JADE - A GENERAL PURPOSE LANGUAGE

Dinu GUȚU*, Adelia BRĂGUȚA, Andrei SĂRĂTEANU,
Alexandra KONJEVIC, Milena COJUHARI

Department of Software Engineering and Automatics, Group FAF-213, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chișinău, Republic of Moldova

*Corresponding author: Dinu Gutu, dinu.gutu@isa.utm.md

Scientific coordinator: Alexandru VDOVICENCO, University Lecturer

Abstract. *General-purpose programming languages are crucial tools for the development of computer programs and are created to offer a broad range of capabilities and flexibility. The idea of general-purpose languages, their traits, their significance in contemporary software development, and how one can be developed are all covered in this article.*

Key words*: General purpose language, grammar, interpreter, lexer, parser.*

Introduction

The ability to express complex algorithms and create software systems is made possible by programming languages, which are crucial to software engineering. Despite the fact that there are numerous programming languages on the market right now, developing a brand-new general-purpose language from scratch takes considerable time and knowledge. Nevertheless, learning a new language can offer a lot of advantages, such as greater performance, readability, and expressiveness [1].

In recent years, there has been growing interest in creating new general-purpose programming languages that are better suited to the needs of modern software engineering. These new languages are designed to provide more efficient, scalable, and maintainable solutions to the challenges posed by complex software systems. Researchers and practitioners have been exploring new approaches to programming language design, seeking to improve upon existing languages and create new ones that meet the evolving needs of software development [2].

Creating a general-purpose programming language from scratch is examined in this work, with a focus on the creation of a lexer and the design of the language's grammar. In detail are laid out the theoretical foundations of programming language design, the principles of lexical analysis, parsing, and semantic analysis. The tradeoffs between expressiveness, efficiency, and readability are only a few examples of the practical factors that must be taken into account when creating a language.

A case study of general-purpose programming that makes use of the aforementioned methodologies was offered in this article to illustrate these findings. In addition, the grammar and lexer are discussed in detail, along with any difficulties that can arise during the development process. The analysis shows that it is possible to develop a general-purpose language that is both descriptive and effective with careful planning, attention to detail, and a thorough understanding of the principles of programming and language design.

Language overview

A general-purpose programming language (GPL) is a type of programming language that can be used to create software for a wide variety of application domains. Although their use within a specific domain is somewhat inferior to that of domain specific languages, the existence of libraries that are extensively tested and optimized should in theory bridge the gap between them [3].

Jade is a general-purpose language that is designed to be simple, versatile, and efficient. The language is based on a minimalist syntax that relies heavily on indentation and white space. This makes it easy to write and read code, and reduces the amount of typing required. Jade also supports templates and mixins, which can greatly simplify the development process by reducing the amount of repetitive code that needs to be written.

The languages' abstract syntax tree interpreter will run the program written in a file with .jd extension, or in the command line. The interpretation of the source code will follow the reconstruction of the code into an abstract syntax tree and each sentence will be translated and executed at a time [4]. Its lexer and parser have the role of managing the grammar. This will perform in the following way: the lexer will analyze the input sequence and convert it into a string of tokens, then, the parser will take over the generated tokens and use them to create the abstract syntax tree.

Tokens are the representation of several terminal symbols existing in the grammar. They are composed of the: identifiers, literals, operators and punctuations. These symbols are the main component that makes possible the creation of rules of the set of non-terminal symbols.

Grammar

In formal language theory, a grammar (sometimes referred to as a formal grammar when the context is not specified) describes how to construct strings from an alphabet that are valid in accordance with the syntax of the language. The meaning of the strings or what can be done with them in any context are not described by a grammar; only their form is. A formal grammar is a collection of rules for creating these strings in a formal language. In addition to that, the grammar of a language can be represented using formalized notations like Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF), which provide a concise method to define a language's syntax. Because grammar establishes a computer language's structure, compilers and interpreters use it to parse and evaluate source code [5].

Table 1.

Meta notations

Notations	Meaning
<text>	a nonterminal symbol
"text"	a terminal symbol
text*	a symbol that appears zero or more times
|	an alternative follows

$G = (V_N, V_T, P, S)$
V_N - finite set of non-terminal symbols.
V_T - finite set of terminal symbols.
P - finite production rules.
S - start symbol.

V_T = { "identifiers", "literals", "operators", "punctuation" }

- identifiers: a set of alphanumeric strings starting with a letter or underscore
- literals: a set of numeric, string, and boolean values
- operators: a set of symbols representing arithmetic, logical, and bitwise operations

- punctuation: a set of symbols used for syntax, such as parentheses, braces, and semicolons

V_N = {<Program>, <Statement>, <Expression>, <AssignmentExpression>, <ConditionalExpression>, <LogicalExpression>, <BinaryExpression>, <UnaryExpression>, <CallExpression>, <MemberExpression>, <PrimaryExpression>, <FunctionExpression>, <ObjectExpression>, <ArrayExpression>, <VariableDeclaration>, <IfStatement>, <WhileStatement>, <ForStatement>, <FunctionDeclaration>, <ReturnStatement>, <BlockStatement>, <Identifier>, <Literal>, <NumericLiteral>, <StringLiteral>, <BooleanLiteral>, <Arguments>, <Property> }

S = {<Program>}

P = {

<Program> → <Statement>*
<Statement> → <Expression>
| <VariableDeclaration>
| <IfStatement>.
| <WhileStatement>
| <ForStatement>
| <FunctionDeclaration>
| <ReturnStatement>
<Expression> → <AssignmentExpression>
| <ArrayExpression>
| <ConditionalExpression>
| <LogicalExpression>
| <BinaryExpression>
| <UnaryExpression>
| <CallExpression>
| <MemberExpression>
| <FunctionExpression>
| <PrimaryExpression>
| <ObjectExpression>
| <Literal>
<AssignmentExpression> → <Expression> "=" <Expression>
<ConditionalExpression> → <LogicalExpression> "?" <Expression> ":" <Expression>
<LogicalExpression> → <LogicalExpression> "&&" <LogicalExpression>
| <LogicalExpression> "||" <LogicalExpression>
<BinaryExpression> → <Expression> "+" <Expression>
| <Expression> "-" <Expression>
| <Expression "" Expression>
| <Expression "/" Expression>
| <Expression "%" Expression>
| <Expression "==" Expression>
| <Expression "!=" Expression>
| <Expression "<" Expression>
| <Expression ">" Expression>
| <Expression "<=" Expression>
| <Expression ">=" Expression>
| <Expression "<<" Expression>
| <Expression ">>" Expression>

```
                        | <Expression "&" Expression>
                        | <Expression "|" Expression>
                        | <Expression "^" Expression>

    <UnaryExpression> → "!" <Expression>
                        | "-" <Expression>
                        | "+" <Expression>
                        | "~" <Expression>
    <CallExpression> → <MemberExpression> <Arguments>
    <Arguments> → "(" [<Expression> ("," <Expression>)] ")"
    <MemberExpression> → <PrimaryExpression> ("." <Identifier> | "[" <Expression> "]")*
    <PrimaryExpression> → <Identifier>
                        | <Literal>
                        | "(" <Expression> ")"
                        | <FunctionExpression>
                        | <ObjectExpression>
                        | <ArrayExpression>
    <FunctionExpression> → "function" [<Identifier>] "(" [<Identifier> ("," <Identifier>)]
    ")" <BlockStatement>
    <ObjectExpression> → "{" [<Property> ("," <Property>)] "}" <Property>→ <Identifier>
    ":" <Expression>
    <ArrayExpression> → "[" [<Expression> ("," <Expression>)] "]"
    <VariableDeclaration> → "var" <Identifier> ["=" <Expression>] ";"
    <IfStatement> → "if" "(" <Expression> ")" <BlockStatement> ["else"
    <BlockStatement>]
    <WhileStatement> → "while" "(" <Expression> ")" <BlockStatement>
    <ForStatement> → "for" "(" [<VariableDeclaration>
                        | <Expression>] ";" <Expression> ";" <Expression> ")"
                <BlockStatement>
    <FunctionDeclaration> → "function" <Identifier> "(" [<Identifier> ("," <Identifier>)] ")"
    <BlockStatement>
    <ReturnStatement> → "return" [<Expression>] ";"
    <BlockStatement> → "{" <Statement>* "}"
    <Identifier> → /[a-zA-Z_][a-zA-Z0-9_]/
    <Literal> → <NumericLiteral>
                | <StringLiteral>
                | <BooleanLiteral>
    <NumericLiteral> → /[0-9]+(.[0-9]+)?/
    <StringLiteral> → /"([^"\]|\.)"/
    <BooleanLiteral> → "true"
                        | "false"
}
```

Sample code

The array arr is defined in the following program using the values [1, 5, 9, 11, 0]. Then, it creates the function doubleArray, which multiplies each value in an array by two before returning the transformed array. Next, after supplying the arr array as input, it calls the doubleArray method. All of the values in the initial array are effectively doubled when the resultant array provided by doubleArray is set to the variable arr. As a result, the array will contain the values [2, 10, 18, 22, 0] at program's completion.

```
var arr = [1, 5, 9, 11, 0];
```

```
function doubleArray(arr) {
  var doubledArr = [];
  for (var i = 0; i < arr.length; i++) {
    doubledArr[i] = arr[i] * 2;
  }
  return doubledArr;
}

var doubleArr = doubleArray(arr);
```

Parsing tree

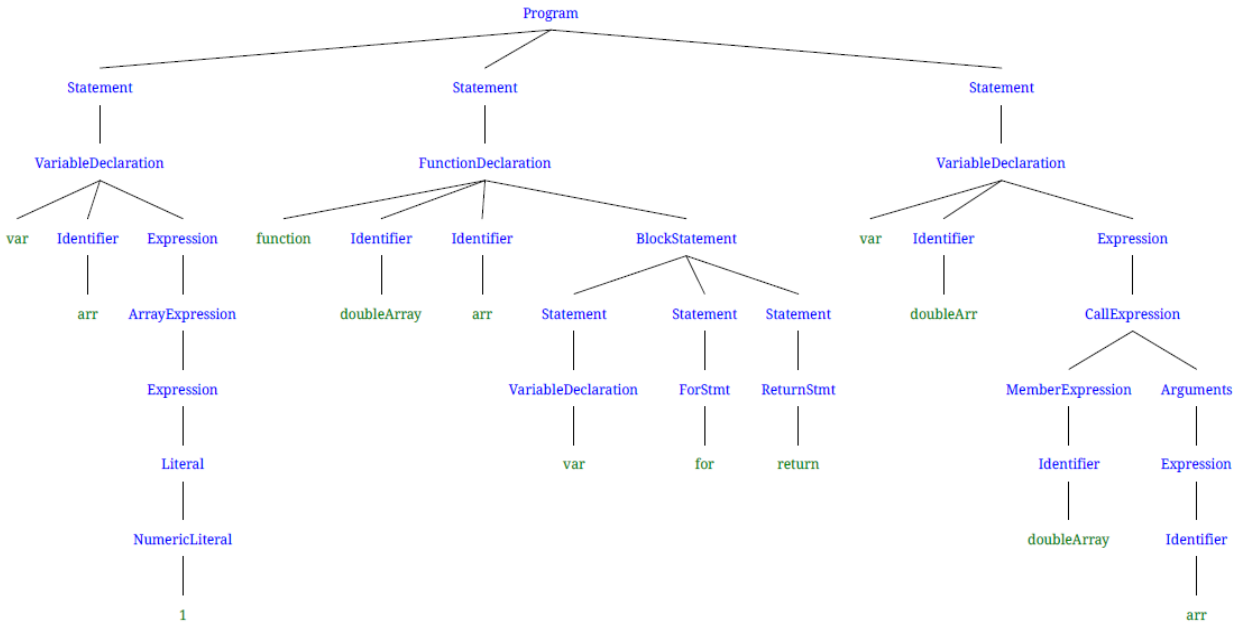

Figure 1. Parsing tree

The parsing tree represents the abstract syntax tree of the program that doubles an input array. It has been significantly reduced due to the big number of branchings that take place. The beginning is the <Program> which disperses into as many statements as needed, in this case, 3. Then each <Statement> will into a <VariableDeclaration> and a <FunctionDeclaration>. The former has a <Identifier> that represents the name of the variable and a <Expression> that can begin to represent what the variable will be. The latter, that is the <FunctionDeclaration>, also separates into identifiers to represent the name of the function and input, and also it branches off into a <BlockStatement> which covers the logic of the function.

Conclusion

Jade is a general-purpose language that is designed to be simple, versatile, and efficient. Numerous things, including software, research projects, mathematical calculations, and so forth, can be done using it. Its simple grammar, which places a heavy emphasis on indentation and white space, enables rapid and simple coding. Additionally, the use of templates and mixins greatly simplifies the development process, reducing the amount of repetitive code that developers need to write.

General-purpose languages, such as Jade, are designed to be flexible and all-round, with the goal of allowing developers to write code that can be used in a wide range of applications. These languages can be used to create any kind of program, including windows software, web applications, and analytical models. They are not constrained to certain industries or program

kinds. The main benefit of general-purpose languages is that they enable programmers to create code that can be applied in a number of situations, allowing them to reuse code and reduce effort redundancy. Additionally, they frequently have sizable, vibrant communities that offer a wealth of tools to aid developers in their learning and skill-building.

In conclusion, general-purpose languages are an important resource for developers who need to work across different domains or who want to write code that is more broadly applicable, even though specialized languages might be more effective in some situations. The final decision about the language will be made in light of the project's particular requirements as well as the preferences of the development team.

References:

1. AABY, A. *Introduction to Programming Languages* [online]. [visited 09.03.2023]. Available: https://web.archive.org/web/20121108043216/http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm.
2. Statistics and Data. [online]. [visited 09.03.2023]. Available: https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021.
3. KOSAR, T. *Comparing General-Purpose and Domain-Specific Languages: An Empirical Study*. In: *Computer Science and Information Systems*. 2010, pp. 247–264
4. VAIDEHI J. *A Deeper Inspection Into Compilation And Interpretation* [online]. [visited 02.03.2023]. Available: https://medium.com/basecs/a-deeper-inspection-into-compilation-and-interpretation-d98952ebc842.
5. ATTENBOROUGH M. Language theory. In: Attenborough M., ed *Mathematics for Electrical Engineering and Computing*. 2003, pp. 479-490.