Jarvas

Igor Masson Calille

Julio Cesar Barboza Filho

Lucas Costa Pessoa Graziano

Luis Felipe Almeida Beserra Matos

Pedro Alcantara Krivochein

Introdução:

Jarvas é uma linguagem de programação baseada em Java que busca facilitar seu uso. O primeiro objetivo é simplificar a sintaxe da linguagem Java ainda mantendo o idioma em inglês. Além disso, a sintaxe do Jarvas foi aprimorada para torná-la mais fácil de ler e escrever, reduzindo a quantidade de código necessário para realizar tarefas comuns. A proposta da criação da linguagem de programação vem do projeto proposto pelo professor Italo Santiago Vega, do quinto semestre de Ciência da Computação. O trabalho em questão se trata da projeção de uma linguagem de programação que forneça um código fonte equivalente em WebAssembly.

Tutorial da Linguagem

Para uma melhor compreensão da nossa linguagem Jarvas, decidimos iniciar com um tutorial abrangente. Nesse tutorial, abordaremos os conceitos básicos da linguagem, como sintaxe, convenções lógicas e operações aritméticas.

Acreditamos que começar com esses tópicos é fundamental para que seja possível compreender a lógica por trás da nossa linguagem e, assim, utilizá-la de forma mais eficiente.

Gramática da Linguagem

prog -> *instrução*

instrução -> *dec*; | *decIf*; | *decWhile*; | *exibir*; | ε

dec -> *tipo ident* | *tipo ident* = *expressão*

expressão -> *expAritmetica* | *integer* |*char* | *boolean* | *float* | *identifier*

decIf-> if (*cond*) {*instrução*}

decWhile -> while (*cond*) {*instrução*}

cond -> *ident opLogico integer* | *ident opLogico char*

exibir -> print (*ident*, *tipo*)

expAritmetica -> *termLogico* | (*termLogico*) | *termLogico opAritmetica expAritmetica* | *termLogico opAritmetica* (*expAritmetica*)

termLogico -> *ident* | *integer*

opAritmetica -> + | - | * | /

opLogico -> == | != | > | < | <= | >=

tipo -> int | float | char | bool

integer -> [0-9]+

float -> [0-9]+.[0-9]+

char -> '[A-Za-z_]'

boolean -> true | false

ident -> [A-Za-z_]+

Hello World

A melhor forma de apresentar a funcionalidade do código jarvas é mostrar a sintaxe através da escrita de algoritmos. Por conta disso, um exemplo introdutório clássico de qualquer linguagem de programação é imprimir "Hello World". Na linguagem de programação Jarvas, a forma mais simples para imprimir "Hello World" é:

```
print("Hello world!");
```

É possível executar o código diretamente no nível do módulo, sem a necessidade explícita de uma função main. Ao contrário de algumas linguagens de programação, como C ou Java, onde um programa começa sua execução a partir de uma função main, em Python, o código no nível superior do módulo é executado conforme é encontrado.

Quando você executa um arquivo em Jarvas, o interpretador começa a executar o código a partir do topo do arquivo e progride linha por linha. Isso significa que qualquer código que estiver fora de funções ou classes será executado imediatamente.

Exemplo de operação aritmética

Como o projeto prioriza as operações aritméticas, optamos por fornecer um exemplo simples que demonstre o uso dessas operações na linguagem Jarvas, visando facilitar a compreensão. A seguir, apresentamos um exemplo de uma operação aritmética básica realizada na linguagem Jarvas:

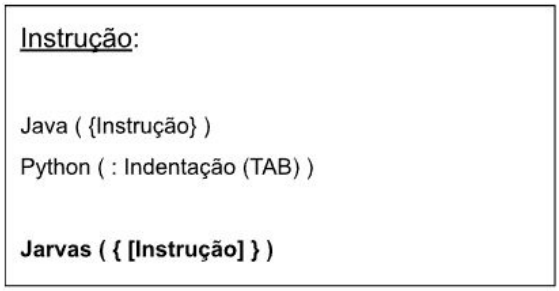
```
int a = 1;
int b = 2;

int c = a + b;

print(c, int);
```

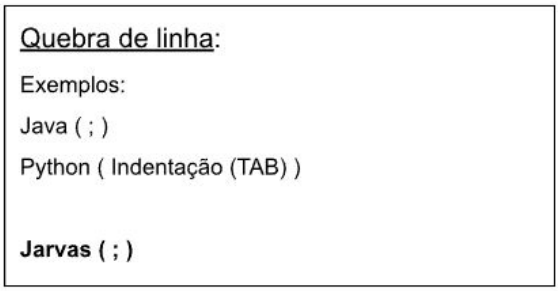
A saída desse código será:

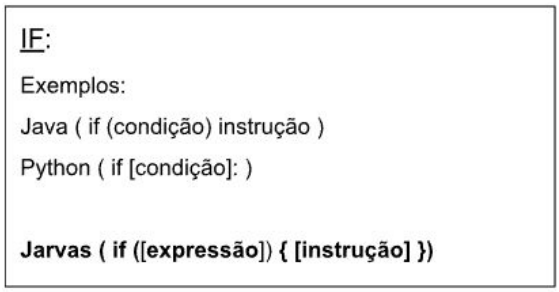
```
3
```

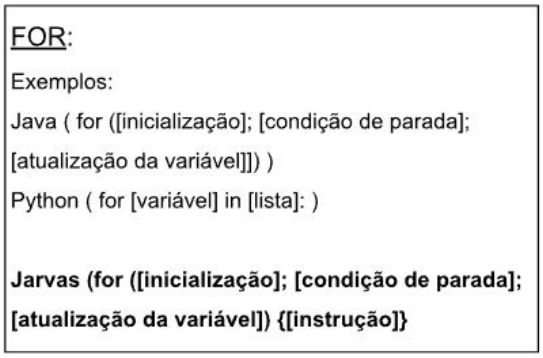
Sintaxe:

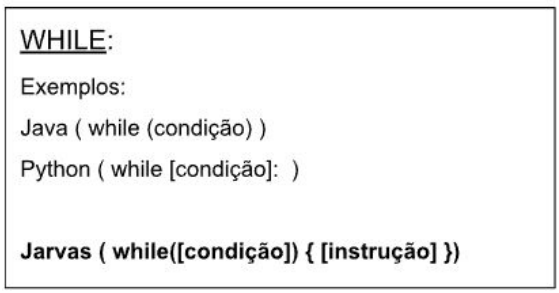
A imagens abaixo apresenta como a linguagem de programação Jarvas irá se comportar através de algumas instruções básicas de qualquer linguagem de programação.











É possível notar que existe muitas similaridades e algumas misturas relacionadas com a linguagem jarvas em relação as linguagens de programação Python e Java.

Manual de referência da linguagem

O manual de referência da linguagem foi inspirado no ***The GNU C Reference Manual*** e no apêndice A do livro ***Apêndice A do livro The C Programming Language***

Convenções Léxicas

Na programação de computadores, as convenções lexicais são as regras que definem como os caracteres são agrupados em tokens em uma linguagem de programação. Tokens são os blocos básicos de construção de um programa e são usados para representar palavras-chave, variáveis, constantes, operadores e outros elementos da linguagem. Na imagem abaixo, é possível ver as convenções léxicas da linguagem de programação Jarvas.

Palavras chaves	Delimitadores	Tipos	Atribuição	Identificadores	Operadores	Condicionais
if	:	int	=	IDENTIFIER	+	==
elif	(	float		INTEGER	-	!=
else	)	string		FLOAT	*	>
while	[	char		STRING	/	>=
for	]	boolean		CHAR		<
do	{			BOOLEAN		<=
return	}					
break	,					
class	.					

Tokens

Tokens são valores de identificação atribuídos a lexemas. Na linguagem Jarvas, existem sete tipos diferentes de classes de tokens: palavras chave, delimitadores, identificadores e literais, atribuição, operadores, condicionais, e de final do arquivo. Espaços em branco são desconsiderados pelo compilador.

Abaixo estão listados todos os tokens da linguagem Jarvas:

IF	ELIF	ELSE	WHILE	FOR	DO
RETURN	BREAK				
PAREN_L	PAREN_R	BRACE_L	BRACE_R	BRACKET_L	BRACKET_R
SEMICOLON	COMMA	DOT	DOUBLE_DOT		
TYPE	INTEGER	FLOAT	BOOLEAN	CHAR	
EQUAL					
PLUS	MINUS	TIMES	DIVIDE		
EQUALS	NOT_EQUAL	GREATER_THAN	LESS_THAN	GREATER_EQUAL	LESS_EQUAL
EOF					

Regex:

Expressões regulares (regex) são uma ferramenta extremamente poderosa e versátil para pesquisar, extrair e manipular texto de maneira eficiente. Elas oferecem uma ampla

gama de recursos e funcionalidades que permitem realizar tarefas complexas de processamento de texto com facilidade.

Na imagem abaixo, apresentamos alguns exemplos ilustrativos de como nosso regex foi desenvolvido para realizar essas operações textuais.

LEXEME	REGEX
if	\bif\b
elif	\belseif\b
else	\belse\b
while	\bwhile\b
for	\bfor\b
do	\bdo\b
return	\breturn\b
break	\bbreak\b
class	\bclass\b
:	:
(	(
)	)
[	[
]	]
{	{
}	}
,	,
.	.
;	;
int	\bint\b
float	\bfloat\b
string	\bstring\b
char	\bchar\b
boolean	\bboolean\b
IDENTIFIER	[a-zA-Z_$][a-zA-Z_$0-9]*

INTEGER	\d+
FLOAT	\d+\.\d+
STRING	\"([^\"]|\.)*\"
CHAR	\'([^\']|\.)\'
BOOLEAN	true|false
=	\b=\b
+	+
-	-
*	*
/	/
==	\b==\b
!=	\b!=\b
>	\b>\b
>=	\b>=\b
<	\b<\b
<=	\b<=\b

Identificadores

Um identificador é uma sequência de símbolos validos usados para identificar um elemento, seja ele uma função ou variável. O primeiro caractere pode ser uma letra maiúscula, minúscula, underscore ou dollar sign ($). Subsequente a essa primeira parte, é possível definir o mesmo padrão: uma ou mais letras maiúsculas, letras minúsculas, underscores ou dollar signs ($) ou dígitos.

Palavras chave

São palavras que são usadas com um propósito específico na linguagem Jarvas e que não podem ser usadas em outro lugar.

`if`	`else`
`while`	`for`

Tipos

Um tipo é um valor fixo. Na linguagem Jarvas os tipos são:

type : INTEGER | FLOAT | CHAR | BOOLEAN

Um tipo inteiro é formado por um ou mais dígitos podendo ser precedido ou não por um sinal de menos ou mais. Exemplo: +1, -1, 129

Um tipo float é formada por um ou mais dígitos, com um ponto marcando o início do número em decimal formado por um ou mais dígitos. Podendo ser precedida ou não por um sinal de menos ou mais. Exemplo: +1.3, -1.0, 129.333

Um tipo caractere é formado por um único caractere entre apóstrofos simples. Exemplo: 'O'

Um boolean é formado ou pelo símbolo ***true*** ou pelo símbolo ***false***

Operadores de soma

São respectivamente + e -

sumOperator : PLUS | MINUS

Operadores de multiplicação

São respectivamente * e /

multiplifierOperator : TIMES | DIVIDE

Operadores relacionais

São respectivamente ==, !=, >, <, >=, <=:

relationalOperator : EQUALS | NOT_EQUAL | GREATER_THAN | LESS_THAN | GREATER_EQUAL | LESS_EQUAL

Expressão inteira

integerExpression : INTEGER multiplifierOperator integerExpression | INTEGER sumOperator integerExpression | '('integerExpression')' | INTEGER

Expressão de float

floatExpression : FLOAT multiplifierOperator floatExpression | FLOAT sumOperator floatExpression | '('floatExpression')' | FLOAT

Expressão booleana

booleanExpression : BOOLEAN relationalOperator booleanExpression | '('BOOLEAN')' | BOOLEAN

Expressão de char

charExpression | '('charExpression')' | CHAR

Expressões

Uma expressão é uma combinação de valores, variáveis, operadores e funções que são avaliados para produzir um resultado.

expression : type | integerExpression | floatExpression | charExpression | stringExpression | booleanExpression

Atribuição

O conceito de atribuição refere-se à ação de atribuir ou associar um valor a uma variável em um programa de computador. É por meio das operações de atribuição que os valores são armazenados em variáveis para posterior uso e manipulação.

variableAssignment : | IDENTIFIER EQUAL expression | 'var' IDENTIFIER EQUAL expression

Declaração de variável

Uma declaração de variável é uma instrução em um programa de computador que reserva um espaço de memória para armazenar um valor e associa um nome a esse espaço de memória.

variableDeclaration : 'var' variableAssignment

Statements de seleção

Statements condicionais, também conhecidos como estruturas de controle condicional, são construções de programação que permitem ao programa tomar decisões com base em condições específicas.

ifStatement : IF '('condition')' '{' block '}'

Plano do projeto

Como plano de projeto da linguagem, utilizamos do cronograma fornecido e os requisitos transcritos pelo professor para realizarmos uma base de construção do projeto. Com o intuito de melhor organizar o desenvolvimento, realizamos uma separação do grupo de acordo os seguintes papéis:

- Gerente de projetos: Julio Cesar Barboza Filho
- Integrador de sistemas: Luis Felipe Almeida Beserra Matos
- Arquiteto de sistemas: Igor Masson Calille
- Testador: Lucas Costa Pessoa Filho
- Guru da linguagem: Pedro Alcantara Krivochein

Ao decorrer da criação da linguagem, encontramos questões e dificuldades que fizeram com que os papeis não fossem rigorosamente seguidos. Dessa forma, cada integrante do grupo se ajudou com o intuito de evitar o sobrecarregamento de um determinado papel.

Além disso, para abordar cada fase da modelagem de implementação de uma forma precisa, utilizamos dos fundamentos essenciais pré-determinados pelo professor a qual uma

linguagem baseada no documento da linguagem Java deve oferecer suporte, sendo estes:

1. Definição de variáveis e tipos de dados
2. Definição de procedimentos com parâmetro
3. Expressões aritméticas, relacionais e lógicas
4. Escopos global e local
5. Geração de código para WebAssembly

Evolução da linguagem

Ao longo do desenvolvimento do Jarvas, aprendemos a importância de tomar decisões embasadas na documentação da linguagem Java e estar abertos ao feedback do professor. À medida que refinamos a sintaxe, a semântica e as funcionalidades da linguagem, buscamos equilibrar aspectos estéticos, didáticos e de usabilidade. Nosso objetivo é criar uma linguagem que seja intuitiva e capaz de atender às necessidades dos programadores que a utilizarem.

Conforme avançávamos no desenvolvimento do compilador, recebemos um comentário do professor em relação a estruturação da sintaxe das instruções, para tornar o código mais intuitivo e lógico, decidimos trocar a delimitação das expressão que antes era representadas como no exemplo a seguir:

```
if:  expressão  {  instrução  }
```

E que foram substituídas para:

```
if ( expressão ) { instrução }
```

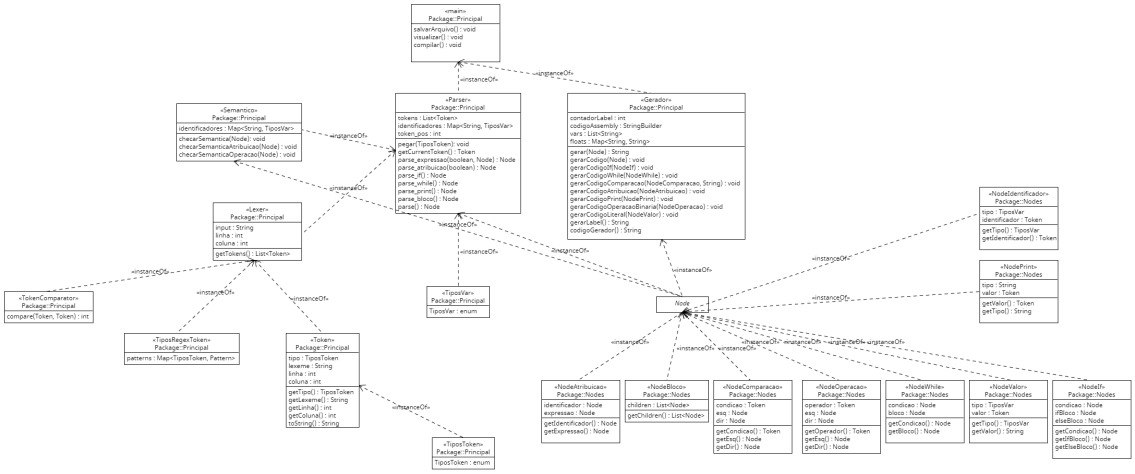
Ou seja, escolhemos delimitar as expressões com a utilização de parênteses como em Java

A evolução desse projeto nos mostra como a criação de uma linguagem de programação é um processo iterativo e desafiador.

Nos baseamos no procedimento de compilação que foi ensinado durante a disciplina de compiladores, por isso, tratamos da análise léxica inicialmente, depois a análise sintática e a análise semântica. Após isso tratamos da questão da geração do código objeto e do código executável.

Após a primeira apresentação tivemos que realizar correções como a citada acima que foi passada pelo professor e seguimos com o projeto sem encontrar problemas que parassem completamente toda a equipe.

Arquitetura do compilador



Ambiente de desenvolvimento

Durante o desenvolvimento do projeto, utilizamos o Visual Studio Code, um editor de código-fonte que oferece suporte a linguagem de programação Java e a documentação em Markdown, que também foi utilizado para realizar testes do compilador.

Como complemento, utilizamos o Live Share, uma extensão do Visual Studio Code que permitiu a colaboração em tempo real entre os integrantes do grupo

Plano de testes

- Funções básicas da linguagem

Nessa etapa, foi feito testes relacionados com atribuição de valores, execução de impressão de valores na tela de saída.

- Operações de lógica

Nessa etapa, foi feito a validação dos operadores lógicos, isso inclui o AND, OR e NOT.controle de fluxo, testes com tipos de dados e afins.

- Operações aritméticas

Nessa etapa, foi realizada uma validação abrangente dos operadores matemáticos existentes na linguagem, a fim de assegurar sua correta implementação e funcionamento. Para isso, foram conduzidos testes rigorosos para garantir que as operações de soma, subtração, divisão e multiplicação estejam sendo executadas de acordo com as regras matemáticas estabelecidas.

- Geração de códigos

  Realizando testes finais para garantir que os códigos estejam sendo gerados de maneira correta, seguindo uma arquitetura adequada e eficiente.

Conclusões contendo lições aprendidas

Uma das principais conclusões obtidas por meio deste projeto é a consolidação e aprofundamento dos conceitos fundamentais das linguagens de programação, bem como a experiência de construir uma linguagem do zero. Para nós, essa empreitada se mostrou desafiadora, uma vez que exigiu a aplicação simultânea dos conceitos apresentados nas aulas de compiladores ministradas pelo professor Italo, juntamente com o desenvolvimento da própria linguagem.

Com esse projeto, nós pudemos expandir nosso conhecimento sobre análise léxica, análise sintática, arquitetura de software e outros conceitos relacionados.

Encontramos diversas dificuldades durante o projeto, e tivemos que nos reunir para resolvê-las da melhor forma possível, os problemas encontrados fortaleceram nossa base conceitual e aumentaram nossa sensibilidade para projetos de maior complexidade.

Outro ponto interessante na confecção do projeto está na questão da documentação, ela é refinada e traz um formato interessante para o grupo trabalhar, com diversos elementos no relatório final, é possível explicar bem o projeto como um todo por esse motivo.

É importante declarar também, que o projeto não foi construído completamente da forma ideal, pois ocorreu um engano da parte de toda na equipe na relação entre o compilador e a gramática da linguagem Jarvas. Sabemos que linguagens como a linguagem C foram moldadas antes de suas gramáticas e mesmo assim funcionaram, mas posteriormente houve essa adição pois se viu como necessário possuir uma gramática para especificá-las. Nossa equipe só foi introduzida ao conceito de gramática após a primeira entrega do projeto, quando a parte relacionada à análise léxica já estava pronta e a parte sintática já estava sendo confeccionada. Houve um erro de interpretação da importância da gramática em uma linguagem por parte da equipe, e acabamos produzindo o compilador primeiro e a gramática depois.

Após pesquisas e conversas com o professor, percebemos que produzir o compilador antes de escrever a gramática da linguagem gera diversos empecilhos, desde ambiguidade até outros problemas que reduzem a confiabilidade da linguagem, porém quando a equipe percebeu isso faltavam poucos dias para a entrega e escolhemos esclarecer o engano que cometemos e assumir essa responsabilidade pois a outra opção seria refazer a maior parte do compilador e entregar algo sem qualidade e talvez não funcional devido ao curto prazo.

A responsabilidade recai sobre a equipe como um todo pela falta de aprofundamento em conceitos relacionados a gramáticas de linguagens de programação, e esse erro com certeza não se repetirá em futuros projetos, pois as consequências desse tipo de engano são consideravelmente preocupantes.

Apêndice contendo a listagem completa do código-fonte do compilador

package Principal; import java.io.File; import java.io.FileNotFoundException; import java.io.FileWriter; import java.io.IOException; import java.util.Scanner;

import Nodes.Node; import Nodes.NodeAtribuicao; import Nodes.NodeBloco; import Nodes.NodeComparacao; import Nodes.NodeIdentificador; import Nodes.NodeIf; import Nodes.NodeOperacao; import Nodes.NodePrint; import Nodes.NodeValor; import Nodes.NodeWhile;

/**

- Main
- / public class Main {

public static void main(String[] args) throws Exception {

```
  //Leitura do arquivo
  File file = new File(args[0]);
  StringBuilder contentBuilder = new StringBuilder();

  try (Scanner scanner = new Scanner(file)) {

      while (scanner.hasNextLine()) {
          String line = scanner.nextLine();
          contentBuilder.append(line).append("\n");
      }

  } catch (FileNotFoundException e) {
      e.printStackTrace();
  }

  String fileContent = contentBuilder.toString();

  //Cria o parser com a lista de tokens
  Parser parser = new Parser(fileContent);

  //Inicia o parser e retorna a Node Raiz da árvore sintática
  var raiz_arvore = parser.parse();

  //Printar a arvore final
  visualizar(raiz_arvore, 0);

  try{
      //Gerar o codigo da arvore em assembly
      Gerador.gerar(raiz_arvore);

      String cod = Gerador.codigoGerador();
      salvarArquivo(cod, "prog_comp.asm");

      //Roda os comandos no cmd para compilar o codigo asm
      compilar();
  }catch(Exception e){
  }
```

}

private static void salvarArquivo(String codigo, String nome_arquivo) {

```
  try (FileWriter writer = new FileWriter(nome_arquivo)) {
      writer.write(codigo);
  } catch (IOException e) {
```

```
        e.printStackTrace();
    }
```

```
}
```

```
private static void visualizar(Node node, int indentLevel) {
```

```
    String indent = " ".repeat(indentLevel * 2);

    if (node instanceof NodeBloco) {
        NodeBloco compositeNode = (NodeBloco) node;
        for (Node child : compositeNode.getChildren()) {
            visualizar(child, indentLevel);
        }
    } else if (node instanceof NodeAtribuicao) {
        NodeAtribuicao assignmentNode = (NodeAtribuicao) node;
        System.out.println(indent + "NodeAtribuicao");
        System.out.println(indent + "  Identificador: ");
        visualizar(assignmentNode.getIdentificador(), indentLevel + 1);
        System.out.println(indent + "  Expressao: ");
        visualizar(assignmentNode.getExpressao(), indentLevel + 1);
    } else if (node instanceof NodeOperacao) {
        NodeOperacao binaryOperationNode = (NodeOperacao) node;
        System.out.println(indent + "NodeOperacao");
        System.out.println(indent + "  Operador: " +
binaryOperationNode.getOperador());
        System.out.println(indent + "  Esquerda: ");
        visualizar(binaryOperationNode.getEsq(), indentLevel + 1);
        System.out.println(indent + "  Direita: ");
        visualizar(binaryOperationNode.getDir(), indentLevel + 1);
    } else if (node instanceof NodeValor) {
        NodeValor literalNode = (NodeValor) node;
        System.out.println(indent + "NodeValor");
        System.out.println(indent + "  Valor: " + literalNode.getValor());
    } else if (node instanceof NodeIf) {
        NodeIf ifNode = (NodeIf) node;
        System.out.println(indent + "NodeIf");
        System.out.println(indent + "  Condicao: ");
        visualizar(ifNode.getCondicao(), indentLevel + 1);
        System.out.println(indent + "  NodeBloco: ");
        visualizar(ifNode.getIfBloco(), indentLevel + 1);
        System.out.println(indent + "  NodeElseBloco: ");
        visualizar(ifNode.getElseBloco(), indentLevel + 1);
    } else if (node instanceof NodeWhile) {
        NodeWhile ifNode = (NodeWhile) node;
        System.out.println(indent + "NodeWhile");
        System.out.println(indent + "  Condicao: ");
        visualizar(ifNode.getCondicao(), indentLevel + 1);
        System.out.println(indent + "  NodeBloco: ");
        visualizar(ifNode.getBloco(), indentLevel + 1);
    } else if (node instanceof NodeComparacao) {
        NodeComparacao nodeC = (NodeComparacao) node;
        System.out.println(indent + "NodeComparacao");
```

```
      System.out.println(indent + "  Condicao: " + nodeC.getCondicao());
      System.out.println(indent + "  Esquerda: ");
      visualizar(nodeC.getEsq(), indentLevel + 1);
      System.out.println(indent + "  Direita: ");
      visualizar(nodeC.getDir(), indentLevel + 1);
  } else if (node instanceof NodeIdentificador) {
      NodeIdentificador nodeC = (NodeIdentificador) node;
      System.out.println(indent + "NodeIdentificador");
      System.out.println(indent + "  Tipo: " + nodeC.getTipo());
      System.out.println(indent + "  Identificador: " +
nodeC.getIdentificador());
  } else if (node instanceof NodePrint) {
      NodePrint nodeC = (NodePrint) node;
      System.out.println(indent + "NodePrint");
      System.out.println(indent + "  Tipo: " + nodeC.getTipo());
      System.out.println(indent + "  Valor: " + nodeC.getValor().getLexeme());
  }
```

```
}
```

```
public static void compilar(){
```

```
  try {

      String command = "nasm -f elf32 prog_comp.asm -o prog_comp.o";

      Process process = Runtime.getRuntime().exec(command);

      command = "gcc -m32 prog_comp.o -o programa";

      process = Runtime.getRuntime().exec(command);

      process = Runtime.getRuntime().exec("chmod +x ./programa");
  } catch (IOException e) {
      e.printStackTrace();
  }
```

```
} }
```

```
package Principal;

import java.util.ArrayList; import java.util.HashMap; import java.util.List; import
java.util.Map;

import Nodes.Node; import Nodes.NodeAtribuicao; import Nodes.NodeBloco; import
Nodes.NodeComparacao; import Nodes.NodeIdentificador; import Nodes.NodeIf; import
Nodes.NodeOperacao; import Nodes.NodePrint; import Nodes.NodeValor; import
Nodes.NodeWhile;

public class Gerador { private static int contadorLabel = 0; private static
StringBuilder codigoAssembly = new StringBuilder();
```

```
public static List<String> vars = new ArrayList<String>();
public static Map<String, String> floats = new HashMap<String, String>();

public static String gerar(Node node) {
    codigoAssembly.setLength(0);
    gerarCodigo(node);
    return codigoAssembly.toString();
}

private static void gerarCodigo(Node node) {
    if (node instanceof NodeBloco) {
        NodeBloco node_aux = (NodeBloco) node;
        for (Node aux : node_aux.getChildren())
            gerarCodigo(aux);
    } else if (node instanceof NodeIf) {
        gerarCodigoIf((NodeIf) node);
    } else if (node instanceof NodeAtribuicao) {
        gerarCodigoAtribuicao((NodeAtribuicao) node);
    } else if (node instanceof NodeOperacao) {
        gerarCodigoOperacaoBinaria((NodeOperacao) node);
    } else if (node instanceof NodeValor) {
        gerarCodigoLiteral((NodeValor) node);
    } else if (node instanceof NodePrint) {
        gerarCodigoPrint((NodePrint) node);
    } else if (node instanceof NodeWhile) {
        gerarCodigoWhile((NodeWhile) node);
    }
}

private static void gerarCodigoIf(NodeIf node) {
    String labelCondicao = gerarLabel();
    String labelFim = gerarLabel();

    if (node.getCondicao() instanceof NodeComparacao) {
        gerarCodigoComparacao((NodeComparacao) node.getCondicao(), labelCondicao,
labelFim);
    } else {
        codigoAssembly.append("\n").append("CMP EAX, 1").append("\n");
    }

    codigoAssembly.append("JE ").append(labelCondicao).append("\n");

    gerarCodigo(node.getElseBloco());
    codigoAssembly.append("\n").append("JMP ").append(labelFim).append("\n");

    codigoAssembly.append(labelCondicao).append(":").append("\n");
    gerarCodigo(node.getIfBloco());
    codigoAssembly.append("\n").append("JMP ").append(labelFim).append("\n");

    codigoAssembly.append(labelFim).append(":").append("\n");
}
```

```
private static void gerarCodigoWhile(NodeWhile node) {
    String labelInicio = gerarLabel();
    String labelFim = gerarLabel();

    codigoAssembly.append(labelInicio).append(":").append("\n");

    if (node.getCondicao() instanceof NodeComparacao) {
        gerarCodigoComparacao((NodeComparacao) node.getCondicao(), labelInicio,
labelFim);
    } else {
        codigoAssembly.append("CMP EAX, 1").append("\n");
        codigoAssembly.append("JNE ").append(labelFim).append("\n");
    }

    gerarCodigo(node.getBloco()); // Adicionado para gerar o código do bloco do while

    codigoAssembly.append("\n").append("JMP ").append(labelInicio).append("\n");

    codigoAssembly.append(labelFim).append(":").append("\n");
}

private static void gerarCodigoComparacao(NodeComparacao node, String labelVerdadeiro,
String labelFim) {
    gerarCodigo(node.getEsq());
    codigoAssembly.append("\n").append("PUSH EAX").append("\n");
    gerarCodigo(node.getDir());
    codigoAssembly.append("\n").append("POP EBX").append("\n");

    Token operador = node.getCondicao();

    switch (operador.getTipo()) {
        case EQUALS:
            codigoAssembly.append("\n").append("CMP EAX, EBX").append("\n");
            codigoAssembly.append("JE ").append(labelVerdadeiro).append("\n");
            break;
        case NOT_EQUAL:
            codigoAssembly.append("\n").append("CMP EAX, EBX").append("\n");
            codigoAssembly.append("JNE ").append(labelVerdadeiro).append("\n");
            break;
        case GREATER_THAN:
            codigoAssembly.append("\n").append("CMP EAX, EBX").append("\n");
            codigoAssembly.append("JG ").append(labelVerdadeiro).append("\n");
            break;
        case GREATER_EQUAL:
            codigoAssembly.append("\n").append("CMP EAX, EBX").append("\n");
            codigoAssembly.append("JGE ").append(labelVerdadeiro).append("\n");
            break;
        case LESS_THAN:
            codigoAssembly.append("\n").append("CMP EAX, EBX").append("\n");
            codigoAssembly.append("JL ").append(labelVerdadeiro).append("\n");
            break;
        case LESS_EQUAL:
```

```
            codigoAssembly.append("\n").append("CMP EAX, EBX").append("\n");
            codigoAssembly.append("JLE ").append(labelVerdadeiro).append("\n");
            break;
    }
}

private static void gerarCodigoAtribuicao(NodeAtribuicao node) {
    gerarCodigo(node.getExpressao());
    NodeIdentificador iden = (NodeIdentificador) node.getIdentificador();

    String var = iden.getIdentificador().getLexeme();
    if (!vars.contains(var))
        vars.add(var);

    if (iden.getTipo() == TiposVar.FLOAT)
        codigoAssembly.append("\n").append("fstp dword
[").append(var).append("]").append("\n");
    else
        codigoAssembly.append("\n").append("MOV [").append(var).append("],
EAX").append("\n");
}

private static void gerarCodigoPrint(NodePrint node) {
    switch(node.getTipo()){
        case "int":
            codigoAssembly.append("\n\npush dword
[").append(node.getValor().getLexeme()).append("]\npush dword printint\ncall
printf\nadd esp, 8");
            break;
        case "bool":
            codigoAssembly.append("\n\npush dword
[").append(node.getValor().getLexeme()).append("]\npush dword printint\ncall
printf\nadd esp, 8");
            break;
        case "char":
            codigoAssembly.append("\n\npush dword
[").append(node.getValor().getLexeme()).append("]\npush dword printchar\ncall
printf\nadd esp, 8");
            break;
        case "float":
            codigoAssembly.append("\n\nfld dword
[").append(node.getValor().getLexeme()).append("]\nsub esp, 8\nfstp qword [esp]\npush
printfloat\ncall printf\nadd esp, 12");
            break;
    }
}

private static void gerarCodigoOperacaoBinaria(NodeOperacao node) {
    gerarCodigo(node.getDir());
    codigoAssembly.append("PUSH EAX").append("\n");
    gerarCodigo(node.getEsq());
    codigoAssembly.append("POP EBX").append("\n");
```

```
    Token operador = node.getOperador();
    switch (operador.getTipo()) {
        case PLUS:
            codigoAssembly.append("ADD EAX, EBX").append("\n");
            break;
        case MINUS:
            codigoAssembly.append("SUB EAX, EBX").append("\n");
            break;
        case TIMES:
            codigoAssembly.append("IMUL EAX, EBX").append("\n");
            break;
        case DIVIDE:
            codigoAssembly.append("MOV EDX, 0").append("\n");
            codigoAssembly.append("IDIV EBX").append("\n");
            break;
    }
}

private static void gerarCodigoLiteral(NodeValor node) {
    String val = node.getValor();
    val = val.replace("true", "1");
    val = val.replace("false", "0");

    String label_float = "";
    if (val.contains(".")){
        label_float = gerarLabel();
        floats.put(label_float, val);

        codigoAssembly.append("\nfld dword [").append(label_float).append("]\n");
    }else{
        if (val.replace(".", "").matches("\\d+") || val.contains("'")){
            codigoAssembly.append("\nMOV EAX, ").append(val).append("\n");
        }else{
            codigoAssembly.append("\nMOV EAX, [").append(val).append("]\n");
        }
    }
}

private static String gerarLabel() {
    contadorLabel++;
    return "L" + contadorLabel;
}

public static String codigoGerador() {
    StringBuilder codigoFinal = new StringBuilder();
    codigoFinal.append("section .data").append("\n");
    for (var aux : vars)
        codigoFinal.append("    " + aux + ":  dd 0").append("\n"); // Declaração da
variável x
    for (var aux : floats.keySet())
        codigoFinal.append("    " + aux + ":  dd " + floats.get(aux)).append("\n"); //
```

```
Declaração da variável x
    codigoFinal.append("    " + "printint: db " + '"' + "%d" + '"' + ", 10,
0").append("\n");
    codigoFinal.append("    " + "printchar: db " + '"' + "%c" + '"' + ", 10,
0").append("\n");
    codigoFinal.append("    " + "printfloat: db " + '"' + "%f" + '"' + ", 10,
0").append("\n");
    codigoFinal.append("\n");
    codigoFinal.append("section .text").append("\n");
    codigoFinal.append("    global main").append("\n\n");
    codigoFinal.append("extern printf, exit");
    codigoFinal.append("\n\n");
    codigoFinal.append("main:").append("\n");
    codigoFinal.append(codigoAssembly);

    codigoFinal.append("\nMOV EAX, 1").append("\n").append("XOR EBX,
EBX").append("\n").append("INT 0x80").append("\n");
    return codigoFinal.toString();
}
```

}

---

package Principal; import java.util.ArrayList; import java.util.List; import java.util.regex.Matcher; import java.util.regex.Pattern; import java.util.Collections;

public class Lexer { public String input; public int linha; public int coluna;

```
public List<Token> tokens;

public Lexer(String input){
    this.input = input;
    this.tokens = new ArrayList<Token>();
}

public List<Token> getTokens() {
    //Separa o codigo por linhas
    String[] lines = input.split("\\r?\\n");

    int i = 0;
    for (i = 0; i < lines.length; i++) { //Roda por todas as linhas

        coluna = 0;
        String line = lines[i];

        //Ele roda por todos os lexemes dessa linha, Tokenizando ate a linha estar
vazia
        while (!line.replace(" ", "").isEmpty()) {
            System.out.println(line);

            Matcher matcher;
            for(var tipo : TiposToken.values()){ //Rodar pelos tipos de token,
procurando pelo regex
```

```
                Pattern tipo_pattern = TiposRegexToken.patterns.get((TiposToken)
tipo);

                matcher = tipo_pattern.matcher(line);

                if (matcher.find()){ //Encontrou
                    String value = matcher.group();
                    coluna = matcher.end() - value.length();
                    tokens.add(new Token(tipo, value, i, coluna)); //Adiciona na lista
de tokens
                    line = line.substring(0, coluna) + " " +
line.substring(matcher.end());
                }
            }
        }
    }

    //Adiciona o fim do arquivo
    tokens.add(new Token(TiposToken.EOF, "", lines.length, 0));

    //Ordena a lista de tokens por linha e coluna
    Collections.sort(tokens, new TokenComparator());

    //Devolve a lista
    return tokens;
}
```

}

```
package Principal; import java.util.ArrayList; import java.util.HashMap; import
java.util.List; import java.util.Map;

import Nodes.Node; import Nodes.NodeAtribuicao; import Nodes.NodeBloco; import
Nodes.NodeComparacao; import Nodes.NodeIdentificador; import Nodes.NodeIf; import
Nodes.NodeOperacao; import Nodes.NodePrint; import Nodes.NodeValor; import
Nodes.NodeWhile;

public class Parser {
```

```
private List<Token> tokens;
private Map<String, TiposVar> identificadores = new HashMap<String, TiposVar>();
private int token_pos;

public Parser(String fileContent) {
    Lexer lexer = new Lexer(fileContent);
    this.tokens = lexer.getTokens();
    token_pos = 0;
}

private void pegar(TiposToken tipo) throws Exception {
    Token currentToken = getCurrentToken();
    System.out.println(currentToken);
    if (currentToken.getTipo() == tipo) {
```

```
        token_pos++;
    } else {
        throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());
    }
}

private Token getCurrentToken() throws Exception {
    if (token_pos >= tokens.size()) {
        throw new Exception("Erro de sintaxe. Fim inesperado do arquivo.");
    }
    return tokens.get(token_pos);
}

/*
*
* <valor>;
* <valor> <operador> <valor>;
* <valor> <operador> (<valor> <operador> <valor>);
* (<valor> <operador> <valor>);
* (<valor>);
*
*/
private Node parse_expressao(boolean com_valor, Node valor) throws Exception {
    //Retornar Valor ou Expressao
    Token currentToken = getCurrentToken();

    if (currentToken.getTipo() == TiposToken.BOOLEAN){
        pegar(TiposToken.BOOLEAN);
        return new NodeValor(currentToken, TiposVar.BOOL);
    }

    Node node_valor;

    if(!com_valor){

        if(currentToken.getTipo() == TiposToken.PAREN_L){
            pegar(currentToken.getTipo());
        }

        currentToken = getCurrentToken();

        node_valor = new NodeValor(currentToken, TiposVar.INT);
        //Checar valor
        if(currentToken.getTipo() == TiposToken.INTEGER || currentToken.getTipo() ==
TiposToken.FLOAT || currentToken.getTipo() == TiposToken.IDENTIFIER)
            pegar(currentToken.getTipo());
        else
            throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());

        currentToken = getCurrentToken();
```

```
        //Fim?
        if(currentToken.getTipo() == TiposToken.SEMICOLON || currentToken.getTipo() ==
TiposToken.EQUALS || currentToken.getTipo() == TiposToken.LESS_EQUAL
        || currentToken.getTipo() == TiposToken.LESS_THAN || currentToken.getTipo() ==
TiposToken.GREATER_EQUAL || currentToken.getTipo() == TiposToken.GREATER_THAN
        || currentToken.getTipo() == TiposToken.NOT_EQUAL){
            return node_valor;
        }

        if(currentToken.getTipo() == TiposToken.PAREN_R){
            return node_valor;
        }

    }else{
        node_valor = valor;
    }

    //Logo: expressao
    currentToken = getCurrentToken();

    if(currentToken.getTipo() == TiposToken.BRACE_L){
        return node_valor;
    }

    //Checar operador
    Token op = getCurrentToken();
    if(currentToken.getTipo() == TiposToken.PLUS || currentToken.getTipo() ==
TiposToken.MINUS || currentToken.getTipo() == TiposToken.TIMES)
        pegar(currentToken.getTipo());
    else
        throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());

    return new NodeOperacao(op, node_valor, parse_expressao(false, null));
}

/*
 *
 * <tipo> <identificador> = <expressao>;
 *
 */
private Node parse_atribuicao(boolean inicializar) throws Exception {
    Token currentToken = getCurrentToken();

    TiposVar tipo = null;
    if (inicializar){
        switch(currentToken.getTipo()){
            case VAR_INT:
                pegar(currentToken.getTipo());
                tipo = TiposVar.INT;
                break;
```

```
            case VAR_BOOL:
                pegar(currentToken.getTipo());
                tipo = TiposVar.BOOL;
                break;
            case VAR_FLOAT:
                pegar(currentToken.getTipo());
                tipo = TiposVar.FLOAT;
                break;
            case VAR_CHAR:
                pegar(currentToken.getTipo());
                tipo = TiposVar.CHAR;
                break;
            default:
                throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());
        }
        currentToken = getCurrentToken();
        if (!identificadores.containsKey(currentToken.getLexeme()))
            identificadores.put(currentToken.getLexeme(), tipo);
    }else{
        if (identificadores.containsKey(currentToken.getLexeme()))
            tipo = identificadores.get(currentToken.getLexeme());
    }

    Token identifierToken = getCurrentToken(); //Guarda o identificador
    NodeIdentificador identificador = new NodeIdentificador(tipo, identifierToken);

    pegar(TiposToken.IDENTIFIER);

    pegar(TiposToken.EQUAL);

    currentToken = getCurrentToken();

    Node final_expression = null;
    switch(tipo){
        case BOOL:
            final_expression = new NodeValor(getCurrentToken(), TiposVar.BOOL);
            pegar(TiposToken.BOOLEAN);
            tipo = TiposVar.BOOL;
            break;
        case CHAR:
            final_expression = new NodeValor(getCurrentToken(), TiposVar.CHAR);
            pegar(TiposToken.CHAR);
            tipo = TiposVar.CHAR;
            break;
        case FLOAT:
        case INT:
            Node expression = parse_expressao(false, null);

            currentToken = getCurrentToken();

            final_expression = expression;
```

```
            if (currentToken.getTipo() == TiposToken.PAREN_R){
                pegar(TiposToken.PAREN_R);

                final_expression = parse_expressao(true, expression);
            }

            currentToken = getCurrentToken();
            if (currentToken.getTipo() == TiposToken.PAREN_R)
                pegar(TiposToken.PAREN_R);

            break;
        default:
            throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());
    }

    pegar(TiposToken.SEMICOLON);

    return new NodeAtribuicao(identificador, final_expression);
}

/*
 *
 * if (<expressao>) {}
 * if (<expressao> == <expressao>) {}
 *
 */
private Node parse_if() throws Exception {
    pegar(TiposToken.IF);

    pegar(TiposToken.PAREN_L);

    Token currentToken = getCurrentToken();
    Node nodeFinal = null;
    if (currentToken.getTipo() == TiposToken.BOOLEAN){
        nodeFinal = new NodeValor(currentToken, TiposVar.BOOL);
        pegar(currentToken.getTipo());
        pegar(TiposToken.PAREN_R);
    }else{

        Node expression = parse_expressao(false, null);
        Node final_expression_left = expression;
        if (currentToken.getTipo() == TiposToken.PAREN_R){
            pegar(TiposToken.PAREN_R);
            final_expression_left = parse_expressao(true, expression);
        }
        currentToken = getCurrentToken();

        if (currentToken.getTipo() == TiposToken.PAREN_R){
            nodeFinal = new NodeValor(currentToken, TiposVar.BOOL);
            pegar(currentToken.getTipo());
```

```
        }else{
            currentToken = getCurrentToken();

            Token cond = currentToken;

            if (currentToken.getTipo() == TiposToken.EQUALS || currentToken.getTipo()
== TiposToken.LESS_EQUAL || currentToken.getTipo() == TiposToken.LESS_THAN
                || currentToken.getTipo() == TiposToken.GREATER_EQUAL ||
currentToken.getTipo() == TiposToken.GREATER_THAN)
                pegar(currentToken.getTipo());
            else
                throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());

            expression = parse_expressao(false, null);
            Node final_expression_right = expression;
            currentToken = getCurrentToken();
            if (currentToken.getTipo() == TiposToken.PAREN_R){
                pegar(TiposToken.PAREN_R);

                final_expression_right = parse_expressao(true, expression);
            }

            nodeFinal = new NodeComparacao(cond, final_expression_left,
final_expression_right);
        }
    }

    Node blocoIf = parse_bloco();
    currentToken = getCurrentToken();

    Node blocoElse = null;
    if (currentToken.getTipo() == TiposToken.ELSE){
        pegar(currentToken.getTipo());
        blocoElse = parse_bloco();
    }

    return new NodeIf(nodeFinal, blocoIf, blocoElse);
}

/*
 *
 * while (<expressao>) {}
 * while (<expressao> == <expressao>) {}
 * while (<valor> == <expressao>) {}
 *
 */
private Node parse_while() throws Exception {
    pegar(TiposToken.WHILE);

    pegar(TiposToken.PAREN_L);
```

```
    Token currentToken = getCurrentToken();

    Node nodeFinal = null;
    if (currentToken.getTipo() == TiposToken.BOOLEAN){
        nodeFinal = new NodeValor(currentToken, TiposVar.BOOL);
        pegar(currentToken.getTipo());
        pegar(TiposToken.PAREN_R);
    }else{
        Node expression = parse_expressao(false, null);
        Node final_expression_left = expression;
        if (currentToken.getTipo() == TiposToken.PAREN_R){
            pegar(TiposToken.PAREN_R);
            final_expression_left = parse_expressao(true, expression);
        }
        currentToken = getCurrentToken();

        if (currentToken.getTipo() == TiposToken.PAREN_R){
            nodeFinal = new NodeValor(currentToken, TiposVar.BOOL);
            pegar(currentToken.getTipo());
        }else{
            currentToken = getCurrentToken();

            Token cond = currentToken;

            if (currentToken.getTipo() == TiposToken.EQUALS || currentToken.getTipo()
== TiposToken.LESS_EQUAL || currentToken.getTipo() == TiposToken.LESS_THAN
                || currentToken.getTipo() == TiposToken.GREATER_EQUAL ||
currentToken.getTipo() == TiposToken.GREATER_THAN)
                pegar(currentToken.getTipo());
            else
                throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());

            expression = parse_expressao(false, null);
            Node final_expression_right = expression;
            currentToken = getCurrentToken();
            if (currentToken.getTipo() == TiposToken.PAREN_R){
                pegar(TiposToken.PAREN_R);

                final_expression_right = parse_expressao(true, expression);
            }

            nodeFinal = new NodeComparacao(cond, final_expression_left,
final_expression_right);
        }
    }

    Node bloco = parse_bloco();

    return new NodeWhile(nodeFinal, bloco);
}
```

```
/*
 *
 * print(<valor>)
 *
 */
private Node parse_print() throws Exception {
    pegar(TiposToken.PRINT);

    pegar(TiposToken.PAREN_L);

    Token valor = getCurrentToken();

    pegar(valor.getTipo());

    pegar(TiposToken.COMMA);

    Token tipoValor = getCurrentToken();

    pegar(tipoValor.getTipo());

    pegar(TiposToken.PAREN_R);

    pegar(TiposToken.SEMICOLON);

    return new NodePrint(valor, tipoValor);
}

/*
 * parse_bloco
 *
 * Roda por um bloco de Nodes
 */
private Node parse_bloco() throws Exception {
    pegar(TiposToken.BRACE_L);

    NodeBloco nodeBloco = new NodeBloco();
    while (getCurrentToken().getTipo() != TiposToken.BRACE_R) {
        Token currentToken = getCurrentToken();
        switch(currentToken.getTipo()){
            case VAR_INT:
                nodeBloco.adicionar(parse_atribuicao(true));
                break;
            case VAR_BOOL:
                nodeBloco.adicionar(parse_atribuicao(true));
                break;
            case VAR_CHAR:
                nodeBloco.adicionar(parse_atribuicao(true));
                break;
            case VAR_FLOAT:
                nodeBloco.adicionar(parse_atribuicao(true));
                break;
            case IDENTIFIER:
```

```
                nodeBloco.adicionar(parse_atribuicao(false));
                break;
            case IF:
                nodeBloco.adicionar(parse_if());
                break;
            case WHILE:
                nodeBloco.adicionar(parse_while());
                break;
            case PRINT:
                nodeBloco.adicionar(parse_print());
                break;
            default:
                throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());
        }
    }

    pegar(TiposToken.BRACE_R);

    return nodeBloco;
}

public Node parse() throws Exception {
    System.out.println("Tokens (Lexer): ");
    System.out.println(tokens);

    System.out.println("Iniciando Parser");

    //Bloco Inicial
    NodeBloco main = new NodeBloco();

    //Enquanto houver tokens, verificar qual a proxima frase
    while (token_pos < tokens.size()) {
        Token currentToken = getCurrentToken();
        switch(currentToken.getTipo()){
            case VAR_INT:
                main.adicionar(parse_atribuicao(true));
                break;
            case VAR_BOOL:
                main.adicionar(parse_atribuicao(true));
                break;
            case VAR_FLOAT:
                main.adicionar(parse_atribuicao(true));
                break;
            case VAR_CHAR:
                main.adicionar(parse_atribuicao(true));
                break;
            case IDENTIFIER:
                main.adicionar(parse_atribuicao(false));
                break;
            case IF:
                main.adicionar(parse_if());
```

```
                break;
            case PRINT:
                main.adicionar(parse_print());
                break;
            case WHILE:
                main.adicionar(parse_while());
                break;
            case EOF:
                return main;
            default:
                throw new Exception("Erro de sintaxe. Token inesperado: " +
currentToken.getTipo());
        }
    }

    Semantico.checarSemantica(main);

    return main;
}
```

}

```
package Principal;

import java.util.ArrayList; import java.util.HashMap; import java.util.List; import
java.util.Map;

import Nodes.Node; import Nodes.NodeAtribuicao; import Nodes.NodeBloco; import
Nodes.NodeComparacao; import Nodes.NodeIdentificador; import Nodes.NodeIf; import
Nodes.NodeOperacao; import Nodes.NodeValor; import Nodes.NodeWhile;

public class Semantico {

public static Map<String, TiposVar> identificadores = new HashMap<String, TiposVar>();

public static void checarSemantica(Node node) throws Exception {
    System.out.println(node.getClass().getName());
    if (node instanceof NodeBloco) {
        NodeBloco node_aux = (NodeBloco) node;
        for (Node aux : node_aux.getChildren())
            checarSemantica(aux);
    } else if (node instanceof NodeIf) {
        NodeIf nodeIf = (NodeIf) node;
        checarSemantica(nodeIf.getCondicao());
        checarSemantica(nodeIf.getIfBloco());
        checarSemantica(nodeIf.getElseBloco());
    } else if (node instanceof NodeAtribuicao) {
        checarSemanticaAtribuicao((NodeAtribuicao) node);
    } else if (node instanceof NodeOperacao) {
        checarSemanticaOperacao((NodeOperacao) node);
    } else if (node instanceof NodeWhile) {
        NodeWhile nodeWhile = (NodeWhile) node;
        checarSemantica(nodeWhile.getCondicao());
```

```
        checarSemantica(nodeWhile.getBloco());
    } else if (node instanceof NodeIdentificador) {
        NodeIdentificador nodeIdentificador = (NodeIdentificador) node;
        identificadores.put(nodeIdentificador.getIdentificador().getLexeme(),
nodeIdentificador.getTipo());
    }
}

private static void checarSemanticaAtribuicao(NodeAtribuicao node) throws Exception{
    NodeIdentificador identificador = (NodeIdentificador) node.getIdentificador();
    checarSemantica(node.getIdentificador());
    checarSemantica(node.getExpressao());

    if (node.getExpressao() instanceof NodeComparacao) {
        checarSemantica(node.getExpressao());

        if (identificador.getTipo() != TiposVar.INT){
            throw new Exception("Erro semantico: Atribuicao de tipos diferentes");
        }
    } else if (node.getExpressao() instanceof NodeIdentificador) {
        NodeIdentificador identificador_2 = (NodeIdentificador) node.getExpressao();
        if (identificador.getTipo() != identificador_2.getTipo()){
            throw new Exception("Erro semantico: Atribuicao de tipos diferentes");
        }
    } else if (node.getExpressao() instanceof NodeValor) {
        NodeValor valor = (NodeValor) node.getExpressao();
        if (valor.getValor().matches("\\d+")){
            if (identificador.getTipo() == TiposVar.BOOL){
                throw new Exception("Erro semantico: Atribuicao de tipos diferentes");
            }
        }else{
            if(!valor.getValor().matches("true|false") && identificador.getTipo() ==
TiposVar.BOOL){
                throw new Exception("Erro semantico: Atribuicao de tipos diferentes");
            }
        }
    }
}

private static void checarSemanticaOperacao(NodeOperacao node) throws Exception{
    checarSemantica(node.getDir());
    checarSemantica(node.getEsq());

    if (node.getDir() instanceof NodeIdentificador && node.getEsq() instanceof
NodeIdentificador) {
        NodeIdentificador dir = (NodeIdentificador) node.getDir();
        NodeIdentificador esq = (NodeIdentificador) node.getEsq();

        if (dir.getTipo() != esq.getTipo()){
            throw new Exception("Erro semantico: Operacao entre tipos diferentes");
        }
    }
```

```
    if(node.getDir() instanceof NodeValor && node.getEsq() instanceof NodeValor){
        NodeValor dir = (NodeValor) node.getDir();
        NodeValor esq = (NodeValor) node.getEsq();
        TiposVar tipo_dir = dir.getTipo();
        TiposVar tipo_esq = esq.getTipo();

        if (identificadores.containsKey(dir.getValor()))
            tipo_dir = identificadores.get(dir.getValor());

        if (identificadores.containsKey(esq.getValor()))
            tipo_esq = identificadores.get(esq.getValor());

        if (tipo_dir != tipo_esq){
            throw new Exception("Erro semantico: Operacao entre tipos diferentes");
        }
    }
    if(node.getDir() instanceof NodeIdentificador && (node.getEsq() instanceof
NodeValor
    || node.getEsq() instanceof NodeComparacao || node.getEsq() instanceof
NodeOperacao)){
        NodeIdentificador dir = (NodeIdentificador) node.getDir();

        if (dir.getTipo() != TiposVar.INT){
            throw new Exception("Erro semantico: Operacao entre tipos diferentes");
        }
    }
    if((node.getDir() instanceof NodeValor || node.getDir() instanceof NodeOperacao ||
node.getDir() instanceof NodeComparacao)
    && node.getEsq() instanceof NodeIdentificador){
        NodeIdentificador esq = (NodeIdentificador) node.getEsq();

        if (esq.getTipo() != TiposVar.INT){
            throw new Exception("Erro semantico: Operacao entre tipos diferentes");
        }
    }
}
```

}

---

package Principal; import java.util.Map; import java.util.regex.Matcher; import java.util.regex.Pattern;

/**

- TiposRegexToken
- / public class TiposRegexToken {

public static Map<TiposToken, Pattern> patterns = Map.ofEntries(

```
  // PALAVRAS CHAVE
  Map.entry(TiposToken.IF, Pattern.compile("if\\b")),
  Map.entry(TiposToken.ELSE, Pattern.compile("\\belse\\b")),
  Map.entry(TiposToken.WHILE, Pattern.compile("\\bwhile\\b")),
```

```
  Map.entry(TiposToken.FOR, Pattern.compile("\\bfor\\b")),
  Map.entry(TiposToken.DO, Pattern.compile("\\bdo\\b")),
  Map.entry(TiposToken.RETURN, Pattern.compile("\\breturn\\b")),
  Map.entry(TiposToken.BREAK, Pattern.compile("\\bbreak\\b")),
  Map.entry(TiposToken.CLASS, Pattern.compile("\\bclass\\b")),

  //FUNCOES
  Map.entry(TiposToken.PRINT, Pattern.compile("\\bprint\\b")),

  // DELIMITADORES
  Map.entry(TiposToken.PAREN_L, Pattern.compile("\\(")),
  Map.entry(TiposToken.PAREN_R, Pattern.compile("\\)")),
  Map.entry(TiposToken.BRACE_L, Pattern.compile("\\{")),
  Map.entry(TiposToken.BRACE_R, Pattern.compile("\\}")),
  Map.entry(TiposToken.BRACKET_L, Pattern.compile("\\[")),
  Map.entry(TiposToken.BRACKET_R, Pattern.compile("\\]")),
  Map.entry(TiposToken.SEMICOLON, Pattern.compile(";")),
  Map.entry(TiposToken.COMMA, Pattern.compile(",")),
  Map.entry(TiposToken.DOUBLE_DOT, Pattern.compile("\\:")),

  // IDENTIFICADORES E LITERAIS
  Map.entry(TiposToken.VAR_INT, Pattern.compile("int")),
  Map.entry(TiposToken.VAR_BOOL, Pattern.compile("bool")),
  Map.entry(TiposToken.VAR_CHAR, Pattern.compile("char")),
  Map.entry(TiposToken.VAR_FLOAT, Pattern.compile("float")),
  Map.entry(TiposToken.IDENTIFIER, Pattern.compile("[a-zA-Z_$][a-zA-Z_$0-
9]*")),
  Map.entry(TiposToken.INTEGER, Pattern.compile("\\b(?!\\d+\\.\\d+)\\d+\\b")),
  Map.entry(TiposToken.FLOAT, Pattern.compile("\\d+\\.\\d+")),
  Map.entry(TiposToken.CHAR, Pattern.compile("\'([^\']|\\.)\'")),
  Map.entry(TiposToken.BOOLEAN, Pattern.compile("true|false")),

  // ATRIBUICAO
  Map.entry(TiposToken.EQUAL, Pattern.compile("(?<!<>=)=(?!=)")),

  // OPERADORES
  Map.entry(TiposToken.PLUS, Pattern.compile("\\+")),
  Map.entry(TiposToken.MINUS, Pattern.compile("\\-")),
  Map.entry(TiposToken.TIMES, Pattern.compile("\\*")),
  Map.entry(TiposToken.DIVIDE, Pattern.compile("\\/")),

  // CONDICIONAIS
  Map.entry(TiposToken.DOUBLE_EQUALS, Pattern.compile("(?<![!=])\\={2}(?!=)")),
  Map.entry(TiposToken.NOT_EQUAL, Pattern.compile("\\b!=\\b")),
  Map.entry(TiposToken.GREATER_EQUAL, Pattern.compile(">=")),
  Map.entry(TiposToken.LESS_EQUAL, Pattern.compile("<=")),
  Map.entry(TiposToken.GREATER_THAN, Pattern.compile("(?<!>)>")),
  Map.entry(TiposToken.LESS_THAN, Pattern.compile("<")),

  // FIM
  Map.entry(TiposToken.EOF, Pattern.compile("FIMDOARQUIVO"))
```

```
    ); }
```

---

package Principal; public enum TiposToken { // PALAVRAS CHAVE IF, ELSE, WHILE, FOR, DO, RETURN, BREAK, CLASS,

```
//FUNCOES
PRINT,

// DELIMITADORES
PAREN_L, PAREN_R, BRACE_L, BRACE_R, BRACKET_L, BRACKET_R, SEMICOLON, COMMA,
DOUBLE_DOT,

// IDENTIFICADORES E LITERAIS
VAR_INT, VAR_BOOL, VAR_FLOAT, VAR_CHAR, IDENTIFIER, FLOAT, INTEGER, CHAR,
BOOLEAN,

// CONDICIONAIS
EQUALS, NOT_EQUAL, GREATER_THAN, LESS_THAN, GREATER_EQUAL, LESS_EQUAL,

// ATRIBUICAO
EQUAL,

// OPERADORES
PLUS, MINUS, TIMES, DIVIDE,


//FIM
EOF
```

}

---

package Principal; public enum TiposVar { INT, BOOL, CHAR, FLOAT }

---

package Principal; public class Token { private TiposToken tipo; private String lexeme; private int linha; private int coluna;

```
public Token(TiposToken tipo, String lexeme, int linha, int coluna) {
    this.tipo = tipo;
    this.lexeme = lexeme;
    this.linha = linha;
    this.coluna = coluna;
}

public TiposToken getTipo() {
    return tipo;
}

public String getLexeme() {
    return lexeme;
}

public int getLinha() {
```

```
    return linha;
}

public int getColuna() {
    return coluna;
}

@Override
public String toString() {
    return "Token{" +
            "tipo=" + tipo +
            ", lexeme='" + lexeme + '\'' +
            ", linha=" + linha +
            ", coluna=" + coluna +
            '}';
}
```

}

---

package Principal; import java.util.Comparator;

public class TokenComparator implements Comparator { @Override public int compare(Token token1, Token token2) { int linhaComp = Integer.compare(token1.getLinha(), token2.getLinha()); if (linhaComp != 0) { return linhaComp; } else { return Integer.compare(token1.getColuna(), token2.getColuna()); } } }

---

package Nodes;

public abstract class Node {}

---

package Nodes;

public class NodeAtribuicao extends Node { private Node identificador; private Node expressao;

```
public NodeAtribuicao(Node identificador, Node expressao) {
    this.identificador = identificador;
    this.expressao = expressao;
}

public Node getIdentificador(){
    return identificador;
}

public Node getExpressao(){
    return expressao;
}
```

}

---

package Nodes;

import java.util.ArrayList; import java.util.List;

public class NodeBloco extends Node { private List children;

```
public NodeBloco() {
    children = new ArrayList<>();
}

public void adicionar(Node child) {
    children.add(child);
}

public List<Node> getChildren(){
    return children;
}
```

}

---

package Nodes;

import Principal.Token;

public class NodeComparacao extends Node { private Token condicao; private Node esq; private Node dir;

```
public NodeComparacao(Token condicao, Node esq, Node dir) {
    this.condicao = condicao;
    this.esq = esq;
    this.dir = dir;
}

public Token getCondicao(){
    return condicao;
}

public Node getEsq(){
    return esq;
}

public Node getDir(){
    return dir;
}
```

}

---

package Nodes;

import Principal.TiposVar; import Principal.Token;

public class NodeIdentificador extends Node{ private TiposVar tipo; private Token identificador;

```
public NodeIdentificador(TiposVar tipo, Token identificador){
    this.tipo = tipo;
    this.identificador = identificador;
}
```

```
public TiposVar getTipo(){
    return tipo;
}

public Token getIdentificador(){
    return identificador;
}
```

}

```
package Nodes;

public class NodeIf extends Node { private Node condicao; private Node ifBloco;
private Node elseBloco;

public NodeIf(Node condicao, Node ifBloco, Node elseBloco) {
    this.condicao = condicao;
    this.ifBloco = ifBloco;
    this.elseBloco = elseBloco;
}

public Node getCondicao(){
    return condicao;
}

public Node getIfBloco(){
    return ifBloco;
}

public Node getElseBloco(){
    return elseBloco;
}
```

}

```
package Nodes;

import Principal.Token;

public class NodeOperacao extends Node { private Token operador; private Node esq;
private Node dir;

public NodeOperacao(Token operador, Node esq, Node dir) {
    this.operador = operador;
    this.esq = esq;
    this.dir = dir;
}

public Token getOperador(){
    return operador;
}

public Node getEsq(){
```

```
    return esq;
}

public Node getDir(){
    return dir;
}
```

}

```
package Nodes;

import Principal.TiposVar; import Principal.Token;

public class NodePrint extends Node{ private String tipo; private Token valor;

public NodePrint(Token valor, Token tipo) {
    this.valor = valor;
    this.tipo = tipo.getLexeme();
}

public Token getValor(){
    return valor;
}

public String getTipo(){
    return tipo;
}
```

}

```
package Nodes;

import Principal.TiposVar; import Principal.Token;

public class NodeValor extends Node { private TiposVar tipo; private Token valor;

public NodeValor(Token valor, TiposVar tipo) {
    this.valor = valor;
    this.tipo = tipo;
}

public String getValor(){
    return valor.getLexeme();
}

public TiposVar getTipo(){
    return tipo;
}
```

}

```
package Nodes;

public class NodeWhile extends Node { private Node condicao; private Node bloco;
```

```
public NodeWhile(Node condicao, Node bloco) {
    this.condicao = condicao;
    this.bloco = bloco;
}

public Node getCondicao(){
    return condicao;
}

public Node getBloco(){
    return bloco;
}
```

}

Referências

ALFRED, Aho. Compilers: Principles, Techniques, and Tools. Boston, EUA: Pearson Education, 1986. ISBN 0-321-48681-1.

The GNU C Reference Manual. https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Expressions. Acesso em 04 junho 2023

KERNIGHAN, Brian W.; Ritchie, Dennis M (1978). The C Programming Language (em inglês). Upper Saddle River, New Jersey: Prentice hall. 228 páginas