CREATING A GENERAL PURPOSE LANGUAGE: DESIGN AND IMPLEMENTATION OF PRISM

Valeria COZLOV[1*], Tudor SÎRGHI[1], Beatricia GOLBAN[1], Ion VASLUIAN[1]

[1]*Department of Software Engineering and Automatics, FAF-213, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova*

*Corresponding author: Valeria Cozlov, valeria.cozlov@isa.utm.md

Scientific coordinator: Alexandr VDOVICENCO, lector-assistant, DISA.

Abstract. *This article presents the development of a new general-purpose programming language, PRISM. The language was designed to be simple, intuitive, and versatile, with features that cater to both beginners and experienced programmers. The article discusses the language's grammar, data structures, and built-in functions, highlighting its potential applications and benefits for various industries.*

Key words*: General Purpose Language, Grammar, Abstract Syntax Tree (AST), Interpretation, Parse Tree.*

Introduction

Special purpose (domain specific) and general purpose languages are the two types of high-level programming languages used in computer programming. Although general purpose languages can be used in a wide range of computer applications, hardware setups, and operating systems, they are fundamentally asynchronous and non-deterministic. They enable portability by allowing the same software to execute equally on a single CPU or on a network of computers [1].

Creating a general-purpose programming language can be a complex and challenging task, but it is also a highly rewarding one. A general-purpose language is designed to be used for a wide variety of applications and can be used to write software for everything from web applications to operating systems.

The process of creating a language involves defining its syntax, grammar, and semantics, as well as designing its features and capabilities. This can be done using various tools and techniques, such as lexers, parsers, and compilers.

When designing a language, it's important to consider factors such as readability, ease of use, and performance. The language should also be designed to allow for efficient execution of programs and offer features that make it easy for developers to write and maintain code.

Overview of language implementation

One of the key features of PRISM is its simplicity. The language has a clean, easy-to-read syntax that is designed to be intuitive for developers. It also supports type inference, which makes it easier to write code without having to specify the data types of variables.

PRISM provides a rich set of data types, including, but not limited to:

- integers;
- floating-point numbers;
- strings;
- arrays.

It also supports built-in functions for common operations, such as string manipulation and mathematical calculations. Interpreting PRISM involves understanding its syntax, grammar, semantics, and features. Here are some steps that are followed to interpret this language:

1. Read the language documentation: The first step is to read the language documentation to understand the syntax, grammar, and semantics of the language. The documentation should provide examples and explanations of how the language works.
2. Understand the lexical structure: The next step is to understand the lexical structure of the language. This involves understanding the basic building blocks of the language, such as keywords, identifiers, literals, and operators.
3. Identify the grammar rules: Once the lexical structure is understood, the grammar rules of the language should be identified. This involves understanding the syntax of the language and how different components of the language fit together.
4. Identify the language constructs: The next step is to identify the language constructs, such as loops, conditionals, functions, and classes. These constructs define how code is organized and executed in the language.
5. Implement a parser: After understanding the grammar rules and constructs of the language, a parser can be implemented to parse the language code. The parser takes in the code as input and produces an abstract syntax tree (AST) as output.
6. Implement an interpreter: Once the AST is produced, an interpreter can be implemented to execute the code. The interpreter reads the AST and executes the code according to the rules of the language.

Grammar of the GPL

A mechanism is required to characterize it in order to investigate languages mathematically. Informal descriptions in English are frequently insufficient due to the imprecision and ambiguity of everyday language. Though it has limitations, the set notation is more appropriate. The concept of grammar must therefore be introduced for this reason [2].

A grammar *G* is defined as a quadruple *G = (V, T, S, P)*, where:

- *V* is a finite set of objects called non-terminal symbols;
- *T* is a finite set of objects called terminal symbols;
- $S \in V$ is a special symbol called the start variable;
- *P* is a finite set of productions

Notation:

<> - Nonterminal parameter;

Bold – terminal parameter

| - separator of alternatives

V = {<code>, <statement>, <variable_declaration>, <comment>, <text>, <integer_literal>, <float_literal>, <string_literal>, <boolean_literal>, <array_literal>

<expression>, <function_call>, <identifier>, <binary_operator>,

<unary_operator>, <type_name>, <function_name>, <digit>, <equality_operator>, <arithmetic_operator>, <comparison_operator>, <condition_operator>, <location>, <block>}

T = { if, else, from, for, to, downto, while, 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 = {<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>

<expression> → <unary_operator> <expression>| (<expression>)

<block> → {<statement>}

<variable_declaration> → <type_name> <identifier>= <expression> | <function _call>

<function_call> → <function_name> ([<expression>])

<identifier> → <letter>*|<digit>*
<type_name> → **int** | **float** | **string** | **boolean** | [[ ]]
<function_name> → <identifier>
<location> → <identifier>
<binary_operator> → <equality_operator>| <comparison_operator>| <condition_operator>| <arithmetic_operator>
<text> → <string_literal>
<comment>→ //
<unary_operator> → **not** | **-**
<equality_operator> → **==** | **!=**
<arithmetic_operator> → **+** | **-** | ***** | **/** | **%**
<comparison_operator> → **>** | **<** | **<=** | **>=**
<condition_operator> → **and** | **or**
<literal> → <integer_literal>|<boolean_literal>|<float_literal> |<string_literal> |<array_literal>
<integer_literal> → [-] <digit>
<boolean_literal> → **true** | **false**
<float_literal> → <integer_literal>. <digit>*
<string_literal> → "ASCII characters"
<array_literal>→ [<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**}

Input and output

The code for the PRISM program must be written in a file with the .psmx extension or directly in the command line for the Abstract Syntax Tree Interpreter to run the program.

The command line will show the direct output of a specific .psmx file. An executable file won't be produced because PRISM is an interpreted language and these files are only compatible with compiled programming languages like C or C++.

Conclusion

In conclusion, creating a new general-purpose language is a complex and challenging task that requires a deep understanding of programming concepts, syntax, and semantics. PRISM, the newly created language, is designed to be easy to learn and use, with a modern syntax and a rich set of features that enable developers to create powerful and efficient applications. With its focus on simplicity, clarity, and flexibility, PRISM has the potential to become a popular language among developers who value productivity and expressiveness.

References

1. GÉRARD BERRY *Real time programming: special purpose or general purpose languages* [online], 1989, [cite: 06.03.2023] Available: https://hal.inria.fr/inria-00075494/document
2. PETER LINZ *An introduction to formal languages and automata* [online], 2012, [cite: 06.03.2023] Available: https://fall14cs.files.wordpress.com/2017/04/an-introduction-to-formal-languages-and-automata-5th-edition-2011.pdf