DOMAIN SPECIFIC LANGUAGE FOR SOLVING GENETICS PROBLEMS

Vasile CEBAN, Damian GROSU, Lina SCRIPCA*, Andrei ZACATOV

Software Engineering, FAF-201, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Scripca, Lina, lina.scripca@isa.utm.md

Abstract. *This article describes a Domain Specific Language for solving genetics problems that require the usage of Punnett Square. Subsequently, this paper delves into the syntaxis, functionality and creation of the language, as well as the steps required for its proper usage for maximum automatization of the steps for problem solution.*

Key words*: domain-specific language, genetics, punnett square, character inheritance.*

Introduction

A Domain Specific Language (DSL) is a language designed to provide a notation tailored towards an application domain and is based only on the relevant concepts and features of that domain [1]. Or, in other words, it is a language designed specifically for ease of usage for experts of a specific domain, with supported for syntactic sugar relevant to that specific domain.

A Punnett square problem is a problem that given the allele (symbolic notations of the genes carrying a certain hereditary character) of a set of parents, determines the probability of that character's propagation in the next generation. For its solution, a parent's genotypical constitution is broken down into possible gamete sets, and then combined with all the other possible gametes of the other parent to obtain a square that would show the distribution of characters of their kids [2].

While the solution seems pretty straightforward, the complexity of finding it rises exponentially with the number of genes that are being monitored, going from 16 possible cases for 2 sets of alleles, towards 256 cases for 4, creating a large margin for human error on the basis of attention during their analysis.

Thus, the domain of genetics, and, specifically, the branches that study the Mendelian character inheritance and population diversity using Punnett square, could benefit from the automatization of problem solution provided by a DSL.

Solution Concept

The proposed DSL will help solve genetics problems using The Punnett square. The basic tool will generate predictions of the genotypes of a particular cross or breeding experiment.

The language will be intuitive and require beginner level coding skills and knowledge in genetics. With the proposed DSL the users will write a code where they will specify the parent, the notation of the alleles, which alleles are dominant or recessive as well as other necessary parameters for the studied problem, after which the DSL will be interpreted, executed and computed to get as output the solutions to the problem, be they a Punnett Square, the possible genotype of a parent, or information about the inherited genotypes and phenotypes of the next generation, along with their probability.

Computational Model of the DSL

The basic items of computation in the proposed language are the data held in variables, types of which will be described in the following paragraph. The problem description model that was chosen envisions the language as being an object-oriented imperative language, following a Control-Driven execution style. The main computational methods are build-in and user defined functions, such as creating a new generation (via a **cross** function), or processing already achieved data (via a **find** <property> statement).

Data Structures

The proposed language will contain Atomic Data types and Object Types. The atomic data types will be closely related to already existing languages, while the Object Types will be custom structures in a vocabulary closely known by people in the specific domain.

Atomic types will include:

Nil / Null - type for everything uninitialized.

Boolean - logical 1 or 0, using 1-byte values.

String - A pointer to an immutable Cstring (zero-terminated string)

Number - An IEEE 754 binary64 type number, more commonly known as a "double" type variable. Using this datatype for both integers and real numbers aims to reduce the amount of variable types and hence the confusion, while still maintaining a more than decent integer value range (exact representation for numbers in the range of ±2^53 exactly and up to ±2^54 with rounding to the nearest multiple of 4).

Composite / Object types will be represented by:

Gene - A basic unit of genetics, that will contain information about the dominance properties of the allele variants, their phenotypical label, location, as well as the existence of codominant variants.

Parent - an object containing genes, predefined methods and properties, as well as a possibility to add custom methods to aid in generalizing the language for broader tasks.

Generation – an object containing several parent candidate organisms as well as the probability of their manifestation within the generation.

Variables can be initialized with the appropriate variable type keywords followed by the names of the variables separated by commas. Most atomic data types will be defined with an appropriate default value, 0 or false for a Boolean, 0.0 for a number variable, while other types will be initialized as Nil/Null. One can also assign values to already defined variables via a set command.

First Iteration functionality

The first iteration of the proposed language is aiming to provide the following operations to the end users through the indicated functions:

cross parent **x** parent **–** computation of a new generation using the genetical composition of a set of parents as the base.

find field genes – computation of all possible genotypic variants containing the genes declared within the program and the allele given within the function, returning the indicated field as the result.

pred generation – computation of all possible parent variants based on the traits inherited by a generation.

estimate generation number – computation of the solution for a population problem, given the trait inheritance probability within a generation and the total number of expected individuals within it.

Grammar definition

The proposed DSL contains the following grammar and production rules, with the additional notation legend explication offered in Tab. 1:

Table 1

Additional Grammar notations

notation	**explanation**
<foo>	nonterminal
notation	**explanation**
foo	terminal
[x]	zero or one occurrence of x
x^*	zero or more occurrence of x
x^+	one or more occurrence of x
|	alternative

G(L) = {V_n, V_t, S, P}, where V_n - neterminal symbols, V_t – terminal symbols, S – Starting symbol and P – finite set of production rules.

V_n = {<program>, <statements>, <declarations>, <assignments>, <flow structures>, <computations>, <io>, <type>, <id>, <alphanum>, <alpha>, <bigalpha>, <smallalpha>, <digit>, <char>, <field>, <expression>, <condition>, <operator>, <number>, <string>}

V_t = {; , “,”, “ ”, genes, parent, generation, boolean, string, number , +, -, =, ->, <, <=, >=, ==, !=, and, or, ?, :, if, then, else, while, do, end, a…z, A…Z, /, . , find, cross, pred, estimate, =, print, all }

S = {<program>}

P = { ≤program> → < statements >$^+$

<statements> → <declarations> | <assignments> | <flow structures> | <computations> | <io>

<declarations> → <type> <id>[,<id>] ;

<type> → genes | parent | generation | boolean | string | number

<id> → <alpha>< digit >*< alpha >*

<alphanum> → <alpha> | <digit> | <char>

<char> → / | ? | . | ; |“ ”|

<alpha> → <bigalpha> | <smallalpha>

<smallalpha> → a|…|z

<bigalpha> → A|…|Z

<digit> → 0|…|9

<assignments> → set [<field>] <id> = <expression> ; | set [<field>] <id> = <computations> ;
| set dom: <bigalpha> -> <smallalpha>;

<field> → label | dom | phenotype | codominance | loaction

<flow structures> → if <condition> then < statements >$^+$ [else < statements >$^+$] end;
| <condition> ? <statements> : <statements>;
| while <condition> do < statements >$^+$ end;

<condition> → <id> <operator> <id> | <id> <operator> <expression>

<expression> → <number> | <string> | true | false |

<number> → < digit >$^+$[. < digit >$^+$]

<string> → “< alphanum >$^+$”

<operator> → < | > | <= | >=| == | != | and | or

<computations> → find <field> <id>[;] | cross <id> x <id>[;] | pred < id >$^+$[;]
| estimate <id> <number>[;]

<io> → print <id>; | print <field> [<id> | all <expression>];

}

Parsing tree

The image below (Fig.1) describes the parsing of the proposed DSL grammar to obtain a workable program in the language.

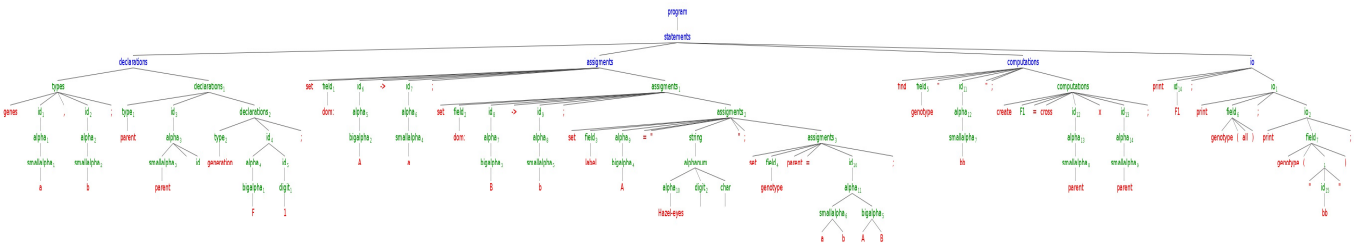

Figure 1 Parsing Tree

With a better resolution image available upon request or at [3].

Syntax example:

In Fig.2 is presented an example of a working program with its concept output. The example described is the same as the one obtained through the Parsing Tree.

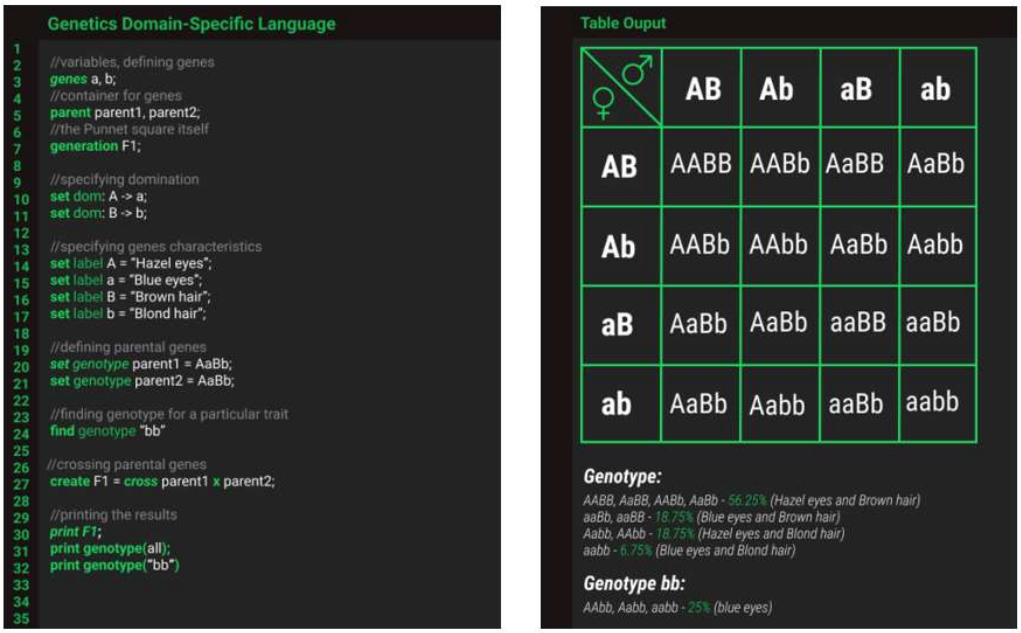

Figure 2 DSL program and output

Conclusions

The purpose of this article was to showcase the functionality, syntaxis and creation of a new Domain Specific Language for solving problems associated with Punnett Square, a tool that would help automatize the solution of those problems, while mitigating user attention error.

The proposed DSL is easy to use, with datatypes and syntactic sugar that resembles the natural solution steps that interested parties, such as genetics consultants, students, and animal and plant breeder, are already familiar with, while also implementing certain programming concepts, such as loops and control structures, that would benefit more programming immersed individuals.

Because of lack of existence of alternatives, the described prototype is but a small solution to a overarching problem that would also benefit from additional research and professional input.

References:

1. KOSAR, T., MARTI, P.E., BARRIENTOS, P.A. AND MERNIK, M.. A preliminary study on various implementation approaches of domain-specific language. *Information and software technology*, *50*(5), 2008, pp.390-405.
2. THOMSON, N. AND STEWART, J. Secondary school genetics instruction: making problem solving explicit and meaningful. *Journal of Biological Education*, *19*(1), 1985, pp.53-62.
3. GitHub repository of project: Parsing Tree: https://github.com/AlmightyCrickityCrick/punnett-dsl/blob/main/syntax_tree.png