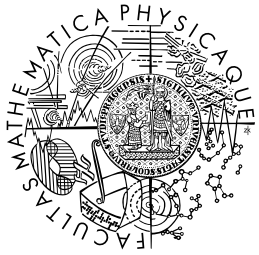

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Přemysl Šťastný

lsql-csv

Department of Applied Mathematics

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

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

Dedication.

Title: lsql-csv

Author: Přemysl Šťastný

Department: Department of Applied Mathematics

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

Abstract: Abstract.

Keywords: key words

Contents

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 at 1970s [3] and is one of the most used query languages. Despite being standardized in 1987 by the International Organization for Standardization [4], there are virtually no implementations that adhere to it fully [3]. Standard SQL is a typed language (Every data column and data value belongs to some data type) [5] 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 (eg. *q* [6], *CSV SQL* [7], *trdsql* [8] or *csvq* [9]), 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 shell.

Haskell is a language with great features for working with the text [10] and therefore it was selected for the task of implementation of lsql-csv.

[1] Or author doesn't know about it

1. User documentation

lsql-csv is a tool for CSV file data querying from the 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 shell.

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* ($\geq 4.9 < 4.17$), *text* ($\geq 1.2 < 1.3$) and *containers* ($\geq 0.5 < 0.7$) installed. (The package boundaries given are identical to cabal boundaries.) Run then:

```
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 dependencies for you.

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 unit tests for you.

1.2 lsql-csv – quick introduction

LSQL, the language of *lsql-csv*, 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.

1.2.1 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 Unix/Linux scripting enough. 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.

/etc/passwd has the following columns[11]:

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

/etc/group has the following columns[12]:

1. group name
2. password
3. numerical group ID
4. user list

Hello World

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

This will print the second (*&1.2*) and the first column (*&1.1*) of csv file on stdin. If you know SQL, you can read it like *from stdio S select S.second, S.first.*

Commands are split by commas into blocks. The first block is (and always is) the from block. There are file names or - (stdin) separated by space. The second block is the select block, also separated by space.

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 previous 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 *from /etc/passwd P select * where P.UID >= 1000*. As you can see, 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
```

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 switch:

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

The output will be:

```
Adam
Petra
Karel
```

As you can see, we can reference named columns by a name. Named switch *-n* enables first-line headers. If named columns are enabled, 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 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 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}'
```

The output will be in both cases still the same.

Simple join

Let's say, I am interested in the default group names of users. We need to join to 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. Select, from and arithmetic expression. In all select and from expressions, you can use expansion and wildcards just like in bash[13].

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 *by* block, which is equivalent of *group by* in SQL.

Basic sorting

Let's say, you want to sort your users 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 a select expression: Write them without space.

But how does the interpreter know the ends of the value name or value expression? You must use quotes for it - quotes themselves can't be part of the value name. As an example, let's try to format our basic grouping example.

Let's try it!

```
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 if block accepts an arithmetic expression and the other accepts a select expression. What if we needed to run an arithmetic expression inside a 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 user and its group excluding the default group. If you know SQL, you can read it as *from /etc/passwd P, /etc/group G select P.1, G.1 where P.1 in G.4*.

How does *in* work? It's one of the basic string level "consist". If 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
```

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 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.

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 short command line help and exits before doing anything else.

```
-n
--named
```

Enables first-line naming convention in csv files. This works only on input files. Output is always without first-line column names.

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

Changes default primary delimiter. The default value is *,*.

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

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

1.3.2 Datatypes

There are 4 datatypes considered: *Bool*, *Int*, *Double*, *String*. *Bool* is either true/false, *Int* is at least a 30-bit integer, *Double* 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 *[0-9]+* matches
- *Double*, if *[0-9]+.[0-9]+(e[0-9]+)?* matches
- *String*, if none of the above matches

1.3.3 Joins

Join means, that you put multiple input files into 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 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 ->
```

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

FROM -> FROM_NAME=FROM_FILE OPTIONS
FROM -> FROM_FILE 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 -> ONEARG_FUNCTION(ARITHMETIC_EXPR)
ARITHMETIC_EXPR -> ARITHMETIC_EXPR OPERATOR ARITHMETIC_EXPR
ARITHMETIC_EXPR -> (ARITHMETIC_EXPR)
// Logical negation
ARITHMETIC_EXPR -> ! ARITHMETIC_EXPR
ARITHMETIC_EXPR -> - ARITHMETIC_EXPR

SELECT_EXPR -> ATOM_SELECTOR SELECT_EXPR
SELECT_EXPR ->

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

// eg. 1.0, "text", 'text', 1
ATOM -> CONSTANT
// eg. &1.1
ATOM -> COLUMN_NAME
ATOM -> pi
ATOM -> e
ATOM -> true
ATOM -> false
ATOM -> $(ARITHMETIC_EXPR)
ATOM -> AGGREGATE_FUNCTION(SELECT_EXPR)
ATOM -> ONEARG_FUNCTION(ARITHMETIC_EXPR)
```

```
// # is not really char:
// two atoms can be written without space
// and will be (string) appended,
// if they are separated using quote chars:
//   left atom must end or right atom must begin with quote char
// This rule doesn't apply inside ARITHMETIC_EXPR
ATOM ~~> ATOM#ATOM


AGGREGATE_FUNCTION -> cat
AGGREGATE_FUNCTION -> sum
AGGREGATE_FUNCTION -> count
AGGREGATE_FUNCTION -> max
AGGREGATE_FUNCTION -> min
AGGREGATE_FUNCTION -> avg

//All trigonometric functions in radian
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

//The length of the string
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

OPERATOR -> ++    //append

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

From block accept specific grammar (as specified in the grammar description), select, by, and sort block select expression (*SELECT_EXPR* in the grammar) and if block arithmetic expression (*ARITHMETIC_EXPR* in the grammar).

Every source file has a number and may have multiple names - assign name, the name given to the source file by *ASSIGN_NAME=FILE_PATH* syntax in from block, and default name, which is given the path to the file or - in case of stdin in from block.

Each column of a source file has a number and may have a name (if the named option is enabled for the given source file).

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

If there is a collision in naming (two source files have the same name or two columns under the same source file have the same name), then the behavior is undefined.

Exotic chars

There are some chars that cannot be in symbol names (column names). For simplicity, we can suppose, they are everything but alphanumerical chars excluding -, ., *&* and _. Also first char of a symbol name must be non-numerical to not be considered as an exotic char. Referencing names containing exotic chars without quotes is unsupported.

It is possible to reference columns with names with exotic chars using ‘ quote - like *‘EXOTIC NAME‘*. The source file name is always part of the column name from the syntax perspective of language - it must be inside the quotes.

Quote chars

There are 3 quotes (‘, ” and ’) used in Lsql. ” and ’ are always quoting a string. The ‘ quote is used for quoting symbol names.

These chars can be used for fast appending. If two atoms inside *SELECT_EXPR* are written without space and are separated using the quotes, they will be appended. For example, *abc”abc”* means: append column abc to the string abc.

Constants

There are 3 types of constants. String, Double, and Int. Everything quoted in ” or ’ is always String constant. Numbers without *[0-9]+* are considered Int constant and numbers *[0-9]+.[0-9]+* Double constant.

Operator precedence

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

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

Select expression

They are similar to bash expressions[13]. They are made by atom selector expressions separated by whitespaces. These expressions are expanded, evaluated, and matched to column names, constants, aggregate functions, or arithmetic expressions.

Every atom selector expression can consist

- Wildcard (Each wildcard will be expanded to multiple statements during processing)
- Bash brace expansion (e.g. 22..25 ->22 23 24 25) [13]
- Arithmetic expression in *$(expr)* format
- Quotes *'anything'* to prevent wildcards and expansions
- Quotes *"* or *'* to insert string
- Call of aggregate function *AGGREGATE_FUNCTION(next select block)* - there cannot be any space after FUNCTION
- Call of single arg function *ONEARG_FUNCTION(arithmetic expression)* - there cannot be any space after FUNCTION
- Constants
- Reference to a column name

If you want to concatenate strings without *++* operator, you can write: *a.1","a.2*.

Please, keep in mind, that operators must be put inside arithmetic expressions, or they will be matched to a column name or aggregate function.

Arithmetic expression

The statement uses mainly classical awk logic.[14] You can use keywords >, <, <=, >=, ==, ||, &&, +, -, *, /,...

Select blocks

These blocks determine output. They accept select expressions and are evaluated and printed in a delimitered format.

Every select block must contain at least one reference to the column name, or the behavior is undefined.

Examples of select blocks:

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

This will print columns 3, 4, 5, and 6 from the first file.

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

This will print the 6th, 5th, and 4th of all files whose name begins with ax.

From blocks

There must be exactly one from block at the beginning of the command. The block can contain any files (and - specifies standard input). You can use any syntax you would otherwise use in bash to select these files (wildcards, expansion,...) [13]. You can also overname the file using *NAME=stmt*. If there is more than 1 matching of stmt, the files will be named *(NAME, NAME1, NAME2,...)*.

Example:

```
/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*.

If *filename* is put inside ‘ quotes, no wildcard or expansion logic will apply to it.

You can also add custom attributes to files in the format *FILE -aX –attribute=X -b*. The attributes will be applied to all files which will be matched using *FILE* bash expression.

Possible attributes

```
-n
--named
```

It means that csv file has the first line with the names of the columns

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

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

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

This changes the primary delimiter.

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

This changes the secondary delimiter char.

Example:

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

Currently, commas and CHARs, which are also quotes in Lsql, are not supported as delimiters.

If block

This block always begins with *if*. They accept arithmetic expressions, which should be convertible to Bool - either String *false*/*true*, Int (*0 false*, anything else *true*), or Bool. Rows with true are printed or aggregated, and rows with false are skipped.

Filtering is done before the aggregation.

You can imagine if block as where clause in SQL.

By block

This statement always begins with *by* and the rest of the block is a select expression. There can be only one By block in the whole command.

The by block is used to group the resulting set by the given atoms.

You can imagine by block as the group by clause in SQL.

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

If there is an aggregate function present without by block present, aggregation runs over all rows at once.

Sort block

This block can be at the end of the command. It begins with the *sort* keyword and the rest is a select expression.

The sort block determines the order of the final output - given atoms are sorted in ascending order.

You can imagine a 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. One way is through Makefile and the second through Cabal. By running the following command it generates the build folder and 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* ($\geq 4.9 < 4.17$), *text* ($\geq 1.2 < 1.3$) and *containers* ($\geq 0.5 < 0.7$)) installed. The package boundaries given are identical to 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 is called by:

```
make test
```

It should always succeed before any commit 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.

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 a primary representation for 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 *lsql-csv* binary. It parses the arguments, checks, whether help optional argument was called or no argument at all was given, and either shows help or further call run from *Lsql.Csv.Main* from library.

Library is split into 5 different namespaces. Its usage is not strictly defined, but this can be said

- *Lsql.Csv* – This namespace contains the entry point
- *Lsql.Csv.Core* – This namespace contains the logic of evaluation
- *Lsql.Csv.Lang* – This namespace contains parsers
- *Lsql.Csv.Lang.From* – This namespace cointains parsers for from block
- *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 and functions for getting specific types of blocks from list of *Block*.
- *Lsql.Csv.Core.Evaluator* – This module contains the evaluator of *lsql-csv* program.
- *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.
- *Lsql.Csv.Core.Tables* – This module contains the definition of *lsql-csv* data types, classes over them and types *Table* and *Column* and functions over them for manipulation of them.
- *Lsql.Csv.Lang.Args* – A module for command line argument parsing.
- *Lsql.Csv.Lang.BlockChain* – This module contains a main parser of blocks other than from block.
- *Lsql.Csv.Lang.BlockSeparator* – This module contain a preprocessor parser, which splits command into list of strings - one string per one block.

- *Lsql.Csv.Lang.Options* – This module implements a common *Option* type and it is parsers for from blocks and command line optional arguments.
- *Lsql.Csv.Lang.Selector* – This module implements the selector expression and arithmetic expression parsers.
- *Lsql.Csv.Lang.From.Block* – This module contains the from block parser. It loads initial *SymbolMap*.
- *Lsql.Csv.Lang.From.CsvParser* – This module contains *CsvParser* called by *parseFile*, which loads input CSV files.
- *Lsql.Csv.Main* – This module contains the starting point for *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 Entry point

The entry point is in module *Lsql.Csv.Main* in function *run*. The function first calls preprocesor in *Lsql.Csv.Lang.BlockSeparator*, which splits command into list of strings. Then *SymbolMap* with input is loaded using *Lsql.Csv.Lang.From.Block* and after that rest of blocks are parsed using *Lsql.Csv.Lang.BlockChain*. Program is then evaluated using *Lsql.Csv.Core.Evaluator* and finally output generated by *Lsql.Csv.Utils.CsvGenerator*.

3. Analysis

Why the language have been made the way it is? Why it is so inspired by SQL and is not just next implementation of SQL? Why it is implemented the way it is? Further chapter is about design decision of the language.

3.1 Why SQL at first place?

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

- Oracle DB have used it since 1979 as a first commercially available implementation [15].
- MySQL have used it since 1994, since its original development started [16]
- PostgreSQL have used it since 1996, since it was created [17].
- MSSQL have used it since 1989, since its initial release [18].

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 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 argument 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 column and data value belongs to some data type) [5], which implied many design choices. Furthermore, SQL itself require large amount of text to be written, before it can be executed.

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

```
select A.dataX from data A where A.dataX > 1000
```

This simple SQL query shows $dataX > 1000$ from table *data*. 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 query. It is simply said the another reason, why we will not just implement another SQL implementation.

3.3 Why number references?

Where the 35% difference happened. One and the main reason, why is, that we allowed referencing *dataX* as *.2*. This is also possible due to nature of CSV file, where columns have their index [1]. Normally, in 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, new columns may be added or removed. Just because somebody removed a column, it is not wanted, that developers needed to change some of the queries so they comply with the new database layout.

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

Also *lsql-csv* is rather a tool for daily life and simple scripts rather than tool for development of medium sized or large sized project, like SQL is. This adds much more flexibility in what can language do (like number references).

3.4 Why renaming standard 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 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.' [19].

The tool for reverting the order of sorted output already exist: *tac*. Piped together it creates the wanted output.

Similar case, the function for second filtering of grouped by (in SQL named *having*) output have 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 usage of *echo* called before *lsql-csv* if needed.

3.6 Why blocks are delimited by comma?

The author thinks, it is more readable like this. Developers in SQL usually use upper case and new line writing to delimiter the blocks and comma is just more

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

easy to write than switching on and off caps lock, holding shift key or making multiline input.

3.7 Why there are two types of expression?

Why there are arithmetic and select expression with different grammars? We think, that addition of special select expression further allows the user to write shorter queries. It allows us to introduce wildcard and curly brackets expansion for the user, which will not be possible otherwise.

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 user to import CSV data to SQL database and use standard SQL if he needs more complicated joins.

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

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

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

As CSV parsing is not 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* will be primary text representation.

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

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

For larger dataset users are encouraged to use standard SQL database.

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 of standard SQL database and do the queries over it. This approach requires definition of schema to 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.

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

By using a standard SQL database you gain the advantage of better performance, typed dataset, indexes and large amount of build-in functions

4.2 Using standard Unix tools

It is possible to do 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 using *awk* to

```
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 the 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 *lsql-csv* join have $\mathcal{O}(nm)$ time complexity, while standard Unix tools have for join written above $\mathcal{O}(n \log n + m \log m)$ time complexity, so for larger dataset, it might be more beneficial to use them.

[1] It is possible, it can be written in shorter and more readable form, but 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* [6]
- *CSV SQL* [7]
- *trdsql* [8]
- *csvq* [9]

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

The advantage of it is you don't have to learn 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 by 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, it will take to much code to be written for any query.

Conclusion

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/info/rfc4180 [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] ISO 9075:1987: Information technology – Database languages SQL – part 1: Framework (SQL/Framework). Standard, International Organization for Standardization, Geneva, CH, June 1987.

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

[6] q - text as data. https://github.com/harelba/q, January 2022. [Online; accessed 18-February-2024].

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

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

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

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

[11] 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].

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

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

[14] 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].

[15] Usha Krishnamurthy et al. *Oracle Database SQL Language Reference, 19c*. 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].

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

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

[18] 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].

[19] 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].

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment