Faculty of Business and Economics
The University of Hong Kong

Data Processing Domain-Specific Language in Clojure

Research Report

20 March, 2022

CHOI Chong Hing

Supervised by Dr. Matthias Buehlmaier

Abstract

Clojure is one of the dialects of Lisp, developed with the code-as-data philosophy. Together with the powerful macro system, a variety of Domain-specific Languages (DSL) could be developed conveniently. This research project focus on integrating panda style data processing DSL into Clojure, exploring the limits of Clojure. The developed data processing DSL is based on an existing data processing library, tech.ml.dataset[1]. It provides the basic data processing operations for the project. This development of the DSL has three stages: conceptual, fundament and syntax design. The conceptual phase involves experimenting with Clojure macro system and defining the project goal. The Fundament stage involves the development of the fundamental pipeline of the DSL, following the logical processing order of the SELECT statement in Structured Query Language (SQL). Lastly, during the syntax design part, the limits of Clojure syntax was explored.

Acknowledgements

I would like to express my special thanks of gratitude to Dr Matthias Buehlmaier for giving me this valuable chance to get involved in this research project. This project would not be possible without his guidance and support throughout the project.

Contents

List of Figures

List of Tables

Abbreviations

DSL	Domain-specific Language
GPL	General-purpose Language
RDBMS	Relational Database Management System
SQL	Structured Query Language

1 Introduction

1.1 Overview

Clojure is a functional programming language, a dialect of Lisp. It is excellent for concurrency operations with concise syntax and immutable data structures. It simplifies concurrency or multi-threaded programming due to its immutable core data structures. The in-built macro system in the Lisp languages with the code-as-data philosophy also enables huge flexibility in programs.

1.2 Macros in Clojure

The macro system in Clojure allows the compiler to be extended by code. It reads the input code as data, phrasing it into a different code for execution. The transformation using the macros could be defined by the user.

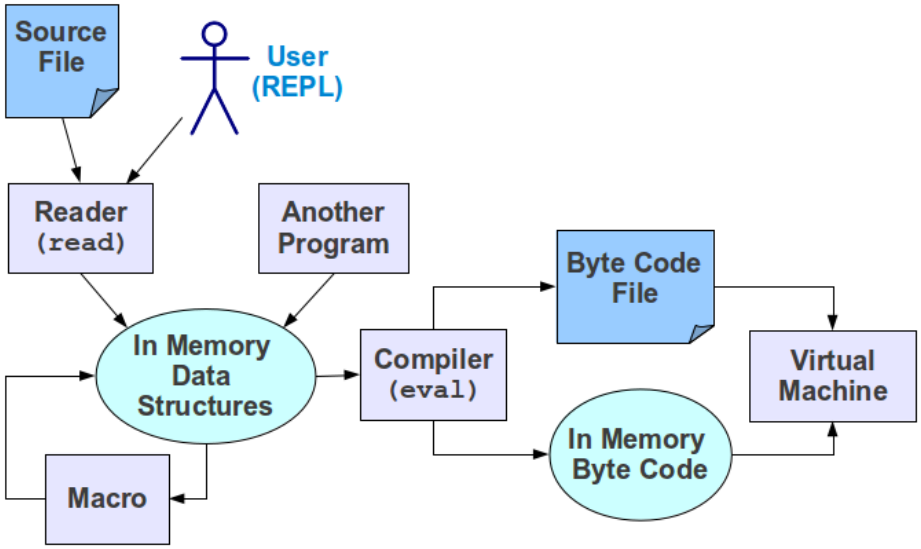

Figure 1.1: Clojure Compilation Overview [2]

Figure 1.1 shows the overview of the Clojure compilation. The macro system lies in a loop in the memory data structures, it represents the transformation between code and data through macros. Such a feature provides an excellent foundation for defining a syntax in Clojure, subject to its fundamental syntax.

1.3 Domain-specific Language

Domain-specific Language is a computer language, declared syntax or grammar that is specialised in a specific application. In contrast to General-purpose Language (GPL), the implementation of DSL is designed with specific goals in that application domain. The use of macros in Lisp dialects enables developers to rewrite source code at compile-time, making implementation of DSL more convenient. As one of the Lisp dialects, Clojure also inherits such an advantage. In addition to macros, the heavy use of core data literals in Clojure also gives an extensive developing opportunity in implementing DSLs.

1.4 Objectives

In this project, a DSL extension to the existing data processing library, tech.ml.dataset[1], will be developed. A generic query using core data literal serves as the foundation of the DSL. This enables huge flexibility in defining the syntax, subject to Clojure's limitation.

1.5 Report Outline

This progress report is structured into five chapters. The first chapter offers a brief overview of Clojure and the background of the project. It also gives the objectives of the project.

Chapter two analyses the methodology of the project. The fundamental pipeline of the DSL is explained in this chapter. This includes the introduction to Structured Query Language,

Chapter three shows the development process of the project. It also presents the project schedule.

Chapter four explains the results of the project. The details of the pipeline and syntax design will be discussed in this chapter.

Chapter five concludes the report, summing up all the results made in the project.

2 Methodology

2.1 Introduction

This chapter explains the methods used in the development of the DSL in this project. The pipeline of the DSL will be discussed. The logic of query operation in SQL and the DSL will also be explained in this chapter.

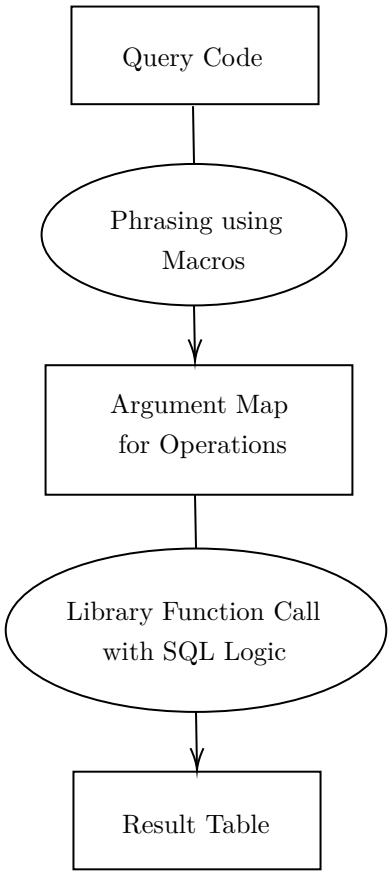

Figure 2.1: Overview of Methodology

Figure 2.1 shows the overview of the methodology of the project. The DSL takes the query code written in the custom syntax as input, phrasing it into a Clojure map containing the arguments of different operations. The library functions are called according to the SQL query logic, returning the data table.

2.2 Structured Query Language

Structured Query Language is a declarative query language designed for managing the data in a Relational Database Management System (RDBMS).

2.2.1 SELECT Statement Logical Processing Order

The SELECT statement query of the SQL is very similar to the query of the DSL developed in this project. The logical processing order of the SELECT statement has been adopted in the DSL.

Table 2.1: Logical Processing Order of the SELECT Statement[3]

Order	Operations	Description
1	FROM	Specifies a table, view, table variable, or derived table source, with or without an alias, to use in the Transact-SQL statement
2	ON	Specifies arbitrary conditions or specify columns to join
3	JOIN	Retrieves data from two or more tables based on logical relationships between the tables
4	WHERE	Specifies the search condition for the rows returned by the query
5	GROUP BY	Divides the query result into groups of rows
6	WITH CUBE WITH ROLLUP	Extend functions for GROUP BY
7	HAVING	Specifies a search condition for a group or an aggregate
8	SELECT	Specifies the columns to be returned by the query
9	DISTINCT	Specifies to return only distinct values
10	ORDER BY	Sorts data returned by a query in SQL Server
11	TOP	Specifies the number of records to return

Table 2.1 shows the logical processing order of a SELECT statement in SQL. The development of the pipeline in this project has referenced its logic, making a generic query possible.

2.2.2 DSL Logical Processing Order

In contrast to the complete query of the SELECT statement in SQL, the DSL developed has fewer operations involved. Table 2.2 shows the logical processing order of the DSL developed.

Table 2.2: Logical Processing Order of the DSL

Order	Operations	Description
1	WHERE	Specifies the search condition for the rows returned by the query
2	ROW	Specifies the row index for the rows returned by the query
3	GROUP BY	Divides the query result into groups of rows
4	HAVING	Specifies a search condition for a group or an aggregate
5	SELECT	Specifies the columns to be returned by the query
6	ORDER BY	Sorts data returned by a query

The DSL developed supports only a few operations shown in Table 2.2. However, one may discover that the DSL supports the uses of row index for query, which is not supported in the SELECT statement in SQL.

2.3 Summary

This chapter explained the methodology used in this project. The logical processing order of the SELECT statement in SQL and DSL are explained in this chapter. The next chapter will show the development process of this project.

3 Project Schedule

3.1 Overview

This chapter presents the project schedule and the development stages for the DSL in the project. The achievements of different stages are also discussed in this chapter.

3.2 Project Schedule

Table 3.1 shows the project schedule for the project.

Table 3.1: Project Schedule

Timeline	Task
Nov 21	Literature Review
Dec 21	Conceptual Phase
Jan 22	Fundaments Development
Feb 22	Syntax Design
Mar 21	Additional Feature

Literature Review

During this phase, a comprehensive review of Clojure was done. This includes some coding practices and documentation review. This has solidated the coding skills in Clojure, helping to understand Clojure's abilities and limitations. It also helped us understand functional programming in general.

Conceptual Phase

The conceptual phase mainly focused on deciding the topic of this research topic. With the understanding developed in the previous stage, the topic of developing a DSL has been decided for the project at this stage. The pipeline for the DSL was also decided and designed during this phase.

Fundaments Development

During this stage, a generic query with an argument map was developed. It serves as the foundation of the project DSL. It includes a combination of data query operations, including selecting rows and columns, filtering with conditions and group by. The generic query should be able to execute multiple operations. To achieve such promise, the processing steps of SQL has been used as a reference in designing the generic query. This corresponds to the 'Library Function Call with SQL Logic' in Figure 2.1.

Syntax Design

With the generic query foundation implemented, the design of syntax is the next step. In this stage, the mix of Clojure literals and macros are used, depending on the syntax design. The macro system in Clojure, and other Lisp dialects, allows developers to extend the compiler by manipulating code expression at compile-time. It suspends the compilation of an expression and brings the expression for user manipulation. Expressions with wrong grammar or unknown keywords could be correctly and manually compiled via macros, making different expressions possible in Clojure. With the use of macros, the implementation syntax will be much more flexible. Ultimately, a syntax with pandas-Clojure style is designed and implemented. This corresponds to the 'Phrasing using Macros' in Figure 2.1.

Additional Feature

During this stage, additional features like sorting and selecting all columns were added to the existing DSL system. These have been implemented by adjusting the syntax and the fundamentals of the DSL system. Debug was also done during this stage.

4 Project Results

This chapter explains the project results, including the DSL syntax design. The design will be discussed in Section 4.1. Some examples will be displayed in Section 4.2.

4.1 Syntax Structure

The syntax of the DSL takes a data table in the first input and query arguments in the second input. The argument input has three sections: row selection section, column selection section and optional section, separated by the symbol '`&`'.

```
dt-get data '[ROW-SELECTION-SECTION & COLUMN-SELECTION-SECTION & OPTIONS]
```

Figure 4.1: DSL Query Syntax

Figure 4.1 shows an overview of the query syntax. Here, `dt-get` is a macro because of the use of the symbol '`&`'.

Row Selection Section

The first section of the argument input is the row selection section. It corresponds to the WHERE, HAVING and ROW operations in Table 2.2. The user could either select the rows using filters or by row index. The use of a filter would override row index selection.

```
[col filter-function]
```

Figure 4.2: Row Selection by Filter

Figure 4.2 shows the syntax of row selection using a filter. `col` refers to the column to be filtered, `:*` can be used to include all rows. `filter-function` refers to the filtering function. This is one of the powerful features - the filtering function can any custom function returning a boolean result. One can define a filtering function for the selection using Clojure built-in fast function syntax: `#{ ... }`. This is valid as long as it returns a boolean.

```
row-index
```

Figure 4.3: Row Selection by Index

Figure 4.3 shows the syntax of row selection using row index. `row-index` refers to the index of the desired row.

```
[col filter-function] row-index
```

Figure 4.4: Row Selection with Both Filter and Row Index

Figure 4.4 shows the case where filtering overrides the use of row index. In this case, the filtering function would override the row index. The pipeline will ignore the `row-index` part, making this expression equivalent to the expression in Figure 4.2.

Column Selection Section

The second section of the argument input is the selection of columns.

```
col
```

Figure 4.5: Column Selection

Figure 4.5 shows the syntax of column selection. `col` refers to the column selected, `:*` can be used to represent all columns.

Optional Section

The third section of the argument section is the optional section. This section specifies all the optional operations, including the GROUP BY and SORT BY operations.

```
operation-keyword operation-arguments
```

Figure 4.6: Optional Operation

Figure 4.6 shows the syntax of an optional operation. `operation-keyword` refers to the operation keyword for the program to identify the operation. It includes `:group-by` and `:sort-by`. `operation-arguments` refers to the corresponding operation arguments, subject to the operation.

```
:group-by col
```

Figure 4.7: Group by

Figure 4.7 shows the syntax of a group by operation. `col` refers to the column(s) to be grouped.

```
:sort-by col sort-by-function
```

Figure 4.8: Sort by

Figure 4.8 shows the syntax of a sort by operation. `col` refers to the column to be sorted. `sort-by-function` refers to the sorting function, with `<` (ascending order) as default. Similar to the filtering function, the sorting function can be any custom function returning a boolean result. It can also be Clojure operator like `<` or `>`, `clojure.core/compare` or custom `java.util.Comparator`.

Aggregate Function

With the group-by operation is implemented, aggregate functions are also needed to be implemented in the syntax.

```
aggregate-keyword col
```

Figure 4.9: An Aggregate Column

Figure 4.9 shows the syntax of an aggregated column. `aggregate-keyword` specifies the aggregated function. `col` refers to the column to be aggregated. One could directly replace the aggregated column syntax in any column argument. Table 4.1 shows the complete aggregate functions available and the corresponding aggregate keywords.

Table 4.1: Aggregate Functions

Aggregate Function	Keyword
Minimum	`:min`
Maximum	`:max`
Mode	`:mode`
Summation	`:sum`
Standard Deviation	`:sd`
Skew	`:skew`
No of Valid Rows	`:n-valid`
No of Missing Rows	`:n-missing`
Total No of Rows	`:n`

4.2 Examples

This section displays some of the examples with the DSL syntax. `data` in Table 4.2 will be used in the examples.

Table 4.2: `data` Used in the Examples

:age	:name	:salary
31	a	200
25	b	500
18	c	200
18	c	370
25	d	3500

Example 1

Select rows with salary > 300, age < 20

```
dt-get data '[[:salary #(< 300 %)] [:age #(> 20 %)] & :*]
```

:age	:name	:salary
18	c	370

Example 2

Group rows by age with sum of salary > 1000, show age and sum of salary

```
dt-get data '[[:sum :salary #(< 1000 %)] & :age :sum :salary & :group-by :age]
```

:age	:salary-sum
25	4000

Example 3

Group rows by age, show age, sum of salary and standard deviation of salary, sorted by standard deviation of salary in descending order

```
dt-get data '[:* & :age :sum :salary :sd :salary & :group-by :age :sort-by :sd :salary >]
```

:age	:salary-sum	:salary-sd
25	4000	2121.32
18	570	120.21
31	200	0

Example 4

Group rows by age and name, show age, name and sum of salary

```
dt-get data '[:* & :age :name :sum :salary & :group-by :age :name]
```

:age	:name	:salary-sum
31	a	200
25	b	500
18	c	570
25	d	3500

5 Conclusion

This project has developed a DSL that integrates pandas style queries into Clojure with the help of the powerful macros system. It explores the abilities and limits of Clojure in creating new syntax. The resulting DSL can perform a basic query with row selection, column selection, group by operation and sorting.

References

[1] "Techascent/tech.ml.dataset: A clojure high performance data processing system." (), [Online]. Available: https://github.com/techascent/tech.ml.dataset.

[2] A. Ortiz. "Clojure macros." (), [Online]. Available: https://arielortiz.info/s201911/tc2006/clojure_macros/clojure_macros.html.

[3] Microsoft. "Select (transact-sql) - sql server." (), [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/queries/select-transact-sql?redirectedfrom=MSDN&view=sql-server-ver15.