Supporting Information

FFMDFPA：A FAIRification Framework for Materials Data with No-Code Flexible Semi-Structured Parser and API

Bing He[a], Zhuming Gong[a], Maxim Avdeev[c,d], Siqi Shi[b,e]

[a] *School of Computer Engineering and Science, Shanghai University, Shanghai 20444, China*

[b] *Materials Genome Institute, Shanghai University, Shanghai 200444, China*

[c] *Australian Nuclear Science and Technology Organisation, Locked Bag 2001, Kirrawee DC NSW 2232, Australia*

[d] *School of Chemistry, The University of Sydney, Sydney 2006, Australia*

[e] *State Key Laboratory of Advanced Special Steel, Shanghai Key Laboratory of Advanced Ferrometallurgy, School of Materials Science and Engineering, Shanghai University, Shanghai 200444, China*

*E-mail: sqshi@shu.edu.cn (Siqi Shi)

S1. Definition of terminologies

Materials calculation: In this paper, "materials computation" specifically refers to the use of high-throughput computing in the field of materials science to explore and analyze large-scale datasets and computational models rapidly and efficiently. This approach greatly expedites the discovery, design, and characterization of new materials with desired properties. By harnessing parallel computing, automation, and data-driven techniques, materials computation enables the swift screening and evaluation of numerous potential materials. In the realm of traditional materials science research, the process of experimentally synthesizing and characterizing materials has long been known to be a time-consuming and costly endeavor. However, high-throughput computing emerges as a promising solution to tackle this challenge head-on. By harnessing the power of computer simulations, data analysis, and advanced algorithms, high-throughput computing endeavors to expedite the screening and evaluation of an extensive array of potential materials. This transformative approach encompasses several key components and processes that collectively drive its efficacy in materials science. These include data generation, the establishment of comprehensive databases and repositories, computational screening, accurate property prediction, design optimization, meticulous data analysis, and crucial experimental validation. Through the seamless integration of these elements, high-throughput computing revolutionizes the landscape of materials science research, enabling researchers to efficiently explore and exploit a vast range of material possibilities.

Data exchange: In this paper, the term "data exchange" pertains to the intricate process of disseminating, conveying, and amalgamating scientific data within the realm of materials science. This encompassing procedure encompasses a diverse array of data exchanges, such as experimental findings, computational models, simulation outputs, material properties, characterization data, and metadata, among other pertinent information. The efficacy of data exchange assumes a pivotal role in propelling advancements in materials science research, fostering collaborative efforts, stimulating innovation, and facilitating knowledge discovery. It serves as a conduit for the dissemination of invaluable information, bolsters the reproducibility of experiments and simulations, enables the validation and verification of results, and expedites overall progress within the field. The principal facets and advantages of materials science data exchange encompass the sharing of research discoveries, the promotion of collaboration and interdisciplinary research, the integration and

extraction of data, the establishment of standards and representation of metadata, the cultivation of open science practices, and the enhancement of reproducibility.

Data services: In this paper, the term "data services" encompasses a range of services, platforms, and infrastructures that are specifically designed to support the management, planning, analysis, and dissemination of scientific data within the field of materials science. These services play a crucial role in facilitating access, integration, and utilization of data, thereby empowering researchers to gain valuable insights, make groundbreaking discoveries, and advance their research objectives. Key aspects and functionalities of these data services include comprehensive data management, meticulous data curation, seamless data integration, efficient data access and sharing mechanisms, adherence to data standards, and promotion of interoperability, among other essential features. Notably, web platforms such as AFLOW, Materials Project, OQMD, and various others fall under this category, exemplifying the diverse range of data services available to researchers in the materials science domain.

Algorithm integration: This paper focuses exclusively on the integration of algorithms within the realm of materials science, aiming to combine multiple computational algorithms or models in order to obtain a holistic comprehension of materials and their properties. This intricate process involves the amalgamation of diverse theories, computational methods, and experimental approaches, with the ultimate goal of enhancing the accuracy, efficiency, and predictive capabilities of material simulation and analysis. In the field of materials science, researchers employ a variety of algorithm models to simulate and forecast the behavior of materials across different scales, encompass atomic and molecular interactions as well as macroscopic properties. Nevertheless, due to the intricate nature of materials, no single algorithm model can fully encapsulate all the complexities and phenomena involved. Consequently, the integration of algorithms endeavors to surmount these limitations by merging multiple methods into a unified framework. Key facets of algorithm integration encompass multiscale modeling, hybrid simulation methods, data-driven approaches, software frameworks, and tools, among others. Overall, the integration of algorithms in materials science strives to transcend the constraints imposed by individual algorithm models, culminating in an all-encompassing framework that enhances the accuracy and predictive capabilities of simulations. This integration not only facilitates the execution of multiscale modeling in material simulation but also supports material

design and fosters scientific discoveries within the realm of materials science.

Algorithm models: Algorithm models in materials science are sophisticated computational frameworks that are constructed based on computer algorithms and mathematical formulas. These models play a crucial role in describing and predicting the properties, behavior, and performance of materials. By employing numerical calculations and simulations, they effectively address complex problems within the field of materials science and offer the capability to interpret and forecast experimental results. Algorithm models serve as powerful tools within the realm of materials science, empowering researchers to gain profound insights into diverse aspects of materials and anticipate the properties and behavior of novel materials. They provide essential theoretical and computational foundations that underpin the advancement of materials science and facilitate material design endeavors. It is worth noting that the integration of algorithms and the development of associated software tools, mentioned earlier, are firmly rooted in the foundation of algorithmic models.

Software tools: This paper refers to software tools in the field of materials science. These tools, which encompass computer programs and applications, play a crucial role in material research, analysis, simulation, and design. By providing researchers and engineers with a computational platform, these software tools enable them to delve into, model, visualize, and analyze materials across various scales and levels of complexity. The contributions of diverse types of software tools have been instrumental in advancing the field of materials science. Notable examples include density functional theory (DFT) software packages like VASP and Quantum ESPRESSO, molecular dynamics (MD) simulators such as LAMMPS and GROMACS, crystallography and structure visualization tools like VESTA and PyMOL, high-throughput screening software packages like AFLOW, pymatgen, AiiDA, machine learning and data analysis tools like TensorFlow and PyTorch, as well as fundamental programming languages like Python and C++.

S2. Overview of DSL syntax and examples

The syntax of the DSL is divided into two main types, as follows, shown in Fig. S1:

1. {% ... %} for statements
2. {{ ... }} for expressions

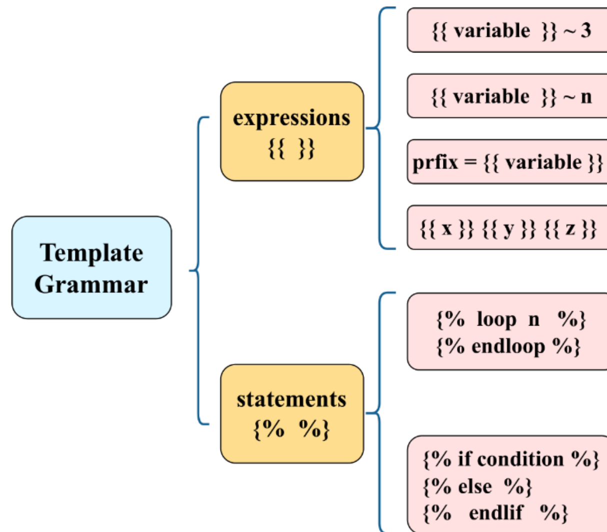


Fig. S1 Overview of the template grammar.

The following systematically introduces the syntax and semantics of the DSL and gives examples of practical usage.

{{ variables }} {{ variables _name }} ~ 3 {{ variables _name1 }} ~ variables {{ variables _name2 }} ~ N **a**	Configuration = {{ variable }} {{ x }} variable {{ y }} {{ x }} {{ y }} {{ z }} {{ x }} = {{ y }} = {{ z }} **b**	{{ element_number | sum }} ~ 3 {{ element_number | toFloat}} ~ 3 {{ element_number | toInt}} ~ 3 **c**
{{ element }} ~ 3 {{ element }} ~ sum(elements) {% loop sum(element) %} {{ coordinates }} ~ 3 {% endloop %} **d**	{{model}} {% if Direct %} {{ coordinates }} ~ 3 {% else %} {{ coordinates }} ~ 6 {% endif %} {% if model==3 %} {{ coordinates }} ~ 3 {% endif %} {% if 3<= model <=6 %} {{ coordinates }} ~ 3 {% endif %} **e**	{% loop 3 %} {{ lattice_vector }} ~ 3 {% endloop %} {% loop n %} {{ coordinates }} ~ 3 {% endloop %} {% loop elements_num %} {{ coordinates }} ~ 3 {% endloop %} {% loop sum(elements_num) %} {{ coordinates }} ~ 3 {% endloop %} **f**

Fig. S2 Use cases of each statement in the DSL.

Templates. A template is simply a text file or string that describes data in text form corresponding to the structure it represents. Templates contain variables and expressions (which can be thought of as placeholders for data at the current location) and labels (which control the logic of the template). Statements in a template should correspond to the data on a line-by-line basis.

Variables. Template variables are defined by the users and indicate the naming of the current position data. Variable names consist of any combination of alphanumeric characters and the underscore symbol ("_") but may not start with an underscore and number. The granularity of variable

matching data is set to 1 by default, and users can use a tilde (~) to control the number of matchings. The match number can be set as specific numbers, contextual variables, and N. Fig. S2a shows the use cases.

Contextual Variables. Contextual variables combine a regular string and variables, representing part of the data in a string. Fig. S2b shows the use cases.

Filter. Variables can be modified by filters. Filters are separated from the variable by a pipe symbol (|) and may have optional arguments in parentheses. Multiple filters can be chained. The output of one filter is applied to the next. Fig. S2c shows the use cases.

Functions. When a filter is used to process variables, the original data extracted in variables are overwritten, thus there is a problem with subsequent references to the original variables. As a result, when a reference variable is needed and further processing is required, we provide another syntactic form of filter, which we call a function. Fig. S2d shows the use cases.

Tags. Tags, such as {% tag %}, are control structures used in templates to control the flow of a program. They allow users to implement loops and logical operations. The *if* statement can be used to test if a variable is defined, not empty, and not false or for numerical comparisons. Meanwhile, the loop statement is designed to match text table data and can be set to specific values, contextual

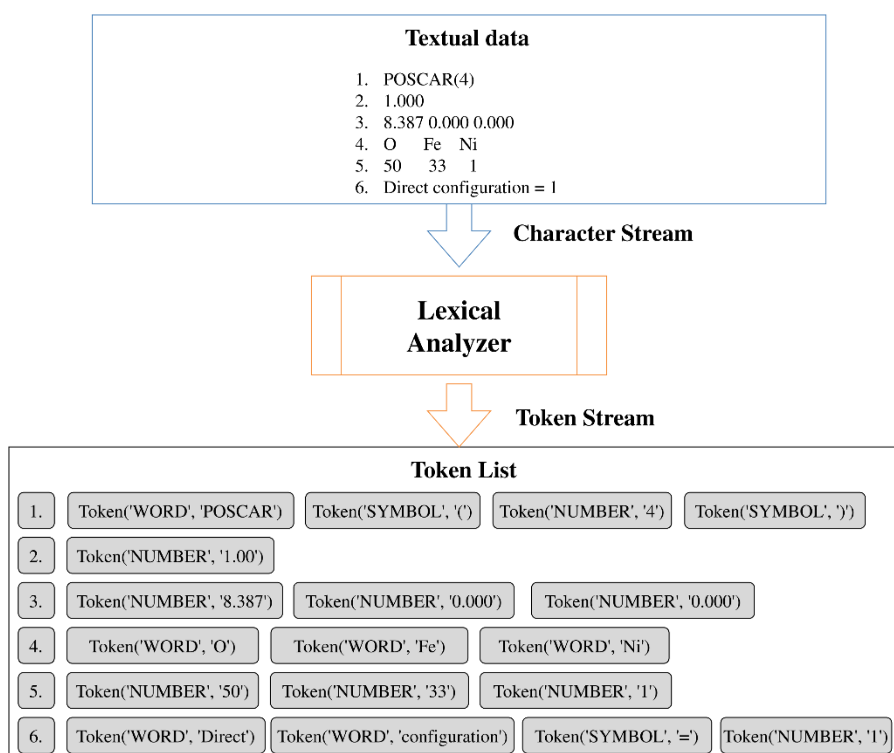


Fig S3. Example of lexical analysis. According to the defined syntax rules, the input data is scanned and converted, and the result is a token list, which is divided into three main types of tokens: words, numbers, and symbols.

variables, or N, where it will retrieve data corresponding to the number of rows or until the end of the file. Examples of these use cases can be found in Fig. S2e and Fig. S2f.

S3. Lexical analysis of data

The input data undergoes a crucial step known as lexical analysis, which involves the generation of tokens. Tokens serve as the basic units of data when templates and data align harmoniously. The tokenization process encompasses the establishment of rules and the thorough scanning of the text to identify words, punctuation marks, and other relevant elements. To provide a visual representation of this process, please refer to Figure S3. Furthermore, by incorporating additional tokenization rules, it becomes possible to achieve a more accurate representation of the data, enabling the identification of specific elements like chemical formulas within the text.

S4. Key features of the input data and the patterns of the parsed results

After the initial parsing of the input data and the creation of the semantic model of the template language, the next step involves matching the data with the semantic model. Prior to the data matching process, it is necessary to define a symbol table that serves two important purposes: storing the extracted data and performing semantic checks. The variable names in the symbol table are provided by the template, while the corresponding values of these variables are provided by the data. The final outcome of this process is a JSON string that represents the result obtained by combining the template and the data. When parsing the vast amount of semi-structured data generated by the computational materials science community and stored in textual form, our attention has been focused on the following key aspects:

1. Completeness of data parsing. To ensure the completeness of data parsing, when dealing with semi-structured data in text formats, it is essential to take into account various levels of data extraction. These levels include complete file extraction, data block extraction, and data line extraction. To cater to these specific scenarios, we employ three distinct function interfaces: parse(), match(), and search(). Each interface is designed to handle its respective scenario effectively.
2. Extraction of multiple rows of repeated data. In the semi-structured generated by the computing software as textual forms, it is common to find multiple consecutive lines of data that convey the same information, such as coordinates in POSCAR files and XDATCAR files for each data frame. In such scenarios, a loop statement becomes crucial to efficiently extract the required data. By utilizing a loop function, one can easily specify the desired number of

repeated lines or utilize the value of N to indicate the end of the file.

3. Basic logic analysis. In some files, the data exists in different forms, determined by the context, such as the form of coordinates in POSCAR files, e.g., fractional vs. Cartesian. In this case, we can use if statements to make selection based on the input.

The parsed result is formatted into JSON or XML. It can be easily extended to other formats, taking advantage of its independent internal structure. This enables easy conversion to different formats while ensuring scalability. The parsed results are organized in a style that follows several rules:

1. When the number of template matches is one, the value in the key-value pair is represented as a number or a string.
2. When the number of template matches is greater than one, the values in the key-value pair are stored in an array.
3. For complex strings containing symbols, if the number of template matches is one, then the characters are merged directly.
4. For the parsing result of the loop statement, naming conflicts need to be taken into account, so a two-dimensional array is used to represent the whole table, and a one-dimensional array is used to represent the data for each cycle.

S5. File formats supported by the parser

This parser is primarily designed for semi-structured data saved in text formats generated by computational software. However, we do not provide a detailed list of specific supported formats. This is because the parser is designed for a particular type of data that exhibits certain characteristics and can be parsed using this format, regardless of specific format restrictions. The characteristics of this type of data are explained in the S4. Therefore, we have not conducted extensive research, testing, and enumeration of all supported formats. Furthermore, as long as the data conforms to the characteristics of this type, the parser has the capability to parse data in that format. For example, we can customize the content of the text and use this parser for parsing, or if experimental data saved in text form also meets these characteristics, it can be parsed using this parser. Therefore, listing out the specific supported formats goes against the design principles of the parser.

The syntax design of the template language plays a crucial role in determining the supported formats that the parser can handle. In the case of this particular template language, it has been meticulously crafted based on the formats produced by VASP software. This meticulous design enables the successful parsing of various files, including POSCAR, XDATCAR, INCAR, KPOINTS,

OUTCAR, and many others. Although we haven't explicitly mentioned all the computational chemistry software packages and file formats that can be parsed, it is worth noting that the parser possesses the capability to construct suitable templates for semi-structured data in textual form generated by packages such as Gaussian and Quantum ESPRESSO. Moreover, the flexibility of the syntax design allows for easy expansion to accommodate new data patterns that may arise in the future, ensuring that unresolved formats can be handled seamlessly.

S6. Test environment and test methods

Test Environment:

Operating system: Windows 11 (version 22H2)

Processor: AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz

Memory: 16 GB RAM

Python version: 3.9.12

Pytest version: 7.3.0

Pytest-benchmark version:4.0.0

Test Method:

We used the --benchmark-autosave option to save benchmark results to disk, and ran each test case five times to ensure consistent results.

Here is an example of the command used to run the benchmark tests:

```
pytest benchmark_test.py --benchmark-autosave --benchmark-sort=mean    --benchmark-min-rounds=5 -v
```

In this command, we specify the --benchmark-autosave option to automatically save the benchmark results to disk, --benchmark-sort=mean to sort the results by the mean execution time, and --benchmark-min-rounds=5 to run each test case at least five times for consistency. Finally, we use the -v option to output verbose information about the test results.

We store the data, code and results of our tests in code repository, see section CODE AVAILABILITY. Test files are stored in a dedicated 'test' directory, raw data files are stored in a 'testdata' directory under the test directory, and the test output is recorded in the 'benchmark' directory.

S7. Character set and filtering examples of the API

Since the main goal of API is to be user-friendly, i.e., the syntax must be easy to read and understand, it is necessary to choose the right characters and set their appropriate meanings. The following table

shows the character set settings and their corresponding meanings. Table S1. shows the character set and their corresponding meanings.

Table S1. List of the symbols used to represent logic operations in the Search API syntax

Symbol	Description	Example
" "	Accurate Matching	"SiO2"
-	Exclusion	-Li
|	Or	Si | Li
+	And	+O
*	Placeholder	*STR*
,	Hyphen	+O,-Li
>	Greater than	nelements>3
<	Less than	nelements<3
=	Equality	nelements=3

The Query-API can filter a series of search results that meet different criteria, including exact match, fuzzy match, and multiple conditions match for both numerical and string data. Examples are shown in Table S2. For the filtering of numerical values, we provide six basic operations based on general comparison logic and they are

<"equal"|"not equal"|"less"|"greater"|"no greater"|"no less">

A query string using key-value pairs combined with symbols can restrict the results returned to values associated with specific keywords to a specific range, while the syntax is so intuitive that it does not need to be explained in detail.

For string matching, we mainly refer to the usage in the regular expression, with the double quotes for exact matching and the placeholder "*" for fuzzy matching. The basic matching logic is

<"equal"|"not equal"|"contain"|"start with"|"end with">

The API also supports multi-conditional queries joined by "AND" and "OR" logical terms, with query conditions nested by parentheses.

Table S2. Examples of filtering

Filter	Query Syntax	Result Description
Number filtering	Key=1.234	key equals 1.234
	Key!=1.234	key not equal 1.234
	Key>1.234	key greater than 1.234
	Key<1.234	key lesser than 1.234
	Key>=1.234	key greater than or equal to 1.234
	Key<=1.234	key is less than or equal to 1.234

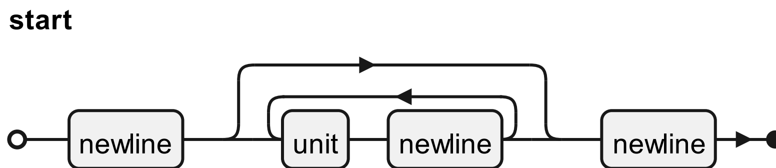
String filtering	Key=" value" Key=*value Key=value* Key=*value* Key!=value	key is exactly the "value" key ends with "value" key starts with "value" key contains the substring "value" key is not "value"
Multiple conditions filtering	key_a=value_a AND key_b=value_b key_a=value_a OR key_b=value_b (key_a=value_a AND key_b=value_b) OR (key_c=value_c OR key_d=value_d)	The results that match for key_a=value_a and key_b=value_b The results that match for key_a=value_a or key_b=value_b The results that match for a condition that key_a=value_a and key_b=value_b or condition that key_c=value_c or key_d=value_d

S8. The EBNF grammar of the template language syntax and its railroad diagram

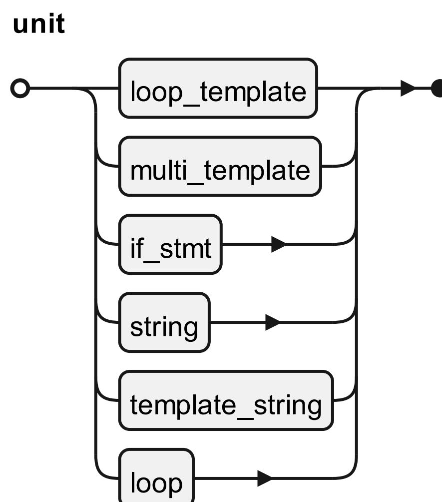
The following EBNF rules are based on the ISO/IEC 14977 standard.

(* BEGIN EBNF *)

start = newline, {unit, newline}, newline ;



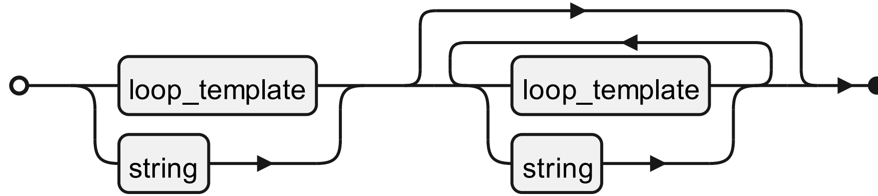
unit = loop_template | multi_template | if_stmt | string | template_string | loop ;

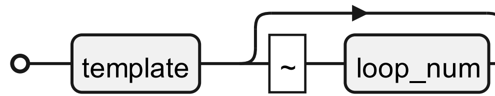


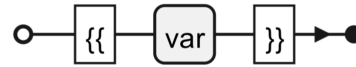
multi_template = (loop_template | string), { loop_template | string } ;

loop_template = template, ["~", loop_num] ;

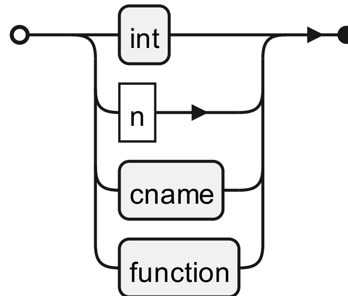
template = "{{", var, "}}" ;



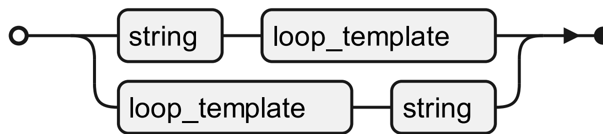




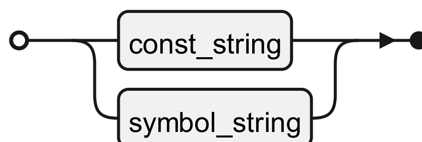
loop_num = int | "n" | cname | function ;

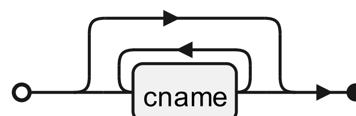


template_string = (string, loop_template) | (loop_template, string) ;

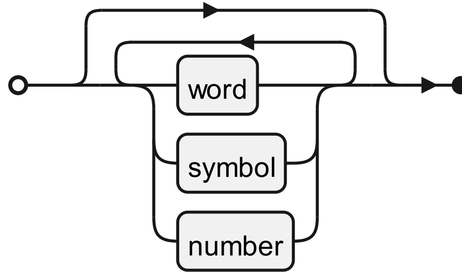


string = const_string | symbol_string ;
const_string = { cname } ;

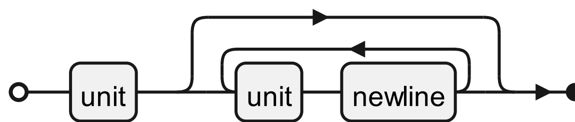




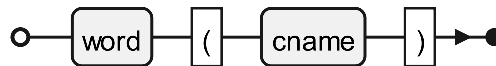
symbol_string = {word | symbol | number} ;



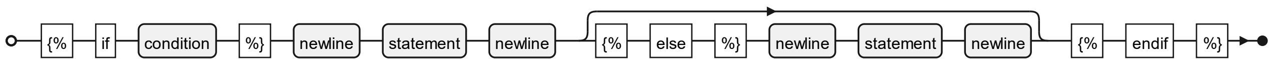
statement = unit, {unit, newline} ;



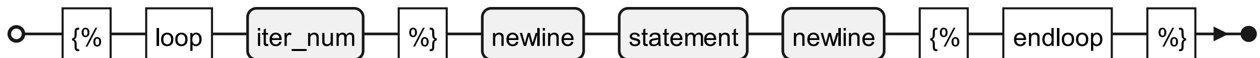
function = word, "(", cname, ")" ;



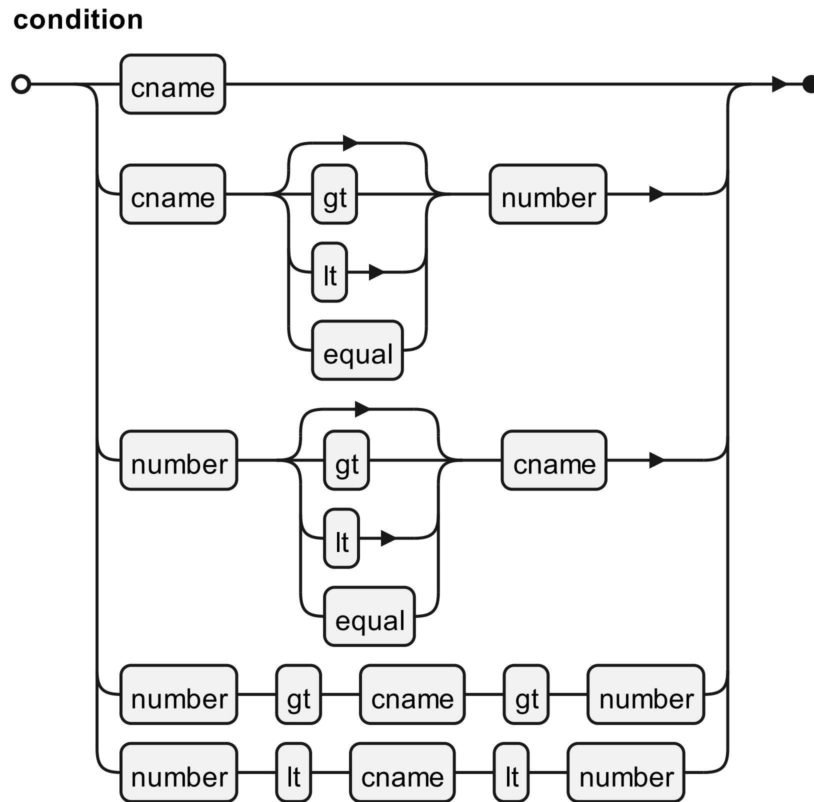
if_stmt = "{%", "if", condition, "%}", newline, statement, newline,
["{%", "else", "%}", newline, statement, newline], "{%", "endif", "%}" ;



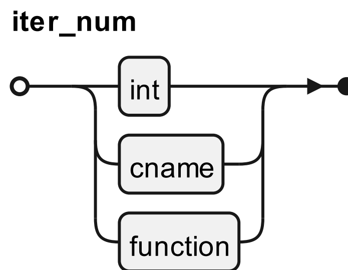
loop = "{%", "loop", iter_num, "%}", newline, statement, newline, "{%", "endloop", "%}" ;



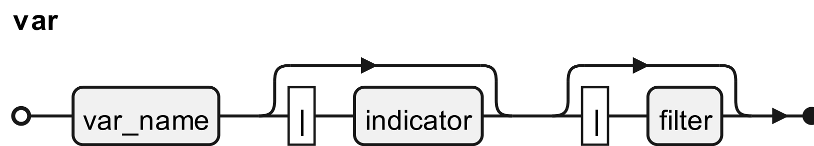
condition = cname | cname, [gt | lt | equal], number | number, [gt | lt | equal], cname
| number, gt, cname, gt, number | number, lt, cname, lt, number ;



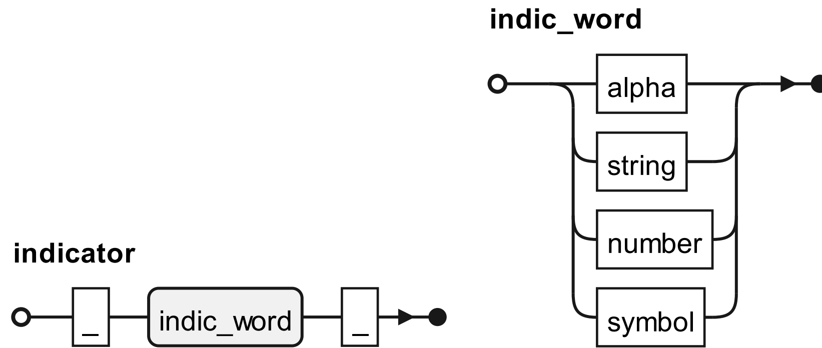
iter_num = int | cname | function ;



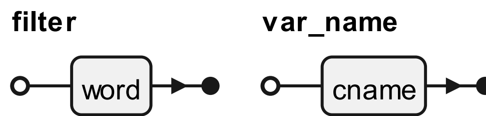
var = var_name, ["|", indicator], ["|", filter] ;



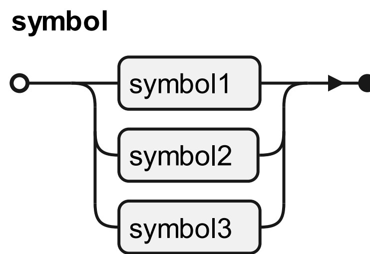
indicator = "_", indic_word, "_" ;
indic_word = "alpha" | "string" | "number" | "symbol" ;



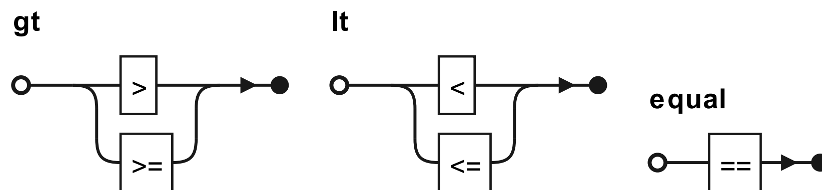
filter = word ;
var_name = cname ;



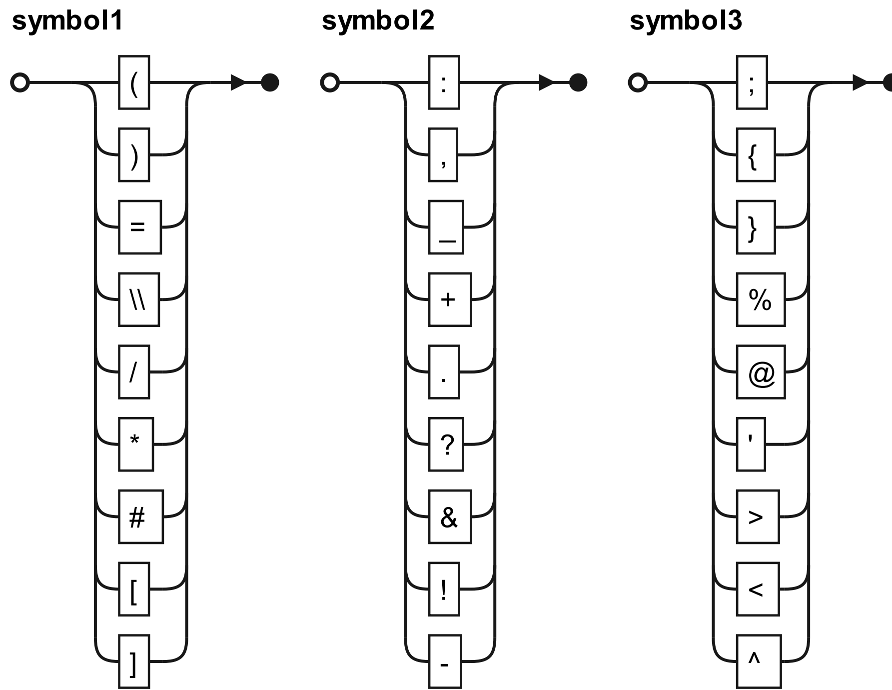
symbol = symbol1 | symbol2 | symbol3 ;



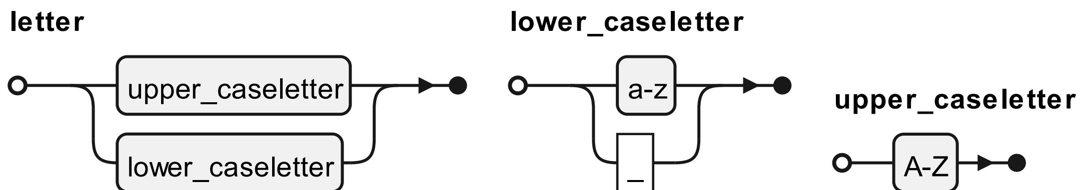
gt = ">" | ">=" ;
lt = "<" | "<=" ;
equal = "==" ;



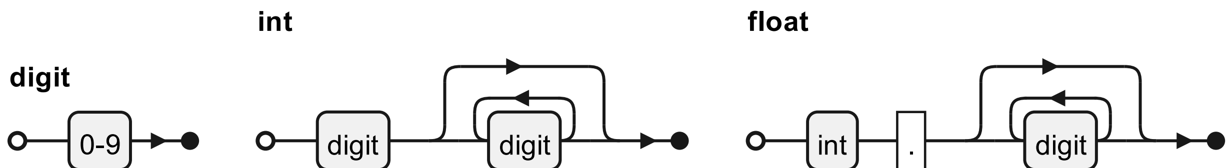
symbol1 = "(" | ")" | "=" | "\\" | "/" |"*" | "#" | "[" | "]" ;
symbol2 = ":" | "," | "_" | "+" | "." |"?" | "&"| "!" | "-" ;
symbol3 = ";" | "{" | "}" | "%" | "@" | """ | ">" | "<" | "^" ;



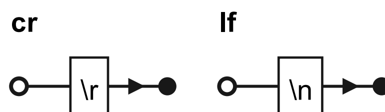
letter = upper_caseletter | lower_caseletter ;
upper_caseletter = A-Z ;
lower_caseletter = a-z | "_";



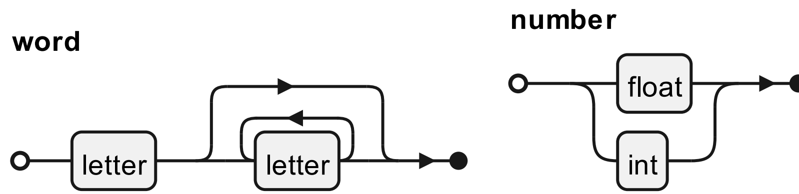
digit = 0-9 ;
int = digit, {digit};
float = int, ".", {digit};



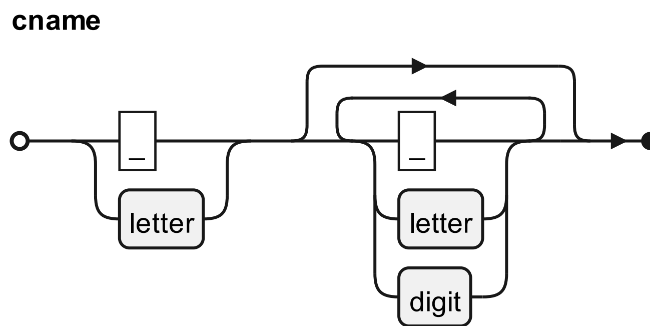
cr = "\r";
lf = "\n";



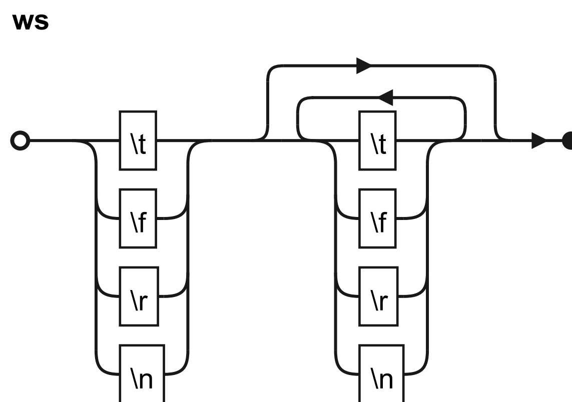
word = letter, {letter};
number = float | int;



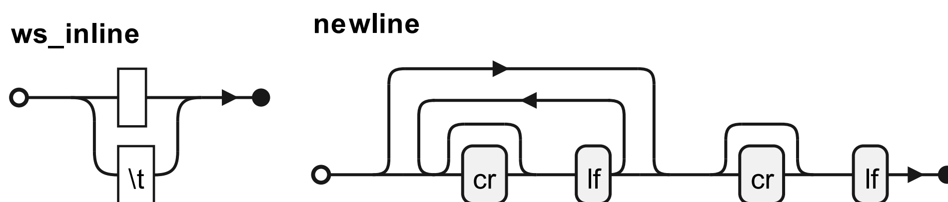
cname = ("_" | letter), {"_" | letter | digit};



ws = ("\t" | "\f" | "\r" | "\n"), {"\t" | "\f" | "\r" | "\n"};



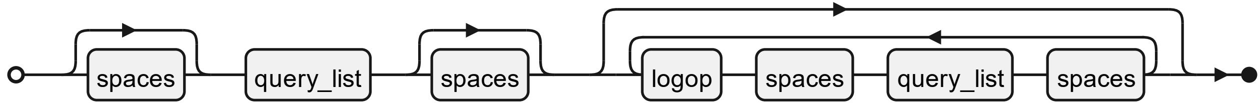
ws_inline = (" "|"\t");
newline = ([cr], lf), {[cr], lf};



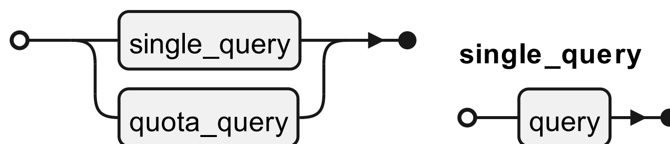
(* END EBNF *)

S9. The EBNF grammar of the query syntax and its railroad diagram

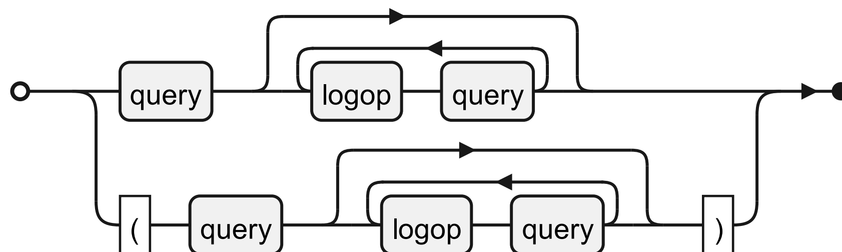
The following EBNF rules are based on the ISO/IEC 14977 standard.
(* BEGIN EBNF *)
start = [spaces], query_list, [spaces], {logop , spaces , query_list , spaces} ;



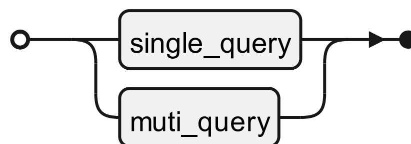
query_list = single_query | quota_query ;
single_query = query ;



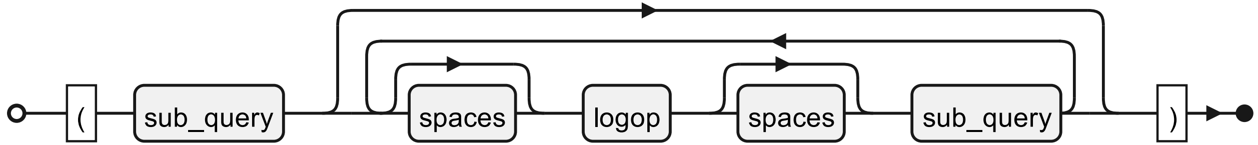
muti_query = query, {logop, query} | "(", query, {logop, query}, ")" ;



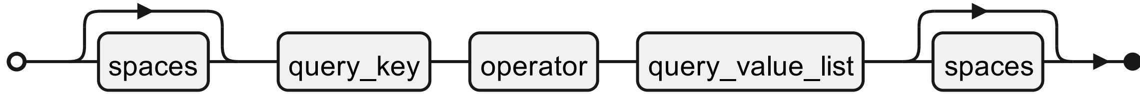
sub_query = single_query | muti_query ;



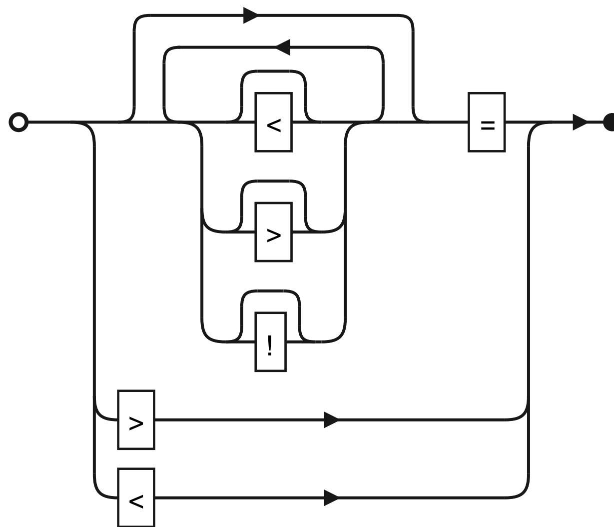
quota_query = "(", sub_query, { [spaces], logop,[spaces],sub_query }, ")" ;



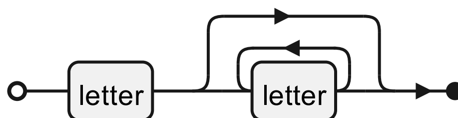
query = [spaces], query_key , operator , query_value_list, [spaces] ;



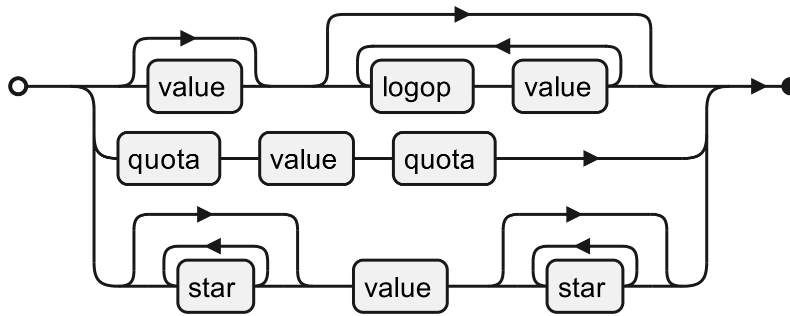
operator = {["<"] | [">"] | ["!"]}, "=" | ">" | "<" ;



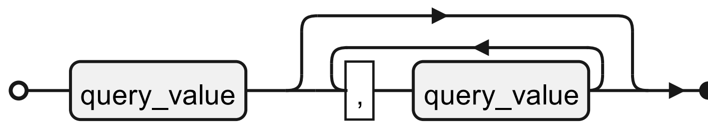
query_key = letter, {letter} ;



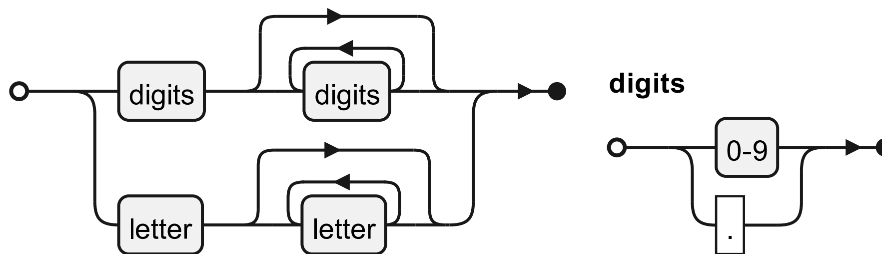
query_value = [value], {logop, value} | quota, value, quota | {star}, value, {star} ;



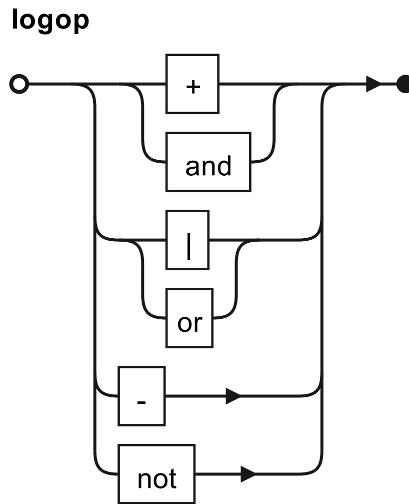
query_value_list = query_value, {",", query_value} ;



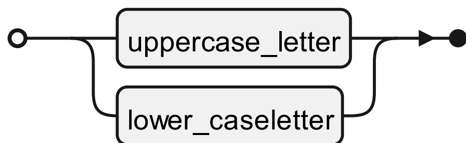
value = digits, {digits} | letter, {letter } ;
digits = 0-9 | "." ;

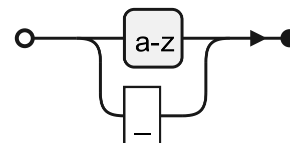


logop = (" + " | " and ") | (" | " | "or") | (" - " | " not ") ;

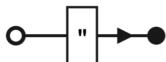


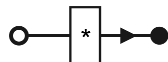
letter = uppercase_letter| lower_caseletter ;
uppercase_letter = A-Z;
lower_caseletter = a-z | "_";

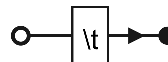


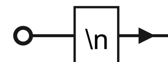


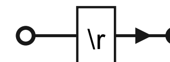
quota = "" ;
star = "*" ;
tab = "\t" ;
nl = "\n" ;
cr = "\r" ;
vt = "\v" ;
ff = "\ff" ;

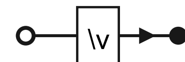


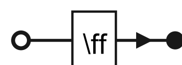




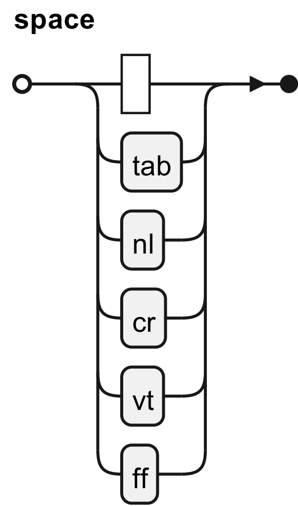


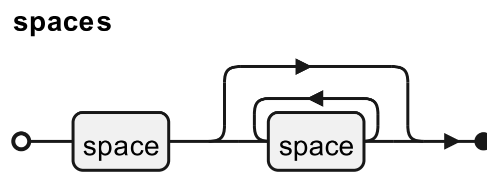






space = " " | tab | nl | cr | vt | ff ;
spaces = space, {space} ;





(* END EBNF *)