MARGAY - GENERAL PURPOSE LANGUAGE DEVELOPMENT

Valeria DUBINA*, Ilie TODIRAȘCU, Maria-Madalina UNGUREANU, Marcel VLASENCO

Department of Software Engineering and Automatics, FAF-203, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Valeria Dubina, valeria.dubina@isa.utm.md

Abstract. *This work was created in the context of a Problem Based Learning (PBL) project, the main purpose of which was the analysis of compilers, interpreters and programming languages and further developement of a General Purpose Language (GPL) called Margay in Go Programming Language.*

Key words*: General Purpose Language (GPL), Abstract Syntax Tree Interpreter, Go, Grammar*

Introduction

Like all good inventions, programming languages were born out of people's desires and need to make their lives easier. After moving from manual labour to electrical signals, the rewiring needed to get the computer to perform a task turned out to be a hassle, so an assembly language that allows programmers to pass instructions directly to the CPU was invented.

Such impressive achievements were possible only thanks to the contributions of the best engineers of the last centuries. They found a more efficient ways to communicate with computer hardware and extended the practical applications of Programming Languages and – Compilers and Interpreters [1].

By gaining technical insights through the implementation of a general-purpose language, the aim of this article is to form a solid understanding of the underlying principles behind the inner workings of the programming languages and interpreters. The focus is on how the tools used by developers came to be and why. The intention is to acquire a much stronger understanding of the computer science fundamentals and how they can be applied for creating new ground-breaking innovations.

Language overview

For Margay Abstract Syntax Tree Interpreter – Go [2] programming language is used as intermediate language that will execute the program written in .margay file or directly in command line. This language was chosen for its infrastructure. Docker, Kubernetes, and Prometheus are some of Go's most commonly developed infrastructures.

The phases of interpreting the margay source code will chase the following steps:

- Reconstruction of the code into an abstract syntax tree (AST)
- Program execution according to the AST tree.
- Each sentence is analysed one at a time

Each value encountered in the interpretation process is wrapped in a structure that meets this particular type of object interface: *Integer, String, Boolean, Float, Null, Array* and *Functions, If-Else loop, For loop.*

There are several data types in Margay – number (integer or float), boolean, string, however, all types are not explicitly specified. In Margay arrays are also present, but all of them are one dimensional and accept all existing data type including functions. Declaration of an array is similar to variable declaration and the specification of array components are denoted in square brackets after the "=" character and are separate by a comma.

Assignment is permitted for all types, as it is shown in the grammar below:

- boolean: <boolean literal> → true | false
- integer: <integer literal> → <integer>

- float: <float literal> → <float>
- string: <string literal> → **"**<text>**"**

Functions invocation includes:
- passing argument values;
- performing the function's content;
- returning, with a possible outcome, to the main.

Some of the control structures implemented are:
- If/else:

This type of the if statement is a common one regarding the semantics. The first step represents the evaluation of the <expression>. In the case when the result of the evaluation is true, is executed the true section of this control structured function. In the other case, when the result doesn't correspond to the initial condition and is false then the else section is executed, if it has been set. To prevent ambiguity when matching an else section with its if statement, the Margay mandates that true and else parts be contained in braces.
- For:

The for statement acts as while loop with the exact same functionality. It loops through a block of code while some conditions are true. To prevent overflow, it is mandatory to increment (or decrement) the value of condition statement.

Basic computational model supported by the language are:
- **Addition**: Symbol: "+" Variable types: *integer, float, string*. Exceptions: When one of the input variables is of type string - the concatenation operation will be executed
- **Subtraction** Symbol: "- "Variable types: *integer, float*
- **Multiplication** Symbol: "*" Variable types: *integer, float*
- **Division** Symbol:" /" Variable types*: integer, float*

Supplementary computational functionalities as square-root, raise to a power, factorial etc. would be implemented as an additional library to the GPL, developed using those basic operations that would be already defended in GPL.

Some of the comparison operation include: **>, =, <=, ==,** and != . This operation will output true or false depending of the given values. Their behaviour is similar to other programming language with no change.

Logical operation such as *and* with key element "*&&"*, *or* with key "||", not with key element "!" are also implemented. This logical operation follows the principles of programming languages and act accordingly.

Reference Grammar:

Meta notation:
< > - nonterminal param;
bold – terminal param;
| - separates alternatives;

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

V_n = {<program>, <statement>, <return statement>, <expression statement>, <expression>, <prefix expression>, <infix expression>, <if expression>, <for expression>, <identifier>, <literal>, <function>, <function call>, <array>, <array call>, <map>, <map call > , <map element>, <prefix operator>, <infix operator>, <assignment operator>, <parameter>, <argument>, <array element>, <integer literal>, <letter>,<string literal>, <boolean literal> ,<float literal>, <digit>, <text>,<character>,<symbol> }

V_t = { ; , “,” , return, if, else, for, fn, true, false, +, -, /, *, == , != , > , <, >= , <=, || , &&, =, 0…9, a…z, A…Z, ! , @ , # ,$, %, ., {,}, (,), [,] }
S = {<program>}

P = {
<program> → <statement> **;** | <statement> **;** <program>
<statement> → <return statement> | <expression statement>
<return statement> → **return** <expression>
<expression statement> → <expression>
<expression> → <prefix expression> | <infix expression> | <if expression> | <for expression> | <identifier> | <literal> | <function> | <function call> | <array> | <array call> | <map> | <map call>
<prefix expression> → <prefix operator> <expression>
<infix expression> → <expression> <infix operator> <expression> | <identifier> <assignment operator> <expression>
<if expression> → **if (** <expression> **) {** <program> **}** | **if (** <expression> **) {** <program> **} else {** <program> **}**
<for expression> → **for (** <expression> **) {** <program> **}**
<function> → **fn**(<parameter>) **{** <program> **}**
<parameter> → ε | <identifier> | <identifier> , <parameter>
<function call> → <identifier>**(** <argument> **)**
<argument> → ε | <expression> | <expression> , <argument>
<array> → **[** <array element> **]**
<array element> → ε | <expression> | <expression> , <array element>
<array call> → <identifier>**[** <integer literal> **]**
<map> → **{** <map element> **}**
<map element> → ε | <literal> : <expression> | <literal> : <expression> , <map element>
<map call> → <identifier>**[** <literal> **]**
<identifier> → <letter> <identifier>
<infix operator> → **+ | - | / | * | == | != | > | < | >= | <= | || | &&**
<prefix operator> → **- | !**
<assignment operator> → **=**
<literal> → <string literal> | < boolean literal> | <integer literal> | <float literal>
<boolean literal> → **true** | **false**
<integer literal> → <integer>
<float literal> → <float>
<string literal> → **“**<text>**”**
<integer> → <digit> | <digit><integer>
<float> → <integer>**.**<integer>
<text> → ε | <character> | <character><text>
<character> → <letter> | <symbol> | <digit>
<digit> → **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
<letter> → **a | A | b | B | ... | z | Z**
<symbol> → **! | @ | # | $ | % | ...**
}

Input and Output

In order for the Abstract Syntax Tree Interpreter to execute the program written in Margay - it is needed for the code to be written in a file with .margay extension, or the code can be written directly in command line.

The straight output of a specific .margay file will be displayed in command line. Due to the fact that Margay is an interpreted language, an executable file will not be provided, because these types of files are specific to compiled programming languages such as C or C++.

Conclusion

To sum up, this article presents an overview on the process of designing Margay general purpose language, the main logic of it and the intended grammar. One of the main goals of the article is to make a much stronger understanding of the computer science fundamentals by avoiding to step of a learning curve and focusing on how programming tools came to be. The intended language is not a strict one, it provides the developer the freedom of decision-making, but the familiar syntax of other high programming language will be kept in order to make it simple for beginners in this field.

References:

1. RAKIA BEN SASSI *Compiler vs. Interpreter: Know The Difference And When To Use Each Of Them* [online], 19.01.2021, [cite: 25.02.2022]
   Available: https://betterprogramming.pub/compiler-vs-interpreter-d0a12ca1c1b6
2. THORSTEN BALL *Writing an INTERPRETER in go*. Germany: Thorsten Ball, 2018.