DOMAIN SPECIFIC LANGUAGE FOR ASTROLOGY

Mirela VEREBCEANU, Ion DODON, Nicu LAPTEDULCE, Nichita RAILEAN

Technical University of Moldova

***Abstract**: In this article describes a Domain Specific Language for astrology. It is named Astro and uses .astro extension. The Astro domain specific language has the purpose to gather astrology data from an API offered by astrologyapi.com and show it to the user. The grammar of this domain specific language is simple so that it cannot confuse the user with many different functions and tricks. It is focused on defining person info that is handled through variables and showing info for those persons by calling specific functions. These specific functions are actually types of requests to the API (ex: wster_horoscope, lunar_metrics, general_sign_reports). The API offers almost any information related to astrology and it is categorized in Calculations, Life Reports, and Compatibility Reports. The ANTLR was used in order to define the grammar and to create the lexer and parser. Antlr is also a domain specific language with the purpose to create other DSLs. In order to invoke ANTLR and to generate lexer and parser as target language is used Java. Java will be the engine of the DSL. It uses an SDK offered by astrologyAPI.com.*
***Keywords**: DSL, astrology, API, ANTLR, parse tree, lexer.*

Introduction

The purpose of the *Astro DSL* (Domain Specific Language) is to have a more flexible environment when working with astrology data [1]. There are many desktop applications and websites that offer graphical ways to query astrological information and to present to the user. But, no one of them offers a good way to show worksheets, to save them in files so that it will be easier to edit. With the continuous development of the *Astro DSL*, it is possible to implement functionalities such as comparing characteristics of different persons since the DSL focuses on creating *person* entities and showing their uniqueness. Another real functionality of the DSL is to query data based on multiple person data. Also would be possible to create data structures composed of person entities such as arrays.

All *Astro* keywords are lowercase and case-sensitive. The language allows comments that should start with // and are terminated by the end of the line. White spaces may appear between any *lexical token*. White space is defined as one or more spaces or tabs. Keywords and variable names must be separated by whitespaces.

1. Reference Grammar

In Table 1 are represented meta-notation used for specifying the grammar.

Table 1 Meta notations

<foo>	Means foo is nonterminal
foo	(in bold font) means that **foo** is a terminal
[x]	Means zero or one occurrence of x. x is optional
x^+	Means at least one or more occurrences of x
|	Separates alternatives

Below are represented grammar productions for Domain Specific Language for Astrology, Astro:
<source_code> -> <statement> *
<statement> -> <variable_declaration> **;** | <print_statement> **;**
<variable_declaration> -> **define** <varible_name> = <variable_value>

<variable_value> -> **person (** <date_of_birth>
<time_of_birth>
[<ptzone_house_node_type>]
) | calculate <calculation_type> **for** <variable_name> [**and** <variable_name>]
<date_of_birth> -> **day :** <digit>[<digit>] **, month :** <digit>[<digit>] **,**
year : <digit><digit><digit><digit> **,**
<time_of_birth> -> **, hour :** <digit>[<digit>] **, min :** <digit>[<digit>] **,**
<location_of_birth> -> **, lat :** <float> **, lon :** <float> **, tzone :** <float>
<ptzone_house_node_type> -> **, prediction_timezone :** <digit><digit>*
, house_type : " <alpha>$^{+}$ **" , node_type : "** <alpha> $^{+}$**"**
<print_statement> -> **print** <variable_name>
<varible_name> -> <alpha_num>$^{+}$
<calculation_type> -> **western_horoscope** | **western_chart_data** | **tropical_transists/daily** | **tropical_transits/weekly** | **tropical_transits/monthly** | **solar_return_details** | **solar_return_planets** | **solar_return_house_cusps** | **solar_return_planet_aspects** | **lunar_metrics** | **composite_horoscope** | **synastry_horoscope** | **tropical_transits_timing/monthly** | **tropical_transits_timing/daily** | **general_ascendant_report** | **general_sign_report** | **general_house_report** | **romantic_personality_report** | **personalized_planet_prediction/daily** | **life_forecast_report** | **ramantic_forecast_report** | **friendship_report** | **karma_destiny_report** | **love_compatibility_report** | **romantic_forecast_couple_report** | **zodiac_compatibility**
<alpha_num> -> <alpha>|<alpha>$^{+}$<digit>$^{+}$
<digit> -> **0**|...|**9**
<float> -> <digit>$^{+}$**.**<digit>$^{+}$
<alpha> -> **a**|...|**z**|**A**|...|**Z**

2. Semantics and semantic rules

First of all the code syntax is checked by the parser and this will be done with the help of the *ANTLR* tool. If all the specifications from the grammar are respected the script will run successfully otherwise it will display in the console a message that will tell that something is wrong. For further development, the error message could be more explicit and tell the line where the error appeared [2].

Astro DSL consists of two types of statements which are a variable declaration that can include a function call or defining a composed variable of type person, and the other type - the **print** statement. The **print** statement prints data in the console in JSON format. In a further development of the DSL, the **print** statement might be replaced with a show statement which will show the result in a graphical more friendly way. **Print** statement can be considered as a function with one argument and this argument can be any type of variable. If the argument is a variable of the type person then it will print all the person's data. If the argument is a variable that it's value has been given by a type of statement which starts with the keyword calculate, then the print statement will show the calculated results concerning astrology.

When the code we can write the statement one after the other and separate them by semicolons. The only rule to keep in mind is that the variable should be declared before they are used. There is no special bloc for variable declaration, neither for the function call.

We can declare a variable, then call a function and again it is possible de declare another variable.

Statements are executed one after another from top to bottom, similar to the scripting language.

Astro DSL has some characteristics of scripting language and if we talk about purpose, it has characteristics of query language since behind the scenes it fetches astrology data from an *API*.

3. Data types

There are three basic types - *int, float*, and *string*. In addition, there is a *struct* data type, *Person*, which is a complex data type. It consists of the next variables: *Day, month, year, hour, min* and optional prediction *timezone*, all this are personal data of a user, which are of the type int. *Lat, lon,* and *tzone* variables represent the location of birth and are of the *float* data type.

4. Variable scopes and rules

As it was said in Semantics sections the *DSL* supports variable declaration and there is only one main rule. This rule is that variables should be defined before they are used. It means that variables should be defined upper in comparison with the line where it is used in a function call or in a print statement.

Since *Astro DSL* doesn't have blocks of code that can be (imbricate) there are no variable scopes based on blocks of code. The scope of each variable is the area below the line where it was defined. In order to define a variable, the **define** keyword should be used in front of the variable name then the = assignment operator followed by a function call which normally starts with **calculate** keyword or **person** keyword which indicates that it follows to be defined data about a person.

5. Type of assignments

The *assignment operators* are equal (=) and colon(:), which assigns the value of its right operand to its left operand. The simple assignment operator is used to assign a value to a variable. The colon (:) is used to assign a value of *float*, *alpha* or several *digits* number to *day, month, year, hour, min lat, lon, tzone*, prediction *timezone*, *house type*, and *node type*.

6. Example of code & Parsed tree by ANTLR

Below is represented a small piece of code, which define a variable named *result1* in which is stored the calculations given for the *wester_horoscope* depending on the data which are stored in variable *ion*. The *print* statement displays on screen the data from the variable *result1*.

```
define results1 = calculate western_horoscope for ion;
define results2 = calculate love_compatibility_report for vasile and ana;
print results1;
```

In Table 2 are represented the *tokens* and *lexemes* for the given above code.

Table 2 Tokens and Lexemes

Token	Lexeme
keyword	define calculate for and print
identifier	results1 results2 ion vasile ana
calculation type	western_horoscope love_compatibility_report
assignment	=
semicolon	;

In Figure 1 is represented the parsed tree obtained in ANTLR.

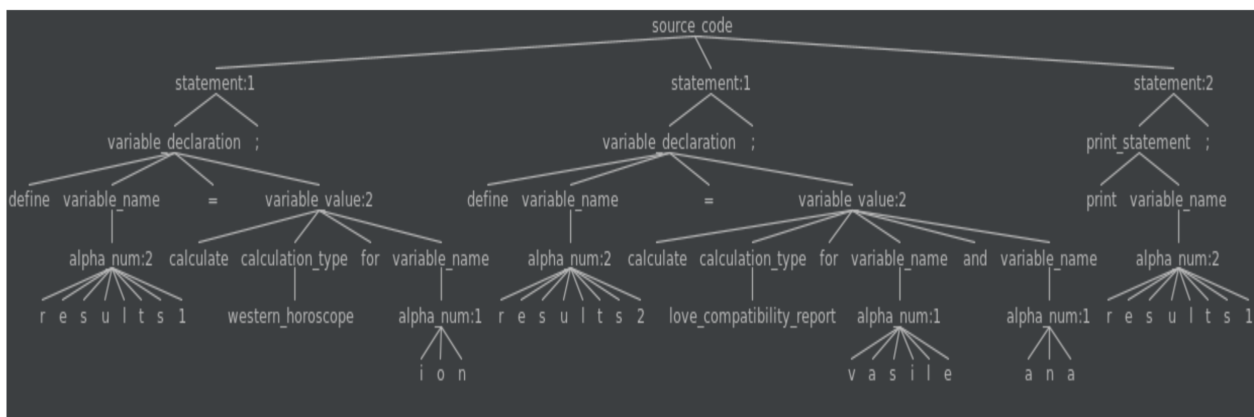

Figure 1 Parse tree in ANTLR

Conclusion

This project is focused only on *Western Astrology*. The purpose of this project is to gather data from *Astrology API* and give it to the user. The results met all the criteria and all the expectations. For further development is planned to enhance the *DSL* and algorithms. To create the *DSL* there are two main steps: creations of the parser and processing the parse tree in a known general purpose language. *ANTLR* is the tool without which it could be difficult to create the language. When the *Astro* code is run, the first step is the creation of the *lexer*, then the *tokens* object is created, then the parser and finally the *parse tree*. All this depends on the code input. All this is done in *Java*, but it could be done in many other languages. After we have the parse tree, it is processed and in this way, the specific action is taken.

Bibliography

1. https://astrologyapi.com/
2. https://tomassetti.me/
3. Markus Voelter, *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*, 2010-2013