DOMAIN-SPECIFIC LANGUAGE FOR ANALYZING MEDICAL RESULTS

**Mihai VIERU*, Vlad POLISCIUC, Daniela GLIGA,
Ecaterina GREBENNICOVA, Anastasia ZAGORODNIUC**

Department of Software Engineering and Automation, Group FAF-223, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Republic of Moldova

*Corresponding Author: Mihai Vieru, mihai.vieru@isa.utm.md

Coordinator: **Gabriel ZAHARIA**, Technical University of Moldova

Abstract. *This article delves into the potential benefits that a Domain-Specific Language (DSL) could bring to the medical field. It highlights how a DSL could enhance the analysis of medical results by offering greater precision, quicker analysis times, and lower chances of mistakes. Furthermore, it elaborates on the potential for DSLs to integrate seamlessly with existing medical software systems, enhancing interoperability and data sharing across platforms. Additionally, it points out the advantages of using a DSL for data management tasks such as gathering, updating, and maintaining records about patients' illnesses, enabling healthcare professionals to access and analyze critical information with ease. The utilization of DSLs could also foster a more personalized approach to patient care, allowing for treatments and medical advice to be tailored more accurately to individual patient profiles. Lastly, the article speculates on the future role of DSL in medicine, emphasizing its ongoing contribution to the interpretation and analysis of medical data, and predicts a significant shift in how medical professionals interact with technology, ultimately leading to more efficient and effective patient care.*

Key-words: *healthcare, data management, data interoperability, software systems integration.*

Introduction

Evaluating medical outcomes is crucial in healthcare, offering critical information for diagnosing, treating, and preventing various health conditions [1]. Yet, sifting through vast amounts of medical data from diverse sources poses significant challenges, particularly for medical practitioners needing more in-depth technical expertise.

Domain-specific languages (DSLs) emerge as a viable answer to these issues, introducing a programming language designed specifically for the medical domain. This article introduces a DSL crafted for medical result evaluation. It starts by examining the domain analysis, addressing the main obstacles in analyzing medical data. The article then outlines the DSL, focusing on its principal characteristics such as its capacity to streamline intricate data evaluation processes and enhance the precision of health diagnoses.

The DSL's syntax is crafted to be intuitive and straightforward for users. Additionally, the article explores the DSL's potential effects on the healthcare industry, such as better patient health outcomes and decreased expenses. In summary, the article offers an in-depth look at a specialized language for analyzing medical data, shedding light on its development, application, and the advantages it brings.

Domain Analysis

Multi-center healthcare data sharing faces significant challenges due to privacy regulations and the heterogeneity of data, which are critical hurdles in advancing medical research across various fields such as neuroscience, genetics, drug discovery, and disease diagnosis and prognosis. The backbone of successful machine learning algorithms, particularly in these areas, relies on having access to sufficiently large datasets with the necessary annotations [2]. In order to reach the impressive performance levels of deep learning - which is known for its intricate models with millions or even billions of parameters - it requires even more data samples than typical machine learning algorithms. Yet, the medical datasets that are frequently made accessible for machine learning research are usually too small and don't have the thorough annotations needed for detailed analysis.

Grammar

Grammar in the context of computer science and programming languages, is a formal set of rules for specifying the syntax of a language. It defines how words, symbols, and other elements (tokens) can be combined to produce valid sentences or expressions within that language [3]. We note grammar as:

G = (V_N, V_T, P, S), where

- V_N - abstract symbols that represent placeholders for collections of phrases or sentence structures, expandable into sequences of terminal and/or non-terminal symbols according to production rules.
- V_T - terminal symbols are the language's basic, indivisible elements like keywords and operators, not further reducible by grammar rules.
- P - rules that define how non-terminal symbols can be replaced with sequences of terminal and/or other non-terminal symbols, with each rule outlining a construction method for language phrases, featuring a single non-terminal symbol on the left and a sequence of symbols on the right.
- S - special non-terminal symbol from which all valid sentences in the language can be derived.

Grammar consists of a series of production rules that describe the structure of valid sentences in the language. These rules are used by parsers and compilers to understand and process the language [4].

In our DSL, V_N and V_T are presented as next:

V_N = {<Program>, <StatementList>, <Statement>, <VariableDeclaration>,
<VariableDeclarator>, <Expression>, <BinaryExpression>,
<MultiplicativeExpression>, <PrimaryExpression>, <Literal>, <StringLiteral>,
<ParanthesizedExpression>, <NumericLiteral>}

V_T = {"NUMBER", "STRING"}

```
<Program> ::= "MAIN_STRUCT" "{" <StatementList> "}"
<StatementList> ::= <Statement> | <Statement> <StatementList> | ε
<Statement> ::= <VariableDeclaration>
<VariableDeclaration> ::= <VariableDeclarator> ","
<VariableDeclarator> ::= "DECLARATOR" "DECLARATOR_OPERATOR"
    <Expression>
<Expression> ::= <BinaryExpression>
<BinaryExpression> ::= <MultiplicativeExpression> | <MultiplicativeExpression>
    "ADDITIVE_OPERATOR" <MultiplicativeExpression>
<MultiplicativeExpression> ::= <PrimaryExpression> | <PrimaryExpression>
    "MULTIPLICATIVE_OPERATOR" <PrimaryExpression>
<PrimaryExpression> ::= <ParanthesizedExpression> | <Literal>
<ParanthesizedExpression> ::= "(" <Expression> ")"
<Literal> ::= <StringLiteral> | <NumericLiteral>
<NumericLiteral> ::= "NUMBER"
<StringLiteral> ::= "STRING"
```

These non-terminal and terminal symbols help us determine how patients should provide their input data based on their illnesses.

Below we provide an example of patient data that could be introduced into the system. The patient completes all the needed info about its analyses for specific illnesses that he has.

```
Obesity {
      pregnancies: (1+2),
      diagnosis 2,
      treatment 2,
      glucose: 2,
      bloodPressure: 2,
      skinThickness: 2,
      insulin: 2,
      bmi: 2,
      diabetesPedigreeFunction 2,
      age: (2+3)*2,
}
```

Lexer

A lexer, short for lexical analyzer or tokenizer, is a crucial component in the process of compiling or interpreting code [5]. Its main job is to read the input source code and break it down into meaningful pieces known as tokens. Tokens can represent keywords, identifiers, literals (like numbers and strings), and operators that the programming language syntax defines. Overall, to obtain the lexer we desire and tokenize the input we have to follow simple rules which are described easily in the code:

```
Tokens: list[list[str]] = [
 [r"\A\s+", "WHITESPACE"],
 [r"\A," , ","],
 [r"\A[(]", "("],
 [r"\A[)]", ")"],
 [r"\A[{]", "{"],
 [r"\A[}]", "}"],
 [r"\A\bCreate Template\b", "INIT_STRUCT"],
 [r"\A\bname\b", "STRUCT_NAME"],
 [r"\A\bparams\b", "STRUCT_PARAMS"],
 [r"\A\btarget\b", "STRUCT_TARGET"],
 [r"\A\bdata\b", "STRUCT_DATA"],
 [r"\A\bdeclare\b", "NEW_STRUCT"],
 [r'\A\w+\.\w+', "METHOD_CALL"],
 [r'\A"""([\s\S]*?)"""', "BCOMMENT"],
 [r"\A\#.*$", "COMMENT"],
 [r'\A:(?!:)', "DECLARATOR_OPERATOR"],
 [r'\A=(?!=)', "DECLARATOR_OPERATOR"],
 [r"\A[^\s\W\d]+", "VARIABLE"],
 [r'\A[+\-]', "ADDITIVE_OPERATOR"],
 [r'\A[*\/]', "MULTIPLICATIVE_OPERATOR"],
 [r"\A\d+", "NUMBER"],
 [r'\A"[^"]*"', "STRING"],
 [r"\A'[^']*'", "STRING"]
```

The various components of the input text, including whitespace, punctuation, comments, operators, numbers, and strings, are identified by these patterns. To ensure that the entire input is either tokenized in accordance with the established patterns or stopped if an unmatchable sequence is discovered, the lexer stops upon encountering unrecognized sequences or the end of the input.

Parsing

Parsing involves analyzing text to understand its meaning based on a set of grammatical rules. In the context provided, the goal is to interpret medical test results by identifying the types of tests, their results, and reference ranges according to a predefined grammar. A common approach for parsing context-free grammars like this one is to use a technique called recursive descent parsing [6]. Here is how it could be done for this grammar:

Initially, the process starts by evaluating the <Program> rule, which then progresses into examining a <StatementList>. It determines whether the upcoming token is a <Statement>, a combination of <Statement> <StatementList>, or none, leading to a corresponding function call or an error. For a <Statement>, it further invokes a parsing function dedicated to <VariableDeclaration>, which then calls for <VariableDeclarator>. The <VariableDeclarator> checks for specific operators before moving on to an <Expression> parsing function. Expressions are broken down into binary or multiplicative forms, depending on the operators and tokens encountered, and further into primary expressions, which include literals and parenthesized expressions. The algorithm emphasizes a hierarchical and recursive parsing strategy, ensuring that every component of the program is syntactically valid by matching specific patterns and data types, such as numbers and strings, thereby streamlining the parsing process into an efficient, step-by-step analysis of the program's structure.

Recursive descent parsing works by recursively calling parsing functions for each rule in the grammar until the entire input is parsed [7]. If the input is valid according to the grammar, a parse tree is constructed that represents the structure of the input. If the input is not valid, an error is returned. Using these parsing rules and grammar we have to define the patient's illness we want to check and provide the tokens that represent test results that we have listed already in the code.

Conclusions

In conclusion, the introduction of Domain-Specific Languages (DSLs) into the medical field marks a fundamental change in how medical data is analyzed and interpreted. By tailoring programming language features specifically for healthcare applications, DSLs have streamlined complex data analysis processes, significantly enhancing the accuracy of diagnoses and patient care. This specialization has not only reduced the time and effort involved in managing and analyzing medical data but has also reduced the risk of errors, leading to better patient outcomes and more efficient healthcare delivery. While DSLs have significantly helped with data diversity and system compatibility in healthcare, their integration faces ongoing adoption and improvement challenges, yet their role is set to grow, further transforming healthcare technology and its ability to serve patients better.

References

[1] L. Pantaleon, "Why measuring outcomes is important in health care," *PMC*. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6430924/. [Accessed: Mar. 23, 2024].

[2] Q. Chang, "Mining multi-center heterogeneous medical data with distributed synthetic learning," *PMC*. [Online]. Available: Mining multi-center heterogeneous medical data with distributed synthetic learning - PMC (nih.gov). [Accessed: Mar. 23, 2024].

[3] "Formal Grammar," *StudySmarter*. [Online]. Available: https://www.studysmarter.co.uk/explanations/computer-science/theory-of-computation/formal-grammar/. [Accessed: Mar. 18, 2024].

[4] F. Pereira, "The semantics of grammar formalism seen as computer languages," *Harvard University*. [Online]. Available: http://nrs.harvard.edu/urn-3:HUL.InstRepos:2309658. [Accessed: Mar. 18, 2024].

[5] "What Is Lexical Analysis?," *Coursera*. [Online]. Available: https://www.coursera.org/articles/lexical-analysis. [Accessed: Apr. 2, 2024].

[6] K. Sikkel and A. Nijholt, "Parsing of context-free languages," *SpringerLink*. [Online]. Available: Parsing of Context-Free Languages | SpringerLink. [Accessed: Apr. 2, 2024].

[7] "Parsing," *University of Rochester*. [Online]. Available: https://www.cs.rochester.edu/u/nelson/courses/csc_173/grammars/parsing.html. [Accessed: Apr. 4, 2024].