THE DEVELOPMENT OF A DOMAIN SPECIFIC LANGUAGE FOR EMAIL SORTING

Andy-Constantin CIOBANU[1], Cristian Cătălin MIRON[1], Dumitru STRELEȚ[1], Radu-Vasile ARAMĂ[1], Andreea BÎRSAN[1*]

[1] *Technical University of Moldova, Faculty of Computers, Informatics and Microelectronics, Department of Software Engineering and Automation, Group FAF-193, Chisinau, Republic of Moldova*

*Corresponding author: Bîrsan Andreea, birsan.andreea@isa.utm.md

Abstract: *The goal of domain-specific languages (DSLs) is to increase the productivity of software engineers by abstracting from low-level boilerplate code. Introduction of DSLs in the software development process requires a smooth workflow for the production of DSLs themselves. This paper discusses the implementation of a user-friendly DSL intended to work as an email querying method. On further research about this topic, we tackle the problem that the domain specific languages solve, the type of data that is going to be processed, the computational model and its grammar design.*

Key words: *DSL, BNF, parse tree, grammar, email.*

Introduction

Abstraction is the key to progress in software engineering. By encapsulating knowledge about low level operations in higher-level abstractions, software developers can think in terms of the higher-level concepts and save the effort of composing the lower-level operations. Conventional abstraction mechanisms of general-purpose programming languages such as methods and classes, are no longer sufficient for creating new abstraction layers [2]. The design and use of appropriate DSLs are a key part of domain engineering, by using a language suitable to the domain at hand – this may consist of using an existing DSL or GPL, or developing a new DSL. Creating a domain-specific language, rather than reusing an existing language, can be worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than an existing language would allow and the type of problem in question reappears sufficiently often.

Domain analysis

The project centers around email management and sorting, as it can have a lot of versatility in the way that it can be of use. This domain encompasses a wide range of utility varying from personal to more business-oriented purposes. Depending on the service that is being used, certain features might not be consistent over the entire scope of the email services, particularly regarding the sorting options of one's emails. Thus, through implementing a DSL that is going to serve a similar purpose for emails as what the structured query language does to databases, it is a possibility to make it easy and efficient to manage the messages that are being received through these particular platforms by using various focal classifications.

Language overview

The basic computation that the proposed DSL performs consists of querying large data, emails in this case, abstracting from the general-purpose language that involves knowledge of programming principles and, at the same time, methods of data parsing. It will filter and sort data based on a given configuration file, given an input or a source database. The DSL will hide the logic of querying data giving the user the possibility to ask for a collection of data in a form of a language, rather than developing a whole application. As far as the computational model, the language uses the declarative model which is specific to the domain of querying or gaining data.

The main data structure is the email, as an abstract data structure. By being a representation of the declarative model of programming, the DSL does not include an explicit data declaration that will affect the computation of the main program.

The DSL supports only sequential Control Structure due to the fact that the language is just a set of predefined rules with a limited type of input for each rule. The rules can be defined as a set of flags in execution of a script. Every email is stored in the same way and has approximately the same structured data columns, like body, subject, folder, time when was received and when was read etc. Due to this situation, it would be a bad choice to write any selectional and/or repetitional control structures (especially in the first versions of the language), because it would considerably raise the complexity and difficulty of the language that is pointed only to find and extract emails by a set of rules. The DSL control structure is very similar to BASH scripts like grep or find, where user input is limited on predefined set of rules. As it can be seen in Fig.1, rules are not linked to each other, and each rule is optional (if no rule at all (email {}) then blank output.

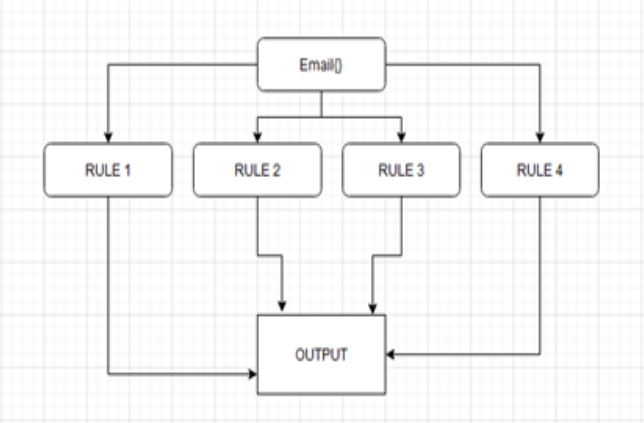

Figure 1. Control flow chart

A potential program written using the proposed DSL will require as an input a collection of records, most probably in JSON format or XML, which are the emails on which the operations will be performed. The records will be received from a database and they must have specific fields and properties to be processed by the program. As an output, the program will return the filtered or processed records, depending on the client's specification.

Grammar design

When describing languages, Backus-Naur form (BNF) is a formal notation for encoding grammars intended for human consumption [3]. Many programming languages, protocols or formats have a BNF description in their specification, Tab. 1.

BNF Representation of the EQL grammar:

Table 1

Meta Notations

<foo>	means foo is a nonterminal.
foo	(in **bold** font) means that **foo** is a terminal i.e., a token or a part of a token.
[x]	means zero or one occurrence of x, i.e., x is optional; note that brackets in quotes ′[′′]′ are terminals.
x∗	means zero or more occurrences of x.
x+,	a comma-separated list of one or more x's.
{ }	large braces are used for grouping; note that braces in quotes ′ {′′ }′ are terminals.
|	separates alternatives.
V_T	terminal symbols.
V_N	non-terminal symbols.
P	productions - rules of the grammar.

V_T = {email, {, }, time:, subject:, sortby:, folder:, to:, from:, cc:, forwarded:, read:, body:, attachments:, TIME, NAME, YES, NO, .., „, ASC, DESC, y, d, -, @., *}

V_N = {<filter>, <destination>, <time>, <subject>, <property>, <content>, <sorting>, <folder>, <to>, <from>, <cc>, <datevalue>, <subjectvalue>, <forwarded>, <read>, <body>, <attachments>, <sortingvalue>, <foldervalue>, <destinationvalue>, <attachementsvalue>, <bodyvalue>, <boolvalue>, <date>, <star>, <day>, <year>, <wordlist>, <word> , <string>, <sortvalue>, <email>, <fextension>, <file>, <digit>, <interval>, <letter>, <date_digit>, <date_day>, <date_month>, <date_year>}

P={

<query> → **email {**<filter>∗ | <star>**}**

<filter> → <destination> | <time> | <subject> | <property> | <content> | <sorting> | <folder>

<destination> → <to> | <from> | <cc>

<time> → **time:** <datevalue>

<subject> → **subject:** <subjectvalue>

<property> → <forwarded>| <read>

<content> → <body> | <attachments>

<sorting> → **sortby:** <sortingvalue>

<folder> → **folder:** <foldervalue>

<to> → **to:** <destinationvalue>

<from> → **from:** <destinationvalue>

<cc> → **cc:** <destinationvalue>

<forwarded> → **forwarded:** <boolvalue>

<read> → **read:** <boolvalue>

<body> → **body:** <bodyvalue>

<attachments> → **attachments:** <attachementsvalue>

<datevalue> → <date><star><date> | <date><star> | <star><date> | <star><day> | <day><star> | <star><year> | <year><star>

<subjectvalue> → <wordlist> | <word> | <string>

<sortingvalue> → <parameter> <sortvalue>

<paramater> → **TIME** | **NAME**

<foldervalue> → <string>

<destinationvalue> → <wordlist> | <word> | <string> | <email> | <star>

<bodyvalue> → <wordlist> | <word> | <string>

<attachementsvalue>→ <boolvalue>| <word> | <fextension> | <file> | <digit> | <interval>

<wordlist> → <word>+

<year> → <date_digit>+ **y**

<day> → <date_digit>+ **d**

<sortvalue> → **ASC** | **DESC**

<boolvalue> → **YES** | **NO**

<date> → <date_day> **-** <date_month> **-** <date_year>

<email> → <word> **@.** <word> **.** <word>

<word> → <letter>+

<fextension> → <star> **.** <word>

<file> → <word> **.** <word>

<interval> → <digit> **..** <digit>

<digit> → 0 | 1 | 2 | ... | 9

<digit_date> → 1 | 2 | ... | 9

<string> → **„** <letter>* **”**

<letter> → a | b | ... | z | A | B | ... | Z |

<date_day> → 1 | 2 | ... | 31

<date_month> → 1 | 2 | ... | 12

<date_year> → 1950 | 1951 | ... | 2100<star> → *****}

Example of program:

email {
to: arama.radu@gmail.com
time: 23-09-2021
subject: laboratory
read: yes
folder: "university"
}

For the given piece of code we obtained the next Parse Tree, Fig. 2.

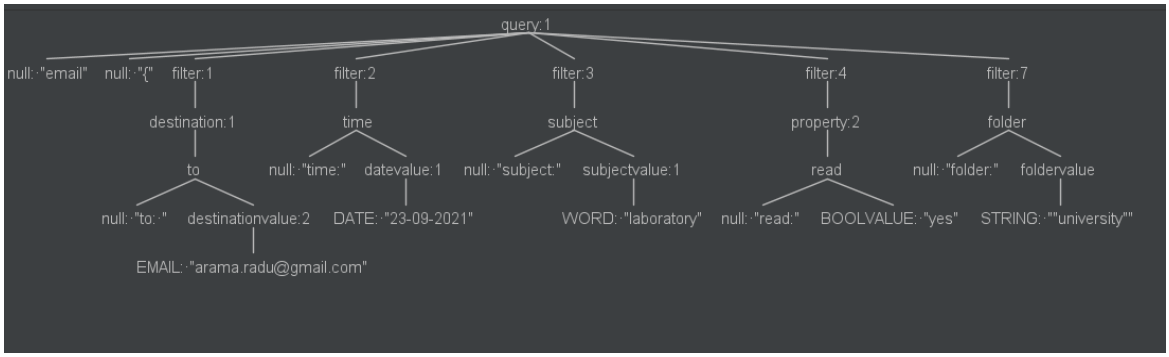

Figure 2. Parser tree

Conclusion and future works

Domain-specific languages (DSLs) are languages tailored to a specific application domain. They have many potential advantages in terms of software engineering, ranging from increased productivity to the application of formal methods. This paper introduces a new DSL concept, which is centered around email querying and its goal is to represent an easily sizeable solution for the end-user. After implementing the language by deconstructing every important element of a DSL, it is essential to consider in the future how intuitive it might be for the end-user, along with making it more specific by adding new variables depending on what works best for the domain that was chosen.

References

1. LÄMMEL, Ralf & VISSER, Joost & SARAIVA, João. (2008). Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007.
2. Tomaž KOSAR, Sudev BOHRA, Marjan MERNIK, Domain-Specific Languages: A Systematic Mapping Study, Information and Software Technology, Volume 71, 2016.
3. MERNIK, Marjan & HEERING, Jan & SLOANE, Anthony. When and How to Develop Domain-Specific Languages. ACM Comput. Surv.. 37. 316-. 10.1145/1118890.1118892, 2005.