Lisp Query Notation—A DSL for Data Processing

Anders Hoff
inconvergent@gmail.com

ABSTRACT

This paper introduces Lisp Query Notation (LQN). A Common Lisp library, DSL and command-line utility for manipulating text files, and structured data such as JSON and csv, and Lisp Source code.

First we introduce the motivation and design principles. Then we present the LQN syntax, and demonstrate how to use LQN as a library to manipulate data structures in CL. Moreover we demonstrate how to use LQN from the command-line. After describing several operators and their syntax in more detail we finally describe a few possibilities for improvements and further work.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

demonstration, command-line utility, data processing, domain specific language, structured data, functional programming, common lisp

ACM Reference Format:
Anders Hoff. 2024. Lisp Query Notation—A DSL for Data Processing. In *Proceedings of European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/11001584

1 INTRODUCTION

Lisp Query Notation (LQN) [1] is a Domain Specific Language (DSL), Common Lisp (CL) library, and command-line utility for text and data processing. It draws inspiration from other well-known text processing tools, such as Sed, AWK, and jq[2]. In particular LQN mimics jq's tacit style and chaining of operations.

We start by describing the motivation and main features of the query language. Further we introduce the LQN CL library, before we make the seamless transition to using the LQN terminal commands. Finally we comment on some implementation details and challenges; performance improvements; and potential further work.

2 MOTIVATION, DESIGN AND IMPLEMENTATION

LQN started as an exercise. The primary motivation beyond that is to develop a terse, but intuitive language that makes it fast and convenient to write small (sometimes throw-away) programs; primarily on the command line. Where all—or most—of the processing can be done in the same language. It should also be possible to fall back to conventional CL when LQN is incomplete, inconvenient or insufficient.

It should handle common data formats such as plain text, JSON, csv and Lisp data. Moreover it should handle tasks encountered by e.g. data scientist, data engineers, and when making Generative Art. All of which are practices that require data wrangling. The latter is particularly relevant as we use CL for our art practice. As such it is convenient to export, import and process data in a format native to Common Lisp.

2.1 Compiler

The current compiler performs all code generation in a single pass. Because of this inherent simplicity the core of the compiler is only about 400 lines of code. The syntax is flexible enough that mixing the LQN syntax with CL and other libraries does not appear to present much friction.[3]

2.2 Internal Data Representation

In order to handle multiple data formats LQN always loads all input data into native CL objects. Primarily vectors and hash-tables. E.g. text files are read into vectors of strings; whereas JSON is read into vectors and hash-tables, depending on the structure.

```
[ { "id": "1",
    "objs": [ { "obj": "Ball",
                "is": "round" } ] },
  { "id": "2",
    "msg": "Hi",
    "objs": [ { "obj": "Box",
                "is": "empty" },
              { "obj": "Yak",
                "is": "shaved" },
              { "obj": "Computer" } ] },
  { "id": "3",
    "msg": "Hello!",
    "objs": [ { "obj": "Paper" },
              { "obj": "Bottle",
                "is": "empty" } ] } ]
```

Listing 1: Contents of `dat.json`

2.3 CL Library

In the first example we use the function, jsnloadf, to load some JSON from a file. The contents of dat.json are in listing 1.

```
* (in-package :lqn)
* (jsnloadf "dat.json")
> #(#<HASH-TABLE :COUNT 2 {1793}>
```

[1] https://github.com/inconvergent/lqn (v. 2.0.1)
[2] https://jqlang.github.io/jq/

[3] So far LQN has only been tested in SBCL on Ubuntu 22.04 LTS, with a limited number of other libraries.

ELS'24, May 06–07, 2024, Vienna, Austria

https://doi.org/10.5281/11001584

```
    #<HASH-TABLE :COUNT 3 {1E43}>
    #<HASH-TABLE :COUNT 3 {2B33}>)
```

We see that the JSON file has been loaded into a CL `vector` with three `hash-tables`. Storing data in `hash-tables` and `vectors` internally facilitates easy data manipulation and extraction. For the LQN compiler as well as the user. We can use `ldnout` to serialize data before printing it:

```
* (ldnout (jsnloadf "dat.json"))
> #(((:ID . "1")
     (:OBJS . #(((:OBJ . "Ball")
                 (:IS . "round")))))
    ...
    ((:ID . "3")
     (:MSG . "Hello!")
     (:OBJS . #(((:OBJ . "Paper"))
                ((:OBJ . "Bottle")
                 (:IS . "empty"))))))
```

Note that `ldnout` serializes to a combination of `vectors`, and `alists` with `keyword` keys. We will use the acronym LDN—"Lisp Data Notation"—to refer to this particular way of serializing such a nested structure.[4]

The primary entry point to the LQN compiler is the `qry` macro. The following is a query that simply returns the input. As we will see shortly, _ is used to refer to the current data in any operator or context.

```
* (defvar *dat* (jsnloadf "dat.json"))
* (ldnout (qry *dat* _))
```

2.4 Queries & the Get Operator

There are several different ways to get, select or iterate data in LQN. The simplest is `(@ ...)`, which can be used to access a particular key, index or path. The optional second argument is used as a default value. Here are some examples:

```
* (qry *dat* (@ :1/msg))
> "Hi"
* (qry *dat* (@ :1/abc :missing))
> :MISSING
* (ldnout (qry *dat* (@ :*/msg)))
> #("Hi" "Hello!")
* (ldnout (qry *dat* (@ :*/msg :nope)))
> #(:NOPE "Hi" "Hello!")
* (ldnout (qry *dat* (@ :*/objs/*/obj)))
> [ [ "Ball" ],
    [ "Box", "Yak", "Computer" ],
    [ "Paper", "Bottle" ] ]
```

This makes it easy to quickly look at parts of a data structure. For brevity we omit calls to `ldnout` in the examples from now on.

2.5 Pipe, Map and Filter

Data can be chained, iterated and filtered in a few different ways. First we consider these three operators:

- `(||...)`: pipe the result of each clause to the next clause. Returns the last result. We will see that it usually isn't necessary to use pipe explicitly;
- `#(...)`: map these clauses across a `vector`. If there are multiple clauses they are wrapped in a pipe. Returns new `vector`;
- `[...]`: filter `vector` by one or more clauses. Returns new `vector`.

The following is an example where we chain a map and filter together. First we select the `id` from each item and convert it to an integer; then we drop odd values.

```
* (qry *dat* (|| #( (@ :id) int! )
                 [ evenp ] ))
> #(2)
```

We note that bare symbols (`int!`, `evenp`) inside their respective operators are called as functions with _ as the first argument. Moreover, `qry` will wrap all arguments beyond the first in an implicit pipe operator. So we get the same result if we write this:

```
* (qry *dat* #( (@ :id) int! ) [ evenp ])
```

You can also use arbitrary expressions:

```
* (qry *dat* #( (@ :id) int! (+ 10 _) )
               [ (< _ 13) ])
> #(11 12)
```

If there are multiple clauses in the filter, the default is to include items that match either clause. E.g. `[evenp (< _ 2)]` to select even numbers as well as numbers smaller than 2.

To require multiple clauses to be satisfied, use the `+@` modifier:

```
* (qry *dat* #( (@ :id) int! )
               [ +@oddp (+@< _ 2) ])
> #(1)
```

Similarly, the `-@` modifier drops items on some condition; in this case the number 1:

```
* (qry *dat* #( (@ :id) int! )
               [ oddp (-@= _ 1) ])
> #(3)
```

The full behaviour of the filter modifiers is explained in the documentation. They can be combined to some extent, but if the behaviour of the modifiers do not suit your situation, you can use regular CL, as we have seen already.[5]

2.6 LQN on the Command-line

The transition to use LQN in the terminal is virtually seamless. Currently there are three different entry points to LQN: Namely the commands `tqn`, `jqn` and `lqn`; for `txt`, JSON, and Lisp data respectively. They expect different input data formats, but they all behave in (nearly) the same way. You can always output data to any format from either terminal command, as you can see in this excerpt from the output of `tqn -h` on the command-line:

```
$ tqn -h
> Usage:
    tqn [options] <qry> [files ...]
    cat sample.csv | tqn [options] <qry>
```

[4] In time we might add support for Extensible Data Notation (`edn`), which is a little more more pleasant to look at. LDN will do for now.

[5] We also note that the behaviour of these modifiers is one of the open questions of the overall design of LQN.

```
Options:
  -v prints the compiled qry before the
  result. For debugging.
  -j output as JSON.
  -l output to readable lisp data (LDN).
  -t output as TXT [default].
  -z preserve empty lines in TXT.
  ...
```

2.7 Processing text

To start, here is a query that splits the incoming `string` at every `"x"`, before it converts each new `string` to uppercase (`sup`). `splt` will trim off any white space by default.

```
$ echo 'a b c x def x 27'\
  | tqn '(splt _ :x) sup'
> A B C
  DEF
  27
```

```
abc
1
33
def
abcdefghi
7
```

Listing 2: Contents of `dat.txt`

Notice that the first expression receives the entire incoming `string` as its input. Whereas "bare" top-level `symbols` inside the implicit pipe operator are called on each individual item in the incoming `vector`; i.e. they are shorthand for the map operator.

Next we read from the `txt` file `dat.txt`. You can see the contents in listing 2. This query finds strings that contain the substring `"ghi"`, as well as all items that can be parsed as an integer by `int!?`:

```
$ tqn '[:ghi int!?]' dat.txt
> 1
  33
  abcdefghi
  7
```

We have seen the filter operator already. But note that now a `keyword` is used as shorthand for case insensitive substring search.[6] `strings` do the same, except then the case is required to match.

The syntax is the same as we saw when using `LQN` as a library. So we can use modifiers to require multiple substring matches:

```
$ tqn '[:+@abc :+@ghi]' dat.txt
> abcdefghi
```

```
animal,cat,angry
animal,yak,shaved
obj,pen,red
shape,ball,round
shape,box,square
```

Listing 3: Contents of `dat.csv`

[6]There is no explicit support for `regex` in `LQN` yet. But using existing libraries is trivial.

2.8 Transforming with Selectors

A frequent task when handling data structures like in listing 1 is iterating all items to perform some selection and/or transformation. `LQN` has several operators for this purpose:

- `{...}`: select keys from a `hash-table` into a new `hash-table`;
- `#{...}`: select keys from `vector` of a `hash-tables` into a new `vector` of `hash-tables`;
- `#[...]`: select keys from `vector` of a `hash-tables` into a new `vector`.

So to select the `id` and `msg` fields from all items we can do this:

```
$ jqn '#{ :id :msg }' dat.json
> [ { "id": "1", "msg": null },
    { "id": "2", "msg": "Hi" },
    { "id": "3", "msg": "Hello!" } ]
```

If you also want to transform some keys you can do this instead:

```
$ jqn '#{ (:id (+ 10 (int! _)))
          (:?@msg sup) }' dat.json
> [ { "id": 11 },
    { "id": 12, "msg": "HI" },
    { "id": 13, "msg": "HELLO!" } ]
```

Again we see that bare symbols are interpreted as a function with the current value as the only argument. Whereas expressions are evaluated as they are.

Notice that we have used the `?@` modifier to handle that the first item is missing a field. In all are three modifiers to augment the behaviour of selectors:

- `?@`: include if the key exists and its value is not `NIL`;
- `%@`: include only if our expression is not `NIL`;
- `-@`: drop this key.

Consider this query to see how the `%@` modifier only includes the `msg` key if the length of the `string` is greater than 3.

```
$ jqn '#{ :id
          (:%@msg (and (> (size? _) 3)
                       (sup _))) }
      ' dat.json
> [ { "id": "1" },
    { "id": "2" },
    { "id": "3", "msg": "HELLO!" } ]
```

Sometimes you want everything except some keys. The `-@` modifier combined with _ allows us to discard or override keys. As such, the following will yield the same output as we just saw:

```
$ jqn '#{ _ :-@objs
            (:%@msg (and (> (size? _) 3)
                         (sup _))) }
      ' dat.json
```

Selectors, and all other operators can be nested. Here is one more example where we use nested selectors, and print the result as newline separated `JSON`.

```
$ jqn -tjm '#[ (:objs #[ (:obj sdwn)
                         (:?@is sup) ]) ]
```

```
           ' dat.json
> ["ball","ROUND"]
  ["box","EMPTY","yak","SHAVED","computer"]
  ["paper","bottle","EMPTY"]
```

LQN could use more utilities for handling csv files properly. But here is a more complex expression to illustrate how to group items by the first column of the csv file in listing 3; before printing the output as JSON.

```
$ tqn -j '#( (splt _ ",") )
          (?grp (@) (new$ :id (@ 1)
                          :is (@ 2)))
         ' dat.csv
> { "animal": [
      { "id": "cat", "is": "angry" },
      { "id": "yak", "is": "shaved" } ],
    "obj": [...],
    "shape": [...] }
```

2.9 Other Operators

There are several other operators we haven't covered. Here are a few compressed examples that demonstrate additional operators, in combination with what we have seen already:

- ?srch: search nested data with custom expressions. E.g. this will find all symbols in a lisp file and sort them as strings:

  ```
  $ lqn -t "(?srch symbolp) (uniq _)
            (sort _ #'string-lessp)" <file>
  ```

- ?txpr: search and replace nested data. The following will alter the id of any JSON object with at least two items.

  ```
  $ jqn '(?txpr (>= (size? (@ :items)) 2)
                {_ (:id (str! "new-" _))})
        ' [file]
  ```

- ?fld: reduce with a function or expression. This expression will parse and sum integers from the second column in a csv file:

  ```
  tqn '#((splt _ ",") (@ 1) int!?) [is?]
       (?fld 0 +)' [file]
  ```

- ?rec: repeat an expression while the/an expression is T. The following will iteratively add the next Fibonacci number to the end of the incoming vector, until it reaches a number larger than 50:

  ```
  $ echo '#(1 1)' |  lqn '
     (?rec (<= (@ -1) 50)
           (cat* _ (apply* + (tail _ 2))))'
  ```

LQN also has a number of other utilities for numbers, vectors, hash-tables, strings, symbols and lisp data. Such as: getting ranges, indices or keys; concatenating, compacting and combining; comparing; and checking and coercing types. The LQN documentation covers this in more detail.

3 NOTES ON PERFORMANCE

The current implementation of LQN loads all the input into appropriate data structures before executing any of the transforms. This simplifies the implementation, and makes it possible to do some things that would otherwise be difficult or impossible. However, it also has implications for performance; most notably it can increase memory usage. A possible way alleviate this is to adapt the various operators to support e.g. generators or streams in addition to vectors and hash-tables.

Depending on the environment there might be a noticeable delay when using LQN on the command-line, as SBCL first has to start and load the LQN library, before it can load the input data and execute the query. A possible way to reduced this delay considerably is to create an SBCL image (sb-ext:save-lisp-and-die) where LQN is already loaded. Then use the image to compile and execute the query.

Additionally it is noticeably slower to process data from a pipe on the command-line; compared to having LQN read the same data from a file, or from *standard-input*. We have been unable to find an explanation for this, but it is likely an issue in the current implementation; as opposed to an issue with the overall approach.

4 CONCLUSION & FURTHER WORK

lqn is a young experiment. It has not been tested in many different circumstances, or for different tasks. For this reason it is hard to know if the behaviour—such as defaults, and the order of arguments—of the operators and utilities are convenient. Obviously this will always depend on the use-case. But there is likely room for improvement that will become more apparent with further use.

There is also a discussion to be had about the syntax of several of the operators and modifiers. Particularly the selectors in section 2.8. Maybe the overall syntax can be cleaner, or easier to read?

The language is still missing native utilities for data processing. Such as calculating statistics (median, mean, variance), sorting, and aggregations. And while there are some ways to interact with other terminal commands, there is considerable room for improvement. The same applies to interacting with files and the file system. Both from the terminal and when using LQN as a library.

In a similar vein, it would be an interesting challenge to implement a more interactive command-line interface. Where the CLI interaction and syntax is closer to the LQN language than e.g. bash.

Finally we would like to note that we have already found LQN useful for performing code transformation in (CL) compilers for other DSLs. We did not anticipate this from the beginning, but it will probably shape the direction of further LQN development. In particular it would be helpful to improve utilities and operators when it comes to processing CL data; in particular data that represents source code. In all LQN is a useful little language, with potential for expansion in several dimensions.

5 ACKNOWLEDGEMENTS

Thanks to Jack Rusher, Robert Smith and Rainer Joswig for answering my questions; Zach Beane for making and maintaining Quicklisp; and thanks to the larger Lisp community for all their interesting work.