetails>`, the expression may be derived from a query field.

```
<Line suppress="yes">
<Line suppress="val(m.suppress_line)">
<Line suppress="query1.suppress_line">
```

Default value is `false`, i.e. no suppression.

### Text color

This color is used to render text. It's accepted with both American and British spelling.

```
<Line color="'blue'">
<Line colour="'blue'">
```

Default is `black`. See Color specification.

### Background color

This color is used to render the background rectangle under the text. It's accepted with both American and British spelling.

```
<Line bgcolor="'blue'">
<Line bgcolour="'blue'">
```

Default is `white`. See Color specification.

## Line subsections

### Text element

Two variants are accepted.

```
<Line>
    <field value="expression..." ... />
</Line>
```

and

```
<Line>
    <literal ... >Literal text</literal>
</Line>
```

See the Text element node below.

## Text element

Two variants are accepted.

```
<Line>
    <field value="expression..." ... />
</Line>
```

and

```
<Line>
    <literal ... >Literal text</literal>
</Line>
```

Neither `field` nor `literal` have child nodes, only attributes or XML values.

The two variants are interchangeable, see Text element value below.

### Text element attributes

#### Value

The value of the text element. It's accepted in two ways: in the `value` attribute or as the XML value for the `field` and `literal` nodes.

```
<Line>
    <field value="'This text'" />
    <field>This text</field>
    <literal value="'This text'" />
    <literal>This text</literal>
</Line>
```

The value is parsed as an expression from the `value` attribute. See Expressions.

On the other hand, the value is taken as a literal string from the XML value in the second and fourth examples above.

#### Delayed (precalculated) value

This setting indicates whether the value is "precalculated", i.e. the value for the last row in the data set is used, or it would be the actual value for the current row in the data set.

```
<field delayed="yes" />
<field precalculate="yes" />
```

Default is `false`.

This setting is only applicable for line elements in the Output node node in `<ReportHeader>`, `<ReportFooter>`, `<Fieldheaders>` `<FieldDetails>`, since they are the ones under the `<Report>` scope with a query.

Note that in OpenCReports, an expression may mix references to precalculated variables with non-precalculated variables and query field references. The result will use the precalculated value only for the precalculated variables. Other references will use values derived from the current row in the data set. This is an extension over RLIB.

### Format string

The format string is used to format the value to be displayed in the generated output. For example, to print a number with two decimal places:

```
<field value="3" format="'%.2d'" />
```

The format string is expected to be a string so quoting around it is needed.

See Formatting data for details.

### Width

The field width.

```
<field value="3" format="'%.2d'" width="6" />
```

Default is unset, i.e. the field width is implicitly set to the rendered width (in points) of the field's formatted value as text, using the font name and font size of the field.

If set, the width's unit depends on the Size unit attribute.

When the Size unit attribute is set to rlib, the field width is measured in number of characters.

The character width is queried from the font set by Line font name and Line font size, Report font name and Report font size, or Part font name and Part font size in decreasing order of precedence.

Text element font name and Text element font size DO NOT influence the font width used to determine the field width.

This allows using uniquely set font name and size for individual text elements, including proportional fonts.

When using different font names or font sizes for different text elements in the same <Line>, the text elements are aligned vertically so all text elements are rendered on the same typographic baseline.

### Alignment

It specifies the alignment for the text element. It works in accordance with the Text element width, i.e. the field's formatted value (as text) is aligned inside the specified field width.

```
<field value="3" format="'%.2d'"
      width="6" align="'center' /">
```

The alignment values may be left, right, center or justified.

Justified alignment is like left alignment, but for a multi-line field all lines but the last are justified. This is best used With the default word wrapping. In this case, the spacing between words is enlarged. justified and left behave the same for regular single-line fields.

Default is left.

`OpenCReports` decoupled the data from displaying it. For example, with the PDF output, the text element is not truncated to the element width. Instead, pixel perfect alignment is used together with masking the parts of the value with a bounding box. This is a deviation from RLIB where data was approximately truncated, because it was designed with monospace fonts in mind, and widths were specified in number of characters.

### Text color

This color is used to render text. It's accepted with both American and British spelling.

```
<Line color="'blue'">
<Line colour="'blue'">
```

Default is what's set for Line node, otherwise `black`. See Color specification.

### Background color

This color is used to render the background rectangle under the text. It's accepted with both American and British spelling.

```
<Line bgcolor="'blue'">
<Line bgcolour="'blue'">
```

Default is `white`. See Color specification.

### Font name

The font name attribute specifies the font for the text element's scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Part font_name="Arial">
<Part fontName="Arial">
```

If both forms are specified, `font_name` is used.

Default font name is `Courier`.

### Font size

The font size attribute specifies the font size for the text element's scope. It may be specified in two forms, the first one is the preferred name, the second is for RLIB compatibility:

```
<Part font_size="10">
<Part fontSize="10">
```

If both forms are specified, `font_size` is used.

Default font size is 12.

### Bold font

Whether the text element uses bold font. It overrides the Line bold attribute for this text element.

```
<field bold="yes" />
```

Default is what's set for Line bold attribute. `false` if both are unset.

### Italic font

Whether the text element uses italic font. It overrides the Line italic attribute for this text element. It is accepted in two forms:

```
<field italic="yes" />
<field italics="yes" />
```

Default is what's set for Line italic attribute. `false` if both are unset.

### Web link

This attribute adds a weblink to the text element. E.g. clicking on this text element in the generated PDF will open a browser with the website.

```
<field value="'This is my website'"
       link="'https://github.com/zboszor/OpenCReports'" />
```

Default is no weblink.

### Multi-line (memo) field

This attribute allows breaking long text fields to multiple lines according to the Text element width.

```
<field value="'This is a long text...'"
       width="12" memo="yes" />
```

Default is `false`, i.e. the field is rendered on a single line.

Every line of the text element (regardless if it's a memo field or not) is aligned according to Text element alignment.

### Multi-line field wrapping

This attribute allows wrapping multi-line text at characters as opposed to words.

```
<field value="'This is a long text...'"
       width="12" memo="yes" memo_wrap_chars="yes" />
```

Default is `false`, i.e. the field is wrapped at word boundaries.

In `OpenCReports`, character wrapping adds hyphenation, thanks to Pango rendering. This is a deviation from RLIB.

### Multi-line field row limit

This attribute allows limiting multi-line text with a maximum row number.

```
<field value="'This is a long text...'"
       width="12" memo="yes" memo_max_lines="20" />
```

Default is no limit.

In `OpenCReports`, character wrapping adds hyphenation, thanks to Pango rendering. This is a deviation from RLIB.

### Column number

This attribute is accepted for RLIB compatibility, but it's unused. text with a maximum row number.

```
<field value="'This is a long text...'"
       col="3" />
```

## HorizontalLine

A horizontally drawn line.

```
<Output>
    <HorizontalLine ... />
</Output
```

### HorizontalLine attributes

### Line width

The line width is measured in points, regardless of the Size unit attribute.

```
<HorizontalLine size="3" />
```

Default is `1.0`

### Indentation

Extra indentation for the line. It is measured in points, regardless of the Size unit attribute.

```
<HorizontalLine indent="15" />
```

Default is `0.0`

### Length

The line length.

```
<HorizontalLine length="150" />
```

The line length unit depends on the Size unit attribute. When set to `rlib`, it is measured in number of characters. The is influenced by HorizontalLine font size, Report font size and Part font size in decreasing order, which in turn is influenced by Report font name and Part font name in decreasing order.

When Size unit attribute is set to `points`, the line length is measured in points.

Default is unset, i.e. the line is drawn across the page width.

### Font size

An extra knob to influence line length computation. See HorizontalLine length. It is accepted in two forms, `font_size` is the preferred one, the other is accepted for RLIB compatibility:

```
<HorizontalLine font_size="14" />
<HorizontalLine fontSize="14" />
```

Default is unset, i.e. only Report font size and Part font size would contribute to the horizontal line width computation.

### Suppress

Horizontal lines may be suppressed. If the parent `<Output>` node is in `<FieldDetails>`, the expression may be derived from a query field.

```
<HorizontalLine suppress="yes /">
<HorizontalLine suppress="val(m.suppress_hline)" />
<HorizontalLine suppress="query1.suppress_hline" />
```

Default value is `false`, i.e. no suppression.

### Line color

This color is used to render text. It's accepted with both American and British spelling. For RLIB compatibility, it is also accepted as `bgcolor`, with both American and British spelling.

```
<HorizontalLine color="'blue'" />
<HorizontalLine colour="'blue'" />
<HorizontalLine bgcolor="'blue'" />
<HorizontalLine bgcolour="'blue'" />
```

Default is `black`. See Color specification.

# Image

An image to draw on the page.

```
<Output>
    <Image ... />
</Output
```

After a valid image specification, subsequent Line nodes and HorizontalLine nodes are indented by the image width in the same `<Output>` node, or until an `<ImageEnd>` node is encountered in that `<Output>` node.

## Image attributes

### File name

The file name of the image.

```
<Image value="'filename.jpg'" />
```

Default is unset. It makes the `Image` not shown.

### Suppress

The image may be suppressed.

```
<Image value="'filename.jpg'"
       suppress="yes" />
<Image value="'filename.jpg'"
       suppress="m.suppress_image" />
<Image value="'filename.jpg'"
       suppress="query1.suppress_image" />
```

Default is `false`, i.e. no suppression.

### Type

Accepted for RLIB compatibility.

```
<Image value="'filename.jpg'"
       type="'jpg'" />
```

Default is unset, i.e. autodetect.

Various image formats are supported with autodetection via `gdk-pixbuf`. SVG (Scalable Vector Graphics) is supported via `librsvg`.

### Width

Image width, measured in points regardless of the Size unit attribute.

```
<Image value="'filename.jpg'"
       width="100" />
```

Default is unset. The image would not be shown, unless both width and height are set.

### Height

Image height, measured in points regardless of the Size unit attribute.

```
<Image value="'filename.jpg'"
       height="100" />
```

Default is unset. The image would not be shown, unless both width and height are set.

## Image end

Terminator for a previous image. This node doesn't have any attributes or child nodes. Its purpose is to reset indentation caused by a previous Image node.

```
<Output>
    <ImageEnd/>
</Output
```

## Color specification

Colors may be specified by HTML notation. This contains six hexadecimal digits, representing RGB (red, green, blue) values between 0 and 255 for each color component, prefixed by the `#` character.

```
<Line bgcolor="'#ffff00'" ... >
<HorizontalLine color="'#ff00ff'" ... >
```

Colors may also be specified by hexadecimal notation. This contains six hexadecimal digits, representing RGB (red, green, blue) values between 0 and 255 for each color component, prefixed by `0x`.

```
<Line bgcolor="'0xffff00'" ... >
<HorizontalLine color="'0xff00ff'" ... >
```

Color names may also be specified by name. The following color names are supported for RLIB compatibility. Color names are matched in a case-insensitive way.


## Notes

1. https://github.com/zboszor/OpenCReports
2. https://github.com/SICOM/rlib
3. https://github.com/zboszor/OpenCReports/blob/main/opencreport.dtd
4. https://stackoverflow.com/a/46637835/290085
5. https://www.postgresql.org/docs
6. https://docs.microsoft.com/en-us/sql/odbc/microsoft-open-database-connectivity-odbc
7. https://www.unixodbc.org
8. https://www.iodbc.org
9. https://www.connectionstrings.com/
10. http://packages.qa.debian.org/libp/libpaper.html
11. http://packages.qa.debian.org/libp/libpaper.html

# Chapter 3. Expressions in OpenCReports

## Introduction

There are four data types in OpenCReports: `numeric`, `string`, `datetime` and `error`. Expressions can return any of these types.

A `numeric` expression's result is a number. It uses high numeric precision. Most functions and operators deal with numbers.

A `string` expression's result is a string. Strings can be concatenated or otherwise processed by string functions.

A `datetime` expression may store a date, a time (with or without timezone) or both. Also, it may store a time interval, e.g. `2 months` that can be added to or subtracted from another `datetime` value.

An `error` expression is a special case of strings: it stores a string literal (which is an error message) but it cannot be further processed by functions or operators. Instead, an error is propagated up from subexpressions to the final result of the expression.

## Constants

### String literals

String literals in OpenCReports can be either single or double quoted. Some examples:

```
"apple"
'apple'
"I've eaten an apple"
'This an "apple".'
```

The values of these strings are:

```
apple
apple
I've eaten an apple
This an "apple".
```

We can see how the other quoting character can be used as part of the string value.

String literals can also use BASIC language style double quoting to embed a single quoting character used for quoting the string itself:

```
'apple'''
'apple''pear'
'apple''''pear'
"apple"""
"apple""pear"
"apple""""pear"
```

The values of these strings are:

```
apple'
apple'pear
apple''pear
apple"
apple"pear
apple""pear
```

String literals can also use C language string continuation if there's at least one whitespace character (space, TAB or new line) between doubled quoting characters. String continuation can also switch quoting characters without whitespace between quoting.

```
"apple" "pear"
"apple" 'pear'
"apple"'pear'
```

The value of all these strings is:

```
applepear
```

## Numeric constants

Numeric values in OpenCReports are handled with GNU MPFR with arbitrary precision, by default 256 bits. This allows for very high precision, i.e. very long numbers. This is good for directly using e.g. arbitrary precision `numeric` type columns from PostgreSQL.

Numeric constants can be integer or fractional numbers with or without the so called *e-notation* or scientific notation. Some examples:

```
1
1.234
1e4
1e-4
1.234e-5
```

E-notation means that that number preceding the letter "e" or "E" is multiplied by ten to the power of the number after the letter "e" or "E", the latter being an integer value. The values of the above examples are:

```
1
1.234
10000
0.0001
0.00001234
```

Numbers greater than 0 and less than 1 can be written with or without the leading zero.

```
0.123
.123
```

## Boolean constants

Boolean constants evalutate to numeric constans `1` and `0`. The boolean constants are:

```
yes
no
true
false
```

### Datetime constants

There are no datetime constants per se, although expressions like `stodt('1980-06-30 16:00:00')` (i.e. function calls with constant arguments that result in a `datetime` value) implicitly turn into constants through expression optimization.

### Constant expressions

Constant expressions are ones that contain constant values (of any type) and operators or functions.

## Precalculated expressions

Due to the possibility of precalculated variables expressions, reports are actually run (and layed out) twice. The second run uses the precalculated values from the first run. Because of this, `OpenCReports` allows mixing precalculated Report variables and non-precalculated variables and subexpressions in the same expression. The result is intuitively expected.

The actual report output is generated in the second run.

## Identifiers

Expressions may reference query column names, environment variables, internal variables and user defined Report variables. These references are called *identifiers*. Their values are evaluated during the report execution.

### Identifier names

Identifiers are in the format `domain.identifier` where the domain name or the dot are optional.

OpenCReports is using UTF-8 encoding even in identifier names. Accented characters are accepted as identifiers.

Valid names for `domain` and `identifier` may start with an underscore or UTF-8 letters and may contain underscore, UTF-8 letters and numbers in subsequent characters.

### Query field identifiers

Any valid identifier is by default a query column reference, with or without the domain name. Examples:

```
field_name
field_name5
myquery1.field_name
mező_név
lekérdezés.mező_név
```

By the way, in the above example, `mező_név` means `field_name`, and `lekérdezés.mező_név` means `query.field_name` in Hungarian. The accented characters are a courtesy of UTF-8.

Query field identifiers in expressions are matched during expression resolution. If the domain name is specified, a query with that name must be present in the report, either as the main query or as a follower query. If the domain name is not specified, the field names of the main query and all the follower queries are matched with the expression.

For exceptions (and exceptions from under the exceptions!), see below.

## Special purpose identifier domains

Some domain names carry special meaning for the report.

### Environment variables

Domain `m` indicates the domain of environment variables. The nature of environment variables depend on the languange binding. In C, it's the variables in the operating environment. In PHP, they are simply global PHP variables. Example:

```
m.current_date
```

Since such a setting is controlled outside the report, and for the duration of running the report, its value cannot (or *shouldn't*) change, environment variable references are treated as constants and are optimized as constant at the start of the report execution.

Environment variables can't change during report execution in single threaded applications, *but they can in multi-threaded ones*. By optimizing environment variables into constants in expressions instead of querying the environment every time the same expression is evaluated, potential data races (that may result in inconsistent results) are eliminated or reduced.

### Internal report variables

Domain `r` indicates the domain of internal report variables.

#### *Current page number*

```
r.pageno
```

The current page of the report is maintained by the report layout and is intertwined with running a report. For example, if an expression is evaluated on page 4 of the report, and happens to reference the current page number variable, then this variable will have the value 4 in the result.

#### *Total number of pages*

```
r.totpages
```

This variable carries the total number of pages in the report. Its value is maintained by the report. This variable is inherently precalculated.

#### *Line number*

```
r.lineno
```

This variable is an alias to the `rownum()` function, i.e. it is the current row (line) number in the data set.

Being an alias means that the variable is replaced by the `rownum()` function call in the grammar. Please, be aware, that functions may be overridden by user defined functions. If `rownum()` is overridden, then it will be called with zero arguments by the grammar transformation, which may or may not work for the user defined function and the variable will not work as intended.

### Detail count

```
r.detailcnt
```

This variable works similarly to the row number counter, e.g. `rownum()` or `r.lineno`, except it restarts from 1 at every `<FieldHeaders>`.

With the default behaviour of `<Breaks>` vs `<FieldHeaders>`, i.e. when `<FieldHeaders>` is printed on the top of every page, `r.detailcnt` works as a per page line count value.

When `<Report field_header_priority="low">` is used, the effect may be more emphasized because the value of this variable is reset more often.

### Field value

```
r.value
```

The report field description has an expression for its value in the form of `<field value="..." />`. It also has supplementary expressions, like the foreground and background colors, the format string, and others.

The supplementary expressions may reference the field value, without having to type out the field expression multiple times.

Using `r.value` also helps reducing the report runtime because the value expression is not computed multiple times. This is a manual optimization.

Referencing `r.value` is only possible for supplementary expressions for the same field description, i.e. other XML attributes for a `<field value="..." .../>` line. This variable cannot cross-reference other field descriptions, or anything not in the same scope. For this purpose, there are user Report variables.

### Format string value

```
r.format
```

Similarly to the field value a.k.a. `r.value`, the field description has an an optional expression for its formatting in the form of `<field format="..." />`.

This internal variable may be used by other expressions. It has very few use cases, if any. It only exists for compatibility with RLIB that has this internal variable.

### Expression self reference

```
r.self
```

This variable references the previous result of the expression. It is used in iterative expressions, like in user-defined Report variables. It can be used in any user defined expression.

### Subexpressions of user-defined variables

```
r.baseexpr
r.intermedexpr
r.intermed2expr
```

These variables are references for the three subexpressions that potentially make up a user-defined custom variable. The expressions in order are: base expression, intermediary expressions one and two. Their evaluation order is the same as in the order they are mentioned here.

Actually, there's a fourth subexpression that exists in every user defined variable, namely the result expression. It's reference is simply the user variable reference, see User defined variables. See also Custom variable attributes.

For example, a running average over a data series needs two intermediary expressions: one for the sum of the values, the other for the number of values in the series. The result is the sum of values divided by the number of values.

Their usage is only valid when declaring a custom user defined variable.

### User defined variables

Domain `v` signifies user defined report variables, which can be used in breaks or to shortcut expressions. Example:

```
v.my_variable
```

### Quoted and dot-prefixed identifiers

Both `domain` and `identifier` names may be quoted or unquoted. Quoting names allow using semi-reserved and reserved words as identifiers and also allow special characters in identifier names. Examples:

```
query.field_name1
query."field_name2"
query."field with space in the name"
"query2".field_name3
"query2"."and"
```

### Dot-prefixed identifiers

A dot-prefixed identifier is one where the domain name is not specified, but the identifier name is prefixed with a dot. Examples:

```
.field_name
."field_name"
```

Semi-reserved words are the boolean constants. They can be used as identifiers with dot-prefixed identifier names without a domain name and without quoting:

```
.yes
.no
.true
.false
yes.no
```

The above unquoted identifiers are equivalent with these quoted ones below:

```
."yes"
."no"
."true"
."false"
"yes"."no"
```

Operator names are reserved words, e.g. `and` and `or`. They cannot be used with dot-prefixed operator names without quoting, as it would cause an expression syntax error. But they can be used as quoted identifiers, in case you would want to use such a query name and column name:

```
."and"
."or"
"and"."or"
```

### Quoted special purpose identifier domains

When identifier domains are quoted, they lose their special meaning and the identifiers become query field identifiers. Of course, in this case, such a query name must exist and the query must have a field name specified in the identifier. Examples:

```
"m".current_date
"r".totpages
"v".my_variable
```

## Operators and functions

OpenCReports expressions can use several operators and functions. The operator precedence is mostly as expected from the C programming language. One notable exception is implicit multiplication. The precedence classes are as below, in increasing order of precedence.

### Ternary operator

The ternary operator is as in the C language:

```
expression1 ? expression2 : expression3
```

It's evaluated as follows: if the value of numeric `expression1` is `true` (i.e. non-zero), then the result is the `expression2`, otherwise it's `expression3`. Type of `expression2` and `expression3` may differ, i.e. the result type will be the type of the underlying expression but it can result in runtime errors.

### Logical operators with two operands

Logical OR can be written as `||` or `or`. Example: `a || b`

Logical AND can be written as `&&` or `and`. Logical AND has precedence over OR. Example: `a && b`

### Bitwise operators with two operands

The bitwise operators in this precedence class and in their increasing order of precedence are: bitwise OR (`|`) and bitwise AND (`&`).

### Equality and inequality comparison operators

The equality comparison operator can be written as `=` or `==`.

The inequality comparison operator can be written as `<>` or `!=`.

### Other comparison operators

Less-than (`<`), less-or-equal (`<=`), greater-than (`>`) and greater-or-equal (`>=`).

### Bitwise shifts

Bitwise shift left (`a >> b`) and bitwise shift right (`a << b`).

### Addition and subtraction

`a + b` and `a - b`.

### Multiplication, division and modulo (remainder)

`a * b`, `a / b` and `a % b`.

### Power-of operator

`a ^ b` works as a-to-the-power-of-b.

### Factorial operator

`a!`, the '!' sign used as postfix operator.

### Unary plus and minus, logical and bitwise NOT, prefix increment and decrement

Unary plus (`+a`), unary minus (`-a`), logical NOT (`!a`, '!' used as prefix operator), bitwise NOT (`~a`), prefix increment (`++a`) and prefix decrement (`--a`).

### Postfix increment and decrement

Postfix increment (`a++`) and decrement (`a--`).

### Function calls and implicit multiplication

Function calls execute a function on operands: `function(operand[, ...])`. A function name is a single word known by OpenCReports at the time of parsing, either as a built-in function, or a user-supplied one. The function name cannot have a leading dot or be a domain-qualified identifier.

Implicit multiplication is when two distinct operands are in juxtaposition, in other words they are written side by side without any whitespace. In this case, there is an implied multiplication between them that acts with higher precedence than regular multiplication or division. Implicit multiplication is applicable in these situations:

- A numeric constant juxtaposed with an identifier, the numeric constant is the on the left side.

```
2x
```

- A numeric constant juxtaposed with an expression inside parentheses. The constant can be on either side of the expression.

  ```
  2(a+b)
  (a+b)2
  ```

- An identifier juxtaposed with an expression inside parentheses, the identifier is on the left side of the expression.

  ```
  x(a+b)
  ```

  This is only treated as implicit multiplication if the identifier name is not known as a function name at the time of parsing and there is a single expression inside the parentheses. No expressions, a single expression with a known function name as the identifier, or a series of comma delimited series of expressions are treated as a function call and the function call validity is checked against the number of operands, with a potential parser error. If there's an ambiguity between function names and identifiers provided by data sources, it can be avoided by using dot-prefixed or dot-prefixed and quoted identifiers, or fully qualified identifiers in the form of `query.identifier`.

- An expression inside parentheses juxtaposed with an identifier on the right side.

  ```
  (a+b)a
  ```

- Two expressions inside parentheses juxtaposed with each other.

  ```
  (a+b)(c+d)
  ```

Implicit multiplication is NOT applicable in these situations, besides the exceptions already explained above:

- An identifier juxtaposed with a numeric constant, the numeric constant is the on the right side.

  ```
  x2
  ```

  Since an identifier name may include digits as the second and subsequent characters, the numeric constant, or at least its integer part is simply recognized as part of the identifier name itself according to the token matching. This can also result in syntax errors when not handled with care.

- An identifier juxtaposed with another identifier.

  ```
  ab
  ```

  The reason is the same as in the preceding case: there is only a single identifier according to token matching.

## Parentheses

Parenthesized expressions are always computed first.

## Token matching, precendence and syntax errors

Expression parsing works on two levels: token matching (the job of Flex) and applying grammar (the job of Bison). Token matching breaks up the expression string into tokens in a greedy way: without whitepace delimiters, the longest possible token is chosen.

This may lead to slight confusion when coupled with implicit multiplication. For example, the expression `2e-1e` is broken up into two tokens: `2e-1` juxtaposed with `e`. The first token is interpreted as a numeric constant using *e-notation* (so that it will mean `2 * 10^(-1)`) and the second is the identifier `e`, leading to the meaning `0.2 * e`. This is unambiguous for the computer, but can be somewhat

confusing to the the user reading or writing expressions. To avoid any confusion, don't use implicit multiplication and use whitespace and parentheses gratituously.

Expression parsing handles precedence and whitespaces. For example, these below do not mean the same thing:

```
a++ + ++b
a+++++b
```

The former is obvious, but the latter may be a little surprising: `(a++)++ + b`, but not when considering precedence and the Flex lexer behaviour to match the the longest known token first. In this case, to make the expression unambiguous, whitespace or parenthesis should be used. Another ambiguous example:

```
a++b
```

The above may be interpreted as `a + +b` but since no whitespace is used, Flex is free to interpret it as `a++ b`, because `++` is longer than `+`, so the former is matched first as an operator token. This is a syntax error and expression parsing throws an error for it.

## Alphabetical list of functions

Most functions below operate in this way, unless noted otherwise:

- numeric and bitwise functions with more than two operands take their first operand and perform the same operation using the second, third, etc. operands repeatedly.
- if any of the operands is an error (resulting from runtime processing of a subexpression), then the result will use the exact error of the first operand that is an error.
- if any of the operands is NULL (e.g. the data source is SQL and the field value is SQL NULL) then the result will also be NULL.
- Boolean logic functions treat their operands with 0 being false and anything else (even fractions less than 0.5) as true.
- Bitwise functions treat their operands as 64-bit numeric values, with rounding if they are fractions.
- String arithmetics operate on UTF-8 encoded strings and count in number of UTF-8 characters instead of byte length.

### abs()

Absolute value. Operator `|...|` is a shortcut for this function. It takes one numeric operand.

### acos()

Arc-cosine function. It takes one numeric operand.

### add()

Addition. Operator + is a shortcut for this function. It takes two or more operands of the same type, with all of them being either numeric or string. For string operands, it is equivalent with concatenation, i.e. `concat()` below.

### and()

Bitwise AND. Operator `&` is a shortcut for this function. It takes two or more numeric operands.

### asin()

Arc-sine function. It takes one numeric operand.

### atan()

Arc-tangent function. It takes one numeric operand.

### brrownum()

Current row number of a break since it was last triggered. It takes one string operand which is the name of the break. The row number restarts from 1 at every break boundary.

### ceil()

Rounds its operand to the next higher or equal integer. It takes one numeric operands.

### chgdateof()

It takes two datetime operands. Changes the date part of the first operand to the date part of the second operand.

### chgtimeof()

It takes two datetime operands. Changes the time part of the first operand to the date part of the second operand.

### concat()

String concatenation. It takes two or more string operands.

### cos()

Cosine function. It takes one numeric operand.

### cot()

Cotangent function. It takes one numeric operand.

### csc()

Cosecant function. It takes one numeric operand.

### date()

Returns the current date. It takes zero operands.

### dateof()

It takes one datetime operand. It returns date part of the datetime operand.

### day()

It takes one datetime operand. It returns the day of month value as a number.

### dec()

Decrement by one. It takes one numeric operand. The operator `++` is the shortcut for it, either in prefix or postfix uses.

### dim()

It takes one datetime operand. Given the year and month values of the datetime, this function returns the number of days in the month. E.g. for February in a leap year, it returns 29.

### div()

Division. Operator `/` is a shortcut for this function. It takes two or more numeric operands. The way it works is: take the first operand and divide it by the second and subsequent operands in sequence.

### dtos()

Datetime to string. It takes one datetime operand. The date part of the datetime is formatted according to the date format of the currently set locale.

### dtosf()

Datetime to formatted string. It takes two operands: one datetime and one string. It takes the second (string) operand as a format string and formats the datetime value according to the format string. If the second operand is NULL or empty string, this function behaves like `dtos()`. Otherwise it behaves like `format()` with the operands reversed.

### eq()

Equivalence. Operator `=` and `==` are a shortcuts for this function. It takes two operands of the same type: numeric, string or datetime. The result is numeric value 1 or 0, if the two operands are equal or non-equal, respectively.

### error()

Returns an artificially generated error. It takes one string operand. The result will use the string operand's value as error message. Good for unit testing OpenCReports as done in the `ocrpt_expr_test` example.

### exp()

It takes one numeric operand and returns the exponential of the operand.

### exp10()

It takes one numeric operand and returns 10 to the power of the operand.

### exp2()

It takes one numeric operand and returns 2 to the power of the operand.

### factorial()

Factorial function. It takes one numeric operand. The postfix operator `!` is the alias for this function.

### floor()

Rounds its operand to the next lower or equal integer. It takes one numeric operands.

### fmod()

The result to the value of `x - ny` (`x` and `y` being its two numeric operands), rounded according to the report rounding mode set via `ocrpt_set_rounding_mode()`, where `n` is the integer quotient of `x` divided by `y`, `n` is rounded toward zero. It takes two numeric operands.

### format()

It takes two operands, the first operand is of any type, the second operand is a string. This function formats the first value according to the second operand as a format string. If the first operand doesn't match the expected type in the format string, an error is returned.

It an RLIB compatibility function and is a special case of the `printf()` function. See also Formatting data

### fxpval()

Compatibility function for RLIB. It takes two operands. The type of the first operand may be string containing a numeric value or numeric. If it's a string, then it will be converted to numeric. The type of the second operand is numeric. The function divides the value of the first operand with 10 to the power of the value of the second operand. One use case is that if the data contains prices in cents, then `fxpval(data, 2)` puts the decimal separator to the correct place.

### ge()

Greater-or-equal. It takes two operands of the same type, which can be either numeric, string or datetime operands. The operator `>=` is the shortcut for this function.

### gettimeinsecs()

It takes one datetime operand. It converts the time part of the datetime to seconds elapsed from 00:00:00.

### gt()

Greater-than. It takes two operands of the same type, which can be either numeric, string or datetime operands. The operator `>` is the shortcut for this function.

### iif()

Trinary function. It takes three operands of which the first one is numeric, the second and third operands can be of any type. If the first operand is non-zero (i.e.: "true") then the result will be the value of the second operand, else it will be the third operand. The trinary operator `exp1 ? exp2 : exp3` is the shortcut for this function.

### inc()

Increment by one. It takes one numeric operand. The operator `--` is the shortcut for it, either in prefix or postfix uses.

### interval()

Convert the parameter(s) to an interval subtype of the datetime type. It takes either one string operand or six numeric operands. In the first case, the string is parsed for interval values, like `1 year` or `2 months`, etc., and sets the specific datetime part values. In the second case, the six numeric operands are the values for the datetime parts, in the order of years, months, days, hours, minutes and seconds.

### isnull()

Returns numeric 1 if the operand is NULL, 0 otherwise. It takes one operand of any type.

### land()

Boolean logic AND. Operator `&&` is a shortcut for this function. It takes two or more numeric operands that are treated as boolean logic values. The function is executed until the result is fully determined, i.e. it stops at the first false value.

### le()

Less-or-equal. It takes two operands of the same type, which can be either numeric, string or datetime operands. The operator `<=` is the shortcut for this function.

### left()

Returns the leftmost N characters of a string. It takes two operands, the first operand is the string, the second is the numeric that is handled as an integer and used to determine the returned string length.

### ln()

Alias for `log()`.

### lnot()

Boolean logic NOT. Prefix operator `!` is the shortcut for this function. It returns the negated boolean value of its operand. It takes one numeric operand.

### log()

Natural logarithm. It takes one nueric operand.

### log10()

Base-10 logarithm. It takes one nueric operand.

### log2()

Base-2 logarithm. It takes one nueric operand.

### lor()

Boolean logic OR. Operator `||` is a shortcut for this function. It takes two or more numeric operands that are treated as boolean logic values, with 0 being false and true for anything else. The function is executed until the result is determined fully, i.e. it stops at the first true value.

### lower()

Lowercase conversion. It takes one string operand.

### lt()

Less-than. It takes two operands of the same type, which can be either numeric, string or datetime operands. The operator < is the shortcut for this function.

### mid()

Return characters from the middle of the string. It takes three operands, the first operand is the string, the second and third are numeric values that are handled as an integers. The second operand is the offset to start from and the third operand is the length of the result string in UTF-8 characters. The offset is 1-based just like in BASIC, with offset value 0 being identical to 1. Negative offsets count from the right end of the string, i.e. `mid(s,-n,n)` is equivalent to `right(s,n)`.

### mod()

An alias of `remainder()`. It takes two numeric operands.

### month()

Returns the month value of a datetime. It takes one datetime operand.

### mul()

Multiplication. Operator `*` is a shortcut for this function. It takes two or more numeric operands.

### ne()

Not equal. It takes two operands of the same type, which can be either numeric, string or datetime operands. The operator `!=` and `<>` are shortcuts for it.

### not()

Bitwise NOT. Prefix operator `~` is the shortcut for this function. It returns the bit-by-bit negated value of its operand. It takes one numeric operand.

### now()

Returns the current timestamp. It is run only once during running the report and the same value is used everywhere where this function is used. Practically, it is the time of generating the report. It takes zero operands.

### null()

Generate NULL value using the type of its operand. It takes one operand of any type.

### nulldt()

Generate NULL of the datetime type. It takes no operands.

### nulln()

Generate NULL of the numeric type. It takes no operands.

### nulls()

Generate NULL of the string type. It takes no operands.

### or()

Bitwise OR. Operator `|` is a shortcut for this function. It takes two or more numeric operands.

### pow()

This function raises the first operand to the power of its second operand. It takes two numeric operands. Operator ^ is the shortcut for this function.

### printf()

This function takes one or more operands. The first operand is a string and used as the format string. Subsequent operands have to be of the expected type according to the format string, otherwise an error is returned. If everything is correct, it returns the formatted data as a string.

### proper()

This function takes one string operand. The function returns the string converted lowecase, except the first letter of the first word, which will be uppercase.

### random()

Generate a pseudo-random numeric value between 0 and 1. It takes no operands.

### remainder()

The result to the value of `x - ny` (`x` and `y` being its two numeric operands), rounded according to the report rounding mode set via `ocrpt_set_rounding_mode()`, where `n` is the integer quotient of `x` divided by `y`, `n` is rounded toward to the nearest integer. It takes two numeric operands.

### right()

Returns the rightmost N characters of a string. It takes two operands, the first operand is the string, the second is the numeric that is handled as an integer and used to determine the returned string length.

### rint()

Rounds its operand using the rounding mode set via `ocrpt_set_rounding_mode()`. It takes one numeric operand.

### round()

Rounds its operand to the nearest representable integer, rounding halfway cases away from zero. It takes one numeric operand.

### rownum()

It takes either zero operands or one string operand. If zero operands are passed, it returns the current row number of the dataset. If one string operand is passed, then it returns the current row number in the named query of the dataset. See the follower queries.

### sec()

Secant. It takes one numeric operand.

### settimeinsecs()

It takes two operands, the first operand is a datetime, the second is numeric. This function returns a datetime where the first operand's time part is changed to `N` seconds after 00:00:00, with `N` being the second operand.

### shl()

Bitwise shift left. Shifts the first operand left with the number of bits set indicated the second operand. The operand << is the shortcut for this function. It takes two numeric operands.

### shr()

Bitwise shift right. Shifts the first operand right with the number of bits indicated by the second operand. The operand >> is the shortcut for this function. It takes two numeric operands.

### sin()

Sine. It takes one numeric operand.

### sqr()

Square. It takes one numeric operand.

### sqrt()

Square root. It takes one numeric operand.

### stdwiy()

It takes one datetime operand. This function returns the ISO-8601 week number of the operand as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the new year.

### stod()

Alias for `stodt()`.

### stodt()

It takes one string operand. This function parses the string and tries to convert it to a datetime value. It is smart enough to recognize locale specific formats and standard ISO-8601 formats. Handles whole datetime, date-only and time-only values in the string.

### stodtsql()

Alias for `stodt()`.

### str()

It takes three numeric operands. The first operand is converted to a string with the specified integer and decimal numeric digits, according to the second and third operands.

### sub()

Subtraction. Operator – is a shortcut for this function. It takes two or more numeric operands.

### tan()

Tangent. It takes one numeric operand.

### timeof()

It takes one datetime operand. It returns time part of the datetime operand.

### trunc()

Rounds its operand to the next representable integer toward zero. It takes one numeric operand.

### tstod()

Alias for `stodt()`.

### uminus()

Unary minus. Changes the sign of its numeric operand from positive to negative, or vice versa. Operator unary – is the shortcut of this function. It takes one numeric operand.

### upper()

It takes one string operand. This function converts the string to uppercase.

### val()

Numeric value. If a string operand is given, it returns the converted numeric value. The value of a numeric operand is passed through as is. It takes one numeric or string operand.

### wiy()

It takes one datetime operand. This function returns the week number of the operand as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01.

### wiy1()

It takes one datetime operand. This function returns the week number of the operand as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.

### wiyo()

It takes two operands, one datetime and one numeric. This function returns the week number of the first operand as a decimal number, range 00 to 53, starting with the specified day number as the first day. (0 = Sunday, 1 = Monday, 2 = Tuesday, ...)

### xor()

Bitwise exclusive OR. It takes two or more numeric operands.

### year()

It takes one datetime operand. This function returns the year value of the operand as a numeric value.

# Chapter 4. Report breaks

## Grouping data

`OpenCReports`, being a report generator, works on tabular data: the data consists of ordered (named) columns and ordered or unordered rows.

It is often necessary to group data by certain properties. Imagine a list of employees of a company, grouped by their departments, pay grade, or location of employment. A report may show the list of the employees with visual separation according to any of these properties.

Multiple groupings may be prioritized (nested):

1. by department
2. by pay grade

With the above, in each department, subgrouping would separate employees according to the pay grade in that department.

For this to work, the rows of data must be fed to the report generator in a certain order. For example, in SQL the ordering can be done by:

```
SELECT ...
ORDER BY department,paygrade,employee;
```

This grouping of data is called a *break* in a report generator.

## Report breaks in OpenCReports

Expressions can reference data via the column names of a row. Arbitrary expressions may be used to watch for changes in the value of the expression data breaks. Breaks occurs on the boundary of changes in the expression value.

Prioritization (nesting) of breaks is done according their order of declaration. See the Breaks and Break node nodes.

Visual separation is optionally helped with break headers and footers. See BreakHeader and BreakFooter.

### Example

This XML part below shows a complete example of nested breaks based on the above mentioned real life example.

```
<Report>

    <Breaks>

        <Break>

            <BreakHeader>
                <Output>
                    <Line>
                        <field value="query1.department" />
                    </Line>
                </Output>
            </BreakHeader>

            <BreakFooter>
                <Output>
                    <Line>
                        <literal>End of </literal>
                        <field value="query1.department" />
                    </Line>
```

```
                    </Output>
                </BreakFooter>

                <BreakFields>
                    <BreakField value="query1.department" />
                </BreakFields>

            </Break>

            <Break>

                <BreakHeader>
                    <Output>
                        <Line>
                            <literal width="30" />
                            <field value="query1.paygrade" />
                        </Line>
                    </Output>
                </BreakHeader>

                <BreakFooter>
                    <Output>
                        <Line>
                            <literal width="30" />
                            <literal>End of </literal>
                            <field value="query1.paygrade" />
                        </Line>
                    </Output>
                </BreakFooter>

                <BreakFields>
                    <BreakField value="query1.paygrade" />
                </BreakFields>

            </Break>

        </Breaks>

        <Detail>

            <FieldHeaders>
                <Output>
                    <Line>
                        <literal width="60" />
                        <literal>Employee name</literal>
                    </Line>
                </Output>
            </FieldHeaders>

            <FieldDetails>
                <Output>
                    <Line>
                        <literal width="60" />
                        <field value="query1.employee" />
                    </Line>
                </Output>
            </FieldDetails>

        <Detail>

    </Report>
```

Assuming that Size unit attribute is set to `points`, the indentation would be 30 and 60 points for certain elements (see the empty `<literal>`s) and the result would look like this:

1. Before the first row on every page, the contents of `<FieldHeaders>` is printed.

2. Before the first row, the contents of `<BreakHeader>` is printed for every break declared in the `<Report>` in the order of their declaration.
3. The contents of `<FieldDetails>` is printed for the current row. Repeat until a value change is observed between adjacent rows for a break's expression. In this case, the employees are printed in one block that are in the current paygrade category and working at the current department.
4. When a value change happened between adjacent rows for a break's expression, then this break and every break declared after it triggers. For every triggering breaks, their `<BreakFooter>` is printed in the reverse order of their declaration. This is done using the *previous row*, so if any data used from the row or derived from it (e.g. a variable) and is to be displayed in the footer, it will be valid for the break period that just ended.
5. Before the new row, the contents of `<BreakHeader>` is printed for every break that just triggered. For example, the department's name is not printed if only the paygrade category changed in the same department from the one row to the next.
6. Repeat from step 3 until there are no more data rows.

# Chapter 5. Report variables

## Introduction to report variables

Variables are named aliases for Expressions.

In `OpenCReports`, there are a few variable types:

- freeform expression variables
- pre-defined numeric operations for simple statistics, like summing, counting, or averaging a data series, or finding the highest or lowest values in a data series
- custom variables where the data type and the operation on the data is completely user-defined

Variables may be reset at break boundaries. See Report breaks and the Reset on break attribute.

## Expression variables

For basic expressions, the value of a variable is calculated from the aliased expression using the current row of data from the data source. This can be thought of as a kind of shortcut. A variable calculates the value of a long or often used expression, and the variable result may in turn be used in other expressions. This can save both typing (in the report XML) and report execution time during the report generation.

Here's a complete example of using a variable:

```
<Report>
    <Variables>
        <Variable
            name="var1"
            value="query1.field1 + query2.field2"
            type="expression" />
    </Variables>

    <Detail>
        <FieldHeaders>
            <literal value="'My variable'" />
        </FieldHeaders>

        <FieldDetails>
            <field value="v.var1" />
        </FieldDetails>
    </Detail>
</Report>
```

Note, that in this simple example, there is no difference if the variable is used in the `<field>` or the `query1.field1 + query2.field2` expression. The efficiency of not computing the variable again for the same data row can be observed when the variable is used multiple times and the report processes a huge data set.

### Variables with iterative expressions

An expression may be iterative, where the new value is derived from the previous value of itself. See Expression self reference.

Here's a complete example of using a variable:

```
<Report>
```

```
    <Variables>
        <Variable
            name="var1"
            value="r.self + query1.field1 + query2.field2"
            type="expression" />
    </Variables>

    <Detail>
        <FieldHeaders>
            <literal value="'My variable'" />
        </FieldHeaders>

        <FieldDetails>
            <field value="v.var1" />
        </FieldDetails>
    </Detail>
</Report>
```

The trick is to use the `r.self` internal variable.

Please note, that the above example will likely not work as is, because for the first row, *there is no previous row*. But there is a trick to avoid such problems, namely using the trinary operator (or its equivalent, the `iif()` function) and the `rownum()` to perform only safe computations. (Note that the `value=...` part below is a single line.)

```
<Variable>
    ...
    value="rownum() == 1 ?
           query1.field1 + query2.field2 :
           r.self + query1.field1 + query2.field2"
    ...
</Variable>
```

This example shows the correct operation of an iterative expression. For the first row, set a known good value. For every subsequent rows, the previous row value may be used for deriving the new value from.

## Variable types for simple statistics

There are pre-defined variable types for performing simple statistic calculations. All of them (except data series counting) operate on numeric values.

### Summing a data series

Summing is done via the `sum` variable type. For example, the above spelled out example can be written as:

```
<Report>
    <Variables>
        <Variable
            name="var1"
            value="query1.field1 + query2.field2"
            type="sum" />
    </Variables>

    <Detail>
        <FieldHeaders>
            <literal value="'My variable'" />
        </FieldHeaders>

        <FieldDetails>
            <field value="v.var1" />
```

```
        </FieldDetails>
    </Detail>
</Report>
```

The iterative nature is implicit for the variable's result.

## Counting data in a series

Counting is done via the `count` and `countall` variable types. The difference between the two is that plain `count` does not count NULL data, while `countall` does. It's equivalent to the difference between `COUNT(query1.field1)` and `COUNT(*) SQL`. (The former doesn't count NULL values, the latter does.)

```
<Report>
    <Variables>
        <Variable
            name="var1"
            value="query1.field1"
            type="count" />

        <Variable
            name="var2"
            value="query1.field1"
            type="countall" />
    </Variables>
</Report>
```

## Averaging data in a series

Averaging uses two running expressions behind the scenes. One is the `sum` of data, the other is the `count` of data. The sum is divided by the count.

Here, two different calculation is possible again, depending on which counting mething is used, see above. NULL data contributes 0 to the sum, but the count (the denominator in the division) may differ. The result depends on this detail.

For this reason, `average` and `averageall` both exist.

```
<Report>
    <Variables>
        <Variable
            name="var1"
            value="query1.field1"
            type="average" />

        <Variable
            name="var2"
            value="query1.field1"
            type="averageall" />
    </Variables>
</Report>
```

## Highest and lowest values of a series

Finding the highest and lowest values in a data series is done by the `highest` and the `lowest` variable types.

```
<Report>
    <Variables>
        <Variable
```

```
            name="var1"
            value="query1.field1"
            type="highest" />

        <Variable
            name="var2"
            value="query1.field1"
            type="lowest" />
    </Variables>
</Report>
```

NULL values don't contribute to the results of either variable types, so in an all-NULL series, each variable will give a NULL result, i.e. empty when displayed.

## Custom variables

As seen in Expression variables above, variables are not mysterious. They can be iterative or non-iterative and their operation can be spelled out. On the other hand, the pre-defined variables for doing simple statistics may be limiting. Maybe we need an iteratively calculated value that uses a different type than `number`. This is where `custom` variables may be useful.

For a `custom` variable, all details can be freely defined:

- the base type: `numeric`, `string` or `datetime`; `number` is also accepted as an alias for `numeric`
- the base expression
- two intermediary expressions that both may use the base expression's result, and the second intermediary may also use the first one's result
- the result expression that may use all three expressions' results

For example, the `average` variable works this way behind the scenes as written below.

```
<Report>
    <Variables>
        <Variable
            name="averagevar1"
            type="custom"
            baseexpr="query1.field1"
            intermedexpr="(rownum() == 1 ? 0 : r.self) +
                          (isnull(r.baseexpr) ?
                              0 : r.baseexpr)"
            intermed2expr="r.self +
                           (isnull(r.baseexpr) ? 0 : 1)"
            resultexpr="r.intermedexpr / r.intermed2expr"
            />
    </Variables>
</Report>
```

## Precalculated variables

By default, variables produce results that are valid for the data rows they are derived from. Iterative variables variable produce results that are valid for the current row and preceding rows.

But usually, we are not interested in the *running average*, only in the average of the whole data series.

This is where the *precalculated* variables come in.

Setting a variable to be precalculated is done by the `precalculate="yes"` attribute. `delayed="yes"` is also accepted as an alias. Such a variable (or, actually, its value) can be displayed in e.g. `<ReportHeader>` which is shown upfront before any report details to inform the reader without looking at the last page.

```
<Report>
    <Variables>
        <Variable
            name="var1"
            value="query1.field1"
            type="average"
            precalculate="yes" />
    </Variables>

    <ReportHeader>
        <Output>
            <field value="v.var1">
        </Output>
    </ReportHeader>
</Report>
```

See also Precalculate attribute.

## Resetting a variable on break boundaries

It may also be useful to use a regular or precalculated variable that only considers data rows in break periods. For example printing a running average for detail rows in breaks, or printing the total average calculated for a break period in the header for that period.

For this purpose, variables may be reset on break boundaries.

```
<Report>
    <Breaks>
        <Break name="break1" ... >
            <BreaksHeader>
                <Output>
                    <field value="v.var1" />
                </Output>
            </BreaksHeader>
            <BreaksFields>
                <BreaksField value="query1.field2" />
            </BreaksFields>
        </Break>
    </Breaks>

    <Variables>
        <Variable
            name="var1"
            value="query1.field1"
            type="average"
            precalculate="yes"
            resetonbreak="'break1'" />
    </Variables>
    ...
</Report>
```

To demistify such a variable, here is the equivalent of the above using a `custom` variable. The value returned by the Break row number function automatically resets at every break boundary, so it can be used as below.

```
<Variables>
    <Variable
        name="var1"
```

```
            type="custom"
            baseexpr="query1.field1"
            intermedexpr="(brrownum('break1') == 1 ? 0 : r.self) +
                          (isnull(r.baseexpr) ?
                              0 : r.baseexpr)"
            intermed2expr="r.self +
                           (isnull(r.baseexpr) ? 0 : 1)"
            resultexpr="r.intermedexpr / r.intermed2expr"
            />
    </Variables>
```

# Chapter 6. Formatting

## Formatting functions

Formatting data can be done via the format function, the printf function and the Text element format attribute. After formatting, regardless of the data type that was formatted, the type of the result value is string. This string can be displayed in the report output or processed further as needed.

## Format strings

OpenCReports supports the same set of format strings as RLIB, with extensions. RLIB and OpenCReports support:

- legacy format strings for strings, numbers and datetime values
- "new style" format strings with `!` prefix

OpenCReports also supports a 2nd generation new style format strings with a prefix and a pair of brackets (`{}`) that embed the format strings.

## Legacy format strings

Legacy format strings are like in C, but not always identical.

### Format string for strings

To print a string, the `%s` format string can be used. Examples for Text element format attribute:

```
<field value="query1.field1"
       format="'%s'" >

field value="query1.field1"
       format="'Look, there is a %s there!'" >
```

Example expressions for format function:

```
format(query1.field1, '%s')
format(query1.field1, 'Look, there is a %s there!')
```

Example expressions for printf function:

```
printf('%s, 'query1.field1')
format('Look, there is a %s there!', query1.field1)
```

Supplementary format strings flags are supported. See printf(3)[1]

### Format string for numeric values

To print a number, the `%d` format string can be used. As opposed to the C `printf` format specifier where `%d` is used for integers, this is used for printing fractions, too. Examples for Text element format attribute:

```
<field value="query1.field1"
```

```
        format="'%.3d'" >

field value="query1.field1"
        format="'You have %.2d apples.'" >
```

The same format string can be used for the the format function and the printf function, just like in the previous examples for strings.

Supplementary format strings flags are supported. See printf(3)[2]

### Format string for datetime values

RLIB approximated `strftime()` when printing a datetime value. OpenCReports *uses* `strftime()`. See strftime(3)[3] for the complete set of format string flags.

When a datetime field didn't have an explicit format string, RLIB used the US date format to print the datetime value. In this case, OpenCReports uses the locale specific date format if the report has a locale set.

## New style format strings

RLIB supported "new style" format strings that allowed formatting numeric data as monetary values and allowed to disambiguate between format strings used for different data types. This was needed because some format flags are used in both `printf()`, `strfmon()` and `strftime()`.

### New style format string for strings

This is an extension over RLIB, which didn't have such a notion. In OpenCReports, the new style flag is prefixed with `!&`

```
<field ... format="'!&%s'" ... />
```

### New style format string for numeric data

The new style flag is prefixed with `!#`

```
<field ... format="'!#%5.3d'" ... />
```

### New style format string for monetary data

The new style flag is prefixed with `!$`

```
<field ... format="'!$%=*#150n'" ... />
```

Formatting monetary values uses `strfmon()`. See strfmon(3)[4]

To print the correct currency name, the locale must be set for the report. Only one locale can be set, so a single currency name will be used for every value using monetary formatting.

### New style format string for datetime values

The new style flag is prefixed with `!@` Formatting a datetime value uses `strftime()`.

```
<field ... format="'!@%c'" ... />

<field ... format="'!@%Y-%m-%d'" ... />
```

## Second generation new style format strings

This format string style builds upon the original new style format strings, with the addition of brackets that embed the underlying format strings.

### 2nd gen new style format string for strings

The format string format is `!&{...}`

```
<field ... format="'!&{%s}'" ... />
```

### 2nd gen new style format string for numeric data

The format string format is `!#{...}`

```
<field ... format="'!#{%5.3d}'" ... />
```

### 2nd gen new style format string for monetary data

The format string format is `!${...}`

```
<field ... format="'!${%=*#150n}'" ... />
```

Formatting monetary values uses `strfmon()`. See strfmon(3)[5]

To print the correct currency name, the locale must be set for the report. Only one locale can be set, so a single currency name will be used for every value using monetary formatting.

### 2nd gen new style format string for datetime values

The format string format is `!@{...}` Formatting a datetime value uses `strftime()`.

```
<field ... format="'!@{%c}'" ... />

<field ... format="'!@{%Y-%m-%d}'" ... />
```

### The swiss army knife of formatting

The printf function is the most versatile formatting function in OpenCReports. It does not exist in RLIB. Using the second generation format strings which makes them completely disambiguous, the `printf()` function in OpenCReports allows formatting every data type into a common result string. Example:

```
printf('You had %d %s on !@{%Y-%m-%d} '
      'and %d %s on !@{%Y-%m-%d} in your pocket.',
      6, 'apples', stodt('2022-05-01'),
      2, 'oranges', stodt('2022-05-02'))
```

The result is:

```
You had 6 apples on 2022-05-01 and 2 oranges on 2022-05-02 in
your pocket.
```

## Notes

1. https://man7.org/linux/man-pages/man3/printf.3.html
2. https://man7.org/linux/man-pages/man3/printf.3.html
3. https://man7.org/linux/man-pages/man3/strftime.3.html
4. https://man7.org/linux/man-pages/man3/strfmon.3.html
5. https://man7.org/linux/man-pages/man3/strfmon.3.html

# Chapter 7. C language API reference

## Header file

For using OpenCReports, this single header must be used:

```
#include <opencreport.h>
```

## High level C API

Example code using the high level C API where everything concerning the report (including the data source) is described in the report XML:

```
#include <opencreport.h>

int main(void) {
    opencreport *o = ocrpt_init();

    if (!ocrpt_parse_xml(o, "report.xml")) {
        printf("XML parse error\n");
        ocrpt_free(o);
        return 1;
    }

    ocrpt_set_output_format(o, OCRPT_OUTPUT_PDF);
    ocrpt_execute(o);
    ocrpt_spool(o);
    ocrpt_free(o);
    return 0;
}
```

The above code will load `report.xml`, set the output format to PDF, runs the report and prints it output on `stdout`.

### Report handler initialization

```
opencreport *ocrpt_init(void);
```

### Load a report XML description

This function loads the specified XML file into the report handler. It returns `true` for success, `false` for failure.

```
bool ocrpt_parse_xml(opencreport *o, const char *filename);
```

### Set report output format

```
enum ocrpt_format_type {
    OCRPT_OUTPUT_UNSET,
    OCRPT_OUTPUT_PDF,
    OCRPT_OUTPUT_HTML,
    OCRPT_OUTPUT_TXT,
    OCRPT_OUTPUT_CSV,
```

```
    OCRPT_OUTPUT_XML
};
typedef enum ocrpt_format_type ocrpt_format_type;

void ocrpt_set_output_format(opencreport *o,
                             ocrpt_format_type format);
```

### Run the report

This function executes the report, constructs the result in memory. It returns `true` for success, `false` for failure. It is a failure if the output format is unset.

```
bool ocrpt_execute(opencreport *o);
```

### Dump report result

Dump the report output on the program's standard output channel.

```
void ocrpt_spool(opencreport *o);
```

### Report handler destruction

Calling this function frees up the report handler structure and everything created for it, even the details that were created by the low level API.

```
void ocrpt_free(opencreport *o);
```

## Low level C API

The High level C API is also part of the low level API. The functions described below allow creating a report using program code, or simply fine tuning the report behavior by mostly using the High level C API.

### Numeric behavior related functions

#### Set numeric precision

The default is 256 bits of floating point precision.

```
void ocrpt_set_numeric_precision_bits(opencreport *o,
                                      mpfr_prec_t prec);
```

### Set rounding mode

OpenCReports uses GNU MPFR under the hood and does't hide this fact. The MPFR rounding mode constants are used as is to set the rounding behaviour. The default is `MPFR_RNDN`, round to nearest.

```
void ocrpt_set_rounding_mode(opencreport *o,
```

```
                                mpfr_rnd_t rndmode);
```

## Locale related functions

### Set report locale

Setting the locale for the report does not affect the main program or other threads. Locale setting includes the language, the country. The UTF-8 suffix is necessary. E.g.: `en_GB.UTF-8` or `de_DE.UTF-8`

```
void ocrpt_set_locale(opencreport *o,
                      const char *locale);
```

### Print monetary data in the report locale

A customized monetary printing function was implemented for the purposes of the report which MPFR doesn't provide. It is used in OpenCReports both internally and by unit tests.

```
ssize_t ocrpt_mpfr_strfmon(opencreport *o,
                           char *s, size_t maxsize,
                           const char *format, ...);
```

## Data source and query related functions

The following `enum` and `struct` types are used by OpenCReports for datasources and queries.

```
enum ocrpt_result_type {
    OCRPT_RESULT_ERROR,
    OCRPT_RESULT_STRING,
    OCRPT_RESULT_NUMBER,
    OCRPT_RESULT_DATETIME
};

struct ocrpt_datasource;
typedef struct ocrpt_datasource ocrpt_datasource;

struct ocrpt_query;
typedef struct ocrpt_query ocrpt_query;

struct ocrpt_query_result;
typedef struct ocrpt_query_result ocrpt_query_result;

struct ocrpt_input {
    void (*describe)(ocrpt_query *,
                     ocrpt_query_result **,
                     int32_t *);
    void (*rewind)(ocrpt_query *);
    bool (*next)(ocrpt_query *);
    bool (*populate_result)(ocrpt_query *);
    bool (*isdone)(ocrpt_query *);
    void (*free)(ocrpt_query *);
    bool (*set_encoding)(ocrpt_datasource *,
                         const char *);
    void (*close)(const ocrpt_datasource *);
};
```

```
typedef struct ocrpt_input ocrpt_input;
```

Data sources in this context are "mini drivers". Data source handling is implemented via `ocrpt_input` functions.

Queries are data providers over data sources. They are the actual sources of data using a specific data source. Multiple queries may use the same data source.

### Add an array datasource

Add an array datasource to the report handler. It's optional, as an array datasource called "array" is automatically added to an opencreport structure.

```
ocrpt_datasource *
ocrpt_datasource_add_array(opencreport *o,
                           const char *source_name);
```

### Add an array query

Add an "array query" to the report handler. This adds the array pointer and parameters, so the array datasource input driver can use it. The provided `array` array pointer contains `(rows + 1) * cols` number of `char *` pointers, with the first row being the column (field) names. The `types` array contains `cols` number of `enum ocrpt_result_type` elements to indicate the column data type.

```
ocrpt_query *
ocrpt_query_add_array(opencreport *o,
                      ocrpt_datasource *source,
                      const char *name,
                      const char **array,
                      int32_t rows, int32_t cols,
                      const enum ocrpt_result_type *types);
```

If the `types` pointer is `NULL`, the column values are treated as strings. This is how RLIB operated.

### Add a CSV datasource

Add a CSV datasource to the report handler.

```
ocrpt_datasource *
ocrpt_datasource_add_csv(opencreport *o,
                         const char *source_name);
```

### Add a CSV query

Add a "CSV query" to the report handler. This specifies the file name, so the CSV datasource input driver can load it.

```
ocrpt_query *
ocrpt_query_add_csv(opencreport *o,
                    ocrpt_datasource *source,
                    const char *name,
                    const char *filename,
                    const enum ocrpt_result_type *types);
```

The `types` array pointer is optional. If it is `NULL`, the column values are treated as strings. This is how RLIB operated.

### Add a JSON datasource

Add a JSON datasource to the report handler.

```
ocrpt_datasource *
ocrpt_datasource_add_json(opencreport *o,
                          const char *source_name);
```

### Add a JSON query

Add a "JSON query" to the report handler. This specifies the file name, so the JSON datasource input driver can load it.

```
ocrpt_query *
ocrpt_query_add_json(opencreport *o,
                     ocrpt_datasource *source,
                     const char *name,
                     const char *filename,
                     const enum ocrpt_result_type *types);
```

The JSON file format defined in JSON file datasource contains the way to describe the column data types, which is optional in a JSON file. The `types` array pointer may optionally supplement (or override) the column types. It is only meaningful if the JSON file itself doesn't contain the type description for the columns. If it is `NULL` (and the JSON file doesn't contain type specifiers), the column values are treated as strings. RLIB didn't have JSON input.

### Add an XML datasource

Add an XML datasource to the report handler.

```
ocrpt_datasource *
ocrpt_datasource_add_xml(opencreport *o,
                         const char *source_name);
```

### Add an XML query

Add an "XML query" to the report handler. This specifies the file name, so the XML datasource input driver can load it.

```
ocrpt_query *
ocrpt_query_add_xml(opencreport *o,
                    ocrpt_datasource *source,
                    const char *name,
                    const char *filename,
                    const enum ocrpt_result_type *types);
```

The XML file format defined in XML file datasource contains the way to describe the column data types, which is optional in an XML file. The `types` array pointer may optionally supplement (or override) the column types. It is only meaningful if the XML file itself doesn't contain the type description for the columns. If it is `NULL` (and the XML file doesn't contain type specifiers), the column values are

treated as strings. This is how RLIB operated. The OpenCReports XML input file format is RLIB compatible without the type specification part.

### Add a PostgreSQL datasource

Add a PostgreSQL datasource to the report handler.

```
ocrpt_datasource *
ocrpt_datasource_add_postgresql(opencreport *o,
                                const char *source_name,
                                const char *host,
                                const char *port,
                                const char *dbname,
                                const char *user,
                                const char *password);

ocrpt_datasource *
ocrpt_datasource_add_postgresql2(opencreport *o,
                                 const char *source_name,
                                 const char *conninfo);
```

For the parameters, see PostgreSQL database connection.

### Add a PostgreSQL query

Add a SQL query using the PostgreSQL datasource to the report handler.

```
ocrpt_query *
ocrpt_query_add_postgresql(opencreport *o,
                           ocrpt_datasource *source,
                           const char *name,
                           const char *querystr);
```

### Add a MariaDB datasource

Add a MariaDB datasource to the report handler.

```
ocrpt_datasource *
ocrpt_datasource_add_mariadb(opencreport *o,
                             const char *source_name,
                             const char *host,
                             const char *port,
                             const char *dbname,
                             const char *user,
                             const char *password,
                             const char *unix_socket);

ocrpt_datasource *
ocrpt_datasource_add_mariadb2(opencreport *o,
                              const char *source_name,
                              const char *optionfile,
                              const char *group);
```

For the parameters, see MariaDB database connection.

### Add a MariaDB query

Add an SQL query using the MariaDB datasource to the report handler.

```
ocrpt_query *
ocrpt_query_add_mariadb(opencreport *o,
                        ocrpt_datasource *source,
                        const char *name,
                        const char *querystr);
```

### Add an ODBC datasource

Add an ODBC datasource to the report handler.

```
ocrpt_datasource *
ocrpt_datasource_add_odbc(opencreport *o,
                          const char *source_name,
                          const char *dbname,
                          const char *user, const char *password);

ocrpt_datasource *
ocrpt_datasource_add_odbc2(opencreport *o,
                           const char *source_name,
                           const char *conninfo);
```

For the parameters, see ODBC database connection.

### Add an ODBC query

Add an SQL query using the ODBC datasource to the report handler.

```
ocrpt_query *
ocrpt_query_add_odbc(opencreport *o,
                     ocrpt_datasource *source,
                     const char *name,
                     const char *querystr);
```

### Find a datasource

Find the data source using its name. It returns NULL if the named data source is not found.

```
ocrpt_datasource *
ocrpt_datasource_get(opencreport *o,
                     const char *source_name);
```

### Add a custom datasource

Add a custom data source to the report handler.

```
ocrpt_datasource *
ocrpt_add_datasource(opencreport *o,
                     const char *source_name,
                     const ocrpt_input *input);
```

#### Set the encoding of a datasource

Set the encoding of a datasource in case if it's not already UTF-8, so data provided by it is automatically converted.

```
void
ocrpt_datasource_set_encoding(opencreport *o,
                              ocrpt_datasource *source,
                              const char *encoding);
```

#### Free a datasource

Free a datasource from the opencreport structure it was added to. It's not needed to be called, all datasources are automatically free with `ocrpt_free()`

```
void
ocrpt_datasource_free(opencreport *o,
                      ocrpt_datasource *source);
```

#### Find a query

Find a query using its name.

```
ocrpt_query *
ocrpt_query_get(opencreport *o,
                const char *name);
```

#### Get the current data row from a query

Create (first call) or get the `ocrpt_query_result` array from a query. Output parameter `cols` returns the number of columns in the result array. It must be re-run after ocrpt_navigate_next() since the previously returned pointer becomes invalid.

```
ocrpt_query_result *
ocrpt_query_get_result(ocrpt_query *q,
                       int32_t *cols);
```

#### Get column name

Using the `ocrpt_query_result *` result from `ocrpt_query_get_result()`, the column names can be discovered from a query.

```
const char *
ocrpt_query_result_column_name(ocrpt_query_result *qr,
                               int32_t col);
```

#### Get column data

Using the `ocrpt_query_result *` result from `ocrpt_query_get_result()`, get a pointer to the column data in its internal (hidden) representation.

```
ocrpt_result *
ocrpt_query_result_column_result(ocrpt_query_result *qr,
                                 int32_t col);
```

#### Add a follower query

Add a `follower` query to the `leader` query. The leader is the primary query and the follower will run in parallel with it until the leader runs out of rows. In case the leader has more rows than the follower, then for rows in the leader where there are no follower rows, the follower fields are set to NULL.

```
bool
ocrpt_add_query_follower(opencreport *o,
                         ocrpt_query *leader,
                         ocrpt_query *follower);
```

#### Add an N:1 follower query

Add an N:1 `follower` query to the `leader` query. The leader is the primary query and rows from the follower will be matched using the `match` expression. If there are multiple rows in the follower matching the leader row, then the leader row will be listed that many times. For rows in the leader where there are no matching rows in the follower, the follower fields are set to NULL. It is similar to `LEFT OUTER JOIN` in SQL databases. For creating an `ocrpt_expr` expression pointer, see the next section.

```
bool
ocrpt_add_query_follower_n_to_1(opencreport *o,
                                ocrpt_query *leader,
                                ocrpt_query *follower,
                                ocrpt_expr *match);
```

#### Free a query

Free a query and remove it from the report handler. It's optional. `ocrpt_free()` frees the queries added to the `opencreport` structure.

```
void
ocrpt_query_free(opencreport *o,
                 ocrpt_query *q);
```

#### Start the main query

Start query (or query set) navigation. `q` should be the primary query of the report.

```
void
ocrpt_query_navigate_start(opencreport *o,
                           ocrpt_query *q);
```

#### Navigate to the next query row

Navigate the query (or query set) to the next row. Returns `false` if there was no more rows. in which case the `ocrpt_query_result` arrays for all queries in the query set (returned by previous `ocrpt_query_get_result()` calls contain invalid data.

```
bool
ocrpt_query_navigate_next(opencreport *o,
                          ocrpt_query *q);
```

#### API specific array discovery function

For array data sources and queries, OpenCReports needs a way to to find the data array and the supplementary type identifier array. These are language specific. The below ones are the C specific ones. The override function is also provided to set a new discovery function.

```
typedef void
(*ocrpt_query_discover_func)(const char *,
                             void **,
                             const char *,
                             void **);

void
ocrpt_query_set_discover_func(ocrpt_query_discover_func func);

extern ocrpt_query_discover_func ocrpt_query_discover_array;

void
ocrpt_query_discover_array_c(const char *arrayname,
                             void **array,
                             const char *typesname,
                             void **types);
```

## Expression related functions

Expressions in OpenCReports is explained in the Expressions chapter.

### Parse an expression string

This function parses an expression string and creates an expression tree. It returns a pointer to the `ocrpt_expr` structure.

If an error occurs, it returns NULL and optionally returns the error message in `err` pointer if it's not `NULL`.

The `ocrpt_report` pointer may be valid or NULL. If valid, the expression is bound to this `ocrpt_report`, i.e. it will be freed automatically with `ocrpt_free()`.

```
ocrpt_expr *
ocrpt_expr_parse(opencreport *o,
                 ocrpt_report *r,
                 const char *str,
                 char **err);
```

### Free an expression parse tree

Free an expression parse tree. If it was bound to the passed-in `ocrpt_report`, this association is also deleted. Alternatively, the expression doesn't need to be free if it was bound to a report when it was parsed, as it will be automatically freed when freeing either the report, or the global `opencreport` structure.

```
void
ocrpt_expr_free(opencreport *o,
                ocrpt_report *r,
                ocrpt_expr *e);
```

### Resolve expression references

This function resolves variable (identifier) references in the expression. This is needed to bind query columns to expressions that use them.

```
void
ocrpt_expr_resolve(opencreport *o,
                   ocrpt_report *r,
                   ocrpt_expr *e);
```

### Optimize an expression

This function optimizes an expression so it may needs fewer computation steps during report execution.

```
void
ocrpt_expr_optimize(opencreport *o,
                    ocrpt_report *r,
                    ocrpt_expr *e);
```

### Evaluate an expression

This function evaluates the expression. It returns the expression's `ocrpt_result` result structure. The result must not be freed with `ocrpt_result_free()`. It will be done by `ocrpt_expr_free()`

For expressions with query column references, this function must be called after `ocrpt_query_navigate_next` otherwise the result is not valid.

```
ocrpt_result *
ocrpt_expr_eval(opencreport *o,
                ocrpt_report *r,
                ocrpt_expr *e);
```

### Get expression result without evaluation

This function returns the expression result if it was already evaluated. The result must not be freed with `ocrpt_result_free()`. It will be done by `ocrpt_expr_free()`. Used by unit tests.

```
ocrpt_result *
ocrpt_expr_get_result(opencreport *o,
                      ocrpt_expr *e);
```

### Print an expression tree

Print an expression tree in its processed form on the standard output. Used by unit tests.

```
void
ocrpt_expr_print(opencreport *o,
                 ocrpt_expr *e);
```

### Print an expression tree with subexpressions and their results

Print an expression tree with subexpressions and their results in its processed form on the standard output. Used by unit tests.

```
void
ocrpt_expr_result_deep_print(opencreport *o,
                             ocrpt_expr *e);
```

### Count the number of expression nodes

This function returns the number of expression nodes. Used by unit tests to validate optimizazion.

```
int32_t
ocrpt_expr_nodes(ocrpt_expr *e);
```

### Initialize expression result type

OpenCReports keeps track of the last three query rows and computes three result values for expressions for internal reasons. These functions initialize the type for either the current result or all results of the expression.

```
enum ocrpt_result_type {
    OCRPT_RESULT_ERROR,
    OCRPT_RESULT_STRING,
    OCRPT_RESULT_NUMBER,
    OCRPT_RESULT_DATETIME
};

bool ocrpt_expr_init_result(opencreport *o,
                            ocrpt_expr *e,
                            enum ocrpt_result_type type);

void ocrpt_expr_init_results(opencreport *o,
                             ocrpt_expr *e,
                             enum ocrpt_result_type type);
```

### Set an error string as expression result

```
ocrpt_result *
ocrpt_expr_make_error_result(opencreport *o,
                             ocrpt_expr *e,
                             const char *format, ...);
```

### Set start value flag for an iterative expression

Set whether the iterative expression's first value is computed from its base expression or from its result expression.

```
void
ocrpt_expr_set_iterative_start_value(ocrpt_expr *e,
                                     bool start_with_init);
```

### Get current value of an expression in base type

Get the current value of an expression in a C base type. Used by parsing report description XML files and unit tests.

```
const char *
ocrpt_expr_get_string_value(opencreport *o,
                            ocrpt_expr *e);

long
ocrpt_expr_get_long_value(opencreport *o,
                          ocrpt_expr *e);

double
ocrpt_expr_get_double_value(opencreport *o,
                            ocrpt_expr *e);
```

### Set current value of an expression in a base type

Used by unit tests.

```
void
ocrpt_expr_set_string_value(opencreport *o,
                            ocrpt_expr *e,
                            const char *s);

void
ocrpt_expr_set_long_value(opencreport *o,
                          ocrpt_expr *e,
                          long l);

void
ocrpt_expr_set_double_value(opencreport *o,
                            ocrpt_expr *e,
                            double d);
```

### Set nth value of an expression in a base type

Expressions use `OCRPT_EXPR_RESULTS` number of values. With these functions, any of them can be set. Used by unit tests.

```
void
ocrpt_expr_set_nth_result_string_value(opencreport *o,
                                       ocrpt_expr *e,
                                       int which,
                                       const char *s);

void
ocrpt_expr_set_nth_result_long_value(opencreport *o,
                                     ocrpt_expr *e,
                                     int which,
                                     long l);

void
ocrpt_expr_set_nth_result_double_value(opencreport *o,
                                       ocrpt_expr *e,
                                       int which,
                                       double d);
```

### Compare current and previous values of and expression

Compare current and previous values of and expression and return `true` if they are equal. It's used to implement Report breaks.

```
bool
ocrpt_expr_cmp_results(opencreport *o,
                       ocrpt_expr *e);
```

### Set delayed flag of an expression

```
void
ocrpt_expr_set_delayed(opencreport *o,
                       ocrpt_expr *e,
                       bool delayed);
```

### Set field expression reference for an expression

If `e` contains `r.value`, the expression `rvalue` will be used to resolve this reference.

```
void
ocrpt_expr_set_field_expr(opencreport *o,
                          ocrpt_expr *e,
                          ocrpt_expr *rvalue);
```

## Column data or expression result related functions

The internal type `ocrpt_result` holds values either for query columns or expression results.

### Create an expression result

The returned pointer must be freed with `ocrpt_result_free()`.

```
ocrpt_result *
ocrpt_result_new(void);
```

### Get expression result type

```
enum ocrpt_result_type
ocrpt_result_get_type(ocrpt_result *result);
```

### Copy an expression result

Some internal data is specific to the `opencreport` structure.

```
void
ocrpt_result_copy(opencreport *o,
                  ocrpt_result *dst,
                  ocrpt_result *src);
```

### Print an expression result

Used by unit tests.

```
void
ocrpt_result_print(ocrpt_result *r);
```

### Free an expression result

```
void
ocrpt_result_free(ocrpt_result *r);
```

### Detect whether a column result is NULL

Using the `ocrpt_result *` result from a query column or an expression, detect whether the column value is NULL.

```
bool
ocrpt_result_isnull(ocrpt_result *result);
```

### Detect whether a column result is numeric

Using the `ocrpt_result *` result from a query column or an expression, detect whether the column value is numeric.

```
bool
ocrpt_result_isnumber(ocrpt_result *result);
```

### Get the numeric value of a column result

Using the `ocrpt_result *` result from a query column or an expression, get the numeric column value. It returns NULL if the column is:

- not a numeric result
- NULL

```
mpfr_ptr
ocrpt_result_get_number(ocrpt_result *result);
```

### Detect whether a column result is string

Using the `ocrpt_result *` result from a query column or an expression, detect whether the column value is string.

```
bool
ocrpt_result_isstring(ocrpt_result *result);
```

### Get the string value of a column result

Using the `ocrpt_result *` result from a query column or an expression, get the string column value. It returns NULL if the column is

- not a string result
- NULL

```
ocrpt_string *
ocrpt_result_get_string(ocrpt_result *result);
```

### Detect whether a column result is datetime

Using the `ocrpt_result *` result from a query column or an expression, detect whether the column value is datetime.

```
bool
ocrpt_result_isdatetime(ocrpt_result *result);
```

### Get the datetime value of a column result

Using the `ocrpt_result *` result from a query column or an expression, get the datetime column value. It returns NULL if the column is

- not a datetime result
- NULL

```
const struct tm *
ocrpt_result_get_datetime(ocrpt_result *result);
```

### Detect whether a datetime column result is interval

Using the `ocrpt_result *` result from a query column or an expression, detect whether the datetime column value is interval.

```
bool
ocrpt_result_datetime_is_interval(ocrpt_result *result);
```

### Detect whether a datetime column result has valid date

Using the `ocrpt_result *` result from a query column or an expression, detect whether the datetime column value has valid date.

```
bool
ocrpt_result_datetime_is_date_valid(ocrpt_result *result);
```

### Detect whether a datetime column result has valid time

Using the `ocrpt_result *` result from a query column or an expression, detect whether the datetime column value has valid time.

```
bool
ocrpt_result_datetime_is_time_valid(ocrpt_result *result);
```

## Variable related functions

Variables can be created for a report using the API.

### Create a basic variable

Using this function, any variable type except `OCRPT_VARIABLE_CUSTOM` may be created. For a custom variable, see the next function.

```
enum ocrpt_var_type {
    OCRPT_VARIABLE_EXPRESSION,
    OCRPT_VARIABLE_COUNT,
    OCRPT_VARIABLE_COUNTALL,
    OCRPT_VARIABLE_SUM,
    OCRPT_VARIABLE_AVERAGE,
    OCRPT_VARIABLE_AVERAGEALL,
    OCRPT_VARIABLE_LOWEST,
    OCRPT_VARIABLE_HIGHEST,
    OCRPT_VARIABLE_CUSTOM
};
typedef enum ocrpt_var_type ocrpt_var_type;

ocrpt_var *
ocrpt_variable_new(opencreport *o,
                   ocrpt_report *r,
                   ocrpt_var_type type,
                   const char *name,
                   const char *expr,
                   const char *reset_on_break_name);
```

### Create a custom variable

Create a custom variable of the specified type with the specified subexpressions.

```
ocrpt_var *
ocrpt_variable_new_full(opencreport *o,
                        ocrpt_report *r,
                        enum ocrpt_result_type type,
                        const char *name,
                        const char *baseexpr,
                        const char *intermedexpr,
                        const char *intermed2expr,
                        const char *resultexpr,
                        const char *reset_on_break_name);
```

### Get subexpressions of a variable

Get subexpressions of a previously created basic or custom variable.

```
ocrpt_expr *
ocrpt_variable_baseexpr(ocrpt_var *v);

ocrpt_expr *
ocrpt_variable_intermedexpr(ocrpt_var *v);

ocrpt_expr *
ocrpt_variable_intermed2expr(ocrpt_var *v);

ocrpt_expr *
ocrpt_variable_resultexpr(ocrpt_var *v);
```

### Set precalculate flag for a variable

```
void
ocrpt_variable_set_precalculate(ocrpt_var *var,
                                bool value);
```

### Resolve a variable

Resolve subexpressions of a variable so it can be evaluated correctly.

```
void
ocrpt_variable_resolve(opencreport *o,
                       ocrpt_report *r,
                       ocrpt_var *v);
```

### Evaluate a variable

After evaluation, the result is in the expression returned by `ocrpt_variable_resultexpr()`.

```
void
ocrpt_variable_evaluate(opencreport *o,
                        ocrpt_report *r, ocrpt_var *v);
```

## Break related functions

### Create a break

Create a break. No need to free it, `ocrpt_free()` does it.

```
ocrpt_break *
ocrpt_break_new(opencreport *o,
                ocrpt_report *r,
                const char *name);
```

### Set attribute flags for a break

Set boolean attribute flags for a break, either via explicit boolean value, or from an expression. The flags are not actually used, they are for RLIB compatibility.

```
enum ocrpt_break_attr_type {
    OCRPT_BREAK_ATTR_NEWPAGE,
    OCRPT_BREAK_ATTR_HEADERNEWPAGE,
    OCRPT_BREAK_ATTR_SUPPRESSBLANK,
    OCRPT_BREAK_ATTRS_COUNT
};
typedef enum ocrpt_break_attr_type ocrpt_break_attr_type;

bool
ocrpt_break_set_attribute(opencreport *o,
                          ocrpt_report *r,
                          ocrpt_break *br,
                          const ocrpt_break_attr_type attr_type,
                          bool value);

bool
ocrpt_break_set_attribute_from_expr(opencreport *o,
                                    ocrpt_report *r,
                                    ocrpt_break *br,
                                    const ocrpt_break_attr_type attr_type,
                                    ocrpt_expr *expr);
```

### Get break using its name

Get the pointer to the break using its name.

```
ocrpt_break *
ocrpt_break_get(opencreport *o,
                ocrpt_report *r,
                const char *name);
```

### Get the name of a break

Get the name of the break using its structure pointer.

```
const char *
ocrpt_break_get_name(ocrpt_break *br);
```

### Add a watched expression to a break

```
bool
ocrpt_break_add_breakfield(opencreport *o,
                           ocrpt_report *r,
                           ocrpt_break *br,
                           ocrpt_expr *bf);
```

### Iterate over breaks of a report

Iterate over breaks of a report. The first call needs the iterator list pointer to be set to NULL.

```
ocrpt_break *
ocrpt_break_get_next(ocrpt_report *r,
                     ocrpt_list **list);
```

### Resolve and optimize break fields

```
void
ocrpt_break_resolve_fields(opencreport *o,
                           ocrpt_report *r,
                           ocrpt_break *br);
```

### Check whether the break triggers

```
bool
ocrpt_break_check_fields(opencreport *o,
                         ocrpt_report *r,
                         ocrpt_break *br);
```

### Reset variables for the break

```
void
ocrpt_break_reset_vars(opencreport *o,
                       ocrpt_report *r,
                       ocrpt_break *br);
```

## Function related functions

### Add a user defined function

Add a user defined function by specifying the name, the function pointer that contains the implementation, the number of operands (0 or greater for fixed number or operands, -1 is varying number of operands) and the function mathematical properties that help optimizing it.

```
bool
```

```
ocrpt_function_add(opencreport *o,
                   const char *fname,
                   ocrpt_function_call func,
                   int32_t n_ops,
                   bool commutative,
                   bool associative,
                   bool left_associative,
                   bool dont_optimize);
```

Adding a user defined function with a name of a pre-existing function will override it.

OpenCReports functions are called with the parameters as declared below.

```
#define OCRPT_FUNCTION_PARAMS \
    opencreport *o, \
    ocrpt_report *r OCRPT_UNUSED_PARAM, \
    ocrpt_expr *e
```

OpenCReports functions may be declared with these convenience symbols below.

```
#define OCRPT_FUNCTION(name) \
    void name(OCRPT_FUNCTION_PARAMS)

#define OCRPT_STATIC_FUNCTION(name) \
    static void name(OCRPT_FUNCTION_PARAMS)
```

The above function (`ocrpt_function_add()`) is called with a function pointer which has this type:

```
typedef void
(*ocrpt_function_call)(OCRPT_FUNCTION_PARAMS);
```

### Find a named function

```
const ocrpt_function *
ocrpt_function_get(opencreport *o,
                   const char *fname);
```

### Get number of operands for an expression (function)

In an expression tree, functions are represented as subexpressions with operands. This call may be used by OpenCReports functions to inspect whether the number of operands is in the expected range.

```
int32_t
ocrpt_expr_get_num_operands(ocrpt_expr *e);
```

### Get current value of a function operand

This function is used by OpenCReports functions internally to compute the result from its operands.

```
ocrpt_result *
ocrpt_expr_operand_get_result(opencreport *o,
                              ocrpt_expr *e,
                              int32_t opnum);
```

## Report part and report related functions

### Create a report part

```
ocrpt_part *
ocrpt_part_new(opencreport *o);
```

### Create a row in a report part

```
ocrpt_part_row *
ocrpt_part_new_row(opencreport *o,
                   ocrpt_part *p);
```

### Create a column in report part row

```
ocrpt_part_row_data *
ocrpt_part_row_new_data(opencreport *o,
                        ocrpt_part *p,
                        ocrpt_part_row *pr);
```

### Create a new report

```
ocrpt_report *
ocrpt_report_new(opencreport *o);
```

### Append the new report to a column

```
ocrpt_part *
ocrpt_part_append_report(opencreport *o,
                         ocrpt_part *p,
                         ocrpt_part_row *pr,
                         ocrpt_part_row_data *pd,
                         ocrpt_report *r);
```

### Report part related iterators

Iterators for getting report parts, part rows, columns in rows and reports in columns. Every iterator function must be called the first time with the list pointer set to NULL.

```
ocrpt_part *
ocrpt_part_get_next(opencreport *o,
                    ocrpt_list **list);

ocrpt_part_row *
ocrpt_part_row_get_next(ocrpt_part *p,
                        ocrpt_list **list);

ocrpt_part_row_data *
ocrpt_part_row_data_get_next(ocrpt_part_row *pr,
                             ocrpt_list **list);

ocrpt_report *
ocrpt_report_get_next(ocrpt_part_row_data *pd,
                      ocrpt_list **list);
```

### Set the main query for a report

Set the main query for a report either by the query structure pointer, or by name.

```
void
ocrpt_report_set_main_query(opencreport *o,
                            ocrpt_report *r,
                            const ocrpt_query *query);

void
ocrpt_report_set_main_query_by_name(opencreport *o,
                                    ocrpt_report *r,
                                    const char *query);
```

### Get the current row number of the main query

The row number starts from 1.

```
long
ocrpt_report_get_query_rownum(opencreport *o,
                              ocrpt_report *r);
```

### Resolve all report variables

```
void
ocrpt_report_resolve_variables(opencreport *o,
                               ocrpt_report *r);
```

### Evaluate all report variables

```
void
ocrpt_report_evaluate_variables(opencreport *o,
                                ocrpt_report *r);
```

### Resolve all report breaks

```
void
ocrpt_report_resolve_breaks(opencreport *o,
                            ocrpt_report *r);
```

### Resolve all report expressions

```
void
ocrpt_report_resolve_expressions(opencreport *o,
                                 ocrpt_report *r);
```

### Evaluate all report expressions

```
void
ocrpt_report_evaluate_expressions(opencreport *o,
                                  ocrpt_report *r);
```

## Callback related functions

Certain stages of the report execution can notify the application about the stage being executed or finished.

Every "add a callback" function below return `true` for success, `false` for failure.

### Add a "part added" callback

```
typedef void
(*ocrpt_part_cb)(opencreport *,
                 ocrpt_part *,
                 void *data);

bool
ocrpt_add_part_added_cb(opencreport *o,
                        ocrpt_part_cb func,
                        void *data);
```

### Add a "report added" callback

```
typedef void
(*ocrpt_report_cb)(opencreport *,
                   ocrpt_report *,
                   void *data);

bool
ocrpt_add_report_added_cb(opencreport *o,
                          ocrpt_report_cb func,
                          void *data);
```

### Add a "part iteration" callback

```
bool
ocrpt_add_part_iteration_cb(opencreport *o,
                            ocrpt_part_cb func,
                            void *data);
```

### Add an "all precalculations done" callback

```
typedef void
(*ocrpt_cb)(opencreport *,
            void *data);

bool
ocrpt_add_precalculation_done_cb(opencreport *o,
                                 ocrpt_cb func,
                                 void *data);
```

### Add a "report started" callback

```
bool
ocrpt_report_add_start_cb(opencreport *o,
                          ocrpt_report *r,
                          ocrpt_report_cb func,
                          void *data);
```

### Add a "report done" callback

```
bool
ocrpt_report_add_done_cb(opencreport *o,
                         ocrpt_report *r,
                         ocrpt_report_cb func,
                         void *data);
```

### Add a "new row" callback

```
bool
ocrpt_report_add_new_row_cb(opencreport *o,
                            ocrpt_report *r,
                            ocrpt_report_cb func,
                            void *data);
```

### Add a "report iteration done" callback

```
bool
ocrpt_report_add_iteration_cb(opencreport *o,
                              ocrpt_report *r,
                              ocrpt_report_cb func,
                              void *data);
```

### Add a "report precalculation done" callback

```
bool
ocrpt_report_add_precalculation_done_cb(opencreport *o,
                                        ocrpt_report *r,
                                        ocrpt_report_cb func,
                                        void *data);
```

### Add a "break triggers" callback

```
typedef void
(*ocrpt_break_trigger_cb)(opencreport *,
                          ocrpt_report *,
                          ocrpt_break *,
                          void *);

bool
ocrpt_break_add_trigger_cb(opencreport *o,
                           ocrpt_report *r,
                           ocrpt_break *br,
                           ocrpt_break_trigger_cb func,
                           void *data);
```

## Environment related functions

### Indirect function to get an environment variable

```
typedef ocrpt_result *
(*ocrpt_environment_query_func)(const char *);

extern ocrpt_environment_query_func
ocrpt_environment_get;
```

### Set the environment query function

```
void
ocrpt_environment_set_query_func(ocrpt_environment_query_func func);
```

### C API environment query function

```
ocrpt_result *
ocrpt_environment_get_c(const char *env);
```

## File handling related functions

### Return a canonical file path

The returned path contains only single directory separators and doesn't contains symlinks.

```
char *
ocrpt_canonicalize_path(const char *path);
```

### Add search path

Add a new directory path to the list of search paths. It's useful to find files referenced with relative path.

```
void
ocrpt_add_search_path(opencreport *o,
                      const char *path);
```

### Find a file

Find a file and return the canonicalized path to it. This function takes the search paths into account.

```
char *
ocrpt_find_file(opencreport *o,
                const char *filename);
```

## Color related functions

### Find a color by its name

The function fills in the `ocrpt_color` structure with RGB values in Cairo values (0.0 ... 1.0).

If the color name starts with `#` or `0x` or `0X` then it must be in HTML notation.

Otherwise, the color name is looked up in the color name database in a case insensitive way. If found, the passed-in ocrpt_color structure is filled with the RGB color value of that name.

If not found or the passed-in color name is NULL, depending on the the expected usage (foreground or background color), the `ocrpt_color` structure is filled with either white or black.

```
void
ocrpt_get_color(opencreport *o,
                const char *cname,
                ocrpt_color *color,
                bool bgcolor);
```

## Paper size related functions

Paper size in OpenCReports is handled via libpaper[1].

This structure is used in OpenCReports to represent paper name and size:

```
struct ocrpt_paper {
    const char *name;
    double width;
    double height;
};
typedef struct ocrpt_paper ocrpt_paper;
```

### Get the system default paper

```
const ocrpt_paper *
ocrpt_get_system_paper(void);
```

### Get the paper specified by name

```
const ocrpt_paper *
ocrpt_get_paper_by_name(const char *paper);
```

### Set the global paper

Set global paper using an `ocrpt_paper` structure. The contents of the structure is copied.

```
void
ocrpt_set_paper(opencreport *o,
                const ocrpt_paper *paper);
```

### Set global paper specified by name

Set paper for the report using a paper name. If the paper name is unknown, the system default paper is set.

```
void
ocrpt_set_paper_by_name(opencreport *o,
                        const char *paper);
```

### Get currently set global paper

```
const ocrpt_paper *
ocrpt_get_paper(opencreport *o);
```

### Iterate over paper sizes

Get the next `ocrpt_paper` structure in the iterator. For the first call, the iterator pointer must be NULL. It returns NULL when there are no more papers known to the system.

```
const ocrpt_paper *
ocrpt_paper_next(opencreport *o,
                 void **iter);
```

## Memory handling related functions

Memory handling is done through an indirection, to help with bindings (that may do their own memory handling) override the default.

### Indirect function pointers

```
typedef void *
(*ocrpt_mem_malloc_t)(size_t);

typedef void *
(*ocrpt_mem_realloc_t)(void *,
                       size_t);

typedef void *
(*ocrpt_mem_reallocarray_t)(void *,
                            size_t,
                            size_t);

typedef void
(*ocrpt_mem_free_t)(const void *);

typedef char *
(*ocrpt_mem_strdup_t)(const char *);

typedef char *
(*ocrpt_mem_strndup_t)(const char *,
                       size_t);

typedef void
(*ocrpt_mem_free_size_t)(void *,
                         size_t);
```

```
extern ocrpt_mem_malloc_t ocrpt_mem_malloc0;
extern ocrpt_mem_realloc_t ocrpt_mem_realloc0;
extern ocrpt_mem_reallocarray_t ocrpt_mem_reallocarray0;
extern ocrpt_mem_free_t ocrpt_mem_free0;
extern ocrpt_mem_strdup_t ocrpt_mem_strdup0;
extern ocrpt_mem_strndup_t ocrpt_mem_strndup0;
```

### Allocate memory

```
void *
ocrpt_mem_malloc(size_t sz);
```

### Reallocate memory

```
void *
ocrpt_mem_realloc(void *ptr,
                  size_t sz);
```

### Reallocate array of memory

```
void *
ocrpt_mem_reallocarray(void *ptr,
                       size_t nmemb,
                       size_t sz);
```

### Free memory

```
void
ocrpt_mem_free(const void *ptr);
```

### Duplicate C string

```
void *
ocrpt_mem_strdup(const char *ptr);
```

### Duplicate C string up to the specified length

```
void *
ocrpt_mem_strndup(const char *ptr,
                  size_t sz);
```

### Free a C string

It'a convenience alias for `ocrpt_mem_free()`.

```
void
ocrpt_strfree(const char *s);
```

### Set indirect allocation functions

```
void
ocrpt_mem_set_alloc_funcs(ocrpt_mem_malloc_t rmalloc,
                          ocrpt_mem_realloc_t rrealloc,
                          ocrpt_mem_reallocarray_t rreallocarray,
                          ocrpt_mem_free_t rfree,
                          ocrpt_mem_strdup_t rstrdup,
                          ocrpt_mem_strndup_t rstrndup);
```

## List related functions

These functions implement a single linked list. The list element structure is hidden:

```
struct ocrpt_list;
typedef struct ocrpt_list ocrpt_list;
```

### Get the list length

```
size_t
ocrpt_list_length(ocrpt_list *l);
```

### Make a list from one element

```
ocrpt_list *
ocrpt_makelist1(const void *data);
```

### Make a list from multiple elements

This function can be used with variable number of arguments.

```
ocrpt_list *
ocrpt_makelist(const void *data1, ...);
```

### Get the last element of a list

```
ocrpt_list *
ocrpt_list_last(const ocrpt_list *l);
```

### Get the nth element of a list

```
ocrpt_list *
ocrpt_list_nth(const ocrpt_list *l, uint32_t n);
```

### Append a new element to a list

```
ocrpt_list *
ocrpt_list_append(ocrpt_list *l,
                  const void *data);
```

### Append to list using the last element

This function make appending to the list work *O(1)* instead of *O(n)*.

```
ocrpt_list *
ocrpt_list_end_append(ocrpt_list *l,
                      ocrpt_list **e,
                      const void *data);
```

### Prepend a new element to a list

```
ocrpt_list *
ocrpt_list_prepend(ocrpt_list *l,
                   const void *data);
```

### Remove a data element from a list

```
ocrpt_list *
ocrpt_list_remove(ocrpt_list *l,
                  const void *data);
```

### Get the data element from a list

```
void *
ocrpt_list_get_data(ocrpt_list *l);
```

### Free a list

```
void
ocrpt_list_free(ocrpt_list *l);
```

### Free a list and its data elements

```
void
ocrpt_list_free_deep(ocrpt_list *l,
                     ocrpt_mem_free_t freefunc);
```

## String related functions

For memory safety and higher performance, a wrapper structure is used over C functions.

```
struct ocrpt_string {
    char *str;
    size_t allocated_len;
    size_t len;
};
typedef struct ocrpt_string ocrpt_string;
```

### Create a new string

Create a new string from a C string. The ownership of the input string may be taken over, or the original string's contents are copied.

```
ocrpt_string *
ocrpt_mem_string_new(const char *str,
                     bool copy);
```

### Create a new string with specified allocated length

Create a new string with specified allocated length so future growth can be done without reallocation. The input string is always copied.

```
ocrpt_string *
ocrpt_mem_string_new_with_len(const char *str,
                              size_t len);
```

### Create a string from a formatted string with maximum length

```
ocrpt_string *
ocrpt_mem_string_new_vnprintf(size_t len,
                              const char *format,
                              va_list va);
```

### Create a string from a formatted string

```
ocrpt_string *
ocrpt_mem_string_new_printf(const char *format, ...);
```

### Resize a string

Resize the string to the specified allocated length.

```
ocrpt_string *
ocrpt_mem_string_resize(ocrpt_string *string,
                        size_t len);
```

### Free a string

```
char *
ocrpt_mem_string_free(ocrpt_string *string,
                      bool free_str);
```

### Append a C string of the specified length to a string

```
void
ocrpt_mem_string_append_len(ocrpt_string *string,
                            const char *str,
                            const size_t len);
```

### Append a binary string of the specified length to a string

```
void
ocrpt_mem_string_append_len_binary(ocrpt_string *string,
                                   const char *str,
                                   const size_t len);
```

### Append a C string of unspecified length to a string

```
void
ocrpt_mem_string_append(ocrpt_string *string,
                        const char *str);
```

### Append a byte to a string

```
void
ocrpt_mem_string_append_c(ocrpt_string *string,
                          const char c);
```

### Append a formatted string to a string

```
void
ocrpt_mem_string_append_printf(ocrpt_string *string,
                               const char *format, ...);
```

## Notes

1. http://packages.qa.debian.org/libp/libpaper.html

# Chapter 8. GNU Free Documentation License

GNU Free Documentation License
Version 1.3, 3 November 2008



0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If

there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History"

in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license

from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See https://www.gnu.org/licenses/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c)  YEAR  YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.