GENERAL PURPOSE LANGUAGE

Maria PROCOPII[1*], Alexei CIUMAC[1], Dorin OTGON[1], Andreea MANOLE[1], Alexandru DOBROJAN[1]

[1]*Department of Software Engineering and Automatics, FAF-212, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova*

*Corresponding author: Maria Procopii, maria.procopii@isa.utm.md

Scientific coordinator: Irina COJUHARI, conf. univ., dr., DISA

Abstract. *General-purpose programming languages are those that are meant to handle a broad variety of activities, such as scientific computing, data analysis, web development, and software engineering. The objective of this paper is to offer an overview of general-purpose languages, covering their characteristics, usage, and features. The article examines the advantages and disadvantages of utilizing general-purpose languages, as well as some of the most popular and commonly used general-purpose languages.*

Keywords*: General-purpose language, programming languages, software engineering, design, grammar*

Introduction

A general-purpose programming language is one that is designed to be flexible and adaptable enough to be used for a wide range of programming tasks. These languages are not restricted to a specific application area or field, but may be used to construct a broad variety of software applications, such as desktop and mobile apps, internet applications, and games [1].

General-purpose programming languages provide developers with a wide range of tools and capabilities, making it easier to construct complex programs. They frequently have a high level of abstraction, allowing developers to express complex notions in a plain and succinct manner without having to worry about the underlying complexity of how the machine actually executes the program.

One of the key benefits of using a general-purpose language is its versatility. Instead than learning and utilizing a distinct language for each activity, developers may use the same language to construct a variety of different apps. This facilitates jumping between projects and interacting with other developers who may be working on a variety of apps. A general-purpose program written effectively may take less time to execute than an equivalent program written in a simulation language. This is because a simulation language is designed to represent a wide range of systems using a single set of building blocks, whereas a general purpose program may be tailored to a specific application. While general purpose languages are present on almost every computer, a specific simulation language may be unavailable on the system that the analyst intends to use [2].

Language overview

A very user-friendly general-purpose language, with a small set of syntax structures that can easily build-up complex programs that can solve bigger problems, is the short description of our language. In order to succeed, this general-purpose language has the needed functionality for variables binding, if-else conditionals, functions and cycles.

The language is partially static typed, i.e. in the functions we are obliged to give a type for the parameters of the function, whereas when declaring a variable, for instance, we don't have to precisely specify a type for the variable. Although, the language support all the classical logical operators met in other programming languages, we tend to give a more friendly and easily understandable control flow experience to the user, thus we've add some rarely seen conditional syntax structure, by the use of the *unless* and *then* keywords.

Grammar definition

A computer language's syntax is an essential component. A grammar is a collection of rules that govern how a language's syntax works. It specifies the sequence of symbols and constructs that may be used to create expressions and statements in the language. The grammar of a programming language determines how the language is processed and code is executed.

Grammar in programming languages is related to the language's syntax. The syntax specifies how statements and expressions are built and combined to form more complex expressions. The grammar of a computer language is often specified using formal notation such as Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF) (EBNF). These notations allow for an accurate representation of the syntax of the language, as well as the allowable sequence of symbols and constructs that compose the language's expressions and statements [3].

The syntax of a computer language determines its utility and adaptability. A sophisticated or restrictive syntax may be more difficult to learn and use than a simple and flexible grammar. Grammar also has an impact on linguistic expressiveness. A more expressive grammar may permit more concise and readable code, whereas a less expressive grammar may necessitate more lengthy and difficult-to-read code. The grammar of the proposed language is described as $G = (VN, VT, S, P)$ where VN is thefinite set of non-terminal symbols, VT is the finite set of terminal symbols, S is the start symbol, and$P$ is the finite set of production rules.

VN = { <program>, <statement>, <expression>, <return expression>, <literal>, <id>, <comment>,
<text_char>, <conditional>, <for expression>, <array>, <element>, <function call>,
<args>, <function>, <param>, <bool>, <type>, <letter>, <digit> };
VT = { return, true, false, [], [a-z], [A-Z], [0-9], “, _, !, @ , # , $, %, && , || , > , < , >= , <= , == | !=, *,
/, +, -, ++, --, = };
S = { <program>};
P = {<program> -> <statement> | <statement><program>
<statement> -> <expression> |<expression><statement>
<expression> -> <id> | <literal> | <conditional>| <id><assign><literal>
<id><assign><expression> <for expression> |<array> | <bool><function> |<function call> |
<expression><logical operator><expression> |
<expression><arithmetic operator><expression> |<sufix><id> | <id><suffix> | <comment>
|<returnexpression>
<return expression> -> return <expression> | return
<literal> -> <string> | <int> | <bool>
<id> -> <text_char><id> | <text_char>
<comment> -> //<string>
<text_char> -> <letter><digit><symbol><text_char> | <letter><digit><symbol>
<string> -> "<text_char>"
<int> -> <digit><int> | <digit>
<conditional> -> if (<expression>) {<program>} |unless (<expression>) {<program>} |
if (<expression>) {<program>} else {<program>} |unless (<expression>) {<program>} then
{<program>}<for expression> -> for (<expression>;<expression;<expression>) {<program>} |for
(;<expression>;<expression>) {<program>}
<array> -> [<element>] | []
<element> -> <id>,<element> |<literal>,<element> |<id> | <literal>
<function call> -> <id>(<args>) | <id>()
<args> -> <expression> | <expression>, <args>

<function> -> def <id>(<param>) {<program>} |def <id>() {<program>}
<param> -> <type><id> | <type><id>, <param>
<bool> -> true | false
<type> -> <int> | <string> | <bool>
<letter> -> a | b | ... | z | A | B | ... | Z | _
<digit> -> 0 | 1 | ... | 9 | ε
<symbol> -> ! | @ | # | $ | % | ... | ε
<logical operator> -> && | || | > | < | >= | <= | == | !=
<arithmetic operator> -> * | / | + | -
<sufix> -> ++ | --
<assign> -> =
}

Parsing tree

A parse tree is a hierarchical representation of a string of symbols or code's syntactic structure, generally in the form of a tree or a directed acyclic graph. It is generated by a parser during the parsing process, which is the act of examining a program's syntax and establishing its grammatical structure. Fig. 1 depicts the parsing tree based on the grammar developed in the previous section.

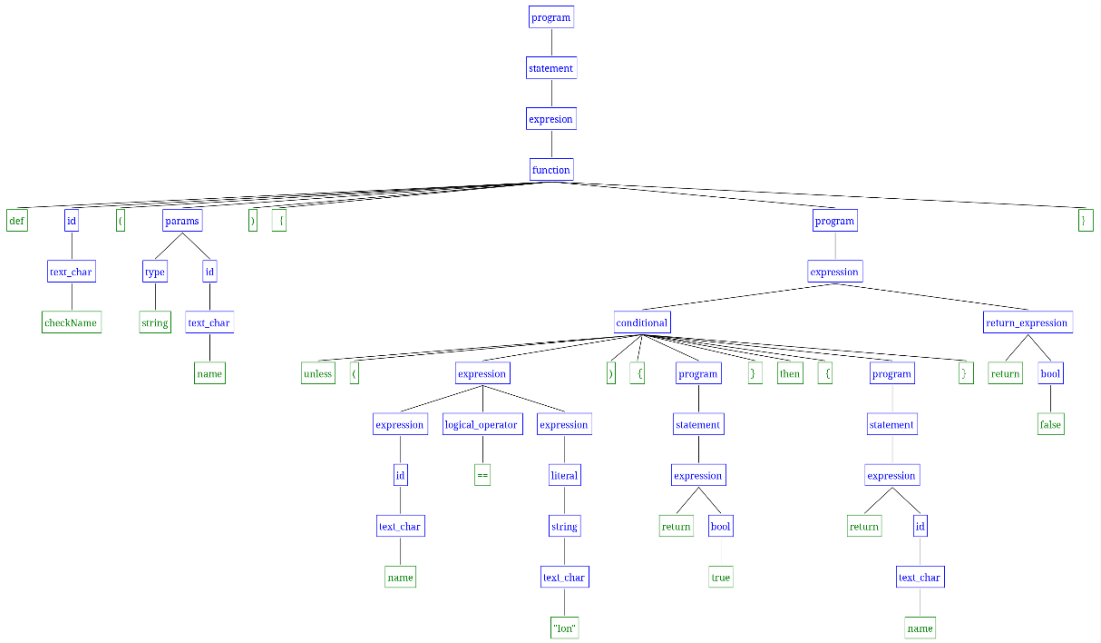

Figure 1. Parsing tree example

Syntax example

The code sample below is an example of syntax.

def checkName(**string** *name*) {
unless (*name* == “Ion”) {
return true
} **then** {
return *name*
}
return false
}

Conclusions

General-purpose programming languages are crucial tools for software development, scientific computing, data analysis, and web development. They are versatile and adaptable, allowing programmers to utilize them in a variety of scenarios. The programming language used is determined by the application domain as well as the programmer's expertise and talents. While general-purpose languages have many advantages, they do have certain disadvantages, such as slower performance or a lack of support for specialized features or libraries. Overall, general- purpose languages are crucial components of modern software development, and their relevance will only rise in the future.

A programming language's grammar is a fundamental component that specifies how the language's syntax operates. It influences the language's usability, flexibility, and expressiveness, making it an important factor to consider when selecting a programming language. A solid knowledge of a programming language's syntax may help programmers produce more succinct,legible, and maintainable code.

References

1. MARGARET ROUSE, TECH TARGET, *What is a General-Purpose Programming Language?* [online]. [accessed 26.02.2023]. Available: https://searchapparchitecture.techtarget.com/definition/general-purpose-programming-language
2. KING FAHD, *Advantages of General Purpose Languages* [online].[accessed 26.02.2023]. Available: https://faculty.kfupm.edu.sa/SE/habboubi/Courses/SE405_STOCHASTIC_SYSTEM_SIMULATION/Advantages_of_Simulation_languages.doc#:~:text=General%E2%80%94purpose%20languages%20allow%20greater,are%20not%20easy%20in%20GP SS.
3. ALFRED V. *AHO, MONICA S. L., Compiles Principles, Techniques, and Tools* [online]. [accessed 28.02.2023]. Available: https://repository.unikom.ac.id/48769/1/Compilers%20-%20Principles%2C%20Techniques%2C%20and%20Tools%20%282006%29.pdf