# DEVELOPING A GENERAL-PURPOSE LANGUAGE

**Alexandru CORNEA[1], Adrian GÎRLEA[1*], Vladislav MASLIHOV[1], Polina PLOTNICOV[1], Ion POALELUNGI[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: Adrian Gîrlea, girlea.adrian@isa.utm.md

***Abstract:*** *This Article provides a summary of the DSL language "CodeTechy". The topics of domain analysis, motivation, functionality, target audience, competitors and others will also be discussed. A particular analysis will be provided to its grammar.*

***Key words:*** *General Purpose Language (GPL), Grammar, Backus-Naur form, lexer, parser.*

**Introduction**.

There is no doubt programming has become one of the most important domains in our life. Because of the high demand on the market more and more people want to study programming and to become a demanded specialist in the domain. Unfortunately, a lot of people get stuck in the beginning, and abandon the idea. In order to help people to better understand the basics and push the students into more advanced topics we plan to develop a General-Purpose Language [1] with a simplified syntax, that will make it more intuitive. In order to simplify the process such, topics that are often considered as more advanced like pointers, memory allocation, a large number of data structures, function declaration and others will not be included.

**Computational Model**

Computational models [2] are mathematical models that are simulated using computation to study complex systems by computer stimulations. Due to multitude of programming languages use different types of models of calculation (i.e., a program that has as its main feature the operations with matrixes will use as a model matrix processing). As our language's purpose is to help people study programming, we won't add anything similar or any computational model focusing on another subject. In order to dispense a product that will help and not confuse people, we will implement a simplified version of a generic programming language.

**Semantics and semantic rules**

Codetechy has simple and quite restrictive rules. All identifiers must be defined (textually) before use. It has at least one valid scope at any point and that is global scope. Also, there are additional local scopes within each block of code. Scopes can be nested. Any identifier introduced must be unique and can't shadow names from outer scopes. Variables can be used only within the scope where they were introduced. After application leaves respective block of code the variable can't be longer accessed.

**Lexical Analysis:**

All Codetechy keywords and identifiers are non-sensitives.

List of keywords: *int, float, string, boolean, array, true, false, if, else, begin, end, for, from, to, downto, while.*

Comments are started by a *//* and last till the end of the line. Horizontal white space may appear between any lexical tokens on a single line. Horizontal white space is defined as one or more spaces or tabs. Keywords and identifiers must be separated by white space or a token that is neither a keyword nor an identifier.

### 1. Data Structure

Primitive data types of our DSL have categories:

- numerical type (int, float);
- string type (string);
- logical type (boolean);

These types have two ways of declaring:

- literal (numerical and string);
- keywords (logical type);

String type literal contains a sequence of characters without identification symbols, like «». Numerical literals are taken as double variables, that is, there is no difference between int and float for the program, both of them are floating-point after the lexer stage. Boolean type Defined by DSL is identified by "true" and "false" keywords.

Furthermore, there are array variants for each of the primitive data types (int [], string []). All arrays are one-dimensional and have a variable length. Array elements are indexed from 0 up to N-1, where N is the length of the array. The usual bracket notation is used to index arrays. Array length can be queried by using the length(a) built-in function.

### 2. Control Structures

- Branching operation (if … else…);
- Recurrence operation (for, while);

Branching operation in our language is defined according to the general standard of many programming languages. The *if* branching expects a Boolean expression, in case the expression is equal to True the statement will execute code written in the statement. Additionally, an *else* branch can be added, in order to execute some code only then the Boolean expression is False.

The while loop also has a structure to other languages. It evaluates the value the expression inside the parenthesis, if the expression is true the loop will execute the code included in the *block statement* delimited by the words (begin, end) until the expression will change to false or until we reach a break statement.

The for-loop's purpose is to execute the code included in the already discussed *block statement* for an n amount of times. Unlike other popular languages the value of the identifier can be only incremented (specified by the keyword *to*) or decremented (specified by the keyword *downto*) in the specified range.

### Reference Grammar

In Tab. 1 are represented notations used in grammar specification

*Table 1*

**Notation**

| **<str>** | **means str is a nonterminal;** |
|---|---|
| str | in bold font means that str is a terminal i.e., a token or a part of token; |
| [x] | Means zero or one occurrence of x, i.e., x is optional; |
| $X^*$ | means zero or more occurrences of x; |
| $X^+$ | A comma-separated list of one or more x's; |
| {} | Large braces are used for grouping; |
| \| | Separates alternatives; |

VN = {<source_code>, <statement>, <variable declaration>, <comment>, <text>, | <location>, <expression>, <function call>, <identifier>**,** <block>, <literal>, <location>, <binary operator>, <unary operator>, <type name>, <function name>, <letter>, <digit>, <equality operator>,

<arithmetic operator>, <comparison operator>, <condition operator>, <string literal>, <boolean literal>, <integer literal>, <float literal>, <string literal>, <array literal>}

VT = { if, else, from, for, to, downto, while, begin, end, int, float, string, boolean, [ ] ,not, and, or, //, -, ==, !=, +, -, *, /, %, >, <, <=, >=, true, false, **“**ASCII characters”, [, ], a , A, b, B, c, C, d, D, e, E, f, F, g, G, h, H, I, I, j, J, k, K, l, L, m, M, n, N, o, O, p, P, q, Q, r, R, s, S, t, T, u, U, v, V, w, W, x, X, y, Y, z, Z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

P = { <source_code> —> <statement> [<comment> <text>]
<statement> —> <variable declaration> | <location> **=** <expression> | <function call> |
**if** <expression> <statement> [**else** <statement>] | **for** <identifier> **from** <expression>
[**to**, **downto**]* <expression> <statement> | **while** <expression> <statement> | <block>
<expression> —> <literal>| <location> | <function call> | <expression> <binary operator>
<expression> | <unary operator> <expression>| **(** <expression> **)**
<block> —> **begin** <statement> **end**
<variable declaration> —> <type name> <identifier> = <expression>
<function call> —> <function name> **( [**<expression> { **,** <expression>} **] )**
<identifier> —> <letter>* | **_** [<letter> | <digit> | **_** ]*
<type name> —> **int** | **float** | **string** | **boolean** | [ **[ ]** ]
<function name> —> <identifier>
<location> —> <identifier> | { **[** <expression> **]** }*
<binary operator> —> <equality operator> | <arithmetic operator> | <comparison operator> |
<condition operator>
<text> —> <string literal>
<comment> —> **//**
<unary operator> —> **not** | **-**
<equality operator> —> **==** | **!=**
<arithmetic operator> —> **+** | **-** | ***** | **/** | **%**
<comparison operator> —> **>** | **<** | **<=** | **>=**
<condition operator> —> **and** | **or**
<literal> —> <boolean literal> | <integer literal> | <float literal> | <string literal> | <array literal>
<boolean literal> —> **true** | **false**
<integer literal> —> [-] <digit> {<digit>} +
<float literal> —> <integer literal>**.** <digit> | <integer literal>**.** {<digit>} +
<string literal> —> **“ASCII characters”**
<array literal> —> **[**<expression>| {**,** <expression>} +**]**
<letter> —> **a** | **A** | **b** | **B** | **c** |**C** | **d** | **D** | **e** | **E** | **f** | **F** | **g** | **G** | **h** | **H** | **i** | **I** | **j** | **J** | **k** | **K** | **l** | **L** | **m** | **M** | **n** | **N** | **o** | **O** | **p** | **P** | **q** | **Q** | **r** | **R** | **s** | **S** | **t** | **T** | **u** | **U** | **v** | **V** | **w** | **W** | **x** | **X** | **y** | **Y** | **z** | **Z**
<digit> —> **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**}

**Example of code and parse tree**

In the following example we declare a new variable in the Codetechy programming language.

The code: *int a_4 =0*

Succeeding the code is being parsed by an abstract syntax tree. In Fig. 1 is represented the tree obtained.

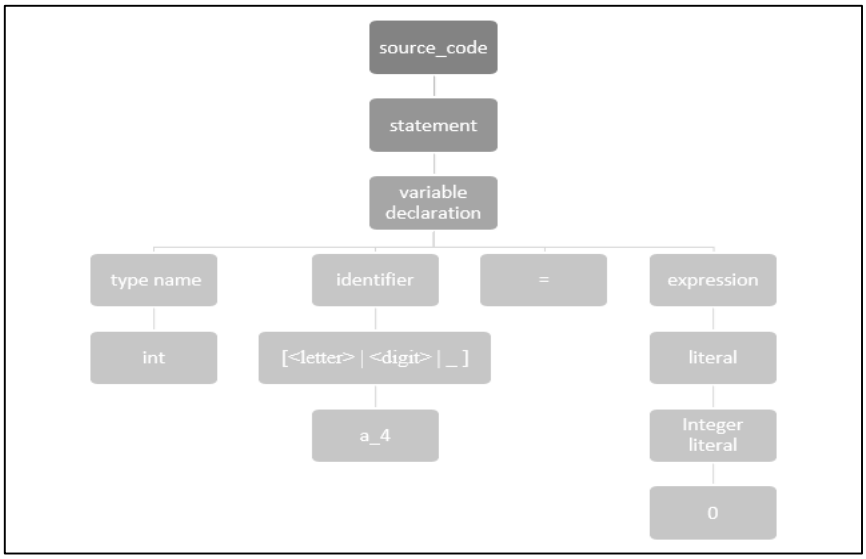


**Figure 1. Parse Tree**

**Conclusion**

In conclusion this paper gives a simplified description of the language developed by our team. The language's syntax will be similar to other GPL on the market but simplified in order to make the education process go without complications. Having an environment to help beginners progress and not get stuck will be our main goal. After all, everyone has to start from something in order to advance and reach the needed requirements to learn more advanced topics or even to find a job on the market.

**References**:

1. General-purpose programming language [online]. [accessed 13.03.2021]. Available: https://en.wikipedia.org/wiki/General-purpose_programming_language
2. Computational Model [online]. [accessed 13.03.2021]. Available: https://en.wikipedia.org/wiki/Computational_model