MyCsv 1.0 - User Manual

Mathieu Poirier and Bastien Rousseau

29th November 2020

Contents

1 Introduction **2**

2 Some examples **2**
2.1 Hello World . . . 2
2.2 Delimiter changing . . . 2
2.3 Basic selection . . . 2
2.4 Building data from scratch . . . 2

3 Concrete syntax and semantics **2**
3.1 Core concepts . . . 3
3.1.1 CSV as unordered data . . . 3
3.1.2 A stateful semantics . . . 3
3.1.3 One CSV at a time . . . 3
3.2 Program's structure . . . 4
3.3 Load, Store, Export . . . 4
3.4 Data expressions and values . . . 5
3.4.1 Algebraic Expressions . . . 5
3.4.2 Values . . . 6
3.5 Indexes . . . 6
3.5.1 Field Indexes . . . 6
3.5.2 Line Indexes, relational and logical expression . . . 6
3.5.3 Cell Indexes . . . 7
3.6 Printing things . . . 7
3.7 Data editing . . . 8
3.7.1 Data selection . . . 8
3.7.2 Data insertion . . . 8
3.7.3 Data modification . . . 9

A Full syntax graph **9**

1 Introduction

`MyCsv` is a DSL (Domain-Specific Language) for CSV file manipulation. It works with a single working CSV simultaneously, named *current CSV* on this document.

`MyCsv` is able to do classical operations on CSV such as selection, projection, insertion or modification. It can also manipulate algebraic and logical expressions for more expressive selections.

On top of those classical data manipulation operations, it is able to print lot of things, and eventually to export CSV data to the JSON format.

2 Some examples

2.1 Hello World

As a mandatory example, `MyCsv` is able to print the famous "Hello world".

```
Print expr "Hello world!"
```

2.2 Delimiter changing

As a basic CSV manipulation oriented language, this simple `MyCsv` program changes the comma delimiter with a semicolon delimiter.

```
Load "/path/to/input.csv" sep=','
Store "relativePath/to/output.csv" sep=';'
```

2.3 Basic selection

The keywords `Select` and `Projection` allows basic data selection. The next program loads a CSV, selects lines for which the age field is more than 18, then keeps only fields `id`, `name` and `age`, and in the end stores this modified CSV in a new file.

```
Load "/path/to/input.csv"
Select age > 18
Projection id name age
Store "/path/to/output.csv"
```

2.4 Building data from scratch

If you don't have any CSV file under your hand... `MyCsv` allows you to craft one *ex-nihilo*!

```
Insert field id: 0
Insert field data1: 0
Insert field data2: 0
Insert field line: [0; Foo; Bar]
Insert field line: [1; Pof; Rad]
Insert field line: [2; Kel; Sto]
Store "/path/to/exNihilo.csv"
```

3 Concrete syntax and semantics

The following section describes the concrete syntax and the semantics of each aspect of `MyCsv` DSL. The full syntax is available in Annex A, displayed as a Syntax Graph (from XText Syntax Graph view in Eclipse). Part of the syntax is given alongside with the presentation.

3.1 Core concepts

This section presents the paradigm of `MyCsv 1.0`. What is a CSV file? What is its meaning? What should be guaranteed or not? What exists along a `MyCsv` execution?

All answers to those questions are in this section.

3.1.1 CSV as unordered data

We conceptualize data of a CSV file as a unordered set of individuals (lines) with characteristics in an arbitrary order (fields).

`MyCsv 1.0` is able to handle CSV files:

- with or without header;
- with any kind of separator;

but isn't able to handle CSV files:

- with String delimiter ;
- with multiple field sharing their name.

Because of this unordered conceptualization, the `MyCsv` language doesn't allow order manipulation.

3.1.2 A stateful semantics

During the interpretation of a `MyCsv` program, there is a state keeping track of important aspects of the execution. This state is made of four parts:

1. the *current header* ;
2. the *current data* ;
3. the *current separator* ;
4. the (implicit) *working directory* (noted *wd*).

The *header* and the *data* are referred together as the *current CSV*. Each operation, either being data modification, file loading or storing, etc, operates regarding this *current CSV*.

The *header* always exists. Whereas in a CSV file, it is optional to present a header, the *current CSV* always has a header. If none is given, one is generated. The statements allowing to load and store CSV file present an option to load or write a CSV file with no header.

On top of this *current CSV*, there is also a *current separator*, it corresponds to the last separator used to load a file, and otherwise corresponds to the comma (as we are working with CSV files, that is, files of *Comma Separated Values*).

The fourth part of the state is the usual one of *working directory* (noted *wd*). The working directory of a `MyCsv` execution is the one from which execution has been launched. There is no way to interact with the *wd* along the execution, neither modification nor evaluation/printing is possible. However the working directory as an incidence on the execution of a `MyCsv` program since the program may refer to a file through a path, and this path may be a relative one. When referring to a relative path, the corresponding file is reached from the *wd*. Note that a way to have a program not influenced by the working directory is to refer only to absolute paths.

3.1.3 One CSV at a time

In the current version of the language, there is no way to manipulate several CSV files together. This has the notable consequence to make it impossible to operate any *join operation* defined by relational algebras.

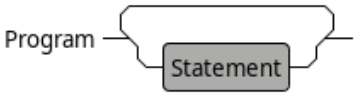

Figure 1: Program

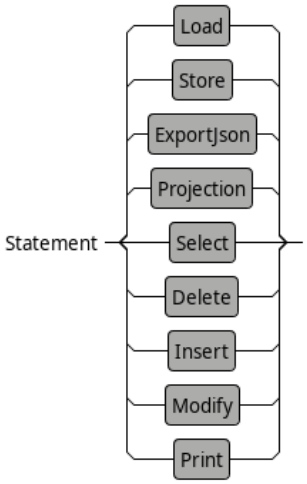

Figure 2: Statements

3.2 Program's structure

As shown in Fig.1, a `MyCsv` program is just a sequence of statements. Each statements either applies a modification on the *current CSV*, produces a printing side-effect or stores/loads a file. Be careful, no modification is saved on the disk before any Store/Export statement.

The available statements are listed in Fig.2, they are presented exhaustively in the next sections.

A common structure of some statements is the notion of *specifier*. A same command may have a specifier which precise its variant, that is, with which kind of argument it works. For example, the `Delete` command has the specifiers `field` and `line` to either delete fields or lines. The command with most specifiers is `Print`. It has as specifiers: `table`, `field`, `line`, `cell` and `expr`.

3.3 Load, Store, Export

There are three commands to manipulate disk files.

- The `Load` command loads the CSV disk file as the *current CSV*. Note that it overrides the previous *current CSV*. By default, the CSV file is supposed to have a header and to use the comma as a field delimiter. If the option `no header` is used, all lines from the file are interpreted as data, and the *current CSV* will use as header for the i-th field the name `field`$(i-1)$ (indexes start at 0, see subsection 3.5). The option `sep="`s`"` changes to s the field delimiter with which the file is interpreted. On top of overriding the *current CSV*, the Load statement override the *current separator*, either with s or with ',' otherwise.

- The `Store` command saves the *current CSV* on a CSV disk file at indicated path. By default, the CSV file is written with the *current header*, even if `no header` option was used at the previous `Load`. The default delimiter is the *current delimiter*. The same options: `no header` and `sep="`s`"` allows respectively to write a CSV with only data lines or with a custom delimiter s.

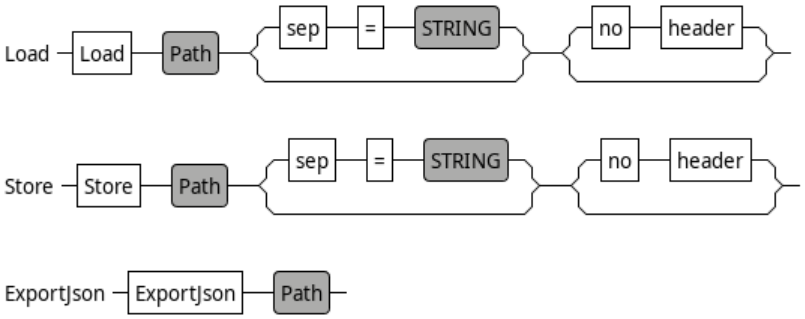

Figure 3: Load

- The `ExportJson` command simply exports the *current CSV* to a JSON file at the given path.

Fig.3 shows the syntax of theses commands.
Note that `MyCsv 1.0` doesn't support string's delimiter in CSV input nor output.

3.4 Data expressions and values

3.4.1 Algebraic Expressions

`MyCsv` supports algebraic expressions on numbers. These are classical expressions, allowing addition, subtraction, multiplication and division. Operator precedence is the usual mathematical one, that is, from most to least priority:

1. Nested expression (surrounded with parenthesis), plain numbers, aggregative operations, and built-in expression;
2. `-` (unary) ;
3. `*`, `/` ;
4. `+`, `-`.

There are no notion of type: all necessary transformations are left at the discretion of the implementation. Usually, there is a notion of integers and floating point numbers.

`MyCsv` provides some aggregative operations over a field:

- `Count` f returns the number of lines
- `Sum` f returns the sum over the field f
- `Product` f returns the product over the field f
- `Mean` f returns the mean over the field f

Each of them take a `field` name as argument and operate with the *current CSV*. Finally, `MyCsv` provides also the built-in expression `NbField` to count the number of fields in the *current CSV*.

Note: no operation on strings is supported (concatenation, etc). Aggregative operations should be computed on numbers only. We encourage implementation to raise an error when an aggregative operation is computed over a field in which data contains Strings. However we let as unspecified the behaviour for such cases.

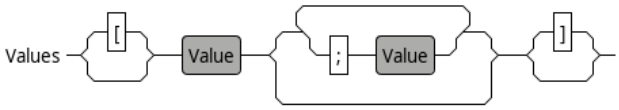

Figure 4: Values

3.4.2 Values

A `Value` is either a String (anything surrounded by double quotes: "example") or an algebraic expression.

Values are used to insert or modify lines or fields. For those operations, the user should write a list of values, separated by semi-colons, and may be surrounded by square brackets.

It is possible to have a list of Value larger (resp. shorter) than the number of fields (when working on lines) or than the number of lines (when working with fields). The semantic for those cases is that the list is truncated (resp. repeats itself from the beginning). This semantic for the length of Values is the same for both of `Insert` and `Modify` commands.

For example, with a *current CSV* having 5 fields:

- `Insert line [1;2;3;4;5;6;7;8]` will truncate the values to insert only `[1;2;3;4;5]`
- `Insert line [1;2;3]` will insert `[1;2;3;1;2]`

The reason motivating the "repeat when too short" semantics is that it allows to specify only one value to fill a whole field or line.

Syntax is given in Fig.4. The attentive reader will notice that the square brackets not only are optional, but they are separately optional. This is not a design error but a funny feature.

3.5 Indexes

This section describes how to access lines, fields and cells. For fields and lines, it is possible to access multiple of them at once. There is no redundancy: each lines (resp. fields) repeated many times are kept only once. As a consequence, it is impossible to duplicate lines (resp. fields).

Note: in `MyCsv`, all indexes start at 0. The first line of data has index 0, the first field has index 0 (and name `field0` if `no header` option was set for last loading).

Note: for all explicit indexes (that is all of them except implicit line index through logical expression), it possible that the user address lines or fields that doesn't exist. No behaviour is specified for such bad practice, but we encourage implementation to raise error of index in such cases. Detection of such errors is possible only at run-time, since we can't predict the size nor the header of input loaded.

3.5.1 Field Indexes

Fields can be indexed only in explicit manners: either by a list of numbers, or by a list of field names, as defined by the header.

3.5.2 Line Indexes, relational and logical expression

There are two ways to index lines:

- explicitly: by a list of numbers ;
- implicitly: through a condition over the lines, which indexes only the lines where the condition is true.

At the core of logical expressions, there are predicates over fields, we name them *relational expressions*. Their syntax is described in Fig.5. For a given line, they allow to compare the value of a given field with an other value (recall that the other value may be a string or a algebraic expression). Comparison predicate is among:

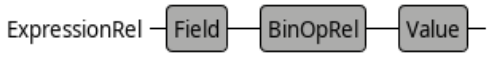

Figure 5: Relational expression

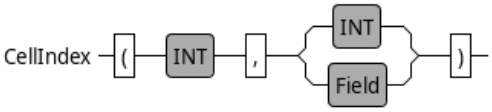

Figure 6: Cell index

- equals: ==
- differs: !=
- lower or equal: <=
- greater or equal: >=
- lower: <
- greater: >

Usual mathematical relations are used to compare numbers, lexicographic order is used to compare strings.

Note: no warranty of any kind is given regarding typing. The behavior may be unpredictable if comparison is made between Strings and numbers. We encourage `MyCsv` implementation to raise errors when such cases occur.

Given those atomic predicates, the *relational expressions*, it is possible to combine them through common propositional logical operators, with the following precedence, from most to least priority:

1. Nested expression (surrounded with parenthesis), relational expression;
2. `not` (unary) ;
3. `and` ;
4. `or`.

Such a combination is called a *logical expression*, and serve as implicit line index.

Example: `Select age>18 and not(name=="Dupond")` selects lines where the value at the field age is greater than 18 and the value at the field name is different than "Dupond".

3.5.3 Cell Indexes

Fig.6 describes how to use cell index. A cell index is basically a pair of a line number and either a field number or a field name.

3.6 Printing things

`Print` commands allows to print some information on the standard output. Fig.7 describes all the ways to print things. The syntax uses five different *specifiers*, allowing to specify what kind of thing will be printed. The necessary argument depends on the given specifier.

Note: there is no standard format to print data. The way information is displayed is left at the discretion of `MyCsv` implementation.

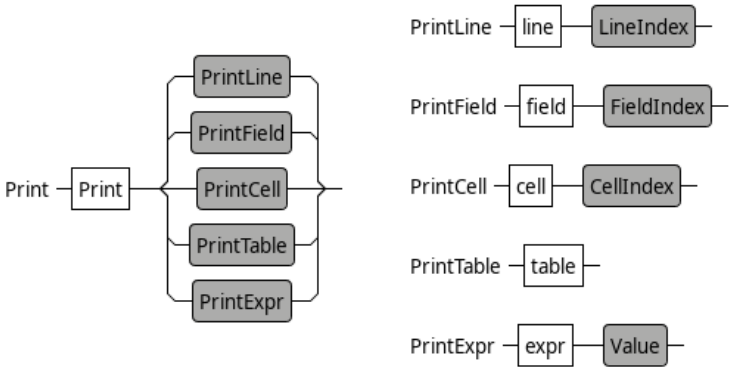

Figure 7: Print

3.7 Data editing

Since the CSV from scratch is allowed, it is not necessary the load a CSV (using a valid `Load` command) before editing data. Note that if you create data from scratch, it would be better to start with line and field insertion before any data modification or deletion. However, you may still try to modify and delete data from an empty set of data! (Note that this will be "successful" only if using implicit indexes -that is logical expression as line index-, otherwise you will need to index absent lines or fields, which is a bad behavior...).

Note: each time there is a data manipulation operation, indexing is updated to correspond to the new data disposition. This update should be done for free in implementation if data is stored in an Array-like data structure. For instance, after `Delete line 0`, the line previously indexed as `line 1` now coincide with `line 0`.

Note: for each data manipulation, the operation has a semantics of simultaneity. It may be in practice implemented sequentially, but for example `Delete line 0 1 2` should delete line 0, 1 and 2 as they are indexed at the beginning of the statement (with a sequential semantics, it would give the counter-intuitive result of deleting lines 0, 2 and 4).

3.7.1 Data selection

The is three ways to select data:

- `Delete` commands, which deletes lines or fields depending on the specifier;
- `Select` command, which selects lines ;
- `Projection` command, which selects columns.

You may think of `Select` as the dual of `Delete line`, and `Projection` as the dual of `Delete field`. Notably, the commands `Select <cond>` and `Delete line not <cond>` have the same behavior.

`MyCsv` implementation should guarantee that fields' order is never changed with the `Projection` command. `MyCsv` implementation should on the contrary guarantee that lines can be re-order with the `Select`. For example: `Select 5 4 3 2 1 0` should result in keeping the six first lines of data but reversing their order.

Note: this may be seen as a weird requirement, but it helps the user in knowing what data is at which index as long as they don't use implicit indexing.

3.7.2 Data insertion

Data can be inserted using `Insert` command. Possible specifier are `line` and `field`, allowing to insert respectively a line or a field. In both cases, a list of values is required: it uses the values grammar, explained

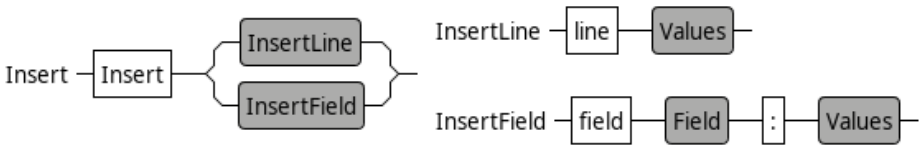

Figure 8: Insert

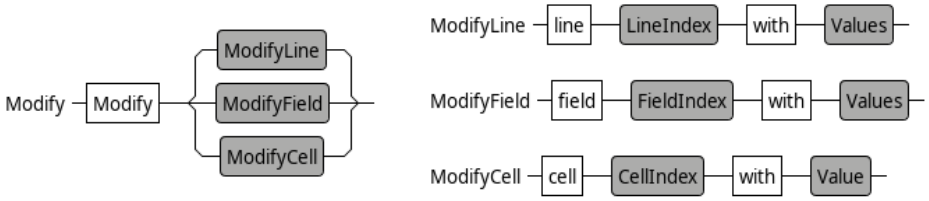

Figure 9: Modify

in Subsection 3.4.2. Syntax of data insertion is given in Fig.8.

Note: the semantics of data insertion is more like data appending. We don't allow any precise way to really *insert* data among other, the `Insert line` (resp. `Insert field`) command just add new data as the new last line (resp. field). This is coherent with our "CSV as unordered data" paradigm presented in subsection 3.1.1.

Note: as we don't handle multiple fields having the same header, behavior is not specified for the insertion of a new field with a name already existing. We encourage implementation to raise an error in such a case.

3.7.3 Data modification

The `Modify` commands change the value of the given lines, fields or cell, depending on the *specifier* used. It is possible to modify multiples lines (or fields) at the same time, in this case, each line (or fields) will contain the same values.

The `Modify` commands only modify the *data*. To modify the *header*, you may use the `Rename field` command.

Note: no warranty of behavior is specified if `Rename field` is used to rename a field with an already existing name of an other field. We encourage implementation to raise error in such a case.

A Full syntax graph

The figures 10 and 11 present the full grammar of `MyCsv`. It is presented as a Syntax Graph, as displayed by the view of the same name in Eclipse IDE.

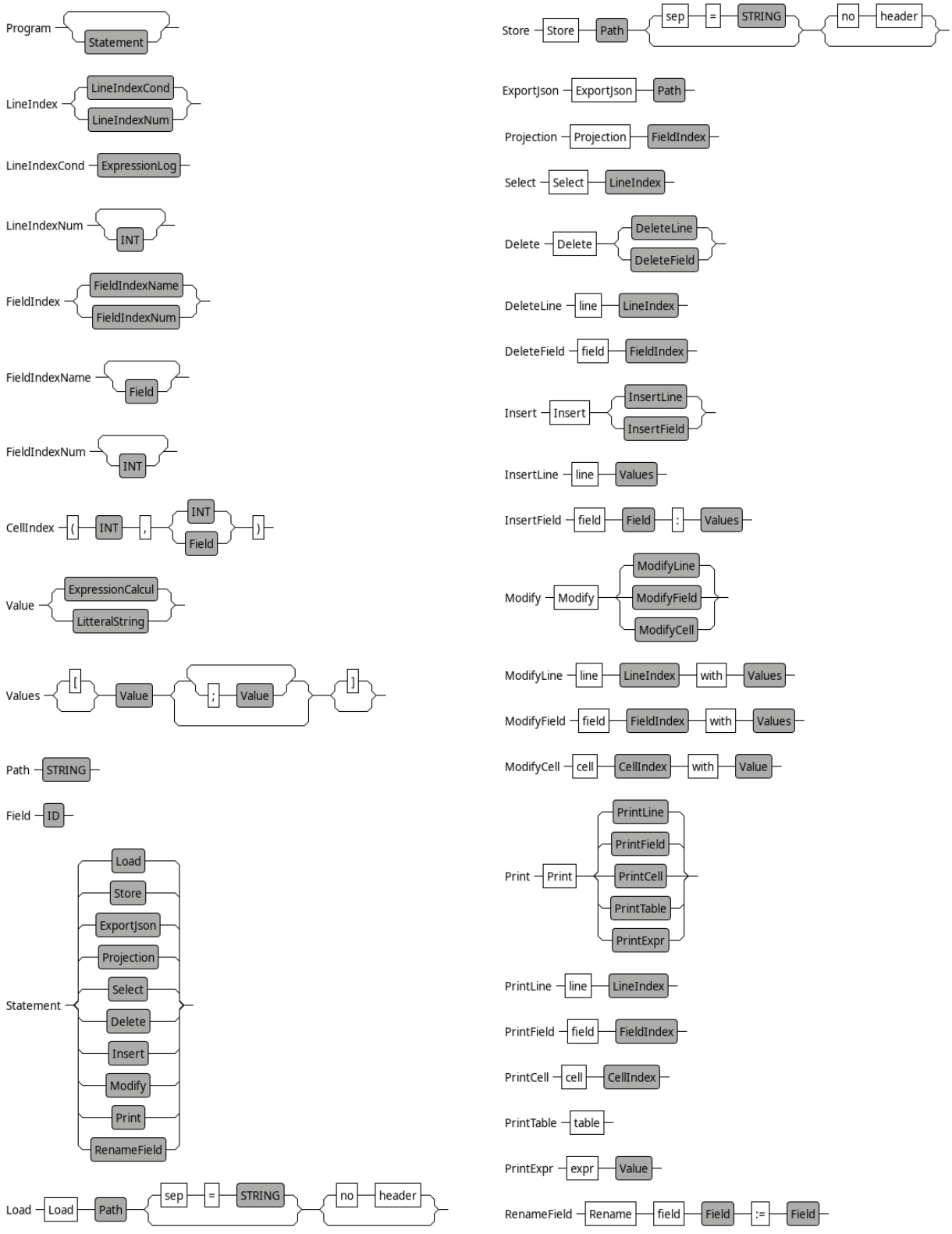

Figure 10: Full grammar - part 1

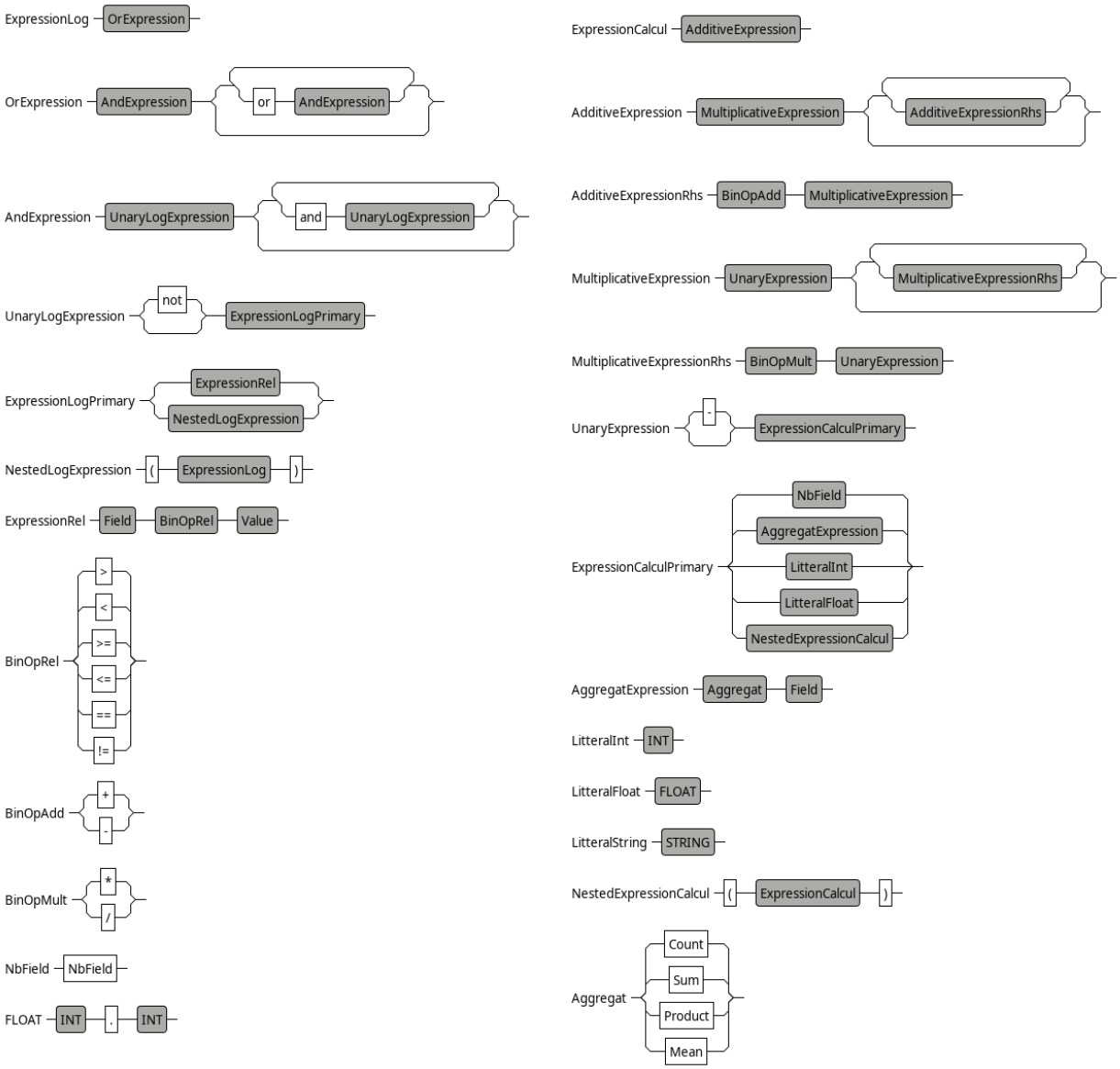

Figure 11: Full grammar - part 2