DESIGN OF DOMAIN SPECIFIC LANGUAGE FOR ASTROLOGICAL CHARTS GENERATION - ALAKIRQL

Alexandru ANDRIEȘ[1], Georgeana GLOBA[1],
Arteom KALAMAGHIN[1*], Dionisie SPATARU[1]

[1]*Departamentul Ingineria Software și Automatică; Facultatea Calculatoare, Informatică şi Microelectronică; Universitatea Tehnică a Moldovei; o. Chișinău; Republica Moldova*

*Corresponding author: Arteom Kalamaghin, arteom.kalamaghin@isa.utm.md

Scientific coordinators: Mariana CATRUC, superior lecturer, UTM; **Irina COJUHARI**, assoc. prof. , UTM

Abstract. *This article represents an analysis of the definition of a* Domain-Specific Language *for work in the field of astrology - AlakirQL. The article's goal is to describe the functionality, specifications, semantic and grammatical features of the AlakirQL, as well as the lexer implementation progress, by analyzing both technical and non-technical subjects.*

Keywords: *Domain-Specific Language, Astrology, formal language, grammar, semantics.*

Introduction

DSLs (Domain-Specific Languages) are becoming increasingly popular tools for solving problems in areas where advanced scripting is not required. Programming provides a wide range of possibilities to create features, solve everyday problems, or routines. Since DSLs are widespread in this day and age, the ability to create automated alternatives for otherwise labored assignments has become effortless in any area of work [1].

In some cases, constructing a domain-specific language from scratch could be a better solution than using any other existent substitutes, since it might allow for a clearer expression of both algorithmic questions and answers in cases where they are regularly encountered. In addition, the improvement of tools has made this task even easier than before.

Domain description

Astrology represents the multitude of spiritual practices that claim to have the ability to discover details about people's lives and some global events based on the positions of different extraterrestrial objects [2].

Nowadays, the most popular application is composition of birth charts. Astrologers use the birth chart to make predictions about future events and to provide guidance for personal growth and development. The composition of the Birth Chart and the provision of tabular data related to this notion will be the primary problem with which the *AlakirQL* will be concerned. Some web resources provide solutions that implement this service, however, we believe that our approach has two big advantages over this form of service delivery. First of all, local use: a web-application requires personal data and contact information to provide results; when used locally your data does not leave the device. Automation: the implementation of control and data structures for repetitive flow and similar data storing will allow users to process large amounts of information more easily, speeding up routine work. It is also worth mentioning another approach to data formatting: special output and reading modules will allow users to save processed data to files in order to be able to return to them in the future.

Grammar

What follows is a definition of the grammar in a form that will help us easily construct a parse tree for a sample piece of code – $G = (S, P, V_N, V_T)$, where: S – the start token; P – a collection of production rules; V_N – lists non-terminals; V_T – a list of terminal symbols.

In Table I are listed meta-notations, understanding of which is essential while reading the grammar specified below.

Table 1.

Meta-notation

Notation	Denotation
<foo>	for non-terminals
foo	for terminal symbols (a token or a part of a token)
[x]	indicates zero or more occurrences of x (is optional; note that brackets in quotes are terminals)
x*	indicates zero or more occurrences of x
x^+,	separated by comma list of one or more x's
|	separates alternatives

S = { <program> }; V_T = { 0..9, A..Z, a..z, WHILE, DECLARE, CONST, (datatype names), (built-in function names), (operators) , _ ,) , (, [,] , { , } , deg, min, sec, year, month, day, hour, minute, second, tables, " , ; }; V_N = { ⟨block⟩, ⟨directive⟩, ⟨var_decl⟩, ⟨declarator⟩, ⟨type⟩, ⟨expr⟩, ⟨statement⟩, ⟨location⟩, ⟨method_call⟩, ⟨method_name⟩, ⟨bin_op⟩, ⟨literal⟩, ⟨id⟩, ⟨alpha_num⟩, ⟨alpha⟩, ⟨digit⟩, ⟨arith_op⟩, ⟨rel_op⟩, ⟨eq_op⟩, ⟨int_literal⟩, ⟨decimal_literal⟩, ⟨string_literal⟩, ⟨angle_literal⟩, ⟨dt_literal⟩ } ; P =

{ ⟨program⟩	→ ⟨directive⟩*
⟨block⟩	→ '{' ⟨directive⟩* '}'
⟨directive⟩	→ ⟨var_decl⟩ | ⟨statement⟩ **;**
⟨var_decl⟩	→ ⟨declarator⟩ ⟨id⟩+, [= ⟨expr⟩+,] | [= '[' ⟨expr⟩+, ']']
⟨declarator⟩	→ **CONST** | **DECLARE**
⟨type⟩	→ **INT** | **STRING** | **ANGLE** | **DATETIME** | **CHART**
⟨expr⟩	→ ⟨location⟩ | ⟨method_call⟩ | ⟨literal⟩ | '(' ⟨expr⟩ ')' | ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩
⟨statement⟩	→ ⟨location⟩+, = ⟨expr⟩+, | ⟨method_call⟩ | **WHILE** ⟨expr⟩ ⟨block⟩
⟨location⟩	→ ⟨id⟩ | ⟨id⟩ '(' ⟨int_literal⟩ ')' | ⟨id⟩.⟨location⟩
⟨method_call⟩	→ ⟨method_name⟩ ([⟨expr⟩+,])
⟨method_name⟩	→ **BUILD_CHART** | **HTABLE_DATA** | **WRITE_TO_FILE** | **PRINT**
⟨bin_op⟩	→ ⟨arith_op⟩ | ⟨rel_op⟩ | ⟨eq_op⟩
⟨literal⟩	→ ⟨int_literal⟩ | ⟨string_literal⟩ | ⟨angle_literal⟩ | ⟨dt_literal⟩
⟨id⟩	→ ⟨alpha⟩ ⟨alpha_num⟩*
⟨alpha_num⟩	→ ⟨alpha⟩ | ⟨digit⟩
⟨alpha⟩	→ a | b | ... | z | A | B | ... | Z | _
⟨digit⟩	→ 0 | 1 | 2 | ... | 9
⟨arith_op⟩	→ + | - | ∗ | /
⟨rel_op⟩	→ < | > | <= | >=
⟨eq_op⟩	→ == | !=
⟨int_literal⟩	→ ⟨decimal_literal⟩
⟨decimal_literal⟩	→ ⟨digit⟩ ⟨digit⟩*
⟨string_literal⟩	→ " ⟨alpha_num⟩* "
⟨angle_literal⟩	→ ' < ' ⟨int_literal⟩ ', ' ⟨int_literal⟩ ', ' ⟨int_literal⟩ ' > '
⟨dt_literal⟩	→ "D⟨int_literal⟩-⟨int_literal⟩-⟨int_literal⟩|⟨int_literal⟩:⟨int_literal⟩" }

Some specifications

An *AlakirQL* program is a sequence of directives that can be divided into two groups. Variables are initiated in the global scope (the only valid one) during field declarations and can be accessed by all methods in the program. Statements deal with processing of data stored in variables and represented by literals.

The *AlakirQL* keywords (PRINT, WHILE etc.) are case-insensitive, yet they are frequently expressed in all capitals; variable names are case-sensitive. White space must be used to separate keywords and identifiers. Any lexical tokens may be separated by white space. One or more spaces, tabs, page and line breaks, and comments are all examples of white space. Sequences starting with an alphabetic character or an underscore form a token with the sequence of characters following it.

Numbers in *AlakirQL* are 64 bit signed floating point num-s. An ⟨alpha_num⟩ is any printable ASCII character, as well as some special characters for zodiac signs. *AlakirQL* does not fully support data structures, but nevertheless uses a field specification access approach for ANGLE, DATE and CHART variables. ANGLE and DATE literals are regular expressions, so they have to be typed in a proper form to be successfully parsed.

Lexing

Lexical analysis in computer science refers to the process of transforming a series of characters into a sequence of lexical tokens (strings with an assigned and thus identified meaning). Almost always, parsing is divided into two smaller tasks. A lexer creates a potentially infinite sequence of tokens for use in the later stages of the parsing process [3]. The lexing module's operation is straightforward: when it traverses a file, it looks ahead. It attempts to assign meaning to the next word by checking in the following order: character, keyword, boolean, date literal, numeric literal, string literal, identifier, and so on. Further, all tokens are added to the list. The complexity of this process is LL(2). Due to space constraints, it is impossible to list all of the details; however, you can fully evaluate the progress by visiting our GitHub repository: https://github.com/prenaissance/alakirql.

Parse tree

A parse tree is a rooted tree that represents the syntactic structure of a string using some context-free grammar. The term parse tree is primarily used in computational linguistics. For the following simple valid code snippet a parse tree was generated: *DECLARE time = "D2001-01-01|00:00"; DECLARE data = HTABLE_DATA(time, "south", 11, "0N"); PRINT(data);*

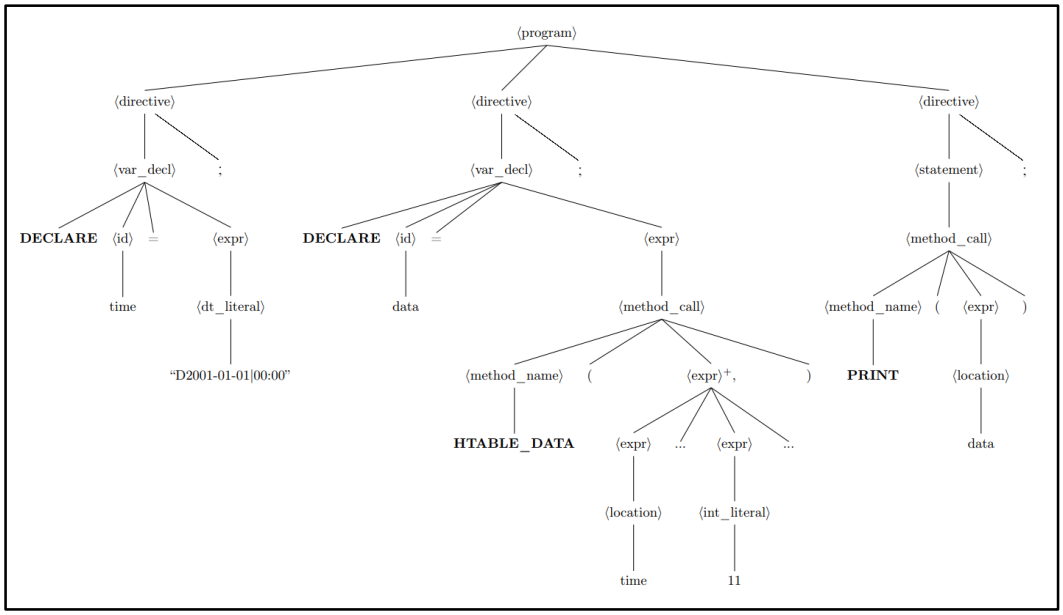

Figure 1. Parse Tree

Conclusion

Careful work at the stage of defining specifications and grammar allows to create a solid base for further development and expansion of the project. Stick to the REPL model and syntax simplification vector by introducing dynamic typing when declaring variables, for example; aim to win over the mass user. Hopefully, the product's final version will meet all expectations and become a useful tool in the field of astrology.

Sources

1. VOELTER, M. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform, 2013.
2. *Oxford Dictionary of English*. Oxford University Press.
3. DOMINUS, M.J. , *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann, 2005.