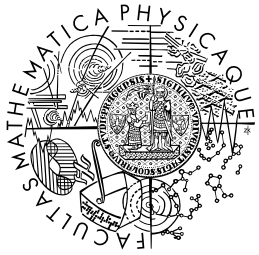

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Přemysl Šťastný

Command-line tool `lsql-csv` for CSV files processing

Department of Applied Mathematics

Supervisor of the bachelor thesis: doc. Mgr. Jan Hubička, Ph.D.
Study programme: Computer Science
Study branch: General Computer Science

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I dedicate my work to my longtime friend Pavla.

I would like to thank the people who helped me create this thesis. Jan Hubička for professional guidance and remarks, my mother, Stanislava Šťastná, for the language proofreading of the text, and Pavla Odehnalová for the persistent support and energy she gave me all the time.

Title: Command-line tool `lsql-csv` for CSV files processing

Author: Přemysl Šťastný

Department: Department of Applied Mathematics

Supervisor: doc. Mgr. Jan Hubička, Ph.D., Department of Applied Mathematics

Abstract: `lsql-csv` is a tool for small CSV file data querying from a shell with short queries. It makes it possible to work with small CSV files like with a read-only relational databases. The tool implements a new language LSQL similar to SQL, specifically designed for working with CSV files in a shell. LSQL aims to be a more lapidary language than SQL. Its design purpose is to enable its user to quickly write simple queries directly to the terminal.

Keywords: relational database, CSV, SQL, Haskell, query language, Unix philosophy, lsql-csv, LSQL

Contents

Introduction **2**

1 User documentation **3**
1.1 Installation . . . 3
1.1.1 Running the unit tests . . . 3
1.2 `lsql-csv`—quick introduction . . . 4
1.2.1 Examples . . . 4
1.3 Usage . . . 10
1.3.1 Options . . . 10
1.3.2 Datatypes . . . 11
1.3.3 Joins . . . 11
1.3.4 Documentation of language . . . 11

2 Developer documentation **21**
2.1 Project building and testing . . . 21
2.2 `String` vs `Data.Text` . . . 21
2.3 Project layout . . . 22
2.4 Modules . . . 22
2.5 Evaluation entry point . . . 23
2.6 Unit tests . . . 23

3 Analysis **25**
3.1 Why SQL in the first place? . . . 25
3.2 Why not implement just another SQL for CSV files? . . . 25
3.3 Why number references? . . . 26
3.4 Why rename standard SQL keywords? . . . 26
3.5 Why some features like descending sort are missing? . . . 26
3.6 Why blocks are delimited by commas? . . . 27
3.7 Why there are two types of expression? . . . 27
3.8 Why there is support only for cross-join and not other types of join? 27
3.9 Why there is no package used for CSV parsing and generating? . 27
3.10 Why `String` is used as primary text representation? . . . 27
3.11 Why joins have $\mathcal{O}(nm)$ complexity? . . . 27
3.12 Why are all operators right-to-left associative? . . . 28

4 Alternative Solutions **29**
4.1 Using SQL database . . . 29
4.2 Using standard Unix tools . . . 29
4.3 By using SQL implementation for CSV files . . . 30
4.4 By using general-purpose programming language . . . 30

Conclusion **31**

Bibliography **32**

Introduction

Database refers to a set of related data accessed through the use of a database management system [1]. CSV files (Comma Separated Value files) are a common way of exchanging and converting data between various spreadsheet programs [2]. Through this definition, we can see even a simple collection of CSV files accessed through some programs may be seen as a database itself.

SQL (Structured Query Language) is a language used to manage data, especially in a relational database management system [3]. It was first introduced in the 1970s [3] and is one of the most used query languages. Despite being standardized in 1986 by the American National Standards Institute [4] and in 1987 by the International Organization for Standardization [5], there are virtually no implementations that adhere to it fully [3]. Standard SQL is a typed language (every data value belongs to some data type) [6] and the language design is therefore not very suitable for type-less databases like a collection of CSV files. Despite that, there are some implementations of SQL (e.g. `q` [7], `CSV SQL` [8], `trdsql` [9], or `csvq` [10]), which tries to implement SQL on CSV files.

SQL itself requires a large amount of text to be written for running even simple queries and the Unix ecosystem misses a tool[1], that would allow running short enough queries over CSV files with similar semantics to SQL. And this is the reason, why `lsql-csv` was created.

`lsql-csv` is a tool for small CSV file data querying from a shell with short queries. It makes it possible to work with small CSV files like with a read-only relational databases. The tool implements a new language LSQL similar to SQL, specifically designed for working with CSV files in a shell.

Haskell is a language with great features for working with the text [11] and therefore it was selected for the task of implementation of `lsql-csv`.

[1] Or author does not know about it.

1. User documentation

`lsql-csv` is a tool for CSV file data querying from a shell with short queries. It makes it possible to work with small CSV files like with a read-only relational database.

The tool implements a new language LSQL similar to SQL, specifically designed for working with CSV files in a shell. LSQL aims to be a more lapidary language than SQL. Its design purpose is to enable its user to quickly write simple queries directly to the terminal—its design purpose is therefore different from SQL, where the readability of queries is more taken into account than in LSQL.

1.1 Installation

It is necessary, you had `GHC` ($\geq 8 < 9.29$) and Haskell packages `Parsec` ($\geq 3.1 < 3.2$), `Glob` ($\geq 0.10 < 0.11$), `base` ($\geq 4.9 < 4.20$), `text` ($\geq 1.2 < 2.2$) and `containers` ($\geq 0.5 < 0.8$) installed. (The package boundaries given are identical to the boundaries in Cabal package file.) For a build and an installation run:

```
make
sudo make install
```

Now the `lsql-csv` is installed in `/usr/local/bin`. If you want, you can specify `INSTALL_DIR` like:

```
sudo make INSTALL_DIR=/custom/install-folder install
```

This will install the package into `INSTALL_DIR`.

If you have installed `cabal`, you can alternatively run:

```
cabal install
```

It will also install the Haskell package dependencies for you.

The package is also published at `https://hackage.haskell.org/package/lsql-csv` in the Hackage public repository. You can therefore also install it directly without the repository cloned with:

```
cabal install lsql-csv
```

1.1.1 Running the unit tests

If you want to verify, that the package has been compiled correctly, it is possible to test it by running:

```
make test
```

This will run all tests for you.

1.2 `lsql-csv`—quick introduction

1.2.1 Examples

One way to learn a new programming language is by understanding concrete examples of its usage. The following examples are written explicitly for the purpose of teaching a reader, how to use the tool `lsql-csv` by showing him many examples of its usage.

The following examples might not be enough for readers, who don't know enough Unix/Linux scripting. If this is the case, please consider learning Unix/Linux scripting first before LSQL.

It is also advantageous to know SQL.

The following examples will be mainly about parsing of `/etc/passwd` and parsing of `/etc/group`. To make example reading more comfortable, we have added `/etc/passwd` and `/etc/group` column descriptions from man pages to the text.

File `/etc/passwd` has the following columns [12]:

1. login name;
2. optional encrypted password;
3. numerical user ID;
4. numerical group ID;
5. user name or comment field;
6. user home directory;
7. optional user command interpreter.

File `/etc/group` has the following columns [13]:

1. group name;
2. password;
3. numerical group ID;
4. user list separated by commas.

Hello World

```
lsql-csv '-, &1.2 &1.1'
```

This will print the second (`&1.2`) and the first column (`&1.1`) of a CSV file on the standard input. If you know SQL, you can read it like `SELECT S.second, S.first FROM stdio S;`.

Commands are split by commas into blocks. The first block is (and always is) the from block. There are file names or `-` (the standard input) separated by whitespaces. The second block in the example is the select block, also separated by whitespaces.

For example:

```
lsql-csv '-, &1.2 &1.1' <<- EOF
World,Hello
EOF
```

It returns:

```
Hello,World
```

Simple filtering

```
lsql-csv -d: '-, &1.*, if &1.3 >= 1000' </etc/passwd
```

This will print lines of users whose `UID` ≥ 1000. It can also be written as:

```
lsql-csv -d: 'p=/etc/passwd, p.*, if p.3 >= 1000'

lsql-csv -d: 'p=/etc/passwd, &1.*, if &1.3 >= 1000'

lsql-csv -d: '/etc/passwd, &1.*, if &1.3 >= 1000'
```

The `-d:` optional argument means the primary delimiter is `:`. In the few examples we used overnaming, which allows us to give a data source file `/etc/passwd` a name `p`.

If you know SQL, you can read it as `SELECT * FROM /etc/passwd P WHERE P.UID >= 1000;`. As you can see, the LSQL style is more compressed than standard SQL.

The output might be:

```
nobody:x:65534:65534:nobody:/var/empty:/bin/false
me:x:1000:1000::/home/me:/bin/bash
```

If you specify delimiter specifically for `/etc/passwd`, the output will be a comma delimited.

```
lsql-csv '/etc/passwd -d:, &1.*, if &1.3 >= 1000'
```

It might return:

```
nobody,x,65534,65534,nobody,/var/empty,/bin/false
me,x,1000,1000,,/home/me,/bin/bash
```

This happens because the default global delimiter, which is used for the output generation, is a comma. The global delimiter changes by usage of the command-line optional argument, but remains unchanged by the usage of the attribute inside the from block.

Named columns

Let's suppose we have a file `people.csv`:

```
name,age
Adam,21
Petra,23
Karel,25
```

Now, let's get all the names of people in `people.csv` using the `-n` named optional argument:

```
lsql-csv -n 'people.csv, &1.name'
```

The output will be:

```
Adam
Petra
Karel
```

As you can see, we can reference named columns by the name. The named optional argument `-n` enables first-line headers. If first-line headers are enabled by the argument, each column has two names under `&X`—the number name `&X.Y` and the actual name `&X.NAME`.

Now, we can select all columns with a wildcard `&1.*`:

```
lsql-csv -n 'people.csv, &1.*'
```

As the output, we get:

```
Adam,21,21,Adam
Petra,23,23,Petra
Karel,25,25,Karel
```

The output contains each column twice because the wildcard `&1.*` was evaluated to `&1.1, &1.2, &1.age, &1.name`. How to fix it?

```
lsql-csv -n 'people.csv, &1.[1-9]*'
```

The output is now:

```
Adam,21
Petra,23
Karel,25
```

The command can also be written as

```
lsql-csv -n 'people.csv, &1.{1,2}'
lsql-csv -n 'people.csv, &1.{1..2}'
lsql-csv 'people.csv -n, &1.{1..2}'
```

The output will be in all cases still the same.

Simple join

Let's say, I am interested in the default group names of users. We need to join tables `/etc/passwd` and `/etc/group`. Let's do it.

```
lsql-csv -d: '/etc/{passwd,group}, &1.1 &2.1, if &1.4 == &2.3'
```

What does `/etc/{passwd,group}` mean? Basically, there are three types of expressions. The select, the from, and the arithmetic expression. In all select and from expressions, you can use the curly expansion and wildcards just like in `bash` [14].

Finally, the output can be something like this:

```
root:root
bin:bin
daemon:daemon
me:me
```

The first column is the name of a user and the second column is the name of its default group.

Basic grouping

Let's say, I want to count users using the same shell.

```
lsql-csv -d: 'p=/etc/passwd, p.7 count(p.3), by p.7'
```

And the output?

```
/bin/bash:7
/bin/false:7
/bin/sh:1
/bin/sync:1
/sbin/halt:1
/sbin/nologin:46
/sbin/shutdown:1
```

You can see here the first usage of the by block, which is equivalent to `GROUP BY` in SQL.

Basic sorting

Let's say, you want to sort your users by `UID` with `UID` greater than or equal to 1000 ascendingly.

```
lsql-csv -d: '/etc/passwd, &1.*, if &1.3 >= 1000, sort &1.3'
```

The output might look like:

```
me1:x:1000:1000::/home/me1:/bin/bash
me2:x:1001:1001::/home/me2:/bin/bash
me3:x:1002:1002::/home/me3:/bin/bash
nobody:x:65534:65534:nobody:/var/empty:/bin/false
```

The sort block is the equivalent of `ORDER BY` in SQL.

If we wanted descendingly sorted output, we might create a pipe to the `tac` command—the `tac` command prints the lines in reverse order:

```
lsql-csv -d: '/etc/passwd, &1.*, if &1.3 >= 1000, sort &1.3' | tac
```

About nice outputs

There is a trick, how to concatenate two values in the select expression: Write them without space.

But how will the interpreter know the ends of the values in a command? If the interpreter sees a char, that can't be part of the currently parsed value, it tries to parse it as a new value concatenated to the current one. You can use quotes for it—quotes themselves can't be part of most value types like the column name or numerical constant.

As an example, let's try to format our basic grouping example.

```
lsql-csv -d: 'p=/etc/passwd,
  "The number of users of "p.7" is "count(p.3)".", by p.7'
```

The output might be:

```
The number of users of /bin/bash is 7.
The number of users of /bin/false is 7.
The number of users of /bin/sh is 1.
The number of users of /bin/sync is 1.
The number of users of /sbin/halt is 1.
The number of users of /sbin/nologin is 46.
The number of users of /sbin/shutdown is 1.
```

As you can see, string formatting is sometimes very simple with LSQL.

Arithmetic expression

So far, we just met all kinds of blocks, and only the if block accepts the arithmetic expression, and the other accepts the select expression. What if we needed to run the arithmetic expression inside the select expression? There is a special syntax `$(…)` for it.

For example:

```
lsql-csv -d: '/etc/passwd, $(sin(&1.3)^2 + cos(&1.3)^2)'
```

It returns something like:

```
1.0
1.0
1.0
0.9999999999999999
...
1.0
```

If we run:

```
lsql-csv -d: '/etc/passwd, $(&1.3 >= 1000), sort $(&1.3 >= 1000)'
```

We get something like:

```
false
false
...
false
true
true
...
true
```

More complicated join

Let's see more complicated examples.

```
lsql-csv -d: 'p=/etc/passwd g=/etc/group, p.1 g.1, if p.1 in g.4'
```

This will print all pairs of users and its group excluding the default group. If you know SQL, you can read it as `SELECT P.1, G.1 FROM /etc/passwd P, /etc/group G WHERE G.4 LIKE '%' + P.1 + '%';` with operator `LIKE` case-sensitive and columns named by their column number.

How does `in` work? It's one of the basic string level "consist". If some string `A` is a substring of `B`, then `A in B` is `true`. Otherwise, it is `false`.

And the output?

```
root:root
root:wheel
root:floppy
root:tape
lp:lp
halt:root
halt:wheel
```

The example will work under the condition, that there isn't any username, which is an infix of any other username.

More complicated...

The previous example doesn't give a very readable output. We can use `group by` to improve it (shortened as `by`).

```
lsql-csv -d: 'p=/etc/passwd g=/etc/group,
  p.1 cat(g.1","), if p.1 in g.4, by p.1'
```

The output will be something like:

```
adm:adm,disk,sys,
bin:bin,daemon,sys,
daemon:adm,bin,daemon,
lp:lp,
mythtv:audio,cdrom,tty,video,
news:news,
```

It groups all non-default groups of a user to a one line and concatenates it delimited by `,`.

How can we add default groups too?

```
lsql-csv -d: 'p=/etc/passwd g=/etc/group,
  p.1 cat(g.1","), if p.1 in g.4, by p.1' |
lsql-csv -d: '- /etc/passwd /etc/group,
  &1.1 &1.2""&3.1, if &1.1 == &2.1 && &2.4 == &3.3'
```

This will output something like:

```
adm:adm,disk,sys,adm
bin:bin,daemon,sys,bin
daemon:adm,bin,daemon,daemon
lp:lp,lp
mythtv:audio,cdrom,tty,video,mythtv
news:news,news
```

The first part of the command is the same as in the previous example. The second part inner joins the output of the first part with `/etc/passwd` on the username and `/etc/group` on the default GID number and prints the output of the first part with an added default group name.

The examples will also work under the condition, that there isn't any username, which is an infix of any other username.

1.3 Usage

Now, if you understand the examples, it is time to move forward to a more abstract description of the language and tool usage.

1.3.1 Options

`-h`
`--help`

Shows a short command line help and exits before doing anything else.

`-n`
`--named`

Enables the first-line naming convention in CSV files. With this option, the first lines of CSV files will be interpreted as a list of column names.

This works only on input files. Output is always without first-line column names.

`-dCHAR`
`--delimiter=CHAR`

Changes the default primary delimiter. The default value is `,`.

`-sCHAR`
`--secondary-delimiter=CHAR`

Changes the default quote char (secondary delimiter). The default value is `"`.

1.3.2 Datatypes

There are 4 datatypes considered: `Bool`, `Int`, `Double`, and `String`. `Bool` is either `true`/`false`, `Int` is at least a 30-bit integer, `Double` is a double-precision floating point number, and `String` is an ordinary char string.

During CSV data parsing, the following logic of datatype selection is used:

- `Bool`, if `true` or `false`;
- `Int`, if the POSIX ERE `[0-9]+` fully matches;
- `Double`, if the POSIX ERE `[0-9]+\.[0-9]+(e[0-9]+)?` fully matches;
- `String`, if none of the above matches.

1.3.3 Joins

Join means, that you put multiple input files into the from block.

Joins always have the time complexity $\mathcal{O}(nm)$. There is no optimization made based on if conditions when you put multiple files into the from block.

1.3.4 Documentation of language

```
lsql-csv [OPTIONS] COMMAND

Description of the grammar:

  COMMAND -> FROM_BLOCK, REST


  REST -> SELECT_BLOCK, REST
  REST -> BY_BLOCK, REST
  REST -> SORT_BLOCK, REST
  REST -> IF_BLOCK, REST
  REST -> LAST_BLOCK

  LAST_BLOCK -> SELECT_BLOCK
  LAST_BLOCK -> BY_BLOCK
  LAST_BLOCK -> SORT_BLOCK
  LAST_BLOCK -> IF_BLOCK


  FROM_BLOCK -> FROM_EXPR

  FROM_EXPR -> FROM_SELECTOR FROM_EXPR
  FROM_EXPR -> FROM_SELECTOR

  // Wildcard and brace expansion
  FROM_SELECTOR ~~> FROM ... FROM
```

```
// Standard input
FROM -> ASSIGN_NAME=- OPTIONS
FROM -> - OPTIONS

FROM -> ASSIGN_NAME=FILE_PATH OPTIONS
FROM -> FILE_PATH OPTIONS


OPTIONS -> -dCHAR OPTIONS
OPTIONS -> --delimiter=CHAR OPTIONS

OPTIONS -> -sCHAR OPTIONS
OPTIONS -> --secondary-delimiter=CHAR OPTIONS

OPTIONS -> -n OPTIONS
OPTIONS -> --named OPTIONS

OPTIONS -> -N OPTIONS
OPTIONS -> --not-named OPTIONS

OPTIONS ->


SELECT_BLOCK -> SELECT_EXPR
BY_BLOCK -> by SELECT_EXPR
SORT_BLOCK -> sort SELECT_EXPR
IF_BLOCK -> if ARITHMETIC_EXPR


ARITHMETIC_EXPR -> ATOM

ARITHMETIC_EXPR -> ARITHMETIC_EXPR OPERATOR ARITHMETIC_EXPR
ARITHMETIC_EXPR -> (ARITHMETIC_EXPR)

// Logical negation
ARITHMETIC_EXPR -> ! ARITHMETIC_EXPR
// Number negation
ARITHMETIC_EXPR -> - ARITHMETIC_EXPR


SELECT_EXPR -> ATOM_SELECTOR SELECT_EXPR
SELECT_EXPR -> ATOM_SELECTOR

// Wildcard and brace expansion
ATOM_SELECTOR ~~> ATOM ... ATOM
```

```
ATOM -> pi
ATOM -> e
ATOM -> true
ATOM -> false

// e.g. 1.0, "text", 'text', 1
ATOM -> CONSTANT
// e.g. &1.1
ATOM -> SYMBOL_NAME

ATOM -> $(ARITHMETIC_EXPR)
ATOM -> AGGREGATE_FUNCTION(SELECT_EXPR)
ATOM -> ONEARG_FUNCTION(ARITHMETIC_EXPR)


// # is not a char:
// Two atoms can be written without whitespace
// and their values will be String appended
// if the right atom begins with a char,
// which can't be a part of the left atom.
//
// E.g. if the left atom is a number constant,
// and the right atom is a String constant
// beginning with a quote char,
// the left atom value will be converted to the String
// and prepended to the right atom value.
//
// This rule doesn't apply inside ARITHMETIC_EXPR
ATOM ~~> ATOM#ATOM


// Converts all values to the String type and appends them.
AGGREGATE_FUNCTION -> cat

// Returns the number of values.
AGGREGATE_FUNCTION -> count

AGGREGATE_FUNCTION -> min
AGGREGATE_FUNCTION -> max
AGGREGATE_FUNCTION -> sum
AGGREGATE_FUNCTION -> avg


// All trigonometric functions are in radians.
ONEARG_FUNCTION -> sin
ONEARG_FUNCTION -> cos
ONEARG_FUNCTION -> tan
```

```
ONEARG_FUNCTION -> asin
ONEARG_FUNCTION -> acos
ONEARG_FUNCTION -> atan

ONEARG_FUNCTION -> sinh
ONEARG_FUNCTION -> cosh
ONEARG_FUNCTION -> tanh

ONEARG_FUNCTION -> asinh
ONEARG_FUNCTION -> acosh
ONEARG_FUNCTION -> atanh

ONEARG_FUNCTION -> exp
ONEARG_FUNCTION -> sqrt

// Converts a value to the String type and returns its length.
ONEARG_FUNCTION -> size

ONEARG_FUNCTION -> to_string

ONEARG_FUNCTION -> negate
ONEARG_FUNCTION -> abs
ONEARG_FUNCTION -> signum

ONEARG_FUNCTION -> truncate
ONEARG_FUNCTION -> ceiling
ONEARG_FUNCTION -> floor

ONEARG_FUNCTION -> even
ONEARG_FUNCTION -> odd


// A in B means A is a substring of B.
OPERATOR -> in

OPERATOR -> *
OPERATOR -> /

// General power
OPERATOR -> **
// Natural power
OPERATOR -> ^

// Integer division truncated towards minus infinity
// (x div y)*y + (x mod y) == x
OPERATOR -> div
OPERATOR -> mod
```

```
// Integer division truncated towards 0
// (x quot y)*y + (x rem y) == x
OPERATOR -> quot
OPERATOR -> rem

// Greatest common divisor
OPERATOR -> gcd
// Least common multiple
OPERATOR -> lcm

// String append
OPERATOR -> ++

OPERATOR -> +
OPERATOR -> -

OPERATOR -> <=
OPERATOR -> >=
OPERATOR -> <
OPERATOR -> >
OPERATOR -> !=
OPERATOR -> ==

OPERATOR -> ||
OPERATOR -> &&
```

Each command is made from blocks separated by a comma. There are these types of blocks.

- From block
- Select block
- If block
- By block
- Sort block

The first block is always the from block. If the block after the first block is without a specifier (`if`, `by`, or `sort`), then it is the select block. Otherwise, it is a block specified by the specifier.

The from block accepts a specific grammar (as specified in the grammar description), the select, the by, and the sort block accept the select expression (`SELECT_EXPR` in the grammar), and the if block accepts the arithmetic expression (`ARITHMETIC_EXPR` in the grammar).

Every source data file has a reference number based on its position in the from block and may have multiple names—the assign name, the name given to the source data file by `ASSIGN_NAME=FILE_PATH` syntax in the from block, and the default name, which is given by the path to the file or `-` in the case of the standard input in the from block.

Each column of a source data file has a reference number based on its position in it and may have a name (if the named option is enabled for the given source file).

If a source data file with the reference number `M` (numbering input files from 1) has a name `XXX`, its columns can be addressed by `&M.N` or `XXX.N`, where `N` is the reference number of a column (numbering columns from 1). If the named option is enabled for the input file and a column has the name `NAME`, it can also be addressed by `&M.NAME` or `XXX.NAME`.

We call the address a symbol name—`SYMBOL_NAME` in the grammar description.

If there is a collision in naming (some symbol name addresses more than one column), then the behavior is undefined.

Exotic chars

Some chars cannot be in unquoted symbol names—exotic chars. For simplicity, we can suppose, they are all non-alphanumerical chars excluding `-`, `.`, `&`, and `_`. Also the first char of a symbol name must be non-numerical and must not be `-` or `.` to not be considered as an exotic char.

It is possible to use a symbol name with exotic chars using `` ` `` quote—like `` `EXOTIC SYMBOL NAME` ``.

Quote chars

There are 3 quote chars (`` ` ``, `"` and `'`) used in LSQL. `"` and `'` are always quoting a `String`. The `` ` `` quote char is used for quoting symbol names.

These chars can be used for `String` appending. If two atoms inside `SELECT_EXPR` are written consecutively without whitespace and the left atom ends by a quote char or the right begins by a quote char, they will be converted to the `String` and will be `String` appended. For example, `&1.1"abc"` means: convert the value of `&1.1` to the `String` and append it to the `String` constant `abc`.

Constants

Constants are in the grammar description as `CONSTANT`. In the following section, we speak only about these constants and not about built-in constant values like `pi` or `true`.

There are 3 datatypes of constants. `String`, `Double`, and `Int`. Every string quoted in `"` chars or `'` chars in an LSQL command is always tokenized as a `String` constant. Numbers fully matching the POSIX ERE `[0-9]+` are considered `Int` constants and numbers fully matching the POSIX ERE `[0-9]+\.[0-9]+` `Double` constants.

Operator associativity and precedence

All operators are right-to-left associative.

The following list outlines the precedence of the `lsql-csv` infix operators. The lower the precedence number, the higher the priority.

Precedence number	Operator
1	`in`, `**`,^
2	`*`, `/`, `div`, `quot`, `rem`, `mod`, `gcd`, `lcm`
3	`++`, `+`, `-`
4	`<=`, `>=`, `<`, `>`, `!=`, `==`
5	||, `&&`

Select expression

Select expressions are in the grammar description as `SELECT_EXPR`. They are similar to the `bash` expressions [14]. They are made by atom selector expressions (`ATOM_SELECTOR`) separated by whitespaces. These expressions are wildcard and brace expanded to atoms (`ATOM`) and are further processed as they were separated by whitespace. (In `bash` brace expansion is a mechanism by which arbitrary strings are generated. For example, `a{b,c,d}e` is expanded to `abe ace ade`, see [14] for details.)

Wildcards and brace expansion expressions are only evaluated and expanded in unquoted parts of the atom selector expression, which aren't part of an inner arithmetic expression.

For example, if we have an LSQL command with symbol names `&1.1` and `&1.2`, then

- the atom selector expression `&1.{1,2}` will be expanded to `&1.1` and `&1.2`;
- the atom selector expression `&1.*` will be expanded to `&1.1` and `&1.2`;
- the atom selector expression `` `&1.*` `` will be expanded to `` `&1.*` ``;
- the atom selector expression `"&1.*"` will be expanded to `"&1.*"`;
- the atom selector expression `$(&?.1)` will be expanded to `$(&?.1)`;
- the atom selector expression `&*$(&1.1)` will be expanded to `&1.1$(&1.1)` and `&1.2$(&1.1)`.

Every atom selector expression can consist:

- A wildcard (Each wildcard is expanded against the symbol name list. If no symbol name matching the wildcard is found, the wildcard is expanded to itself.);
- A `bash` brace expansion expression (e.g. `{22..25}` → `22 23 24 25`) [14];
- An arithmetic expression in `$(expr)` format;
- A call of an aggregate function `AGGREGATE_FUNCTION(SELECT_EXPR)`—there cannot be any space after `FUNCTION`;
- A call of a one-argument function `ONEARG_FUNCTION(ARITHMETIC_EXPR)`—there cannot be any space after `FUNCTION`;
- A constant;

- A built-in constant value;
- A symbol name.

Please, keep in mind, that operators (`OPERATOR`) must be put inside arithmetic expressions.

Arithmetic expression

Arithmetic expressions are in the grammar description as `ARITHMETIC_EXPR`.

The expressions use mainly the classical `awk` style of expressions [15]. You can use here operators (`OPERATOR`) keywords `>`, `<`, `<=`, `>=`, `==`, `||`, `&&`, `+`, `-`, `*`, `/`…

Wildcards and brace expansion expressions are not evaluated inside the arithmetic expression.

Select blocks

Select blocks are referred in the grammar description as `SELECT_BLOCK`. These blocks determine the output. They accept the select expression.

There must be at least one select block in an LSQL command, which refers to at least one symbol name, or the behavior is undefined.

Examples of select blocks:

```
&1.[3-6]
```

This will print the 3rd, 4th, 5th, and 6th columns from the first file if the first file has at least 6 columns.

```
ax*.{6..4}
```

This will print the 6th, the 5th, and the 4th columns from all files whose name begins with ax if the files have at least 6 columns.

From blocks

These blocks are in the grammar description as `FROM_BLOCK`. There must be exactly one from block at the beginning of an LSQL command.

The from block contains input file paths (or `-` in the case of the standard input), and optionally their assign name `ASSIGN_NAME`.

You can use the wildcards and the curly bracket expansion as you were in the `bash` to refer input files [14]. If there is a wildcard with an assign name `NAME` matching more than one input file, the input files will be given assign names `NAME`, `NAME1`, `NAME2`… If there is a wildcard, that matches to no file, it is expanded to itself.

If `FILE_PATH` is put inside ` quotes, no wildcard or expansion logic applies to it.

You can also add custom attributes to input files in the format `FILE_PATH -aX --attribute=X -b`. The attributes will be applied to all files which will be matched against `FILE_PATH`. The custom attributes are referred to as `OPTIONS` in the grammar description.

Examples:

```
/etc/{passwd,group}
```

This will select `/etc/passwd` and `/etc/group` files. They can be addressed either as `&1` or `/etc/passwd`, and `&2` or `/etc/group`.

```
passwd=/etc/passwd
```

This will select `/etc/passwd` and set its assign name to `passwd`. It can be addressed as `&1`, `passwd`, or `/etc/passwd`.

Possible custom attributes

```
-n
--named
```

Enables the first-line naming convention for an input CSV file. With this option, the first line of a CSV file will be interpreted as a list of column names.

```
-N
--not-named
```

You can also set the exact opposite to an input file. This can be useful if you change the default behavior.

```
-dCHAR
--delimiter=CHAR
```

This changes the primary delimiter of an input file.

```
-sCHAR
--secondary-delimiter=CHAR
```

This changes the secondary delimiter of an input file.

Example:

```
/etc/passwd -d:
```

This will select `/etc/passwd` and set its delimiter to `:`.

Currently, commas and `CHAR`s, which are also quotes in LSQL, are not supported as a delimiter or a secondary delimiter in `FILE_PATH` custom attributes.

If blocks

These blocks are in the grammar description as `IF_BLOCK`. They always begin with the `if` keyword. They accept arithmetic expressions, which should be convertible to `Bool`: either `String` `false`/`true`, `Int` (`0` `false`, anything else `true`), or `Bool`.

Rows with the arithmetic expression converted to `Bool` `true` are printed or aggregated, and rows with the arithmetic expression converted to `Bool` `false` are skipped.

Filtering is done before the aggregation.

You can imagine the if block as the `WHERE` clause in SQL.

By blocks

By blocks are referred in the grammar description as `BY_BLOCK`. These blocks always begin with the `by` keyword. They accept the select expression.

There can be only one by block in a whole LSQL command.

The by block is used to group the resulting set by the given atoms for the evaluation by an aggregate function. The by block is similar to the `GROUP BY` clause in SQL.

There must be at least one aggregate function in the select block if the by block is present. Otherwise, the behavior is undefined.

If there is an aggregate function present without the by block present in an LSQL command, the aggregate function runs over all rows at once.

Sort blocks

These blocks are in the grammar description as `SORT_BLOCK`. It begins with the `sort` keyword. They accept the select expression.

The sort block determines the order of the final output—given atoms are sorted in ascending order. If there is more than one atom in the sort block (`A`, `B`, `C…`), the data is first sorted by `A` and in the case of ties, the atoms (`B`, `C…`) are used to further refine the order of the final output.

You can imagine the sort block as the `ORDER BY` clause in SQL.

There can be only one sort block in the whole command.

2. Developer documentation

This chapter is for potential developers of the project.

2.1 Project building and testing

The project has two ways of building. The first way is through `Makefile` and the second is through Cabal. By running the following command it generates the `build` folder and the `lsql-csv` binary under it.

```
make
```

It is necessary to have all Haskell dependencies (`Parsec` ($\geq 3.1 < 3.2$), `Glob` ($\geq 0.10 < 0.11$), `base` ($\geq 4.9 < 4.20$), `text` ($\geq 1.2 < 2.2$) and `containers` ($\geq 0.5 < 0.8$)) installed. The package boundaries given are identical to the Cabal boundaries. Also, it is necessary, that you have `GHC` ($\geq 8 < 9.29$) installed.

The second way of building is through Cabal, which handles all dependencies for you.

```
cabal build
```

The project unit tests require building through `Makefile` and are called by:

```
make test
```

It should always succeed before any commit to the project repository is made.

It is possible to generate Haddock developer documentation by calling:

```
cabal haddock
```

The documentation contains comments on all exported functions. The documentation can be alternatively generated by running:

```
make docs
```

It generates HTML documentation under the `build` folder.

The package is also published at `https://hackage.haskell.org/package/lsql-csv` in the Hackage public repository. The generated documentation is fully browseable there.

2.2 `String` vs `Data.Text`

As there is a long-term discussion in the Haskell community about whether `String` or `Data.Text` should be used as the primary representation of text, I would like to emphasize that in this project, `String` is used as the primary representation of text.

2.3 Project layout

The project is split into:

1. a library, which contains almost all the logic and is placed under `src` folder of the project
2. the `main`, which contains one source file with `Main`, which is the entry point for the `lsql-csv` binary. It parses the arguments, and checks, whether the help optional argument was called and whether no argument at all was given, and either displays the usage message or further call `run` from `Lsql.Csv.Main` in the library—the evaluation entry point.

The library is split into 5 namespaces. Their usage is not strictly defined, but this can be said:

- `Lsql.Csv` – This namespace contains the starting point for an `lsql-csv` evaluation.
- `Lsql.Csv.Core` – This namespace contains the logic of the evaluation.
- `Lsql.Csv.Lang` – This namespace contains parsers for the blocks other than the from block.
- `Lsql.Csv.Lang.From` – This namespace contains parsers for the from block and for CSV files.
- `Lsql.Csv.Utils` – This namespace contains helper functions.

2.4 Modules

The following section is a summary of all modules of the library.

- `Lsql.Csv.Core.BlockOps` – This module contains the `Block` definition representing an LSQL command block and functions for getting a specific type of block from a list of `Block`.
- `Lsql.Csv.Core.Evaluator` – This module contains the evaluator of an `lsql-csv` command.
- `Lsql.Csv.Core.Functions` – This module contains the syntactic tree definition and helper functions for its evaluation.
- `Lsql.Csv.Core.Symbols` – This module contains the definition of `Symbol`, `SymbolMap`, and helper functions for working with them. `SymbolMap` is one of the representations of input data.
- `Lsql.Csv.Core.Tables` – This module contains the definition of `Value`, `Table`, and `Column`, class instancies over them, functions for manipulation of them, and `Boolable` class definition.
- `Lsql.Csv.Lang.Args` – A module for command-line argument parsing.

- `Lsql.Csv.Lang.BlockChain` – This module contains the main parser of the blocks other than the from block.
- `Lsql.Csv.Lang.BlockSeparator` – This module contains the preprocessor parser, which splits an LSQL command into a list of `String`—one `String` per block.
- `Lsql.Csv.Lang.Options` – This module implements the common `Option` type for the from block custom attributes representation and for the command-line optional arguments representation, and its parsers.
- `Lsql.Csv.Lang.Selector` – This module implements the selector expression parser and the arithmetic expression parser.
- `Lsql.Csv.Lang.From.Block` – This module contains the from block parser. It loads the initial `SymbolMap` with input data.
- `Lsql.Csv.Lang.From.CsvParser` – This module contains the `CsvParser` called by the `parseFile`, which loads input CSV files.
- `Lsql.Csv.Main` – This module contains the starting point for an `lsql-csv` evaluation.
- `Lsql.Csv.Utils.BracketExpansion` – This module contains the curly bracket (braces) expansion implementation.
- `Lsql.Csv.Utils.CsvGenerator` – This module contains the CSV generator for the output.

2.5 Evaluation entry point

The evaluation entry point is in the module `Lsql.Csv.Main` in the function `run`. The function first calls the preprocessor in `Lsql.Csv.Lang.BlockSeparator`, which splits the input command into a list of `String`—one `String` per block. Then the `SymbolMap` with input data is loaded using `Lsql.Csv.Lang.From.Block` and after that, the rest of the blocks are parsed using `Lsql.Csv.Lang.BlockChain`.

The loaded data are then processed using the evaluator in `Lsql.Csv.Core.Evaluator` according to the parsed command and finally, the output is generated by `Lsql.Csv.Utils.CsvGenerator`.

2.6 Unit tests

There are many unit tests for testing the functionality of the `lsql-csv` binary.

All unit tests are shell scripts. Each test invokes the `lsql-csv` binary in the `build` project folder with predefined input data, predefined optional arguments, and a predefined LSQL command, and compares the output of the `lsql-csv` invocation with a predefined expected output. If the expected output is the same as the actual output, the test exits with code `0`. If the expected output is different from the actual output, the test exits with code `1`.

All unit tests are split into subfolders of the project folder `tests`. There are these subfolders:

1. `basics` – A basic functionality tests.
2. `blocks` – Tests of all block types and their combinations.
3. `examples` – Tests of a majority of examples in the user documentation.
4. `options` – Tests of the command-line optional arguments, and the from block custom attributes.
5. `operators` – Tests of operators, their precedence, and associativity.
6. `onearg-functions` – Tests of one-argument functions.
7. `aggregate-functions` – Tests of all aggregate functions.

For each test subfolder `FOLDER`, there is a `Makefile` target `test-FOLDER`, which indirectly invokes all the tests in the subfolder. The `Makefile` target `test` indirectly invokes the tests in all subfolders.

3. Analysis

Why the language have been made the way it is? Why it is so inspired by SQL and is it not just the next implementation of SQL? Why it is implemented the way it is? This chapter is about key decisions we made while designing the language.

3.1 Why SQL in the first place?

Why do we talk so much about SQL in the first place? Since it was introduced in the 1970s [3], it has become the de facto standard for many major databases. Just for illustration, we name a few of them.

- Oracle DB has used it since 1979 as the first commercially available implementation [16].
- MySQL has used it since 1994, since its original development started [17].
- PostgreSQL has used it since 1996, since it was created [18].
- MSSQL has used it since 1989, since its initial release [19].

SQL is so much known, that there is a widely used term NoSQL databases as databases opposed to SQL databases.

The main point of making language inspired by SQL is that it brings the advantage of getting a large user base which only needs to understand the difference between SQL and the new language to start using the new language. This is the starting point, from which we further argue about design decisions made.

3.2 Why not implement just another SQL for CSV files?

As mentioned in the introduction, standard SQL is a typed language (every data value belongs to some data type) [6], which implies many design choices.

Furthermore, SQL itself requires a large amount of text to be written, before it can be executed.

One of the ambitions of `lsql-csv` is to allow a user to write shorter queries to get the result. For the example, consider

```
SELECT dataX FROM data.txt WHERE dataX > 1000;
```

This simple SQL query shows `dataX > 1000` from the table `data.txt`. Now, if we have a CSV file `data.txt`, where we know, that `dataX` is a second column, the same query can be written with `lsql-csv` as

```
data.txt, &1.2, if &1.2 > 1000
```

The length difference is about 35% off the original SQL query. It is simply said one of the reasons, why we will not just implement another SQL implementation.

3.3 Why number references?

Where the 35% difference happened? One of the main reasons is, that we allowed referencing *dataX* as `.2`. This is also possible due to the nature of the CSV file, where columns have their index[1]. Normally, in a SQL database, indexes of columns are not considered as something, which should decide about query meaning. This is because the SQL database itself may change, and new columns may be added or removed. Just because somebody removed a column, it is not desirable to require developers to change some queries so they comply with the new database layout.

On the other hand, CSV files are not usually altered as much as SQL databases are. If a developer (or a user) is not sure about the columns present in a CSV file, it is one of the first signs, that he or she should use rather a SQL database than a simple CSV file.

Also, `lsql-csv` is a tool for daily life and simple scripts rather than a tool for the development of medium-sized or large-sized projects, like SQL is. This adds much more flexibility in what can language do without jeopardizing its design goals—things like number references.

3.4 Why rename standard SQL keywords?

Why have we renamed `WHERE` for `if`, `GROUP BY` for `by`, and `SORT BY` for `sort`? Simply said, because the renamed variants are shorter and still do not block a user in understanding, what the query does.

3.5 Why some features like descending sort are missing?

`lsql-csv` is a shell utility and as such, it tries to comply with UNIX philosophy. The summarized version from Doug McIlroy is: "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface." [20]

The tool for reverting the order of output already exists: `tac`. Piped together it creates the wanted output.

In a similar case, the function for the second filtering of the grouped by (in SQL named `HAVING`) output has been not added, because the wanted output may be received by piping the output with another instance of `lsql-csv`.

And why there is no support for creating output with first-line names of columns? Because the wanted output may be simply made with the usage of `echo` called before `lsql-csv` if needed.

[1] Do not confuse with the SQL database index for performance.

3.6 Why blocks are delimited by commas?

The author thinks it is more readable like this. Developers in SQL usually use upper case chars and new line writing to delimiter "the blocks"—this is not so much necessary in LSQL as in SQL due to comma delimiting. A comma is also usually easier to write than switching on and off the caps lock, holding the shift key, or making multiline input.

3.7 Why there are two types of expression?

Why there are the arithmetic and the select expressions with different grammars? We think, that the addition of a specially designed select expression further allows the user to write shorter queries. It allows us to introduce wildcards and curly brackets expansion for a user.

3.8 Why there is support only for cross-join and not other types of join?

The tool is supposed to be simple. We recommend a user to import CSV data to a SQL database and use standard SQL if he needs more complicated joins.

The other reason is, that CSV has no standardized `NULL` value, which is needed for the left or the right outer joins.

3.9 Why there is no package used for CSV parsing and generating?

Very simply said, to limit the number of dependencies. The more dependencies are used in a package, the harder is to compile it, maintain it, and add it to any Linux distribution.

As CSV parsing is not a hard job, it was decided so.

3.10 Why `String` is used as primary text representation?

As the performance gain from using `Data.Text` would not be significant and `String` does not add any more complexity to the code like `Data.Text` does, it was decided that `String` would be the primary text representation.

The tool is simpler and more easily maintainable with `String` than it would be with `Data.Text`.

3.11 Why joins have $\mathcal{O}(nm)$ complexity?

`lsql-csv` is a simple tool for small dataset data querying. As such, it implements only a simple algorithm for joining the tables—cross-join.

For larger dataset joins users are encouraged to use standard SQL databases as `lsql-csv` is not designed for this use case.

3.12 Why are all operators right-to-left associative?

Many modern languages like C++ [21], Rust [22], and C# [23] use combined left-to-right and right-to-left associativity for operators. Given the fact that it might be hard to remember which operators are left-to-right and which operators are right-to-left associative, some expressions in these languages might be hard to interpret.

To comply with Unix philosophy, concretely with the KISS principle—Keep it simple, stupid [24], and to solve the problem mentioned above, it was decided all operators will have the same associativity. The choice of right-to-left instead of left-to-right is just an arbitrary decision as there are arguments for both right-to-left or left-to-right associativity of operators. By this decision, the problem of hard expression interpretation might be solved.

4. Alternative Solutions

What other solutions are there for the given problem? What other approaches can we use, when we are dealing with queries over CSV files? The following chapter is about alternative approaches to the problem.

4.1 Using SQL database

The first obvious solution is importing the dataset to some standard SQL database and doing the queries over it. This approach requires the definition of the schema into which the data will be imported. This is an overhead, which isn't always advantageous to pay as we might need only a simple query to be run over it. The data import might take also a much longer time than a simple `lsql-csv` invocation.

But it might be a very favorable solution, if we need large dataset joins, need a complex query execution or a large amount of simple queries run over it.

By using a standard SQL database you gain the advantage of better performance, typed datasets, indexes, and a larger number of built-in functions.

4.2 Using standard Unix tools

It is possible to do a large amount of work just by using `awk`, `join`, `sort`… For example:

```
lsql-csv -d: '-, &1.*, if &1.3 >= 1000' </etc/passwd
```

This query might be rewritten to `awk`:

```
awk -F: '{ if($3 >= 1000){ print $0 }}' </etc/passwd
```

The main advantage of `lsql-csv` is that it handles some more complex queries more easily. For example:

```
lsql-csv -d: '/etc/{passwd,group}, &1.1 &2.1, if &1.4 == &2.3'
```

This is a simple join query. When written using standard Unix tools, it is:

```
sort -t: -k3,3 /etc/group >/tmp/group.sort
sort -t: -k4,4 /etc/passwd >/tmp/passwd.sort
join -t: -14 -23 /tmp/passwd.sort /tmp/group.sort | cut -d: -f2,8
```

As demonstrated, the `lsql-csv` variant is more readable and shorter[1].

It should be also noted, that an `lsql-csv` join has $\mathcal{O}(nm)$ time complexity, while standard Unix tools have for join written above $\mathcal{O}(n \log n + m \log m)$ time complexity, so for larger datasets, it might be more beneficial to use them.

[1] It is possible, that it can be written in shorter and more readable form, but probably not more than the `lsql-csv` variant.

4.3 By using SQL implementation for CSV files

There are many projects implementing SQL on CSV files. For example:

- `q` [7]
- `CSV SQL` [8]
- `trdsql` [9]
- `csvq` [10]

It is possible to use them to do the job directly with SQL.

The advantage of it is you do not have to learn a new language if you already know standard SQL. The disadvantage is that the queries will be probably longer than would be with `lsql-csv`.

4.4 By using general-purpose programming language

It is not hard to parse and process CSV files with a general-purpose programming language (for example Python).

The advantage of this solution is a much greater flexibility of what you can do with the CSV files. The large disadvantage is, that it will take too much code to be written for any query.

Conclusion

The goal of this thesis was to create a new tool for small CSV file data querying from a shell with short queries. It ought to be simple, comply with Unix philosophy, and use more lapidary language than SQL. It ought to enable its users to quickly write simple queries directly to the terminal.

Our new tool `lsql-csv` implements a new language LSQL similar to SQL, specifically designed for working with CSV files in a shell. It makes it possible to work with small CSV files like with a read-only relational database.

In some of the use cases, the tool provides shorter and more readable commands then you would get by using standard Unix tools and the language is more lapidary than SQL.

The `lsql-csv` package has been uploaded to the Hackage public package repository and therefore is easily installable. It provides automatic Haddock documentation, which is browseable from there. It has also been published on GitHub with the `README.md` file containing the full user documentation including the tutorial and grammar description.

Bibliography

[1] Wikipedia contributors. Database—Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Database&oldid=1200665358, 2024. [Online; accessed 16-February-2024].

[2] Yakov Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, October 2005. Available at https://www.rfc-editor.org/rfc/rfc4180.txt [Online; accessed 16-February-2024].

[3] Wikipedia contributors. SQL—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=SQL&oldid=1205226098, 2024. [Online; accessed 16-February-2024].

[4] ANSI, ANSI X3.135-1986, American National Standard for Information Systems—Database Language SQL. Standard, American National Standards Institute, Washington, D.C., US, October 1986.

[5] ISO 9075:1987: Information technology – Database languages SQL – Part 1: Framework (SQL/Framework). Standard, International Organization for Standardization, Geneva, CH, June 1987.

[6] ISO 9075-1:2023: Information technology – Database languages SQL – Part 1: Framework (SQL/Framework). Standard, International Organization for Standardization, Geneva, CH, June 2023.

[7] q—Text as Data. https://github.com/harelba/q, January 2022. [Online; accessed 18-February-2024].

[8] CSV SQL. https://github.com/alex/csv-sql, May 2021. [Online; accessed 29-February-2024].

[9] trdsql. https://github.com/noborus/trdsql, December 2023. [Online; accessed 29-February-2024].

[10] csvq. https://github.com/mithrandie/csvq, February 2023. [Online; accessed 29-February-2024].

[11] Alejandro Serrano Mena. *Practical Haskell: A real world guide to programming*. Second Edition. Apress, New York, 2019.

[12] passwd(5)—Linux manual page. https://www.man7.org/linux/man-pages/man5/passwd.5@@shadow-utils.html, December 2021. [Online; accessed 21-March-2024].

[13] group(5)—Linux manual page. https://www.man7.org/linux/man-pages/man5/group.5.html, October 2023. [Online; accessed 21-March-2024].

[14] Bash Reference Manual. https://www.gnu.org/software/bash/manual/bash.html, September 2022. [Online; accessed 25-March-2024].

[15] GAWK: Effective AWK Programming: A User's Guide for GNU Awk. https://www.gnu.org/software/gawk/manual/gawk.html, 2023. [Online; accessed 25-March-2024].

[16] Usha Krishnamurthy et al. *Oracle Database SQL Language Reference, 19c, E96310-23.* Oracle, November 2023. Available at https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/sql-language-reference.pdf [Online; accessed 23-February-2024].

[17] Wikipedia contributors. MySQL—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=MySQL&oldid=1205045759, 2024. [Online; accessed 23-February-2024].

[18] Happy Birthday, PostgreSQL! https://www.postgresql.org/about/news/happy-birthday-postgresql-978/, July 2008. [Online; accessed 23-February-2024].

[19] Wikipedia contributors. Microsoft SQL Server—Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Microsoft_SQL_Server&oldid=1210428592, 2024. [Online; accessed 29-February-2024].

[20] Eric S. Raymond. Basics of the Unix Philosophy. http://www.catb.org/~esr/writings/taoup/html/ch01s06.html, September 2003. [Online; accessed 24-February-2024].

[21] C++ Operator Precedence. https://en.cppreference.com/w/cpp/language/operator_precedence, September 2023. [Online; accessed 08-Apr-2024].

[22] The Rust Reference—Expressions. https://doc.rust-lang.org/reference/expressions.html, July 2023. [Online; accessed 08-Apr-2024].

[23] C# specifications—Expressions. https://learn.microsoft.com/en-US/dotnet/csharp/language-reference/language-specification/expressions, July 2024. [Online; accessed 08-Apr-2024].

[24] Wikipedia contributors. KISS principle—Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=KISS_principle&oldid=1198063371, 2024. [Online; accessed 08-April-2024].