O PROCESSO DE CRIAÇÃO DE UMA LINGUAGEM DE PROGRAMAÇÃO PARA A JAVA VIRTUAL MACHINE: UM ENFOQUE PRÁTICO

Matheus de Lima Dimer[1], Matheus Leandro Ferreira[2]

Resumo: Este trabalho apresenta o processo de criação de uma linguagem de programação compilada para a Java Virtual Machine, com o objetivo de proporcionar a compreensão prática das etapas de compilação e execução. A partir da definição de objetivos específicos, buscou-se compreender os princípios teóricos das linguagens de programação, examinar as ferramentas e metodologias existentes, desenvolver uma linguagem funcional baseada em ANTLR4 e JVM, e analisar os desafios e soluções surgidos durante a implementação. A linguagem proposta, denominada Simple Lang, teve sua gramática definida utilizando ANTLR4, enquanto a geração de bytecode foi realizada com o framework ASM. A linguagem permite declaração de variáveis, criação de métodos, estruturas condicionais, laços de repetição e impressão de dados, sendo todo o código-fonte convertido diretamente em bytecode executável. O projeto demonstrou que é possível construir uma linguagem compatível com a JVM com recursos limitados, permitindo a execução de programas escritos em Simple Lang de forma integrada com o ecossistema Java. A abordagem adotada permitiu uma compreensão mais clara da comunicação entre níveis de abstração da computação, cumprindo todos os objetivos propostos. O trabalho ainda abre espaço para futuras evoluções da linguagem, como suporte a arrays, importações externas e estruturas de controle adicionais.

Palavras-chave: compiladores; linguagem de programação; Java Virtual Machine; ANTLR4; ASM; bytecode.

[1]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), matheus-delima@outlook.com

[2]Curso de Ciência da Computação, Universidade do Extremo Sul Catarinense (Unesc), mlf@unesc.net

ABSTRACT: This work presents the creation process of a programming language compiled for the Java Virtual Machine, aiming to provide a practical understanding of compilation and execution steps. Based on specific objectives, it focused on understanding the theoretical principles of programming languages, examining existing tools and methodologies, developing a functional language using ANTLR4 and the JVM, and analyzing challenges and solutions throughout the implementation. The proposed language, named Simple Lang, had its grammar defined using ANTLR4, and its bytecode was generated through the ASM framework. The language supports variable declarations, method definitions, conditional structures, loops, and data output, compiling source code directly into executable bytecode. The project demonstrated that it is feasible to build a JVM-compatible language with limited resources, allowing programs written in Simple Lang to integrate with the Java ecosystem. The approach enabled a clearer understanding of the interaction between different abstraction levels in computing, fulfilling all proposed objectives. The work also paves the way for future enhancements, such as array support, external imports, and additional control structures.

Keywords: compilers; programming language; Java Virtual Machine; ANTLR4; ASM; bytecode.

1 INTRODUÇÃO

O desenvolvimento de linguagens de programação é um campo fascinante e essencial na ciência da computação. Diferentemente da comunicação humana, realizada por meio de linguagens naturais, os computadores interpretam apenas instruções em linguagem de máquina, compostas por sequências de zeros e uns, representando estados de energia elétrica (Rapp, 1985). Esse modelo, embora eficiente para máquinas, é intrinsecamente complexo e impraticável para o ser humano, levando ao surgimento de linguagens mais abstratas, como o Assembly e, posteriormente, linguagens de alto nível como Java (Sanati-Mehrizy; Minaie, 2003). O advento de linguagens de programação de alto nível, acompanhado pelo avanço dos compiladores, proporcionou uma ponte entre o homem e a máquina, permitindo o desenvolvimento de sistemas computacionais sofisticados. Um compilador, como descrito por Aho, Sethi e Ullman (1986), é um programa que traduz um código-fonte de uma linguagem de alto nível para uma linguagem de máquina. Essa transformação facilita a portabilidade entre diferentes hardwares e sistemas operacionais, além de simplificar a programação.

No entanto, essa evolução tecnológica também gerou um afastamento significativo dos desenvolvedores em relação às camadas de baixo nível da computação, criando uma lacuna de conhecimento, mesmo entre profissionais experientes (Lange; Koch, 2010). Por essa razão, estudar os fundamentos de compiladores e linguagens de máquina é essencial para desmistificar os processos computacionais e otimizar o desempenho de software em contextos mais complexos. Este trabalho tem como objetivo geral proporcionar a compreensão do processo de compilação e execução por meio da criação de uma linguagem de programação compilada para a Java Virtual Machine, detalhando as etapas essenciais envolvidas. Além disso, busca-se exemplificar a aplicação prática dessas etapas em um projeto que aprofunde a conexão entre os níveis de abstração da computação e o funcionamento interno da JVM.

Entre os trabalhos correlatos que inspiram e fundamentam a presente pesquisa, destaca-se o projeto "Lenguaje Algorítmico sobre la Máquina Virtual de Java", desenvolvido por Ruiz (2017). Este trabalho apresenta o desenvolvimento de um compilador para a linguagem LAG, direcionada à execução na JVM, com o objetivo de oferecer uma ferramenta prática e educacional para demonstrar conceitos de compilação. O projeto utiliza o ANTLR para processar o código da LAG e gerar bytecodes JVM a partir de representações intermediárias em Jasmin, o que ilustra a viabilidade de linguagens simplificadas na plataforma. Essa abordagem demonstra como conceitos de análise léxica e sintática podem ser aplicados em um ambiente de aprendizado.

Outro estudo relevante é o "C Compiler Targeting the Java Virtual Machine", Pien (1998), que explora a criação de um compilador para um subconjunto da linguagem C direcionado à JVM. O projeto destaca os desafios de adaptar uma linguagem procedural para uma máquina orientada a objetos, utilizando uma classe fictícia para encapsular variáveis e funções. Essa solução permite a execução de programas C na JVM, promovendo portabilidade e eficiência, e demonstra a flexibilidade da máquina virtual em suportar diferentes paradigmas de programação.

Por fim, o artigo "Sulong: Execution of LLVM-Based Languages on the JVM", de Rigger, Grimmer e Mössenböck (2016), aborda a criação de um interpretador para LLVM IR, permitindo a execução de linguagens como C, C++ e Fortran na JVM. O Sulong utiliza o Truffle e o compilador Graal para realizar otimizações em tempo de execução, promovendo integração entre linguagens e garantindo segurança de memória. Este trabalho

evidencia o potencial da JVM como plataforma para execução de linguagens de baixo nível, além de destacar sua eficiência e robustez.

Com relação aos objetivos específicos deste trabalho, consistem em: compreender os princípios teóricos das linguagens de programação e seus elementos constituintes; examinar as ferramentas e metodologias disponíveis para o desenvolvimento de linguagens de programação; desenvolver um projeto prático, utilizando o Antlr4 e a JVM como base, para criar uma nova linguagem de programação; analisar os desafios encontrados durante o processo de criação da linguagem, bem como as soluções adotadas.

2 MATERIAIS E MÉTODOS

Este projeto configura-se como uma pesquisa aplicada, de base tecnológica e natureza explicativa. Seu objetivo central consiste em proporcionar uma compreensão prática dos processos de compilação e execução, mediante o desenvolvimento de uma linguagem de programação compilada direcionada à Java Virtual Machine (JVM). A linguagem proposta apresenta-se como funcional, permitindo que códigos escritos em sua sintaxe sejam compilados e gerem programas executáveis na JVM.

O desenvolvimento do compilador seguiu três etapas principais, inter-relacionadas: (i) a definição da gramática da linguagem; (ii) a análise do código-fonte, realizada por meio das regras estabelecidas na gramática; e (iii) a geração do *bytecode* correspondente às instruções descritas no código-fonte da linguagem desenvolvida.

Para viabilizar essas etapas, foram utilizadas ferramentas e bibliotecas consolidadas no domínio, destacando-se o ANTLR4, empregado na definição e interpretação da gramática, e a biblioteca ASM, utilizada para a geração do *bytecode* compatível com a JVM. Inicialmente, realiza-se a criação da gramática, cuja abordagem detalhada encontra-se na subseção a seguir.

2.1 DEFINIÇÃO DA GRAMÁTICA DE LINGUAGEM

Nessa etapa foi definida a estrutura sintática da nova linguagem, que foi dado o nome de 'Simple Lang'. Esboçou-se então um rascunho de como a linguagem deveria ser, através da escrita de um código-fonte de exemplo, contendo algumas operações básicas almejadas para desenvolvimento. Definiu-se o corpo do programa, sendo uma classe, assim como no Java, para simplificar o entendimento e fazer um paralelo com a lingua-

gem Java, a qual um dos objetivos é ser compatível. Logo após, definiu-se o nome da classe, tendo em seu corpo a definição de três blocos principais: o bloco 'var': para definição de variáveis de escopo da classe, o bloco 'methods': para os métodos da classe, e o bloco 'init': que seria o código onde começa a ser executada a classe, nesse caso, comparável ao construtor de uma classe Java. A Figura 1 demonstra o código ilustrado.

Figura 1 - Código-fonte inicial da linguagem

```
class Hello {
    var {
        string nome = "Matheus";
        int idade = 23;
        float salario = 2000.50;
    }

    methods {
        getAnosParaAposentar(): int {
            return 65 - idade;
        }
    }

    # Método construtor do objeto
    init {
        int anos = getAnosParaAposentar();
        print("Olá " + nome + ", faltam " + anos + " para sua aposentadoria.");
    }
}
```

Fonte: Elaborado pelo autor.

A partir desse protótipo de estrutura da linguagem, vem o processo de tradução para uma gramática válida de ANTLR4. Para isso, foi necessária a criação de um novo projeto Java/Maven, que inclui em suas dependências a biblioteca ANTLR4 em sua versão 4.13.2, e as bibliotecas ASM e ASM-UTIL, ambas na versão 9.4. O projeto utilizou o Java 21 em conjunto com o Maven na versão 3.9.9. Todo o desenvolvimento foi realizado utilizando a IDE IntelliJ IDEA na versão 2024.3.

A definição da gramática foi feita por meio de um arquivo com extensão '.g4', nesse caso criado como 'SimpleLang.g4'. Esse arquivo segue uma estrutura pré-definida pela ferramenta, onde é descrita uma árvore de gramática em forma de código, conforme a Figura 2.

Figura 2 - Definição inicial da gramática

```
SimpleLang.g4
1  grammar SimpleLang;
2
3  // Parser Rules
4
5  program        : classDeclaration+ ;
6  classDeclaration
7                 : CLASS IDENTIFIER LBRACE varSection methodsSection initSection RBRACE ;
```

Fonte: Elaborado pelo autor.

O arquivo começa com a palavra-chave 'grammar', seguida do nome da linguagem. A gramática tem estrutura de chave-valor, em que cada valor pode apontar para outras chaves, formando uma árvore. Na Figura 2, o programa inicia com uma declaração de classe (*classDeclaration*), definida logo abaixo como a palavra-reservada 'class', seguida de um identificador e um bloco entre chaves, com três seções: variáveis, métodos e início. Todos os tokens nessa declaração são chaves que apontam para definições no próprio arquivo. Essa é a lógica básica da definição de gramática com o ANTLR4.

Antes de se prosseguir com o restante da linguagem, definiu-se o conjunto de todas as palavras-reservadas, símbolos e demais tokens que serão válidos na gramática. A Figura 3 mostra a definição de alguns desses símbolos, como chaves para blocos de código, parênteses para declarações e chamadas de métodos, operadores aritméticos (soma, subtração, multiplicação e divisão), lógicos ('and' e 'or') e outros símbolos essenciais.

Figura 3 - Palavras-reservadas e símbolos

```
96   // Palavras-chave
97   CLASS        : 'class' ;
98   VAR          : 'var' ;
99   METHODS      : 'methods' ;
100  INIT         : 'init' ;
101  RETURN       : 'return' ;
102  PRINT        : 'print' ;
103  READ         : 'read' ;
104  IF           : 'if' ;
105  ELSE         : 'else' ;
106  WHILE        : 'while' ;
107  STRING_TYPE  : 'string' ;
108  INT_TYPE     : 'int' ;
109  FLOAT_TYPE   : 'float' ;
110  VOID_TYPE    : 'void' ;
```

```
112  // Símbolos e operadores
113  LBRACE       : '{' ;
114  RBRACE       : '}' ;
115  LPAREN       : '(' ;
116  RPAREN       : ')' ;
117  COLON        : ':' ;
118  SEMICOLON    : ';' ;
119  COMMA        : ',' ;
120  ASSIGN       : '=' ;
121  PLUS         : '+' ;
```

Fonte: Elaborado pelo autor.

Por fim, são definidos os tokens mais complexos, que em vez de serem valores fixos, são, na verdade, regras baseadas em regex. A Figura 4 demonstra esses tokens.

Figura 4 - Demais tokens

```
134  // Outros tokens
135  COMMENT       : '#' ~[\r\n]* -> skip;        // Comentários iniciados com '#' são ignorados
136  BLOCK_COMMENT : '##' .*? '##' -> skip;      // Comentários de bloco entre '##' ... '##'
137  IDENTIFIER    : [a-zA-Z_] [a-zA-Z_0-9]* ;
138  STRING        : '"' .*? '"' ;
139  INT           : [0-9]+ ;
140  FLOAT         : [0-9]+ '.' [0-9]+ ;
141
142  WHITESPACE    : [ \t\r\n]+ -> skip ;         // Espaços em branco, tabs, novas linhas ignorados
```

Fonte: Elaborado pelo autor.

A definição de comentários e espaços em branco utiliza a regra

'skip', que ignora esses trechos no código-fonte. O token 'IDENTIFIER', mostrado na Figura 2, representa identificadores como variáveis ou nomes de métodos, permitindo letras, números e o caractere *underline*. A figura também traz os tipos *string*, *int* e *float*.

Com base nisso, o restante da gramática foi desenvolvido conforme demonstra a Figura 5 e algumas dessas regras serão detalhadas a seguir.

Figura 5 - Partes principais da gramática

```
program       : classDeclaration+ ;
classDeclaration
              : CLASS IDENTIFIER LBRACE varSection methodsSection initSection RBRACE ;

varSection    : VAR LBRACE varDeclaration* RBRACE ;
varDeclaration
              : type IDENTIFIER (ASSIGN expression)? SEMICOLON ;

methodsSection
              : METHODS LBRACE methodDeclaration* RBRACE ;
methodDeclaration
              : IDENTIFIER LPAREN parameterList? RPAREN COLON type block ;

parameterList : parameter (COMMA parameter)* ;
parameter     : type IDENTIFIER ;

initSection   : INIT LBRACE statement* RBRACE ;

block         : LBRACE statement* RBRACE ;

statement     : varDeclaration
              | methodCall SEMICOLON
              | assignment SEMICOLON
              | ifStatement
              | whileStatement
              | returnStatement
              | printStatement
              | readStatement ;

printStatement: PRINT LPAREN expression RPAREN SEMICOLON ;

readStatement : READ LPAREN IDENTIFIER RPAREN SEMICOLON ;

ifStatement   : IF LPAREN expression RPAREN block (ELSE block)? ;
whileStatement
              : WHILE LPAREN expression RPAREN block ;

returnStatement
              : RETURN expression SEMICOLON ;

assignment    : IDENTIFIER ASSIGN expression ;
methodCall    : IDENTIFIER LPAREN argumentList? RPAREN ;
argumentList  : expression (COMMA expression)* ;
```

Fonte: Elaborado pelo autor.

2.2 LEITURA DO CÓDIGO-FONTE COM O PARSER DO ANTLR4

Com a gramática definida e o plugin do ANTLR4 para Maven, são geradas classes Java que interpretam o código-fonte da nova linguagem. A partir da classe principal, o caminho do arquivo-fonte é recebido como parâmetro, lido do sistema e processado pelas classes *Lexer* e *Parser*. O parser, então, gera a árvore do programa, como mostrado na Figura 6.

Figura 6 - Código da classe principal do compilador

```
15  public static void main(String[] args) throws Exception {   matheus.dimer
16      if (args.length ≠ 1) {
17          System.err.println("Uso: java Main <caminho_do_arquivo>");
18          System.exit( status: 1);
19      }
20      String filePath = args[0];
21      CharStream input = CharStreams.fromFileName(filePath);
22
23      SimpleLangLexer lexer = new SimpleLangLexer(input);
24      CommonTokenStream tokens = new CommonTokenStream(lexer);
25      SimpleLangParser parser = new SimpleLangParser(tokens);
26
27      SimpleLangParser.ProgramContext tree = parser.program();
```

Fonte: Elaborado pelo autor.

Desta forma, a ferramenta, ao analisar o código-fonte, posteriormente, irá interpretar as suas estruturas e transformá-las em uma árvore sintática, que será utilizada pelo compilador desenvolvido, a fim de transformar os comandos do programador em instruções da JVM. Ao passar essa árvore para o método 'println' do Java, será impressa a representação dessa árvore em forma de texto, que está presente na Figura 7, demonstrando como funciona esse processo de *parse*.

Figura 7 - Código-fonte com os rótulos da gramática

```
(program
  (classDeclaration class Hello {
    (varSection var {
      (varDeclaration (type string) nome = (expression (literal "Matheus")) ;)
      (varDeclaration (type int) idade = (expression (literal 23)) ;)
      (varDeclaration (type float) salario = (expression (literal 2000.50)) ;)
    })

    (methodsSection methods {
      (methodDeclaration getAnosParaAposentar ( ) : (type int) (block {
        (statement (returnStatement return (expression (numericExpression (operand 65) - (operand
idade))) ;))
      }))
    })

    (initSection init {
      (statement (varDeclaration (type int) anos = (expression (methodCall getAnosParaAposentar ( )))
;))
      (statement (printStatement print (
        (expression (stringConcatenation
          "Olá " + nome + (literal ", faltam ") + anos + (literal " para sua aposentadoria.")
        ))
      ;))
    })
  })
)
 go(f, seed, [])
}
```

Fonte: Elaborado pelo autor.

Com a árvore sintática formada, o compilador pode percorrê-la para reconhecer instruções, dados, tipos e contexto, viabilizando a geração do bytecode, tema do próximo tópico.

Essa navegação é feita por meio do padrão de projeto *visitor*, comum em Java, onde uma classe visita cada nó da árvore com métodos

específicos para tratar suas ramificações. A Figura 8 mostra uma implementação básica desse *visitor*.

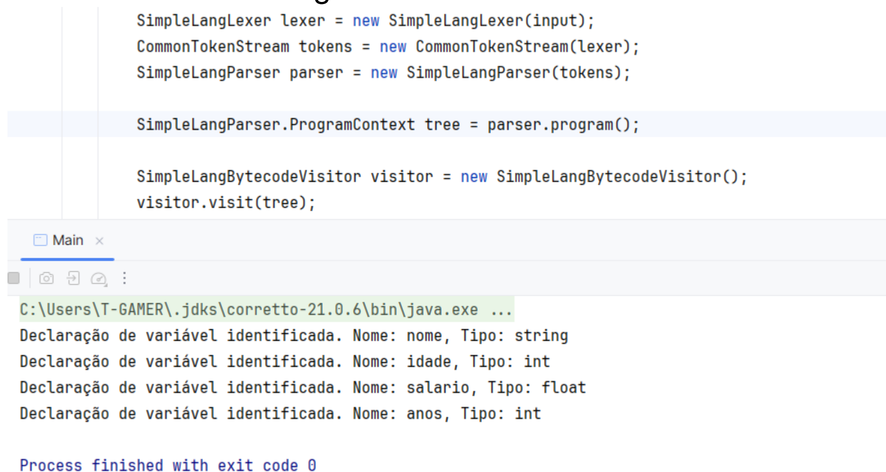
Figura 8 - Início da implementação do *visitor*

```
public class SimpleLangBytecodeVisitor extends SimpleLangBaseVisitor<Void> {  3 usages  new *

    @Override  1 usage  new *
    public Void visitVarDeclaration(SimpleLangParser.VarDeclarationContext ctx) {
        String varName = ctx.IDENTIFIER().getText();
        String varType = ctx.type().getText();

        System.out.println("Declaração de variável identificada. Nome: " + varName + ", Tipo: " + varType);

        return null;
    }
}
```

Fonte: Elaborado pelo autor.

O *visitor* estende a classe 'SimpleLangBaseVisitor', gerada pelo ANTLR4, que possui um método *visit* para cada ramificação da gramática. Na Figura 8, é mostrado o método para visitar uma declaração de variável. Cada método recebe um contexto, que contém as informações do nó, como nome, tipo e possível expressão de atribuição. Para executá-lo, basta instanciá-lo na classe 'Main' e passar a árvore gerada, conforme mostra a Figura 9.

Figura 9 - Uso do *visitor*



Fonte: Elaborado pelo autor.

Ao executar o programa, o compilador consegue identificar todas as declarações de variáveis contidas no código, conforme a saída contida na Figura 9. A partir desse ponto, a implementação do compilador consiste em tratar cada um dos nós da gramática, por meio de seus métodos de visitação correspondentes, retirando as informações necessárias do contexto para transformá-las em *bytecode*, que será abordado na próxima seção.

2.3 ESCRITA DE BYTECODE COM O FRAMEWORK ASM

O passo final de implementação do compilador se dá pela escrita de *bytecode*, onde foi utilizado o framework ASM do Java. A sua classe 'org.objectweb.asm.ClassWriter' possui um conjunto de métodos que fornece todo o suporte necessário para a escrita de instruções da JVM em um arquivo de classe com extensão '.class', o qual é executado pela máquina virtual. Para utilizá-lo, primeiro é necessário instanciá-lo dentro da classe de *visitor* a qual vem sendo trabalhada anteriormente, conforme a Figura 10.

Figura 10 - Uso do ClassWriter

```
10  public class SimpleLangBytecodeVisitor extends SimpleLangBaseVisitor<Void> {  3 usages  new *
11
12      private final ClassWriter classWriter;  4 usages
13      private final String className;  2 usages
14
15      public SimpleLangBytecodeVisitor(String className) {  1 usage  new *
16          this.className = className;
17          this.classWriter = new ClassWriter( flags: ClassWriter.COMPUTE_FRAMES | ClassWriter.COMPUTE_MAXS);
18      }
19
20      public byte[] getBytecode() { return classWriter.toByteArray(); }
23
24      @Override  1 usage  new *
25      public Void visitClassDeclaration(SimpleLangParser.ClassDeclarationContext ctx) {
26          // Criação da classe com ASM
27          classWriter.visit(V1_8, ACC_PUBLIC, className,  signature: null,  superName: "java/lang/Object",  interfaces: null);
28
29          visit(ctx.varSection()); // Bloco var
30          visit(ctx.methodsSection()); // Bloco methods
31          visit(ctx.initSection()); // Método init
32
33          classWriter.visitEnd();
34          return null;
35      }
36  }
```

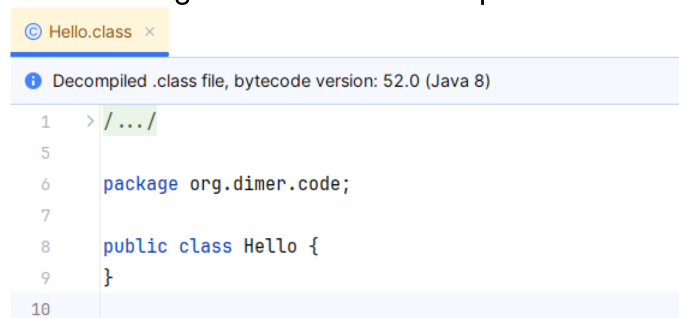
Fonte: Elaborado pelo autor.

Após a instanciação no construtor, a implementação do compilador inicia-se com o método visitClassDeclaration, sendo ele o nó superior da gramática de onde partirão todas as outras ramificações. Nele é constituído o cabeçalho da classe compilada, na chamada do método 'visit' do 'classWriter', contido na linha 27 da Figura 10. Nessa chamada, são passados alguns argumentos que definem as características dessa classe, cabendo destacar entre eles: a versão da classe, que nesse caso será referente ao Java 1.8; o modificador de acesso, nesse caso indicando que é uma classe pública; e o 'superName', que é a classe que se está sendo estendida (na JVM, todas as classes são herdeiras da classe 'Object').

Ainda na Figura 10, após a definição do cabeçalho da classe, há a invocação do método 'visit' para as três seções principais do programa conforme a gramática definida no projeto, que irá resultar na compilação posteriormente dessas seções na ordem apresentada, a partir da implementação dos métodos visitantes correspondentes a cada uma dessas se-

ções e seus filhos, conforme o desenvolvimento do compilador. Ao final, ao realizar a invocação 'classWriter.visitEnd()', é marcado o fim da escrita da classe, e a partir do método 'getBytecode' é recuperado todo o *bytecode* gerado em forma de uma lista de bytes, que será escrita em um arquivo, resultando em uma classe compilada. Ao abrir o arquivo gerado com a IDE, é possível ver a classe gerada descompilada como se fosse uma classe Java (embora tenha sido gerada a partir da nova linguagem), mostrando o esqueleto de uma classe vazia conforme a Figura 11.

Figura 11 - Classe compilada



Fonte: Elaborado pelo autor.

Na seção 2.4 serão abordadas a implementação de algumas das principais estruturas da linguagem. Devido à complexidade, não será possível discorrer sobre todas, mas a implementação completa poderá ser vista no repositório público disponibilizado com o código-fonte do compilador.

2.4 IMPLEMENTAÇÃO DAS ROTINAS DA LINGUAGEM

A implementação da seção de variáveis foi realizada por meio do método visitVarDeclaration, responsável por compilar as variáveis identificadas como atributos da classe, conforme ilustrado na Figura 12. A instrução 'visitField' é utilizada para criar um novo atributo na classe, sendo necessário passar o modificador de acesso, que para essa linguagem será fixo como privado, o nome da variável, e seu *descriptor*, que é a forma com que a JVM representa um tipo de dado. Para isso, é feita a conversão da tipagem da linguagem para o seu respectivo *descriptor*, com a ajuda do método 'typeToDescriptor'.

Figura 12 - Compilação de variáveis

```
41      @Override  1 usage  new *
42      public Void visitVarDeclaration(SimpleLangParser.VarDeclarationContext ctx) {
43          String varName = ctx.IDENTIFIER().getText();
44          String varType = ctx.type().getText();
45          String descriptor = typeToDescriptor(varType);
46
47          classWriter.visitField(ACC_PRIVATE, varName, descriptor, signature: null, value: null).visitEnd();
48
49          // Adiciona a lista de variáveis de classe para ter seu valor preenchido no bloco do construtor
50          var value  = ctx.expression() ≠ null ? getLiteralValue(ctx.expression().literal()) : null;
51          classVariables.put(varName, new Variable(varName, varType, value));
52
53          return null;
54      }
55
56      /**
57       * Converte a tipagem da linguagem para a tipagem da JVM
58       */
59      private String typeToDescriptor(String type) {  1 usage  new *
60          return switch (type) {
61              case "int" → "I";
62              case "float" → "F";
63              case "string" → "Ljava/lang/String;";
64              case "void" → "V";
65              default → throw new IllegalArgumentException("Tipo desconhecido: " + type);
66          };
67      }
```

Fonte: Elaborado pelo autor.

Ao executar o compilador e visualizar o *bytecode* gerado (Figura 13), nota-se que não há distinção entre variáveis de classe e locais — por isso, a variável ’anos’ foi tratada como atributo da classe. As atribuições ainda não ocorrem, pois, segundo a JVM, devem ser feitas no construtor, que será implementado posteriormente. Para isso, as variáveis e seus valores são armazenados em um Map chamado ’classVariables’, para serem usados posteriormente.

Figura 13 - Variáveis compiladas

```
1  public class org/dimer/code/Hello {
2
3      private Ljava/lang/String; nome
4
5      private I idade
6
7      private F salario
8
9      private I anos
10 }
```

Fonte: Elaborado pelo autor.

Para a implementação do construtor, foi implementado o método ’visitInitSection’ conforme a Figura 14. Utilizando o método ’visitMethod’ do objeto ’classWriter’, é criado o método ’init’ da classe (construtor), sendo ele público, e conforme seu *descriptor*, sem parâmetros com retorno *void*.

Figura 14 - Compilação do método construtor

```
58  @Override 1 usage new *
59  public Void visitInitSection(SimpleLangParser.InitSectionContext ctx) {
60      currentMethod = classWriter.visitMethod(ACC_PUBLIC, name: "<init>", descriptor: "()V", signature: null, exceptions: null);
61      currentMethod.visitCode();
62
63      currentMethod.visitVarInsn(ALOAD, varIndex: 0);
64      currentMethod.visitMethodInsn(INVOKESPECIAL, owner: "java/lang/Object", name: "<init>", descriptor: "()V", isInterface: false);
65
66      // Inicializa os valores das variáveis de classe caso existam
67      for (Variable classVariable : classVariables.values()) {
68          if (classVariable.value() == null) {
69              continue;
70          }
71
72          currentMethod.visitVarInsn(ALOAD, varIndex: 0); // Carrega o this
73          currentMethod.visitLdcInsn(classVariable.value()); // Load do valor
74          currentMethod.visitFieldInsn(PUTFIELD, className, classVariable.name(), typeToDescriptor(classVariable.type()));
75      }
76
77      // Passa por todos os comandos do bloco init
78      for (SimpleLangParser.StatementContext statementContext : ctx.statement()) {
79          visit(statementContext);
80      }
81
82      currentMethod.visitInsn(RETURN);
83      currentMethod.visitMaxs( maxStack: 0, maxLocals: 0); // Será calculado automaticamente pelo ASM
84      currentMethod.visitEnd();
85      currentMethod = null;
86
87      return null;
88  }
```

Fonte: Elaborado pelo autor.

A Figura 15 demonstra o *bytecode* gerado pelo compilador nesse ponto. Para que o construtor funcione, a JVM exige que seja invocado também o construtor da classe pai 'Object', que é feito por meio das instruções 'ALOAD 0' e 'INVOKESPECIAL java/lang/Object.<init> ()V'. ALOAD é a instrução responsável por carregar o ponteiro de um objeto na pilha da JVM, sendo zero o índice que contém a referência para o próprio objeto (*this*). Portanto, sempre que for necessário invocar algo (método ou atributo) que esteja dentro da própria classe, deverá primeiro ser carregado o *this*.

Figura 15 - Bytecode do método construtor

```
public class org/dimer/code/Hello {
  private Ljava/lang/String; nome
  private I idade
  private F salario
  private I anos

  // access flags 0x1
  public <init>()V
    ALOAD 0
    INVOKESPECIAL java/lang/Object.<init> ()V
    ALOAD 0
    LDC 23
    PUTFIELD org/dimer/code/Hello.idade : I
    ALOAD 0
    LDC 2000.5
    PUTFIELD org/dimer/code/Hello.salario : F
    ALOAD 0
    LDC "Matheus"
    PUTFIELD org/dimer/code/Hello.nome : Ljava/lang/String;
    RETURN
    MAXSTACK = 2
    MAXLOCALS = 1
}
```

Fonte: Elaborado pelo autor.

Como mostrado na Figura 15, o valor constante vinte e três é carregado com a instrução LDC e, em seguida, atribuído ao atributo *idade* com a instrução PUTFIELD, que consome o *this* e o valor da pilha. O mesmo ocorre para outras variáveis, mudando apenas os valores. Na JVM, todos os argumentos são passados pela pilha, seguindo a ordem exigida por cada

instrução, com o topo sendo o último argumento.

Esse exemplo ajuda a compreender a estrutura básica de uma classe em *bytecode*: tudo se resume a manipular valores na pilha e invocar instruções para reproduzir o comportamento do código-fonte da nova linguagem. Para subtrair dois valores e armazenar o resultado, por exemplo, é preciso carregá-los na pilha, aplicar a instrução ISUB e usar ISTORE para guardar o valor, como ilustrado na Figura 16.

Figura 16 - Bytecode de subtração com atribuição de variável

```
init {
   int anos = 65 - idade;
}
  public <init>()V
    [...]
    LDC 65
    ALOAD 0
    GETFIELD org/dimer/code/Hello.idade : I
    ISUB    Subtrai dois números inteiros e armazena o resultado no topo da pilha
    ISTORE 1    Armazena o valor do topo da pilha no índice 1 (variável local)
    RETURN
    MAXSTACK = 2
    MAXLOCALS = 2
```

Fonte: Elaborado pelo autor.

Na JVM, variáveis locais não existem normalmente após a compilação — elas são acessadas por índices via instruções *store* e *load*, começando do zero (reservado para *this*). Os demais índices são usados para variáveis locais, isolados por método. Por isso, o compilador precisa mapear nomes de variáveis aos seus índices dentro de cada método. Para isso, foi criado o *LocalVariableManager*, que gerencia essa associação e, com o uso de uma pilha de controladores, isola os escopos entre métodos (ver Figura 17).

Figura 17 - Implementação de variáveis locais

```
private final Stack<LocalVariableManager> localVariablesStack = new Stack<>();

[...]

@Override
public Void visitVarDeclaration(SimpleLangParser.VarDeclarationContext ctx) {
    String varName = ctx.IDENTIFIER().getText();
    String varType = ctx.type().getText();
    String descriptor = typeToDescriptor(varType);

    if (currentMethod == null) { // Significa que é variável da classe
        [...] // Variáveis de classe
    } else {
        if (localVariablesStack.isEmpty()) {
            localVariablesStack.push(new LocalVariableManager());
        }

        LocalVariableManager manager = localVariablesStack.peek();
        int varIndex = manager.allocate(new Variable(varName, varType));

        if (ctx.expression() != null) {
            visit(ctx.expression()); // Executa a expressão que determina o valor da variável

            String type = determineTypeOfExpression(ctx.expression());
            [...]
            currentMethod.visitVarInsn(determineStoreCommand(type), varIndex);
        }
    }

    return null;
}
```

Fonte: Elaborado pelo autor.

Após alocar a variável, seu valor é armazenado no índice correspondente com instruções específicas: ISTORE (inteiros), FSTORE (*float*) e ASTORE (objetos). Para recuperar uma variável local, usa-se seu índice com a instrução de *load* adequada (ILOAD, FLOAD etc.). Se não estiver no escopo local, pode ser um atributo da classe, acessado com GETFIELD. A Figura 18 mostra essa implementação.

Figura 18 - Recuperação de valores de variáveis

```
    private void loadVariable(ParserRuleContext ctx, String varName) {
        if (!localVariablesStack.isEmpty()) {
            var manager = localVariablesStack.peek();
            var variable = manager.load(varName);

            if (variable != null) {
                currentMethod.visitVarInsn(determineLoadCommand(variable.type()), variable.index());
                return;
            }
        }

        loadClassVariable(ctx, varName);
    }

    private void loadClassVariable(ParserRuleContext ctx, String varName) {
        currentMethod.visitVarInsn(ALOAD, 0); // Carrega 'this'
        currentMethod.visitFieldInsn(GETFIELD, className, varName, determineDescriptor(ctx,
varName)); // Pega o atributo
    }
```

Fonte: Elaborado pelo autor.

Com a implementação funcional de armazenamento e recuperação de valores das variáveis, foram desenvolvidas as demais funcionalidades da linguagem, sendo algumas descritas a seguir.

2.4.1 IMPLEMENTAÇÃO DA FUNÇÃO PRINT

Para imprimir valores na tela, é utilizado o método 'println' disponível dentro do objeto estático 'System.out' disponível globalmente na JVM. Para isso, utiliza-se das instruções 'GETSTATIC' para recuperar o objeto estático que contém o método de imprimir, conforme mostra a Figura 19.

Figura 19 - Implementação da função *print*

```
@Override
public Void visitPrintStatement(SimpleLangParser.PrintStatementContext ctx) {
    currentMethod.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
    visit(ctx.expression());

    String type = determineTypeOfExpression(ctx.expression());
    String descriptor = Type.getMethodDescriptor(Type.VOID_TYPE, Type.getType(typeToDescriptor(type)));

    currentMethod.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", descriptor, false);
    return null;
}
```

Fonte: Elaborado pelo autor.

Logo após, a expressão contida no método print é compilada, podendo representar uma variável, uma expressão aritmética, uma chamada de método ou uma concatenação de strings. O retorno dessa expressão,

então, é colocado no topo da pilha, e assim se prossegue a chamada do método ’println’ através da instrução INVOKEVIRTUAL.

2.4.2 IMPLEMENTAÇÃO DO LAÇO DE REPETIÇÃO WHILE

O *while* consiste em um *loop* baseado em uma condição. Para compilá-lo, são usadas as instruções de *if* juntamente com três *labels*, o primeiro marcando onde está a condicional, a segunda para o bloco de código do *while*, e a última para o final do *loop*, como apresenta a Figura 20.

Figura 20 - Implementação da função *while*

```
578         @Override  1 usage  matheus.dimer
579         public Void visitWhileStatement(SimpleLangParser.WhileStatementContext ctx) {
580             Label conditionLabel = new Label();
581             Label blockLabel = new Label();
582             Label endLabel = new Label();
583
584             int instruction = determineComparisonInstruction(ctx.expression());
585
586             // Marca o início do bloco da condição
587             currentMethod.visitLabel(conditionLabel);
588
589             if (ctx.expression().booleanExpression() != null) {
590                 visitBooleanExpression(ctx.expression().booleanExpression(), blockLabel, endLabel);
591             } else {
592                 visit(ctx.expression()); // Compila a expressão do while
593                 // Instrução de jump condicional baseado no retorno da expressão
594                 currentMethod.visitJumpInsn(instruction, blockLabel); // Caso true, executa o bloco dentro do while
595                 currentMethod.visitJumpInsn(GOTO, endLabel); // Caso false, vai pro final do while
596             }
597
598             currentMethod.visitLabel(blockLabel); // Início do bloco do while
599             visit(ctx.block());
600
601             // Ao acabar o bloco do while, jump de volta para o bloco da condição do while e executa dnv
602             currentMethod.visitJumpInsn(GOTO, conditionLabel);
603             currentMethod.visitLabel(endLabel);
604
605             return null;
606         }
```

Fonte: Elaborado pelo autor.

Começa-se então pela execução da condicional, usando uma instrução de *jump* condicional correspondente ao tipo de comparação. Caso seja verdadeira, vai para o bloco de código do *while*, caso falsa, pula para o *label* do fim do laço de repetição. Ao final do bloco de código de dentro do *while*, é adicionada a instrução de pulo de volta para o *label* da condicional, e assim se dá o ciclo de repetição, conforme o exemplo da Figura 21.

Figura 21 - Bytecode da função *while*

```
L0
FRAME FULL [org/dimer/code/Hello] []
 ALOAD 0
 GETFIELD org/dimer/code/Hello.nome : Ljava/lang/String;
 LDC ""
 INVOKEVIRTUAL java/lang/String.equals (Ljava/lang/Object;)Z
 IFNE L1   Label do bloco while
 GOTO L2   Label do fim do while
L1
FRAME SAME
 GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
 LDC "Por favor, digite seu nome: "
```

Fonte: Elaborado pelo autor.

No exemplo da Figura 21, o código original verifica se a variável 'nome' contém uma *string* vazia, utilizando-se do método 'equals', que por sua vez retorna zero para diferente e um para igual. Então, com a instrução 'IFNE', caso seja diferente de zero, irá pular para o *label* 'L1', que corresponde ao bloco interno do *while* onde irá ser requisitado o nome novamente.

2.4.3 IMPLEMENTAÇÃO DA CONDICIONAL IF

A implementação da instrução *if* segue os mesmos princípios lógicos aplicados no *while*, utilizando desvios condicionais e marcações por meio de *labels*, sendo eles o bloco de código executado caso a condição for verdadeira e o bloco caso falsa (*else*), que nesse caso é opcional. A Figura 22 demonstra um método de exemplo que utiliza o *if* e seu bytecode compilado.

Figura 22 - Exemplo de função *if* compilada

```
methods {
    anosParaAposentar(): int {
        int resultado;

        if (getIdade() > idadeAposentadoria) {
            resultado = 0;
        } else {
            resultado = idadeAposentadoria - getIdade();
        }

        return resultado;
    }
}
```

```
public anosParaAposentar()I
    ALOAD 0
    INVOKEVIRTUAL org/dimer/code/Hello.getIdade ()I
    ALOAD 0
    GETFIELD org/dimer/code/Hello.idadeAposentadoria : I
    IF_ICMPGT L0
    GOTO L1
   L0
   FRAME SAME
    LDC 0
    ISTORE 1
    GOTO L2
   L1
   FRAME SAME
    ALOAD 0
    GETFIELD org/dimer/code/Hello.idadeAposentadoria : I
    ALOAD 0
    INVOKEVIRTUAL org/dimer/code/Hello.getIdade ()I
    ISUB
    ISTORE 1
    GOTO L2
   L2
   FRAME APPEND [I]
    ILOAD 1
    IRETURN
    MAXSTACK = 2
    MAXLOCALS = 2
```

Fonte: Elaborado pelo autor.

São carregados na pilha os valores de idade e da variável idade de aposentadoria, em seguida, a comparação é realizada pela instrução 'IF_ICMPGT', que em inglês significa "int comparator greater than" traduzido como "comparador de inteiros maior que". Ela também é uma instrução de desvio condicional, caso seja verdadeira a comparação, irá desviar para o *label* 'L0', que contém a operação de armazenamento (*store*) do valor zero na variável resultado. Em seguida, caso o desvio não se concretize, há a instrução 'GOTO L1', que nada mais é que o desvio para o bloco *else* representado pelo bloco 'L1'.

3 DISCUSSÃO E RESULTADOS

O desenvolvimento da linguagem Simple Lang, direcionada à Java Virtual Machine (JVM), demonstrou a viabilidade de criação de uma linguagem de programação funcional e compatível com a plataforma proposta. A implementação atendeu integralmente aos objetivos, contemplando as principais funcionalidades previstas, tais como a declaração e inicialização de variáveis, a utilização de estruturas condicionais (*if*), a execução de laços de repetição (*while*) e a definição de métodos.

As funcionalidades essenciais foram implementadas com sucesso, permitindo que a linguagem oferecesse suporte à manipulação básica de variáveis e ao controle de fluxo de execução. O mecanismo de compilação, ao converter o código-fonte diretamente em *bytecode* por meio do framework ASM, garantiu a execução dos programas resultantes na JVM. A Figura 23 apresenta um exemplo de programa desenvolvido em Simple Lang, acompanhado de sua respectiva saída, demonstrando na prática as capacidades da linguagem criada.

Figura 23 - Resultado final

Código SimpleLang:

```
1 class Hello {
2     var {
3         string nome;
4         int idade;
5         int idadeAposentadoria = 60;
6         float salario;
7     }
8
9     methods {
10        getIdade(): int {
11            return idade;
12        }
13
14        anosParaAposentar(): int {
15            int resultado;
16            if (getIdade() > idadeAposentadoria) {
17                resultado = 0;
18            } else {
19                resultado = idadeAposentadoria - getIdade();
20            }
21            return resultado;
22        }
23
24        formatAposentadoria(): string {
25            int anos = anosParaAposentar();
26            if (anos == 0) {
27                return "Parabéns, você está aposentado.";
28            }
29            return "Anos para se aposentar: " + anos;
30        }
31    }
32
33    init {
34        print("Olá, digite seu nome: ");
35        read(nome);
36
37        while (nome == "") {
38            print("Por favor, digite seu nome: ");
39            read(nome);
40        }
41
42        print("Digite sua idade: ");
43        read(idade);
44        while (idade < 1 or idade > 120) {
45            print("Idade inválida");
46            print("Digite sua idade: ");
47            read(idade);
48        }
49
50        print("Olá " + nome + ", você tem " + idade + " anos de idade.");
51        if (idade < 18 or idade > 70) {
52            print("Você não trabalha.");
53        }
54        print("Cálculo complexo: " + (idade + 10.5 + (20 - 2)));
55        print(formatAposentadoria());
56    }
57 }
```

Resultado:

```
C:\Users\T-GAMER\.jdks\corretto-21.0.6\bin\java.exe ...
Olá, digite seu nome:

Por favor, digite seu nome:
Matheus
Digite sua idade:
800
Idade inválida
Digite sua idade:
17
Olá Matheus, você tem 17 anos de idade.
Você não trabalha.
Cálculo complexo: 45.5
Anos para se aposentar: 43

Process finished with exit code 0
```

Fonte: Elaborado pelo autor.

Durante o processo de desenvolvimento, a tradução da gramática da linguagem para o formato aceito pelo ANTLR4 constituiu um dos principais desafios enfrentados. Essa etapa foi fundamental para que a ferramenta pudesse processar corretamente a sintaxe da linguagem proposta, gerando a árvore sintática utilizada pelo compilador. A escolha do

framework ASM revelou-se adequada, ao oferecer recursos completos para a geração de *bytecode* compatível com as necessidades do projeto, permitindo o controle detalhado sobre o código gerado.

No tocante à comparação com trabalhos correlatos, observa-se que a Simple Lang apresenta avanços significativos em relação a projetos como o de Ruiz (2017) e Pien (1998). No projeto de Ruiz, a linguagem LAG é projetada como ferramenta educacional e depende do uso da ferramenta Jasmin como intermediária para a geração de bytecode, inserindo uma etapa adicional no processo de compilação. Em contraste, a Simple Lang realiza a geração de bytecode diretamente, eliminando intermediários e oferecendo maior controle sobre o processo.

De modo semelhante, o projeto de Pien adaptou a linguagem C para a JVM por meio do encapsulamento de variáveis globais e funções em classes fictícias, viabilizando a execução de código C em um ambiente orientado a objetos. A Simple Lang, por sua vez, foi projetada desde sua concepção para operar de forma integrada à JVM, proporcionando uma integração mais fluida e natural com a plataforma.

Um diferencial adicional da Simple Lang reside na possibilidade de utilização de classes escritas na linguagem dentro de projetos Java, o que evidencia sua integração com o ecossistema Java. Contudo, ainda não há suporte para o caminho inverso, ou seja, a utilização direta de classes Java em programas escritos em Simple Lang, configurando-se como uma limitação e uma oportunidade de aprimoramento futuro.

Além disso, a adoção do ANTLR4 para análise léxica e sintática permitiu maior eficiência na identificação e processamento das estruturas da linguagem, em comparação com projetos que empregam abordagens manuais ou ferramentas menos integradas.

Não obstante, foram identificadas limitações inerentes ao escopo do projeto, entre as quais se destacam a ausência de suporte a tipos de dados mais avançados, estruturas complexas como *arrays* e objetos, bem como a ausência de otimizações de compilação a nível de *bytecode*. Tais limitações foram consideradas decisões estratégicas, visando à manutenção de um escopo simplificado e à priorização das funcionalidades essenciais, de modo a garantir o alcance dos objetivos propostos e a clareza na demonstração dos conceitos abordados.

Com o objetivo de avaliar a performance da linguagem desenvolvida, foi realizado um benchmark comparativo entre um mesmo programa escrito em Simple Lang e em Java. Ambos executam operações simples

de entrada, saída e cálculos. As entradas foram automatizadas via redirecionamento de System.in, e cada versão foi executada 200 vezes. O programa em Java teve média de 214.868 nanossegundos, enquanto a versão em Simple Lang obteve 309.274 nanossegundos, indicando que o Java foi cerca de 30,5% mais rápido. Ainda assim, os resultados demonstram que a linguagem proposta gera *bytecode* funcional e competitivo, considerando seu caráter experimental.

Por fim, o trabalho estabeleceu uma base sólida para futuras explorações, permitindo a expansão das funcionalidades da linguagem sem comprometer a estabilidade e a funcionalidade do compilador desenvolvido.

4 CONCLUSÃO

Este trabalho teve como objetivo geral proporcionar uma compreensão prática do processo de compilação e execução por meio do desenvolvimento de uma linguagem de programação compilada para a Java Virtual Machine (JVM). Para tal, foram detalhadas as etapas essenciais envolvidas na criação de uma linguagem, desde a definição da gramática até a geração de bytecode executável.

A partir dessa proposta, todas as fases que compõem um compilador foram exploradas de forma sistemática, possibilitando a visualização concreta do funcionamento interno da JVM. A linguagem desenvolvida, denominada Simple Lang, consolidou-se como uma ferramenta didática e técnica eficaz, permitindo a demonstração prática dos conceitos teóricos abordados ao longo do projeto.

No desenvolvimento da pesquisa, os objetivos específicos estabelecidos foram integralmente atingidos. Inicialmente, foi realizada a revisão dos princípios teóricos das linguagens de programação, proporcionando a base necessária para a construção da gramática e o entendimento das estruturas internas da linguagem proposta. Posteriormente, foram analisadas ferramentas e metodologias voltadas ao desenvolvimento de linguagens de programação, destacando-se o uso do ANTLR4, utilizado como gerador de analisadores léxicos e sintáticos, e do framework ASM, empregado na geração do *bytecode*.

Com a consolidação dessas definições, procedeu-se à implementação prática da linguagem Simple Lang, utilizando o ANTLR4 e a JVM como base tecnológica. Essa abordagem demonstrou que, mesmo com recursos limitados, é possível conceber uma linguagem funcional e integrada ao ecossistema da JVM. Durante a execução do projeto, foram também

analisados os desafios técnicos enfrentados, destacando-se a adequação da gramática às necessidades da linguagem, o controle de escopos e variáveis na geração do bytecode, bem como a conversão de estruturas lógicas e aritméticas para as instruções da JVM.

Embora os resultados obtidos tenham validado a proposta, algumas limitações foram identificadas. Destacam-se, entre elas, a ausência de suporte a estruturas mais avançadas, como *arrays* e objetos, bem como a inexistência de otimizações no processo de compilação, aspectos que poderão ser explorados em trabalhos futuros. Além disso, a integração entre Simple Lang e classes externas escritas em Java configura-se como uma oportunidade relevante de evolução, ampliando o potencial de aplicação da linguagem proposta.

Adicionalmente, recomenda-se a investigação de técnicas de otimização de *bytecode*, a reorganização das estruturas geradas e a eliminação de redundâncias, com vistas a aprimorar a eficiência dos programas compilados. A expansão da linguagem para suporte a estruturas de controle adicionais, como laços do tipo *for*, bem como a introdução de funcionalidades de importação de bibliotecas externas, também são caminhos promissores para o fortalecimento da Simple Lang.

Conclui-se, portanto, que o trabalho contribuiu significativamente para a compreensão prática do processo de compilação para a JVM, constituindo uma base sólida para estudos futuros e para o aprofundamento do conhecimento na área de desenvolvimento de compiladores e linguagens de programação.

REFERÊNCIAS

AHO, A. V.; SETHI, R.; ULLMAN, J. D. **Compilers: principles, techniques, and tools**. USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN 0201100886.

LANGE, H.; KOCH, A. **Architectures and Execution Models for Hardware/Software Compilation and Their System-Level Realization**. [S.l.: s.n.], 2010. v. 59. 1363-1377 p.

PIEN, J. **C Compiler Targeting the Java Virtual Machine C Compiler Targeting the Java Virtual Machine**. [s.n.], 1998. Acesso em: 16 de set. 2024. Disponível em: <https://digitalcommons.dartmouth.edu/cgi/viewcontent.cgi?article=1186&context=senior_theses>.

RAPP, W. **Fourth generation languages**. [S.l.: s.n.], 1985. v. 6. 38—40 p. ISSN 0745-1075.

RIGGER, M.; GRIMMER, M.; MÖSSENBÖCK, H. **Sulong - Execution of LLVM-based languages on the JVM position paper**. [s.n.], 2016. Acesso em: 19 de set. 2024. ISBN 9781450348379. Disponível em: <https://ssw.jku.at/General/Staff/ManuelRigger/ICOOOLPS16.pdf>.

RUIZ, D. A. R. **Lenguaje algoritmico sobre la maquina virtual de java**. [S.l.], 2017. Acesso em: 23 de ago. 2024. Disponível em: <http://catalogo.escuelaing.edu.co/cgi-bin/koha/opac-detail.pl?biblionumber=20756>.

SANATI-MEHRIZY, R.; MINAIE, A. **A New Role Of Assembly Language In Computer Engineering/Science Curriculum**. Nashville, Tennessee: ASEE Conferences, 2003.