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

TUTORIAL DA 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. 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.

1.1 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.

1.2 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.

1.3 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 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;
  }
```

1.3.1 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.

1.3.2 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.

1.3.3. 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.

1.3.4. 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.

1.3.5 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`.

1.6 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.

2. MANUAL DE REFERÊNCIA

2.1 Introdução

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

2.2 Convenções Léxicas

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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`

2.2.5 Constantes

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

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

2.3 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)

2.4 Expressões

2.4.1 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>])
```

2.4.2 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>
```

2.4.3 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)

2.5 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>;
```

2.5.1 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'.

2.5.2 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>];
```

2.5.3 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>;
```

2.5.4 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.

2.5.5 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.

2.5.6 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. COMPILADOR

Para a produção do compilador da linguagem Minerva, propôs-se uma tradução dirigida por sintaxe. Nessa primeira etapa, desenvolveu-se o analisador léxico e, parcialmente, o analisador sintático.

3.1 Análise Léxica

Com o objetivo de gerar uma lista de tokens a partir da análise léxica, criou-se os seguintes tokens em Java:

```
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
}
```

Figura 01: Definição dos nomes de um Token

Para trabalhar com esses tokens, foi decidido criar um HashMap:

```
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);
}
```

Figura 02: Dicionário de palavras reservadas implementado via HashMap

Para inicializar o processo da Análise Léxica, o código-fonte foi armazenado em um array de caracteres, permitindo que cada símbolo seja analisado pelo lexer. Para esse cenário acontecer, eles serão avaliados até que se encontre um regex correspondente.

Por exemplo, cada lexema começado com letra e seguido por letras ou números, intercalados, no momento em que está sendo analisado é considerado um token do tipo ID. Porém, quando o lexema termina (quando encontra-se algo diferente de letras ou números), ocorrerá uma consulta na lista de palavras reservadas: caso o lexema seja correspondente a algum token mapeado, então o tipo de seu token será alterado para o valor mapeado correspondente. Esse cenário pode ser visto abaixo:

```
} 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;
    }
```

Figura 03: Implementação da análise de um ID - letra(letra|dígito)*

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

Figura 04: método verificarPalavrasReservadas

Tanto para verificação de ID e NUM (que corresponde a números), é preciso utilizar a técnica de look-ahead: o lexer observa o símbolo seguinte ao atual, a fim de definir qual tipo de token ele será classificado. Por exemplo, no caso de um número, o look-ahead serve para concatenar uma sequência de números como um único lexema.

Essa técnica também é utilizada na verificação dos seguintes lexemas: > e <. No caso do menor (<), é preciso realizar o look-ahead para verificar se o símbolo seguinte é um “=”, “-” ou nenhum dos dois. Respectivamente, os tokens apropriados seriam: MENORIGUAL, ATRIB, MENOR.

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

Figura 05: Verificação do token MENOR

```
case "-":
    if(menor){
        tipo = nome.ATRIB;
        b = a + 1;
    }
    else {
        tipo = nome.MENOS;
        b = a;
    }
    adiciona = true;
    break;
```

Figura 06: Verificação dos tokens ATRIB (<-) e MENOS (-)

Para o maior (>), é preciso apenas verificar se o próximo símbolo é um “=”, e, portanto, seu token será MAIORIGUAL, ou se não for seguido por um “=”, ele será apenas MAIOR.

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

Figura 07: verificação do token MAIOR

```
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;
```

Figura 08: Verificação dos tokens IGUAL (=), MAIORIGUAL (>=) e MENORIGUAL(<=).

Para lexemas unitários, onde não se importa quais serão os símbolos seguintes, foi utilizado um switch para definir qual será o tipo do token, como no exemplo abaixo:

```
} 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);
```

Figura 09: Verificação para lexemas unitários

No final da lista de tokens, quando todos os caracteres do código-fonte forem consumidos pelo lexer, será inserido o token EOF, que sinaliza o final do programa.

```
listtokens.add("[@" + indice + ", " +  a + "-" + a + "," + "\"EOF\"" + ", <" + nome.EOF + ">," + line +"]");
```

Figura 10: Inserção do token EOF ao final da análise

3.2 Análise Sintática

Por enquanto, a Análise Sintática desenvolvida até essa etapa reconhece apenas expressões aritméticas, as quais foram definidas a partir das regras gramaticais abaixo:

expression = term ((“+”|”-”|”*”) term)*

term = variable|number (“.”number)*|”(“ exp “)”

condition = “(“ expression (“=”|”<”|”>”|”<=”|”>=”) expression “)”

variable = ID

number = NUM

Por ser uma tradução dirigida por sintaxe, a lista de tokens gerada pelo lexer servirá como entrada para o analisador sintático, que tem como objetivo devolver uma árvore. Para implementá-la, decidiu-se utilizar, inicialmente, os seguintes atributos:

```
public class No
{
    private No pai;
    private No esq;
    private No meio;
    private No dir;
    private Object chave;
```

Figura 11: Definição da classe No

Como ainda não foi possível desenvolver a árvore por completo, a análise foi implementada baseando-se em funções, como no exemplo abaixo, onde trabalha-se com uma recursão indireta:

```
private void fimSintatica(){
    if(countPar == 0) System.out.println("Anlise Sintatica Executada com Sucesso, sem detecção de erros");
    else System.out.println("Erro Sintatico: existe " + Math.abs(countPar) + " parentese(s) incompleto(s)!");
}

private void verNum(Token token){ // Num
    if(token.getNome() == nome.NUM){
        if(primeiraVez == true) primeiraVez = false;
        token = proxToken();
        verExp(token);
    }else verAPar(token);
}

private void verExp(Token token){ // MAIS / MENOS / MULT
    if(token.getNome() == nome.MAIS || token.getNome() == nome.MENOS || token.getNome() == nome.MULT){
        token = proxToken();
        verNum(token);
    } else verPto(token);
}

private void verAPar(Token token){ //APAR
    if(token.getNome() == nome.APAR){
        if(primeiraVez == true) primeiraVez = false;
        countPar++;
        token = proxToken();
        verNum(token);
    }else if(primeiraVez == false) erro(token);
    else verFPar(token);
}

private void verFPar(Token token){ //FPAR
    if(token.getNome() == nome.FPAR){
        countPar--;
        token = proxToken();
        verExp(token);
    } else verPtv(token);
}
```

Figura 12: Parte I da implementação da análise sintática

```
private void verPto(Token token){ // PTO
    if(token.getNome() == nome.PTO){
        token = proxToken();
        verNum(token);
    } else verFPar(token);
}

private void verPtv(Token token){ // PTV
    if(token.getNome() == nome.PTV){
        token = proxToken();
        verEof(token);
    } else erro(token);
}

private void verEof(Token token){ // EOF
    if(token.getNome() == nome.EOF){
        fimSintatica();
    } else erro(token);
}

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

private void erro(Token token){
    System.out.println("Erro Sintatico no Token: " + Tokens.get(i-1));
    System.exit(0);
}
```

Figura 13: Parte II da implementação da análise sintática

A árvore gerada pelo analisador sintático contém dois tipos distintos de nós: os tokens e os nós não terminais. Os nós não-terminais são etiquetas colocadas na árvore, como nós intermediários, que indicam quais os tipos de regras gramaticais que os nós folhas (tokens) fazem parte.

Com isso, a geração da árvore ocorre durante a execução da análise sintática com funções que inserem os nós não terminais (nós internos) e os tokens (folhas da árvore) no decorrer das funções recursivas de verificação da sequência de tokens analisada.

Para melhor ilustrar a geração da árvore sintática, abaixo há como exemplo uma árvore sintática da expressão aritmética 5 + 5 * 2, em que é visto a presença dos nós internos e da raiz com os elementos não terminais Expr e Term, que representam respectivamente as regra gramaticais expression e term. E também, os nós que guardam os tokens, sendo as folhas da árvore os tokens de nome NUM e os nós internos que guardam os tokens MAIS e MULT.

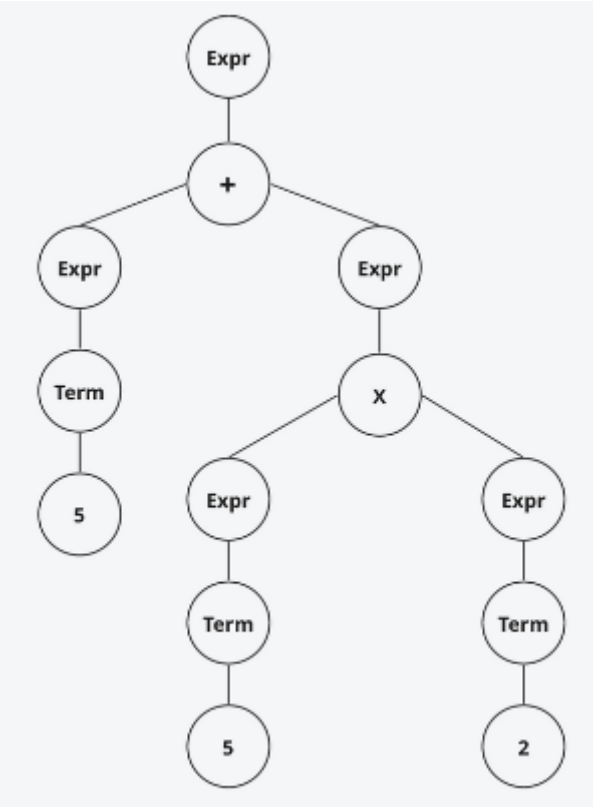

Figura 14: Ideia inicial da árvore sintática

4. 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.