PONTIFÍCIA UNIVERSIDADE CATÓLICA DE SÃO PAULO
FACULDADE DE CIÊNCIAS EXATAS E TECNOLOGIA
CIÊNCIA DA COMPUTAÇÃO

BEATRIZ LOPES RIZZO
JOÃO PEDRO BASSO COURA
JOÃO PEDRO FIGOLS NECO
JÚLIA GACHIDO SCHMIDT
LEONARDO FAJARDO GRUPIONI

RELATÓRIO FINAL
LINGUAGEM MINERVA

SÃO PAULO

2024

BEATRIZ LOPES RIZZO

JOÃO PEDRO BASSO COURA

JOÃO PEDRO FIGOLS NECO

JÚLIA GACHIDO SCHMIDT

LEONARDO FAJARDO GRUPIONI

TUTORIAL DA LINGUAGEM MINERVA

Trabalho da disciplina Compiladores, apresentado à Pontifícia Universidade Católica de São Paulo - PUCSP

Professor: Italo Santiago Vega

SÃO PAULO

2024

SUMÁRIO

1. INTRODUÇÃO

Minerva é uma linguagem de programação de alto nível, procedural e imperativa, inspirada em pseudocódigo e nas linguagens Python, C# e Pascal. Ela tem como princípio seu fácil entendimento para falantes da língua portuguesa e para aqueles que são iniciantes na área da programação, tendo cunho educacional.

A linguagem Minerva é fortemente e estaticamente tipada, permitindo que os erros sejam detectados no momento de compilação, tal qual Java.

2. TUTORIAL DA LINGUAGEM

Inspirado em The C Programming Language, de Kernighan e Ritchie, o objetivo desse tutorial é demonstrar os principais elementos da linguagem Minerva, exemplificando-os com trechos de códigos, permitindo com que seus conceitos, que podem ser abstratos para um iniciante, tornem-se mais palpáveis. Seu propósito é permitir que qualquer pessoa consiga escrever programas em Minerva, já que esse tutorial enfatiza a parte prática. Nele, serão abordados tópicos como: declaração de variáveis, procedimentos e funções, expressões aritméticas e lógicas e entrada e saída de dados.

2.1 Primeiro programa

Assim como em toda linguagem, o primeiro programa a ser escrito é o mesmo: o que imprime `Hello, world` na tela. Em Minerva, o trecho de código que realiza essa tarefa é o seguinte:

```
procedimento principal(){
     imprima(“Hello, world”);
}
```

Deve-se criar um programa em um arquivo cujo nome termine em ".mi", como hello.mi, então compila-se com o comando:

```
mc hello.min
```

Se o programa não apresenta nenhum erro semântico ou sintático, o compilador irá gerar um arquivo executável chamado `hello.algo`. Ao rodar esse comando, irá produzir como saída:

```
hello, world
```

A partir desse exemplo inicial, pode-se perceber o funcionamento de algumas partes principais de um programa escrito em Minerva: independentemente de seu tamanho, ele consiste em procedimentos, funções e variáveis, resumidamente. Um procedimento e uma função contém um bloco de código, o qual é definido por um par de chaves `{...}`, que será executado, podendo retornar um valor ou não. As variáveis, por sua vez, armazenam valores.

No código acima, o exemplo de procedimento é o `principal`. No caso geral, o nome dos procedimentos (e de funções) é arbitrário. Porém, o `principal` tem uma particularidade: a execução do programa, obrigatoriamente, começará por esse procedimento, que no exemplo, não recebe argumentos passados por parâmetros, o que é indicado pela lista vazia `()`.

Tanto uma função como um procedimento são chamados a partir de seus nomes, seguidos de sua respectiva lista de argumentos, escrita entre parênteses. No caso do procedimento `imprima`, que é nativo da linguagem, o argumento é a cadeia de caracteres "`Hello, world`", a qual será impressa na tela, sendo o output do programa.

2.2 Variáveis e Expressões Aritméticas

O próximo programa consiste na aplicação de expressões aritméticas para o cálculo da tabuada de um número em específico.

Esse programa aborda, além do procedimento principal, novos conceitos, como declarações, expressões aritméticas, comandos de iterações e comentários.

```
#imprime a tabuada de um numero especifico#

procedimento principal(){
     int num, max, count, mult;

     num <- 2; #numero da tabuada#
     max <- 10;#maximo em que o numero sera multiplicado#
     count <- 1; #verifica se chega ao max#

     enquanto(count <= max){
          mult <- num * count;
          imprima(mult);
          count <- count + 1;
     }
}
```

No começo do código, está sendo representado um comentário, que nesse caso descreve o que o programa executará. Basicamente, o compilador ignora qualquer caractere escrito entre `#`, portanto recomenda-se abusar dos comentários para o melhor entendimento do código.

Em Minerva, todas as variáveis devem ser declaradas antes de serem usadas, de maneira que seja esclarecido o tipo e o nome de cada variável. É possível, também, declarar múltiplas variáveis como uma lista, tal qual o caso abaixo, onde representa-se números inteiros (`int`):

```
int num, max, count, mult;
```

No código, observa-se a utilização do tipo `int`, que representa números inteiros . Porém, além dele, Minerva suporta os seguintes tipos de dados: `dec`, `carac` e `bool`. Respectivamente, temos a representação de números decimais (ponto flutuante), caracteres, e booleanos - que assumem os valores `verdadeiro` ou `falso`. O tipo `carac` possui uma particularidade: caso seja um único caractere, então ele deve ser colocado entre aspas simples, caso contrário, se ele for uma cadeia de caracteres, então será colocado entre aspas duplas.

Ademais, a instrução de atribuição é feita a partir do operador `<-`. A variável localiza-se no lado esquerdo, ao passo que o valor atribuído à ela deve ficar na direita. No exemplo apresentado anteriormente, as variáveis `num`, `max` e `count` têm os valores `2`, `10` e `1`, respectivamente atribuídos à elas.

```
num <- 2;
max <- 10;
count <- 1;
```

A linguagem Minerva possui dois principais operadores aritméticos: `+` (soma) e `*` (multiplicação). Abaixo, pode-se ver a implementação desses dois operadores:

```
x <- a + b;
y <- a * b;
```

Nesse caso é atribuído à variável `x` o resultado da soma dos valores armazenados nas variáveis `a` e `b`. Similarmente, `y` armazena o resultado da multiplicação entre `a` e `b`.

O loop `enquanto` funciona da seguinte maneira: ele contém uma pré-condição entre parênteses que, enquanto for verdadeira (`count` menor ou igual

a `max`), é executado o bloco de código dentro do loop (as três operações entre as chaves). Ao finalizar, a condição será checada novamente e, se verdadeiro, executa o bloco de novo. Porém, se a condição resultar no valor booleano falso (`count` maior que `max`), o loop acaba (ou nem é executado, caso seja o primeiro teste) e a execução segue para as operações após o loop, caso existam.

O corpo do loop pode conter uma ou mais operações entre chaves, ou com apenas uma operação, não são necessárias as chaves, como:

```
enquanto (n > 5)
     n = n + 2;
```

Em ambos os casos, as instruções controladas pelo `enquanto` serão indentadas por uma tabulação (4 espaços) para facilitar a visualização de quais trechos de código estão dentro do loop. A indentação para o compilador MC não é relevante, mas ajuda a visualização e a leitura do código.

2.3 Repetição, Comandos de Seleção, Expressões Relacionais/Lógicas, Funções e Procedimentos.

Para melhor demonstrar e explicar o uso de comandos de seleção, repetição, funções e procedimentos, será utilizado o código de implementação da função fatorial:

```
funcao int fatorial(int numero);

procedimento principal(){
     int resultado;
     int numero;
     leia(numero);
     resultado <- fatorial(numero);
     se (resultado = -1) entao {
```

```
            imprima("erro");
        } senao {
            imprima(resultado);
        }
    }

    funcao int fatorial(int numero){
        int resultado <- 1;
        int i;
        se (numero < 0) entao{
            resultado <- -1;
        } senao se (numero > 0) entao {
            para i de numero ate 1 passo -1 faca {
                resultado <- resultado * i;
            }
        }
        retorna resultado;
    }
```

2.3.1 Repetição

O `para (i) de (numero) ate (1) passo (-1)` do código acima representa um loop de contagem controlada, onde a variável `i` possui um inicio `(numero)`, um fim `(1)` e um incremento fixo, o `-1`. Ou seja, a variável definida na repetição será decrementada até atingir o valor final, e nesse processo, estará sendo executado o bloco de instruções correspondente.

Além desse comando de repetição, a linguagem Minerva apresenta o `enquanto`, que foi apresentado no tópico 1.2, e ainda o `repita … até que`, o qual representa uma pós-condição. Ou seja, o bloco de código presente no loop sempre será executado no mínimo uma vez, já que a condição será verificada no

final do bloco, podendo ter como resultado `verdadeiro` ou `falso`. Ao ser avaliada, caso a expressão retorne `falso`, a execução terminará.

2.3.2 Comandos de Seleção

Os comandos se, então e senão tem como objetivo permitir que sejam tomadas decisões para que o programa tenha caminhos diferentes a depender de uma ou mais verificações, por meio da seleção. No código em questão, há a utilização do `se (resultado = -1) entao { imprima("erro"); } senao {...}`. Ou seja, após o `se`, dentro dos parênteses é feita uma verificação: caso o `resultado` seja igual a -1, então a expressão ao ser avaliada produz o valor booleano `verdadeiro`, e assim, é executado o primeiro bloco de comandos, definido entre o abre e fecha chaves, após o `entao`. Caso contrário, se a variável `resultado` for diferente de -1, a expressão ao ser avaliada retorna `falso` e então é executado o segundo bloco de comandos delimitado entre as chaves, após o `senão`.

Há também a possibilidade de fazer uma sequência de comandos de seleção, sendo ela a seleção aninhada, que permite a utilização do `senão se`, como demonstrado no exemplo abaixo do cálculo do IMC-Simplificado:

```
carac resultado[20];
double IMC;
leia(IMC);
se (IMC < 18.5)então {
     resultado <- “Abaixo do peso”;
} senão se (IMC < 25) então {
     resultado <- “Peso normal”;
} senão se (IMC < 30) então {
     resultado <- “Sobrepeso”;
} senão {
```

```
        resultado <- "Obesidade";
    }
    imprima(resultado);
```

O uso do `senão se (IMC < 25) então` permite que, ao em vez de entrar em um bloco de comandos, caso a expressão ao ser avaliada produza o valor booleano falso, haja uma nova verificação com o comando de seleção subsequente, `se (IMC < 25)`. Possibilitando, assim, com que as verificações possam ocorrer em um fluxo de execução começando pelo primeiro `se` e, caso o valor retornado pela avaliação da expressão seja `falso`, as próximas condições serão avaliadas até que haja um valor `verdadeiro`, onde será executado o bloco correspondente, ou, em último caso, alcance o último bloco, o do `senão`, executando-o e terminando as avaliações.

2.3.3. Expressões relacionais/lógicas

Além das expressões aritméticas, a linguagem Minerva comporta a utilização de expressões relacionais/lógicas, que permitem comparações entre valores dentro de condições, que podem estar presentes tanto nos comandos de seleção, quanto nos comandos de iteração.

Desta maneira, pode-se utilizar para comparação de valores os operadores `/\` representando o valor lógico "e", `\/` como "ou", `=` como "igual", `>` como "maior" e `<` representa o valor "menor". É possível, também, seu uso de maneira conjugada, dentro de uma expressão: `<=` para "menor ou igual" e `>=` para "maior ou igual". Toda expressão lógica pode retornar dois valores, sendo eles `verdadeiro` ou `falso`.

O uso do `/\` para representação da conjunção lógica "e", e `\/` para a disjunção lógica "ou", permite a utilização de mais de uma comparação dentro de uma expressão relacional/lógica, como demonstrado no código abaixo:

```
bool a <- (1 < 4 /\ 1 > 2);
bool b <- (1 < 4 \/ 1 > 2);
imprima(a);
imprima(b);
```

No código em questão, ao imprimir o valor de `a`, o output será o valor booleano `falso`, enquanto que, ao imprimir o valor presente na variável b, o output será o valor `verdadeiro`. Isso porque, ao avaliar a expressão 1 < 4 o valor a ser produzido é o `verdadeiro`, enquanto o valor a ser produzido na verificação 1 > 2 é o valor `falso`, portanto se houver um `/\` (e) entre as duas expressões, então ambas têm que ser verdadeiras para que a expressão produza o valor `verdadeiro`, caso contrário, produzirá o valor `falso`. Entretanto, caso haja um `\/` (ou), ao menos uma das expressões em questão tem que produzir o valor `verdadeiro` ao ser avaliada para que a expressão completa seja verdadeira.

2.3.4. Funções e Procedimentos

A linguagem Minerva possui duas formas de modularização do código: utilizando funções ou procedimentos. Na matemática, uma função mapeia elementos do domínio no contradomínio, produzindo uma imagem. Uma função, necessariamente, produz um valor de retorno, ao passo que um procedimento não.

Na linguagem Minerva, é obrigatório a existência de um procedimento principal, que corresponde ao início da execução do programa.

Tanto a função quanto o procedimento têm uma assinatura semelhante. No caso da função `funcao int fatorial(int numero);` primeiro vem a palavra reservada `funcao`, indicando que haverá um retorno que encerra o bloco de comandos. Em seguida, é declarado o tipo de retorno, nesse caso `int`. Depois, vem o nome da função que deve sempre começar por letras de [a-zA-Z] podendo conter dígitos e/ou letras, tendo nesse caso o nome `fatorial`, seguido de abre e fecha parênteses contendo dentro uma lista com as variáveis de parâmetro, que pode ser desde nenhuma variável, até um conjunto de variáveis separadas por vírgula `(int`

`var1, carac var2, int var3)`. Na função apresentada como exemplo, há apenas uma variável de parâmetro: `int numero`.

Enquanto isso, para o procedimento, não é possível indicar um tipo de retorno, já que ele não existe. Além disso, é preciso escrever a palavra reservada `procedimento` antes de seu nome. Os outros elementos da assinatura do procedimento (variáveis de parâmetro e nome), seguem as mesmas regras de uma função, apresentadas no parágrafo anterior.

É importante ressaltar que, no início do programa, antes do `procedimento principal()` deve aparecer todas as assinaturas de funções e procedimentos, como uma interface. Após o corpo do procedimento principal, a implementa-se tanto os procedimentos quanto as funções. Uma assinatura é sempre terminada por `;` , enquanto que a implementação sempre deve conter um abre e fecha chaves, onde estará contido suas respectivas instruções, após os parênteses, que delimitam as variáveis de parâmetro.

2.3.5 Entrada e Saída

Para a entrada e saída de dados, a linguagem Minerva é contemplada com dois comandos: `leia()` para entrada de dados via terminal e `imprima()`, para escrever no terminal.

Ao compilar um programa e executá-lo, quando houver um comando `leia()`, o programa irá aguardar até que haja um valor de input no terminal para poder prosseguir com a execução.

```
int numero; leia(numero);
```

O valor será armazenado em uma variável, que deve ter sido declarada antes do comando. No exemplo, o valor do input será armazenado na variável `numero`, do tipo `int`, já que ela é o parâmetro de `leia`.

Já `imprima`, quando chamado na linguagem Minerva, retorna o valor da avaliação de uma expressão ou variável no terminal. Por exemplo, no programa que

descreve função fatorial, ocorre tanto o `leia(numero);` quanto o `imprima(resultado);` que dentro do contexto desse programa, mostrado no tópico 1.3, ao receber um input `6` produz o output `720`.

2.6 Array de Caracteres

Supondo que deseja-se escrever uma frase, ao invés de utilizar diversas variáveis do tipo `carac`, é possível criar um único array de caracteres. Para ilustrar esse cenário, pode-se recorrer ao primeiro programa apresentado neste tutorial:

```
procedimento principal(){
      imprima("Hello, world");
}
```

Modificando-o, é possível exemplificar a utilização de um array de caracteres da seguinte maneira: cria-se um array que armazena a frase `"Hello, world"`, e posteriormente, com o uso do comando `imprima()`, a sentença será impressa na tela. Com as alterações, tem-se:

```
procedimento principal(){
      carac frase[13] <- "Hello, world";
      imprima(frase);
}
```

Conclui-se que, para criar um array, deve-se especificar qual o tipo de dados, seu nome e o tamanho do array.

3. MANUAL DE REFERÊNCIA

3.1 Introdução

Este manual de referência, criado para a linguagem Minerva, é baseado no The C Programming Language, de Kernighan e Ritchie, Apêndice A.

3.2 Convenções Léxicas

3.2.1 Tokens

Existem 6 classes de tokens: identificadores, palavras reservadas, constantes, strings literais, operadores e outros separadores. Espaços em branco, tabulações verticais e horizontais e comentários são ignorados, exceto quando separam tokens. Todavia, um espaço é necessário para separar identificadores, palavras reservadas e constantes.

3.2.2 Comentários

O dígito # inicia um comentário, para finalizá-lo é necessário o segundo uso do #. O que estiver contido entre eles é ignorado pelo compilador.

3.2.3 Identificadores

Um identificador é uma sequência de letras e dígitos, que o primeiro dígito deve ser uma letra. Letras maiúsculas e minúsculas são consideradas diferentes. Identificadores podem ter qualquer comprimento.

3.2.4 Palavras Reservadas

Os seguintes identificadores são palavras reservadas, e não podem ser utilizados de outra forma:

`bool`	`imprima`	`de`	`verdadeiro`
`int`	`leia`	`enquanto`	`falso`
`dec`	`ate`	`faca`	`funcao`
`carac`	`entao`	`para`	`procedimento`
`passo`	`principal`	`procedimento`	`que`
`repita`	`retorna`	`se`	`senao`

3.2.5 Constantes

Existem diferentes tipos de constantes, cada um com seu tipo de dado, sendo eles:

inteiro - constante
decimal - constante
caracter - constante
booleano - constante

3.3 Tipos Básicos

A linguagem Minerva contém 4 tipos básicos de dados, sendo eles:
Tipos de Dados:

`int` - inteiro (16 bits)
`dec` - decimal (64 bits)
`carac` - carácter (8 bits)
`bool` - booleano (1 bit)

3.4 Expressões

3.4.1 Chamadas de Função e Procedimento

Para chamar funções e procedimentos na linguagem Minerva, é necessário que eles estejam dentro do escopo do procedimento principal ou dentro do escopo de outras funções/procedimentos, as quais são diferentes da função/procedimento que está sendo chamado. Ou seja, uma função/procedimento não chama a si mesmo.

Para realizar esse processo, é necessário escrever o nome da função/procedimento, seguido por suas variáveis de parâmetros, delimitadas pelo abre e fecha de parênteses. Portanto, é possível passar como parâmetro valores ou variáveis. Observa-se, todavia, que a quantidade, existência e tipo destes deve obedecer à assinatura da função ou procedimento.

Entretanto, uma função diferencia-se de um procedimento no seguinte mérito: como uma função sempre retorna um valor (porque mapeia um elemento do domínio no contradomínio), então ela deve, quando chamada, estar associada a uma operação de atribuição, a um comando de seleção ou, ainda, a uma expressão.

Abaixo, observa-se a chamada de um procedimento e de uma função:

```
<nome do procedimento>([<variáveis de parâmetro>]);
<nome da função>([<variáveis de parâmetro>])
```

3.4.2 Operadores Aditivos e Multiplicativos

Os operadores principais presentes em Minerva são `+`, para representar a soma e `*`, para representar a multiplicação. Ambos sempre são utilizados em uma expressão aritmética, a qual pode envolver os tipos de dados `int` e `dec`, seguindo as normas padrões da matemática para a ordem das operações.

```
<valor 1> * <valor 2>
<valor 1> + <valor 2>
```

3.4.3 Operadores Relacionais e Lógicos

Em Minerva é possível utilizar 2 operadores relacionais e 5 lógicos. Eles são sempre utilizados dentro de expressões lógicas que, quando avaliadas, sempre produzem os valores booleanos: `verdadeiro` ou `falso`. Abaixo encontram-se os operadores:

Relacionais:

`/\` (e)

`\/` (ou)

Lógicos:

= (igual)

> (maior)

< (menor)

>= (maior igual)

<= (menor igual)

3.5 Declarações

Para a declaração de variáveis na linguagem Minerva, é necessário indicar inicialmente o tipo de dado básico, seguido pelo nome da variável e terminado por um ponto e vírgula após o nome da variável.

```
int <nome da variável>;
dec <nome da variável>;
carac <nome da variável>;
bool <nome da variável>;
```

3.5.1 Tipos Específicos

A linguagem Minerva contempla apenas um tipo específico de dado, sendo ele o array de caracteres, que permite armazenar em um array sequencial um conjunto de valores do tipo `carac`, terminados por '\0'.

3.5.2 Declaração de Array

Para declarar um array na linguagem Minerva é necessário utilizar o tipo de dados básico `carac`, seguido por seu nome, pelo tamanho do array (entre colchetes), como demonstrado abaixo:

```
carac <nome da array>[<tamanho da array>];
```

3.5.3 Inicialização

Para inicializar uma variável na linguagem Minerva, é necessário indicar primeiramente o tipo de dado básico, seguido pelo nome da variável. Depois, utiliza-se o símbolo de atribuição `<-` e um valor do mesmo tipo (que será atribuído a variável).

```
int <nome da variável> <- <valor do tipo int>;
dec <nome da variável> <- <valor do tipo dec>;
carac <nome da variável> <- <valor do tipo carac>;
bool <nome da variável> <- <valor do tipo bool>;
```

3.5.4 Declaração de Seleção

Para tomadas de decisões, é possível utilizar três tipos de comandos: `se`, `senão`, `senão se`.

1) A sintaxe do comando `se` é representada da seguinte maneira:

```
se (<expressão>) então {
      <bloco>
      ...
}
```

Nesse caso, se a expressão, ao ser avaliada, produzir um valor `verdadeiro`, então o bloco será executado.

2) Para `se (`<expressão>`)então {...} senao`, temos:

```
se (<expressão>) então {
      <bloco1>
      ...
} senao {
      <bloco2>
      ...
}
```

Dessa vez, se a expressão produzir um valor `falso`, então, o bloco2 será executado.

3) Para comandos de seleção aninhados:

```
se (<expressão 1>) então {
      <bloco 1>
      ...
} senao se (<expressão 2>){
      <bloco 2>
      ...
} senao se (<expressão 3>){
      <bloco 3>
      ...
} senao{
      <bloco 4>
      ...
}
```

O fluxo de execução começa pelo primeiro `se` e, caso o valor retornado pela avaliação da expressão seja `falso`, as próximas condições serão avaliadas até que haja um valor `verdadeiro`, onde será executado o bloco correspondente, ou, em último caso, alcance o último bloco, o do `senão`, executando-o e terminando as avaliações.

3.5.5 Declaração de Iteração

Para repetições, Minerva oferece três possibilidades: `enquanto {...}`, `repita {...} até que ...` e `para ... de ... ate ... passo ... faca {...}`.

No primeiro caso, utilizando `enquanto`, a sintaxe é representada da seguinte maneira:

```
enquanto (<condição>){
      <bloco>
      ...
}
```

Nesse caso, representa-se uma pré-condição, no qual, enquanto a condição for `verdadeira`, é executado o bloco. Porém, se a condição resultar no valor booleano `falso`, esse bloco nunca será executado.

No segundo caso, utiliza-se `repita {...} até que` … do seguinte modo:

```
repita {
      <bloco>
      ...
} ate que (<condição>);
```

Esse caso exemplifica uma pós-condição, ou seja, o bloco sempre será executado no mínimo uma vez. Após a primeira execução, ele verificará se a

condição é `verdadeira` para poder realizar novamente o mesmo bloco até que a condição seja `falsa`.

Já no terceiro caso, `para ... de ... ate ... passo ... faca {...}` é utilizado da seguinte maneira:

```
para (<variável>) de (<início>) até (<fim>) passo (<incremento>) faca {
    <bloco>
    ...
}
```

Assim, é exemplificado um loop de contagem controlada onde a variável passada terá seu início e seu fim estabelecidos e um incremento fixo. Ou seja, o bloco será executado até que a variável alcance o valor final estabelecido.

3.5.6 Declaração de Funções e Procedimentos

Na matemática, uma função mapeia elementos do domínio no contradomínio, produzindo uma imagem. A partir dessa ideia, Minerva possui duas definições: função e procedimento, assim como Pascal.

Uma função necessariamente produz um valor de retorno, ao passo que um procedimento não.

Na linguagem Minerva, é obrigatório a existência de um procedimento principal, que corresponde ao início da execução do programa.

A representação formal de cada um é:

```
procedimento <nome do procedimento> ([<variáveis de parâmetro>]){...}
```

```
funcao <tipo de dado> <nome da função> ([<variáveis de parâmetro>]){
    ...
    retorna <variável ou expressão>;
}
```

3.6 Gramática

A gramática é um conjunto de regras propostas para ser seguidas ao montar uma determinada linguagem, sendo ela composta por regras de produção, símbolos terminais, não-terminais e símbolo inicial. Para o auxílio da criação da linguagem Minerva, foi criada a seguinte gramática:

funcao -> **funcao** tipo nomeMetodo **(** tipo variável **);**
funcao -> **funcao** tipo nomeMetodo **(** (tipo variável)* **) {** bloco **retorna** (número | variável) **}**

procedimento -> **procedimento** nomeMetodo **(** tipo variável **);**
procedimento -> **procedimento** nomeMetodo **(** (tipo variável)* **) {** bloco **}**

chamadaMetodo - variável **(** variável **) ;**

declaração -> tipo variável **;**
declaração -> tipo variável **<-** (**verdadeiro** | **falso**) **;**
declaração -> tipo variável **<-** termo **;**

atribuiçãoVariável -> chamadaMetodo;
atribuiçãoVariável -> variável **<-** (**verdadeiro** | **falso**) **;**
atribuiçãoVariável -> variável **<-** termo **;**

repetiçãoEnquanto -> **enquanto (** condição **)**
repetiçãoRepitaAte -> **repita {** bloco **} ate que (** condição **)**
repetiçãoRepitaPara -> **para** variável **de** (variável | número) **ate** (variável | numero) **passo** número **faca**

instrução -> **se (**condição**) entao {** bloco **}**
instrução -> **se (**condição**) entao {** bloco **} senao {** bloco **}**
instrução -> **se (**condição**) entao {** bloco **} senao se {** bloco **} senao {** bloco **}**

condição -> expressao (**=** | **<** | **>** | **<=** | **>=**) expressao
expressão -> termo ((**+** | **-** | *****) termo)*
termo -> variável | numero (**.** numero)* | **(** expressão **)**

bloco -> (declaração | chamadaMetodo | instrução | repetição)*
variável -> **ID**
nomeMetodo **-> ID**
numero -> **NUM**
tipo -> **INT | DEC | CARAC | BOOL**

4. PLANO DO PROJETO

O plano do projeto baseou-se, principalmente, nos requisitos propostos pelo professor, bem como nas datas das entregas parciais. Ademais, vale ressaltar que, apesar da existência dos papéis de cada integrante do grupo, eles não foram seguidos à risca, isso porque, devido a complexidade do trabalho, julgou-se necessário a participação de todos para a estruturação de um compilador e a criação de uma linguagem.

A princípio, tem-se cinco requisitos:

1. definição de variáveis e tipos de dados
2. definição de procedimentos com parâmetros
3. expressões aritméticas, relacionais e lógicas
4. escopos global e local
5. geração de código para WebAssembly

Em relação ao terceiro requisito, em primeiro momento, considerou-se expressões aritméticas apenas a soma e multiplicação. A partir dos requisitos, pode-se ter uma ideia de como iniciar o projeto. Em primeiro lugar, decidiu-se a parte léxica, ou seja, a forma na qual a linguagem Minerva seria escrita. Posteriormente, foi possível raciocinar sobre os analisadores léxico e sintático.

5. EVOLUÇÃO DA LINGUAGEM

Na primeira versão da linguagem Minerva, foi desenvolvido os seguintes tipos de dados:

1. int
2. dec
3. carac
4. texto
5. bool

Entretanto, após a primeira apresentação, foi estabelecido que não seria necessário o tipo texto, e que ele poderia ser descartado, já que apenas com o tipo `carac` seria possível trabalhar com cadeias de caracteres. Então, tem-se como tipo de dados:

1. int
2. dec
3. carac
4. bool

Além disso, na primeira versão, tinha-se quatro operações aritméticas: soma, subtração, multiplicação e divisão. Porém, no projeto final, a divisão foi removida. As outras características da linguagem não sofreram nenhuma mudança.

6. ARQUITETURA DO COMPILADOR

Idealmente, a arquitetura do compilador segue o modelo padrão:

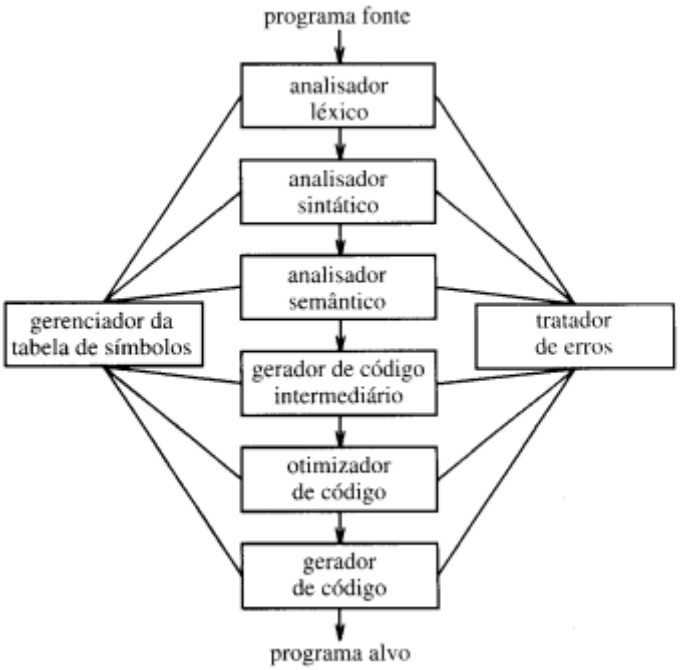

Figura 1: representação das fases de um compilador. Disponível em: AHO, A. V.; SETHI, R.; ULLMAN, J.D. Compiladores: Princípios, técnicas e ferramentas. 2ª ed. São Paulo: Pearson, 2007.

7. AMBIENTE DE DESENVOLVIMENTO

A linguagem Minerva foi implementada por meio da linguagem Java, sendo desenvolvida no BlueJ.

8. PLANO DE TESTES

Primeiramente, o grupo buscou testar as várias possibilidades de implementação que um usuário pode realizar ao utilizar a linguagem Minerva. Para isso, baseando-se na gramática estabelecida, pequenos trechos de código na linguagem programada foram escritos com o objetivo de verificar o funcionamento de determinados conceitos lógicos.

Para começar os testes, foi decidido utilizar um código que representa, no mundo da programação, o primeiro de muitos a ser ensinado ao se falar de uma linguagem:

```
procedimento principal(){
     imprima(“Hello, world”);
}
```

Além disso, uma variação do mesmo também passou na fase de testes com o objetivo de observar o comportamento de um array para atribuições e seu uso como parâmetro em comandos de `imprima`:

```
procedimento principal(){
     carac frase[13] <- “Hello, world”;
     imprima(frase);
}
```

Posteriormente, trechos de códigos um pouco mais complexos foram testados a fim de observar o comportamento de outros tipos de dados como `int`, `dec`, `bool`, além de chamadas de funções previamente declaradas. Segue abaixo o trecho utilizado para isso:

```
funcao int teste(int i);

procedimento principal(){
     int t;
     int i <- 1;
     int k <- 2;
     int soma <- i + (k * 10);

     bool fim <- verdadeiro;

     soma <- ((9+10)*80)-10;

     dec l <- 1.9;

     teste(i);
}
```

Além disso, antes de testar códigos ainda mais complexos, foi testados mais possibilidades de declarações e atribuições de variáveis, assim como chamadas de métodos e a utilização de instruções de seleção:

```
procedimento cadastro(int nome);
funcao int cadastro(int nome);

procedimento principal(){
     int j;
     int nome <- 0;
     int j <- nome + 1;
     int k <- 2 + 1;
     int j <- k;
     bool l <- verdadeiro;

     cadastro(nome);
     i <- i + 1;

     se(1+2 <= 1+2) entao{}
     se(i <= 1+2) entao{}
     se(i) entao{}
     se(10 > i) entao{}
}
```

Por fim, foram testados códigos com comandos de repetição e com mais de uma função, para assim conseguir testar todos os outros comandos presentes na linguagem, sempre tentando identificar os erros no decorrer do programa escrito.

9. CONCLUSÕES

A partir da criação de uma linguagem, foi possível aprender não somente sobre a arquitetura de um compilador e as etapas de compilação, mas também sobre expressões regulares, autômatos e gramáticas.

Durante o desenvolvimento da linguagem, diversas dúvidas conceituais surgiram, levando-nos a buscar respostas para elas, o que, consequentemente, nos guiou a um estudo mais aprofundado na área de compiladores, com destaque para o livro “Compiladores: Princípios, técnicas e ferramentas”, de Alfred V. Aho.

Em resumo, o desenvolvimento da linguagem Minerva não apenas ampliou nosso entendimento de conceitos de programação, como também nos instigou a explorar novos campos do conhecimento.

10. APÊNDICE

Classe Main:

```
public class main{
     public static void main(String Args[]){
               Sintatica app = new Sintatica();
               app.programa();
     }
}
```

Classe Entrada:

```
import java.util.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import javax.swing.JOptionPane;

public class Entrada
{
    public String lerArq(){
        String codigo = "";
         String nome = coletarString("nome do arquivo") +
      ".txt";
         try{
            FileReader arq = new FileReader(nome);
            BufferedReader lerArq = new
BufferedReader(arq);

            String line = null;
            line = lerArq.readLine();

            codigo = codigo + line;

            while (line != null) {
                 line = lerArq.readLine();
                 codigo = codigo + line;
            }

             arq.close();
         }catch (IOException e) {
             System.err.printf("Erro na abertura do
arquivo: %s.\n",
             e.getMessage());
         }
```

```
                return codigo;
            }

            public int coletarInt(String txt){
                  return
Integer.parseInt(JOptionPane.showInputDialog(txt+""));
            }

            public String coletarString(String txt){
                return JOptionPane.showInputDialog(txt+"");
            }

            public void mostrar(String txt){
                JOptionPane.showMessageDialog(null, txt+"");
            }
        }
```

Enum nome:

```
public enum nome {
      NUM, ID, MAIS, MENOS, PTO, PTV, IGUAL, MULT, APAR, FPAR,
      ACHAV, FCHAV, ACOL, FCOL, MAIOR, MENOR, MAIORIGUAL,
      MENORIGUAL, ATRIB, BOOL, IMPRIMA, DE, VERDADEIRO, INT,
      LEIA, ENQUANTO, FALSO, DEC, ATE, FACA, FUNCAO,CARAC,
      ENTAO, PARA, PROCEDIMENTO, PASSO, PRINCIPAL, QUE,
      REPITA,RETORNA, SE, SENAO, ARRAYC, ELCARAC, EOF
}
```

Classe Token:

```
public class Token{
    private nome tk;
    private int a;
    private int b;
    private int index;
    private int line;
    private String lexema;

    public Token(int index, int a, int b, String lexema, nome
tk, int line){
        setIndex(index);
        setA(a);
        setB(b);
        setLexema(lexema);
        setNome(tk);
        setLine(line);
```

```
}

// Getter and Setter for tk
public nome getNome() {
    return tk;
}

private void setNome(nome tk) {
    this.tk = tk;
}

// Getter and Setter for a
public int getA() {
    return a;
}

private void setA(int a) {
    this.a = a;
}

// Getter and Setter for b
public int getB() {
    return b;
}

private void setB(int b) {
    this.b = b;
}

// Getter and Setter for index
public int getIndex() {
    return index;
}

private void setIndex(int index) {
    this.index = index;
}

// Getter and Setter for line
public int getLine() {
    return line;
}


private void setLine(int line) {
```

```
            this.line = line;
        }

        // Getter and Setter for lexema
        public String getLexema() {
            return lexema;
        }

        private void setLexema(String lexema) {
            this.lexema = lexema;
        }

        public String toString(){
            return ("[@" + this.index + ", " +  this.a + "-" +
    this.b + "," + "\"" + this.lexema + "\"" + ", <" + this.tk +
    ">," + this.line +"]");
        }
    }
```

Classe PalavrasReservadas:

```
    import java.util.HashMap;

    public class PalavrasReservadas
    {
          HashMap<String, nome> palavrasReservadas = new
          HashMap<String, nome>();

        public void inicializarPalavrasReservadas(){
            palavrasReservadas.put("int", nome.INT);
            palavrasReservadas.put("bool", nome.BOOL);
            palavrasReservadas.put("imprima", nome.IMPRIMA);
            palavrasReservadas.put("de", nome.DE);
            palavrasReservadas.put("verdadeiro",
    nome.VERDADEIRO);
            palavrasReservadas.put("leia", nome.LEIA);
            palavrasReservadas.put("enquanto", nome.ENQUANTO);
            palavrasReservadas.put("falso", nome.FALSO);
            palavrasReservadas.put("dec", nome.DEC);
            palavrasReservadas.put("ate", nome.ATE);
            palavrasReservadas.put("faca", nome.FACA);
            palavrasReservadas.put("funcao", nome.FUNCAO);
            palavrasReservadas.put("carac", nome.CARAC);
            palavrasReservadas.put("entao", nome.ENTAO);
            palavrasReservadas.put("para", nome.PARA);
```

```
            palavrasReservadas.put("procedimento",
         nome.PROCEDIMENTO);
            palavrasReservadas.put("passo", nome.PASSO);
            palavrasReservadas.put("principal", nome.PRINCIPAL);
            palavrasReservadas.put("que", nome.QUE);
            palavrasReservadas.put("repita", nome.REPITA);
            palavrasReservadas.put("retorna", nome.RETORNA);
            palavrasReservadas.put("se", nome.SE);
            palavrasReservadas.put("senao", nome.SENAO);
        }

        public nome verificarPalavrasReservadas(String lexema){
            nome t;
            t = palavrasReservadas.get(lexema);
            if(t == null) t = nome.ID;
            return t;
        }
    }
```

Classe ‘Léxico’

```
import java.util.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import javax.swing.JOptionPane;

public class Lexica
{
    int indice = 1;

    private ArrayList<String> listtokens = new ArrayList<String>();

    PalavrasReservadas palavrasReservadas = new
PalavrasReservadas();

    public ArrayList<Token> AnLex(char fonte[]){
        palavrasReservadas.inicializarPalavrasReservadas();

        int a = 0, b = -1, line = 1;
        ArrayList<Token> tokens = new ArrayList<Token>();
        Token token;
        String cadeia = "", tk = "erro";
```

```
        boolean adiciona = false, sentinela = false, aspa = false,
aspaD = false, coment = false, menor = false, maior = false, entrou
= false;
        nome tipo = nome.ID;

        for(int i = 0; i < fonte.length; i++){
            var c = fonte[i];

            if(Character.toString(c).matches("#") || coment ==
true){

                cadeia += c;
                coment = true;
                if(cadeia.length() > 1 && c == '#'){
                    cadeia = "";
                    coment = false;
                } else if (i == fonte.length-1){
                    erro();
                }
            } else if(Character.toString(c).matches("\"") || aspaD
== true){
                if(!Character.toString(c).matches("\"")) cadeia +=
c;
                b++;
                aspaD = true;
                if(entrou == false) entrou = true;
                else if(Character.toString(c).matches("\"")){
                    listtokens.add("[@" + indice + ", " +  a + "-"
+ b + "," + "\"" + cadeia + "\"" + ", <" + nome.ARRAYC + ">," +
line +"]");
                    token = new
Token(indice,a,b,cadeia,nome.ARRAYC,line);
                    tokens.add(token);
                    a = b + 1; indice++; cadeia = "";
                    aspaD = false; entrou = false;
                } else if (i == fonte.length-1){
                    erro();
                }
            } else if(Character.toString(c).matches("'") || aspa ==
true){
                if(!Character.toString(c).matches("'")) cadeia +=
c;
                b++;
                //'A'
                //234
```

```
                //5-2 = 3
                aspa = true;
                if(entrou == false) entrou = true;
                else if(Character.toString(c).matches("'")){
                    listtokens.add("[@" + indice + ", " +  a + "-"
+ b + "," + "\"" + cadeia + "\"" + ", <" + nome.ELCARAC + ">," +
line +"]");
                    token = new
Token(indice,a,b,cadeia,nome.ELCARAC,line);
                    tokens.add(token);
                    a = b + 1; indice++; cadeia = "";
                    aspa = false; entrou = false;
                } else if(b-a > 2) erro();

            } else if(Character.toString(c).matches("[a-zA-Z]") ||
sentinela == true){
                cadeia += c;
                b++;

                if(i < (fonte.length - 1) &&
!Character.toString(fonte[i+1]).matches("[a-zA-Z\\d]")){
                   tipo =
palavrasReservadas.verificarPalavrasReservadas(cadeia);
                    listtokens.add("[@" + indice + ", " +  a + "-"
+ b + "," + "\"" + cadeia + "\"" + ", <" + tipo + ">," + line
+"]");
                    token = new Token(indice,a,b,cadeia,tipo,line);
                    tokens.add(token);
                    a = b + 1; indice++; cadeia = "";
                    sentinela = false;
                }else if(i < (fonte.length - 1) &&
Character.toString(fonte[i+1]).matches("\\d")){
                    sentinela = true;
                }
            }
            else if(Character.toString(c).matches("\\d")){
                cadeia += c;
                b++;

                if(i < (fonte.length - 1) &&
!Character.toString(fonte[i+1]).matches("\\d")){
                    listtokens.add("[@" + indice + ", " +  a + "-"
+ b + "," + "\"" + cadeia + "\"" + ", <" + nome.NUM + ">," + line
+"]");
```

```
                    token = new
Token(indice,a,b,cadeia,nome.NUM,line);
                    tokens.add(token);
                    a = b + 1; indice++; cadeia = "";
                }
            } else if(Character.toString(c).matches("\\s")){
                //ignorando
            } else
if(Character.toString(c).matches("[+*;\\.(){}\\[\\]]")){
                cadeia += c;
                b = a;

                String teste = Character.toString(c);
                tipo = nome.MAIS;
                switch(teste){
                    case "*":
                        tipo = nome.MULT;
                        break;
                    case ";":
                        tipo = nome.PTV;
                        break;
                    case ".":
                        tipo = nome.PTO;
                        break;
                    case "(":
                        tipo = nome.APAR;
                        break;
                    case ")":
                        tipo = nome.FPAR;
                        break;
                    case "{":
                        tipo = nome.ACHAV;
                        break;
                    case "}":
                        tipo = nome.FCHAV;
                        break;
                    case "[":
                        tipo = nome.ACOL;
                        break;
                    case "]":
                        tipo = nome.FCOL;
                        break;
                }
```

```
                listtokens.add("[@" + indice + ", " +  a + "-" + b
+ "," + "\"" + cadeia + "\"" + ", <" + tipo + ">," + line +"]");
                token = new Token(indice,a,b,cadeia,tipo,line);
                tokens.add(token);
                a = b + 1; indice++; cadeia = "";
            } else if(Character.toString(c).matches("[<>=\\-]")){
                cadeia += c;

                String teste = Character.toString(c);

                switch(teste){
                    case ">":

if(!Character.toString(fonte[i+1]).matches("\\=")){
                            tipo = nome.MAIOR;
                            b = a;
                            adiciona = true;
                        } else maior = true;
                        break;
                    case "<":

if(!Character.toString(fonte[i+1]).matches("[=\\-]")){
                            tipo = nome.MENOR;
                            b = a;
                            adiciona = true;
                        } else menor = true;
                        break;
                    case "=":
                        if(menor){
                            tipo = nome.MENORIGUAL;
                            b = a + 1;
                        }
                        else if (maior){
                            tipo = nome.MAIORIGUAL;
                            b = a + 1;
                        }
                        else {
                            tipo = nome.IGUAL;
                            b = a;
                        }
                        adiciona = true;
                        break;
                    case "-":
                        if(menor){
                            tipo = nome.ATRIB;
```

```
                            b = a + 1;
                        }
                        else {
                            tipo = nome.MENOS;
                            b = a;
                        }
                        adiciona = true;
                        break;
                }

                if(adiciona){
                    listtokens.add("[@" + indice + ", " +  a + "-"
+ b + "," + "\"" + cadeia + "\"" + ", <" + tipo + ">," + line
+"]");
                    token = new Token(indice,a,b,cadeia,tipo,line);
                    tokens.add(token);
                    a = b + 1; indice++; cadeia = "";
                    adiciona = false;
                    menor = false; maior = false;
                }

            }else{
                erro();
            }
        }
        listtokens.add("[@" + indice + ", " +  a + "-" + a + "," +
"\"EOF\"" + ", <" + nome.EOF + ">," + line +"]");
        token = new Token(indice,a,b,cadeia,nome.EOF,line);
        tokens.add(token);
        System.out.println(listtokens);
        return tokens;
    }

    private void erro(){
        System.out.println("Erro!!");
        System.exit(0);
    }
}
```

Classe Sintática:

```
import java.util.*;
import java.lang.Math;

public class Sintatica
{
```

```
 ArrayList<Object> codigo = new ArrayList<Object>();
 private boolean tipo, nomeMetodo, variavel;
 private int i, limite, countPrinc;
 private Token token;
 private No no;
 ArrayList<Token> Tokens;
 nome nome;

public void programa(){
    Entrada entrada = new Entrada();
    String teste = entrada.lerArq();
    char[] n = teste.toCharArray();

    Lexica lex = new Lexica();
    Tokens = lex.AnLex(n);

    Arvore arvSintatica = AnSint(Tokens);
}

private Arvore AnSint(ArrayList<Token> tokens){
    no = new No("Inicio");
    Arvore arv = new Arvore(no);
    limite = Tokens.size();

    nome nome;

    i = 0;
    countPrinc = 0;

    token = tokens.get(i);

    procedimento();

    return arv;
}

private void funcao(){
    if(token.getNome() == nome.FUNCAO){
            no.acrescentarFilho(token.getNome() + ": " +
            token.getLexema());

            proxToken();

            if(tipo()){
                      tipo = false;
```

```
                no.acrescentarFilho(token.getNome() + ":
                " + token.getLexema());

                     proxToken();

                     nomeMetodo(no);

          }else erro(token);

          Apar(no);

          if(tipo()){
                No param =
                no.acrescentarFilho("parametro");
                tipo = false;
                param.acrescentarFilho(token.getNome() +
                ": " + token.getLexema());
                     proxToken();

                if(variavel()){
                variavel = false;

                param.acrescentarFilho(token.getNome() +
                ": " + token.getLexema());

                proxToken();
          }else erro(token);
     }

     Fpar(no);

     if(token.getNome() == nome.PTV){

          no.acrescentarFilho(token.getNome() + ": " +
          token.getLexema());
                proxToken();

          procedimento();
     }else if(token.getNome() == nome.ACHAV){

          no.acrescentarFilho(token.getNome() + ": " +
          token.getLexema());
                proxToken();

          No bloco = no.acrescentarFilho("bloco");
```

```
                    bloco(bloco);

                    if(token.getNome() == nome.RETORNA){

                         bloco.acrescentarFilho(token.getNome() +
                         ": " + token.getLexema());
                         proxToken();

                           if(token.getNome() == nome.NUM ||
               token.getNome() == nome.ID){

                         bloco.acrescentarFilho(token.getNome() +
                         ": " + token.getLexema());
                            proxToken();

                            if(token.getNome() == nome.PTV){

                               bloco.acrescentarFilho(token.getNom
                               e() + ": " + token.getLexema());
                                proxToken();

                                if(token.getNome() ==
  nome.FCHAV){

                                    no.acrescentarFilho(token.get
                                    Nome() + ": " +
                                    token.getLexema());
                                         proxToken();

                                    Eof();
                                }else erro(token);
                            }else erro(token);
                        }else erro(token);
                    }else erro(token);
                }else erro(token);
            }
      }

private void procedimento(){
    if(token.getNome() == nome.PROCEDIMENTO){

       no.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
        proxToken();
```

```
if(token.getNome() == nome.PRINCIPAL){

    no.acrescentarFilho(token.getNome() + ": " +
    token.getLexema());
    proxToken();
    countPrinc++;

    Apar(no);
    Fpar(no);

    if(token.getNome() == nome.ACHAV){

         no.acrescentarFilho(token.getNome() + ": " +
         token.getLexema());
         proxToken();

         No bloco = no.acrescentarFilho("bloco");
         bloco(bloco);

         if(token.getNome() == nome.FCHAV){

              no.acrescentarFilho(token.getNome() + ":
              " + token.getLexema());
              proxToken();

              Eof();
         }else erro(token);
         }else erro(token);
}else {
    nomeMetodo(no);
    Apar(no);

    if(tipo()){
         No param = no.acrescentarFilho("parametro");
         tipo = false;

         param.acrescentarFilho(token.getNome() + ": "
         + token.getLexema());
         proxToken();

         if(variavel()){
              variavel = false;

              param.acrescentarFilho(token.getNome() +
              ": " + token.getLexema());
```

```
                    proxToken();
                }else erro(token);
            }

            Fpar(no);



            if(token.getNome() == nome.PTV){

                no.acrescentarFilho(token.getNome() + ": " +
                token.getLexema());
                proxToken();

                procedimento();
            }else if(token.getNome() == nome.ACHAV){

                no.acrescentarFilho(token.getNome() + ": " +
                token.getLexema());
                proxToken();

                No bloco = no.acrescentarFilho("bloco");
                bloco(bloco);

               if(token.getNome() == nome.FCHAV){

                    no.acrescentarFilho(token.getNome() + ":
                    " + token.getLexema());

                    proxToken();

                    Eof();
                }else erro(token);
            }else erro(token);
        }
    }else funcao();
}

private void Eof(){
    if(token.getNome() == nome.EOF){
        fimSintatica();
    }else procedimento();
}
```

```
    private void bloco(No bloco){
              while(token.getNome() == nome.SE || token.getNome()
== nome.ENQUANTO || token.getNome() == nome.REPITA ||
token.getNome() == nome.PARA || token.getNome() == nome.IMPRIMA ||
token.getNome() == nome.LEIA || token.getNome() == nome.ID ||
tipo()){
            if(token.getNome() == nome.SE){
                instrucao(bloco);
            }else if(token.getNome() == nome.ENQUANTO){
                repeticaoEnquanto(bloco);
            }else if(token.getNome() == nome.REPITA){
                repeticaoRepitaAte(bloco);
            }else if(token.getNome() == nome.PARA){
                repeticaoRepitaPara(bloco);
            }else if(token.getNome() == nome.IMPRIMA ||
token.getNome() == nome.LEIA){
                No comando = bloco.acrescentarFilho("comando");

                imprimaLeia(comando);
            }else if(token.getNome() == nome.ID){
                Token tokenAnt = token;

                proxToken();

                if(token.getNome() == nome.APAR){
                     No chamada =
                     bloco.acrescentarFilho("chamadaMetodo");

                     No nomeMetodo =
                     chamada.acrescentarFilho("nomeMetodo");

                     nomeMetodo.acrescentarFilho(tokenAnt.getNome(
                     ) + ": " + tokenAnt.getLexema());


                     chamada.acrescentarFilho(token.getNome() + ":
                     " + token.getLexema());
                     proxToken();

                     chamadaMetodo(chamada);
                }else{
                     No atribuicao =
                     bloco.acrescentarFilho("atribuicaoVariavel");
```

```
                    No variavel =
                    atribuicao.acrescentarFilho("variavel");

                    variavel.acrescentarFilho(tokenAnt.getNome()
                    + ": " + tokenAnt.getLexema());

                    atribuicaoVariavel(atribuicao);
                }
            }else declaracao(bloco);
        }
    }

    private void chamadaMetodo(No chamada){
        if(variavel()){
            variavel = false;

            No variavel = chamada.acrescentarFilho("variavel");

           variavel.acrescentarFilho(token.getNome() + ": " +
           token.getLexema());

            proxToken();
        }

        Fpar(chamada);
        Ptv(chamada);
    }

    private void atribuicaoVariavel(No atribuicao){
        if(token.getNome() == nome.ATRIB){

           atribuicao.acrescentarFilho(token.getNome() + ": " +
           token.getLexema());
           proxToken();

           if(token.getNome() == nome.VERDADEIRO || token.getNome()
== nome.FALSO){

                atribuicao.acrescentarFilho(token.getNome() + ": "
                + token.getLexema());
                    proxToken();

                    if(token.getNome() == nome.PTV){
```

```
                atribuicao.acrescentarFilho(token.getNome() +
                ": " + token.getLexema());
                proxToken();
            }else erro(token);
        }else if(token.getNome() == nome.ARRAYC){

            atribuicao.acrescentarFilho(token.getNome() + ": "
            + token.getLexema());
                proxToken();



            if(token.getNome() == nome.PTV){

                atribuicao.acrescentarFilho(token.getNome() +
                ": " + token.getLexema());
                proxToken();
            }else erro(token);

        }else if(token.getNome() == nome.ELCARAC){

                atribuicao.acrescentarFilho(token.getNome() +
                ": " + token.getLexema());
                     proxToken();

            if(token.getNome() == nome.PTV){

                atribuicao.acrescentarFilho(token.getNome() +
                ": " + token.getLexema());
                proxToken();
            }else erro(token);
        }else if(token.getNome() == nome.ID){
            Token tokenAnt = token;

            proxToken();

            if(token.getNome() == nome.APAR){
                No nomeMetodo =
                atribuicao.acrescentarFilho("nomeMetodo");

                nomeMetodo.acrescentarFilho(tokenAnt.getNome(
                ) + ": " + tokenAnt.getLexema());
```

```
                    atribuicao.acrescentarFilho(token.getNome() +
                    ": " + token.getLexema());
                    proxToken();

                    chamadaMetodo(atribuicao);
               }else{
                    No variavel =
                    atribuicao.acrescentarFilho("termo");

                    variavel.acrescentarFilho(tokenAnt.getNome()
                    + ": " + tokenAnt.getLexema());

                    expressao(atribuicao);

                   if(token.getNome() == nome.PTV){

                    atribuicao.acrescentarFilho(token.getNome() +
                    ": " + token.getLexema());
                     proxToken();
                   }else erro(token);
               }
          }else{
               expressao(atribuicao);

               if(token.getNome() == nome.PTV){

                    atribuicao.acrescentarFilho(token.getNome() +
                    ": " + token.getLexema());
                    proxToken();
               }else erro(token);
          }
     }else erro(token);
}

private void imprimaLeia(No comando){
     if(token.getNome() == nome.IMPRIMA){

        comando.acrescentarFilho(token.getNome() + ": " +
        token.getLexema());
        proxToken();

        Apar(comando);
        if(token.getNome() == nome.ID){
             No variavel = comando.acrescentarFilho("variavel");
```

```
            variavel.acrescentarFilho(token.getNome() + ": " +
            token.getLexema());
             proxToken();

            Fpar(comando);
            Ptv(comando);
        }else if(token.getNome() == nome.ARRAYC){

            comando.acrescentarFilho(token.getNome() + ": " +
            token.getLexema());
             proxToken();

            Fpar(comando);
            Ptv(comando);
        }else erro(token);
    }else if(token.getNome() == nome.LEIA){

       comando.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
       proxToken();

       Apar(comando);

       if(token.getNome() == nome.ID){
            No variavel = comando.acrescentarFilho("variavel");

            variavel.acrescentarFilho(token.getNome() + ": " +
            token.getLexema());
            proxToken();

            Fpar(comando);
            Ptv(comando);
        }
    }
}

private void Apar(No bloco){
    if(token.getNome() == nome.APAR){

       bloco.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
       proxToken();
    }else erro(token);
}
```

```
private void Fpar(No bloco){
    if(token.getNome() == nome.FPAR){

       bloco.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
       proxToken();
    }else erro(token);
}

private void Ptv(No bloco){
    if(token.getNome() == nome.PTV){

       bloco.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
        proxToken();
    }else erro(token);
}

private void declaracao(No decl){
    No declaracao = decl.acrescentarFilho("declaracao");

    if(tipo()){
       tipo = false;
       No tipo = declaracao.acrescentarFilho("tipo");

       tipo.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
       proxToken();
    }else erro(token);

    if(variavel()){
       variavel = false;
       No variavel = declaracao.acrescentarFilho("variavel");

       variavel.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
       proxToken();
    }else erro(token);

    if(token.getNome() == nome.ACOL){

       declaracao.acrescentarFilho(token.getNome() + ": " +
       token.getLexema());
       proxToken();
```

```
            if(token.getNome() == nome.NUM){

                 declaracao.acrescentarFilho(token.getNome() + ": "
                 + token.getLexema());
                  proxToken();

                 if(token.getNome() == nome.FCOL){

                      declaracao.acrescentarFilho(token.getNome() +
                      ": " + token.getLexema());
                      proxToken();
                 }else erro(token);
             }else erro(token);
         }

         if(token.getNome() == nome.PTV){

            declaracao.acrescentarFilho(token.getNome() + ": " +
            token.getLexema());
            proxToken();

         }else if(token.getNome() == nome.ATRIB){

            declaracao.acrescentarFilho(token.getNome() + ": " +
            token.getLexema());
            proxToken();

             if(token.getNome() == nome.VERDADEIRO ||
token.getNome() == nome.FALSO){

                 declaracao.acrescentarFilho(token.getNome() + ": "
                 + token.getLexema());
                      proxToken();

                      if(token.getNome() == nome.PTV){

                      declaracao.acrescentarFilho(token.getNome() +
                      ": " + token.getLexema());
                      proxToken();

                 }else erro(token);
             }else if(token.getNome() == nome.ARRAYC){
```

```
            declaracao.acrescentarFilho(token.getNome() + ": "
            + token.getLexema());
                  proxToken();

            if(token.getNome() == nome.PTV){

                  declaracao.acrescentarFilho(token.getNome() +
                  ": " + token.getLexema());
                  proxToken();

            }else erro(token);
      }else if(token.getNome() == nome.ELCARAC){

            declaracao.acrescentarFilho(token.getNome() + ": "
            + token.getLexema());
                  proxToken();

            if(token.getNome() == nome.PTV){

                  declaracao.acrescentarFilho(token.getNome() +
                  ": " + token.getLexema());
                  proxToken();

            }else erro(token);
      }else if(token.getNome() == nome.ID){
            Token tokenAnt = token;

            proxToken();

            if(token.getNome() == nome.APAR){
                  No nomeMetodo =
                  declaracao.acrescentarFilho("nomeMetodo");

                  nomeMetodo.acrescentarFilho(tokenAnt.getNome(
                  ) + ": " + tokenAnt.getLexema());


                  declaracao.acrescentarFilho(token.getNome() +
                  ": " + token.getLexema());
                  proxToken();

                  chamadaMetodo(declaracao);
            }else{
```

```
                    No variavel =
                    declaracao.acrescentarFilho("termo");

                    variavel.acrescentarFilho(tokenAnt.getNome()
                    + ": " + tokenAnt.getLexema());

                    expressao(declaracao);

                   if(token.getNome() == nome.PTV){

                    declaracao.acrescentarFilho(token.getNome() +
                    ": " + token.getLexema());
                     proxToken();
                   }else erro(token);
               }
          }else{
               expressao(declaracao);

               if(token.getNome() == nome.PTV){

                    declaracao.acrescentarFilho(token.getNome() +
                    ": " + token.getLexema());
                     proxToken();
               }else erro(token);
          }
     }else erro(token);
}

private void instrucao(No bloco){
     if(token.getNome() == nome.SE){
        No selecao = bloco.acrescentarFilho("selecao");

        selecao.acrescentarFilho(token.getNome() + ": " +
        token.getLexema());
        proxToken();

        Apar(selecao);
        condicao(selecao);

        Fpar(selecao);

        if(token.getNome() == nome.ENTAO){

             selecao.acrescentarFilho(token.getNome() + ": " +
             token.getLexema());
```

```
 proxToken();

if(token.getNome() == nome.ACHAV){

     selecao.acrescentarFilho(token.getNome() + ":
     " + token.getLexema());
     proxToken();

     No blocoSelecao =
     selecao.acrescentarFilho("bloco");
     bloco(blocoSelecao);

    if(token.getNome() == nome.FCHAV){

     selecao.acrescentarFilho(token.getNome() + ":
     " + token.getLexema());
      proxToken();

        while(token.getNome() == nome.SENAO){

           selecao.acrescentarFilho(token.getNome()
           + ": " + token.getLexema());
            proxToken();

            if(token.getNome() == nome.ACHAV){

                selecao.acrescentarFilho(token.getN
                ome() + ": " + token.getLexema());
                     proxToken();

                blocoSelecao =
                selecao.acrescentarFilho("bloco");
                     bloco(blocoSelecao);

                if(token.getNome() == nome.FCHAV){

                     selecao.acrescentarFilho(toke
                     n.getNome() + ": " +
                     token.getLexema());
                     proxToken();
                }else erro(token);
            }else if(token.getNome() == nome.SE){

                selecao.acrescentarFilho(token.getN
                ome() + ": " + token.getLexema());
```

```
                    proxToken();

                    Apar(selecao);
                    condicao(selecao);

                    Fpar(selecao);

                if(token.getNome() == nome.ENTAO){

                    selecao.acrescentarFilho(toke
                    n.getNome() + ": " +
                    token.getLexema());
                    proxToken();




                if(token.getNome() == nome.ACHAV){

                    selecao.acrescentarFilho(toke
                    n.getNome() + ": " +
                    token.getLexema());
                     proxToken();

                    blocoSelecao =
                    selecao.acrescentarFilho("blo
                    co");
                       bloco(blocoSelecao);


                    if(token.getNome() ==
                    nome.FCHAV){

                          selecao.acrescentarFilho
                          (token.getNome() + ": "
                          + token.getLexema());
                            proxToken();
                       }else erro(token);
                   }else erro(token);
                }else erro(token);
            }
        }
    }else erro(token);
}else erro(token);
```

```
            }else erro(token);
        }
    }

    private void condicao(No cond){
        No condicao = cond.acrescentarFilho("condicao");
        expressao(condicao);

           if(token.getNome() == nome.IGUAL || token.getNome() ==
     nome.MENOR || token.getNome() == nome.MAIOR ||
     token.getNome() == nome.MAIORIGUAL || token.getNome() ==
     nome.MENORIGUAL){

           condicao.acrescentarFilho(token.getNome() + ": " +
           token.getLexema());
           proxToken();

           expressao(condicao);
        }
    }

    private void expressao(No exp){
        No expressao = exp.acrescentarFilho("expressao");
        termo(expressao);

        while(token.getNome() == nome.MAIS || token.getNome() ==
nome.MENOS || token.getNome() == nome.MULT){

           expressao.acrescentarFilho(token.getNome() + ": " +
           token.getLexema());
           proxToken();

           termo(expressao);
        }
    }

    private void termo(No term){
        if(token.getNome() == nome.ID){
           No termo = term.acrescentarFilho("termo");

           termo.acrescentarFilho(token.getNome() + ": " +
           token.getLexema());
           proxToken();
        }else if(token.getNome() == nome.NUM){
           No termo = term.acrescentarFilho("termo");
```

```
        termo.acrescentarFilho(token.getNome() + ": " +
        token.getLexema());
        proxToken();

        if(token.getNome() == nome.PTO){

             termo.acrescentarFilho(token.getNome() + ": " +
             token.getLexema());
                  proxToken();

             if(token.getNome() == nome.NUM){

                  termo.acrescentarFilho(token.getNome() + ": "
                  + token.getLexema());
                  proxToken();
             }
         }
    }else if(token.getNome() == nome.APAR){

        term.acrescentarFilho(token.getNome() + ": " +
        token.getLexema());
        proxToken();

        expressao(term);

        if(token.getNome() == nome.FPAR){

             term.acrescentarFilho(token.getNome() + ": " +
             token.getLexema());
                  proxToken();
         }else erro(token);
    }
}

private void repeticaoEnquanto(No repete){
    if(token.getNome() == nome.ENQUANTO){
        No repeticao = repete.acrescentarFilho("repeticao");

        repeticao.acrescentarFilho(token.getNome() + ": " +
        token.getLexema());
        proxToken();

         if(token.getNome() == nome.APAR){
```

```
                repeticao.acrescentarFilho(token.getNome() + ": " +
                token.getLexema());
                     proxToken();

                     condicao(repeticao);

                if(token.getNome() == nome.FPAR){

                     repeticao.acrescentarFilho(token.getNome() +
                     ": " + token.getLexema());
                     proxToken();

                     if(token.getNome() == nome.ACHAV){

                          repeticao.acrescentarFilho(token.getNome
                          () + ": " + token.getLexema());
                          proxToken();

                          No blocoRepeticao =
                          repeticao.acrescentarFilho("bloco");
                          bloco(blocoRepeticao);

                          if(token.getNome() == nome.FCHAV){

                          repeticao.acrescentarFilho(token.getNome
                          () + ": " + token.getLexema());
                           proxToken();
                        }else erro(token);
                    }else erro(token);
                }else erro(token);
            }else erro(token);
        }
    }

    private void repeticaoRepitaAte(No repete){
        if(token.getNome() == nome.REPITA){
           No repeticao = repete.acrescentarFilho("repeticao");

           repeticao.acrescentarFilho(token.getNome() + ": " +
           token.getLexema());
           proxToken();

           if(token.getNome() == nome.ACHAV){
```

```
repeticao.acrescentarFilho(token.getNome() + ": " +
token.getLexema());
     proxToken();

No blocoRepeticao =
repeticao.acrescentarFilho("bloco");
bloco(blocoRepeticao);

if(token.getNome() == nome.FCHAV){

     repeticao.acrescentarFilho(token.getNome() +
     ": " + token.getLexema());
     proxToken();

     if(token.getNome() == nome.ATE){

          repeticao.acrescentarFilho(token.getNome
          () + ": " + token.getLexema());
          proxToken();

        if(token.getNome() == nome.QUE){

          repeticao.acrescentarFilho(token.getNome
          () + ": " + token.getLexema());
           proxToken();

           if(token.getNome() == nome.APAR){

               repeticao.acrescentarFilho(token.ge
               tNome() + ": " +
               token.getLexema());
                    proxToken();

                    condicao(repeticao);

               if(token.getNome() == nome.FPAR){

               repeticao.acrescentarFilho(token.ge
               tNome() + ": " +
               token.getLexema());
                proxToken();

                if(token.getNome() == nome.PTV){
```

```
                                        repeticao.acrescentarFilho(to
                                        ken.getNome() + ": " +
                                        token.getLexema());
                                          proxToken();

                                    }else erro(token);
                                }else erro(token);
                            }else erro(token);
                        }else erro(token);
                    }else erro(token);
                }else erro(token);
            }
        }

        private void repeticaoRepitaPara(No repete){
            if(token.getNome() == nome.PARA){
               No repeticao = repete.acrescentarFilho("repeticao");

               repeticao.acrescentarFilho(token.getNome() + ": " +
               token.getLexema());
               proxToken();

                if(variavel()){
                    variavel = false;

                    No variavel = repeticao.acrescentarFilho("termo");

                    variavel.acrescentarFilho(token.getNome() + ": " +
                    token.getLexema());

                    proxToken();
                }else erro(token);

                if(token.getNome() == nome.DE){

                    repeticao.acrescentarFilho(token.getNome() + ": " +
                    token.getLexema());
                         proxToken();

                    if(token.getNome() == nome.ID || token.getNome() ==
nome.NUM){
```

```
                        repeticao.acrescentarFilho(token.getNome() +
                        ": " + token.getLexema());
                        proxToken();

                       if(token.getNome() == nome.ATE){

                        repeticao.acrescentarFilho(token.getNome() +
                        ": " + token.getLexema());
                          proxToken();

                           if(token.getNome() == nome.ID ||
token.getNome() == nome.NUM){
                                No termo =
                        repeticao.acrescentarFilho("termo");

                        termo.acrescentarFilho(token.getNome() +  ":
                        " + token.getLexema());
                         proxToken();

                               if(token.getNome() == nome.PASSO){

                                   repeticao.acrescentarFilho(token.ge
                                   tNome() + ": " +
                                   token.getLexema());
                                        proxToken();

                               if(token.getNome() == nome.NUM){
                                   termo =
                                   repeticao.acrescentarFilho("termo")
                                   ;

                                   termo.acrescentarFilho(token.getNom
                                   e() + ": " + token.getLexema());
                                    proxToken();

                                      if(token.getNome() ==
nome.FACA){

                                         repeticao.acrescentarFilho(to
                                         ken.getNome() + ": " +
                                         token.getLexema());
                                           proxToken();
```

```
                                    if(token.getNome() ==
nome.ACHAV){

                                        repeticao.acrescentarFilho(to
                                        ken.getNome() + ": " +
                                        token.getLexema());
                                         proxToken();
                                        No blocoRepeticao =
                                    repeticao.acrescentarFilho("bloco")
                                    ;
                                          bloco(blocoRepeticao);


                                    if(token.getNome() == nome.FCHAV){

                                         repeticao.acrescentarFilho(to
                                         ken.getNome() + ": " +
                                         token.getLexema());
                                          proxToken();
                                      }else erro(token);
                                   }else erro(token);
                                  }else erro(token);
                              }else if(token.getNome() == nome.MENOS){

                               No expressao =
                               repeticao.acrescentarFilho("expressao");

                               expressao.acrescentarFilho(token.getNome
                               () + ": " + token.getLexema());
                               proxToken();

                              if(token.getNome() == nome.NUM){

                                    termo =
                                    expressao.acrescentarFilho("termo")
                                    ;

                                    termo.acrescentarFilho(token.getNom
                                    e() + ": " + token.getLexema());
                                     proxToken();

                                      if(token.getNome() == nome.FACA){

                                          repeticao.acrescentarFilho(to
```

```
                                        ken.getNome() + ": " +
                                        token.getLexema());
                                         proxToken();

                                    if(token.getNome() ==
nome.ACHAV){

                                        repeticao.acrescentarFilho(to
                                        ken.getNome() + ": " +
                                        token.getLexema());
                                         proxToken();


                                        No blocoRepeticao =
                                        repeticao.acrescentarFilho("b
                                        loco");

                                        bloco(blocoRepeticao);

                                    if(token.getNome() ==
nome.FCHAV){

                                             repeticao.acrescentarFil
                                             ho(token.getNome() + ":
                                             " + token.getLexema());
                                               proxToken();
                                            }else erro(token);
                                          }else erro(token);
                                       }else erro(token);
                                    }else erro(token);
                                 }else erro(token);
                              }else erro(token);
                           }else erro(token);
                        }else erro(token);
                     }else erro(token);
                  }else erro(token);
               }else erro(token);
            }
        }

        private boolean tipo(){
            tipo = false;

            if(token.getNome() == nome.INT || token.getNome() ==
nome.BOOL || token.getNome() == nome.DEC || token.getNome() ==
nome.CARAC){
```

```
            tipo = true;
        }

        return tipo;
    }

    private void nomeMetodo(No bloco){
        if(token.getNome() == nome.ID){
            bloco.acrescentarFilho(token.getNome() + ": " +
token.getLexema());
            proxToken();
        }else erro(token);
    }

    private boolean variavel(){
        variavel = false;

        if(token.getNome() == nome.ID){
            variavel = true;
        }

        return variavel;
    }

    private void fimSintatica(){
        if(countPrinc == 1){
            System.out.println("Analise Sintatica Executada com
Sucesso, sem detecção de erros!!!");
            no.imprimirArvore();
        }else if(countPrinc > 1){
            System.out.println("O programa possui mais de um
procedimento principal!!!");
        }else if(countPrinc == 0){
            System.out.println("Nao ha um procedimento
principal!!!");
        }
    }

    private void proxToken(){
        if(i < limite){
            i++;
            token = Tokens.get(i);
        }
    }
```

```
    private void erro(Token token){
        System.out.println("Erro Sintatico no Token: " +
Tokens.get(i));
        System.exit(0);
    }
}
```

Classe Arvore:

```
public class Arvore
{
    private No raiz;

    public Arvore(No no){
        setRaiz(no);
    }

    private void setRaiz(No raiz){
        this.raiz = raiz;
    }

    public No getRaiz(){
        return this.raiz;
    }
}
```

Classe No:

```
public class No
{
    private String chave;
    private ArrayList<No> filhos;

    public No(String chave){
        setChave(chave);
        filhos = new ArrayList<>();
    }

    private void setChave(String chave){
        this.chave = chave;
    }
```

```
    public String getChave(){
        return this.chave;
    }

    public No acrescentarFilho(String chave){
        No filho = new No(chave);
        filhos.add(filho);

        return filho;
    }

    public void imprimirArvore(){
        System.out.println("\nraiz: " + getChave());
        System.out.print("filhos: ");

        if(filhos.size() != 0){
            for(int i = 0; i < filhos.size(); i++){
                No f = filhos.get(i);
                System.out.print(f.getChave() + " ");
            }

            for(int i = 0; i < filhos.size(); i++){
                No f = filhos.get(i);

                if(f.filhos.size() != 0){
                    f.imprimirArvore();
                }
            }
        }

    }
}
```

11. BIBLIOGRAFIA

AHO, A. V.; SETHI, R.; ULLMAN, J.D. Compiladores: Princípios, técnicas e ferramentas. 2ª ed. São Paulo: Pearson, 2007.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A.; SMITH, D.; BIERMAN, G. The Java® Language Specification. Java SE 22 Edition. [S.l.]: [s.n.], 8 fev. 2024.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. The C Programming Language. 2. ed. New Jersey: TBS, 1988.

RAMOS, M. V. M.; NETO, J. J.; VEGA, I. S. Linguagens formais, Teorias e conceitos. 1. ed. São Paulo: Blucher, 2023.

VEGA, I. S. Construção de Compiladores: Teoria e Prática em Java. Série OC2-RD2 / Ciência da Computação, versão 0.2.3. [S.l.]: ISVega, 16 mar. 2024.