Uma imagem contendo comida

Descrição gerada automaticamente

**UNIVERSIDADE SÃO JUDAS TADEU**

**PROJETO A3**

**Teoria da Computação e Compiladores**

**Sumário**

[**1. Introdução 2**](#_t856e9g7vceo)

[**2. Linguagem de programação Brazuca 2**](#_oflqhgpr9fhb)

[**2.1. Estrutura Básica 2**](#_lxt45x78grxm)

[**2.2 Variáveis 3**](#_kt7oczi73gii)

[**2.3 Estrutura de controle se então 3**](#_q21cpojwhrmp)

[**2.4 Estrutura de repetição enquanto 4**](#_7wscehcaciat)

[**2.5 Estrutura de repetição paracada 4**](#_58bntxglu41f)

[**2.6 Operações matemáticas e booleanas 4**](#_34phk5xjb61h)

[**3. Análise léxica 5**](#_7q5bvnevqczz)

[**4. Análise Sintática 6**](#_kf43pzidg5vi)

[**5. Geração do código-objeto Python 8**](#_g7iau3uzlxqr)

[**6. Conclusão 9**](#_qjr80pnwz12r)

[**7. Referências 9**](#_jefzy3v1sk3c)

## **Introdução**

Esse projeto teve como objetivo desenvolver um compilador para realizar a análise léxica e sintática de uma linguagem de programação fictícia e gerar código válido para uma segunda linguagem de programação. Todos os artefatos desse projeto e instruções de instalação se encontram no seguinte repositório:

<https://github.com/d4rkwav3/compilador_projetoA3.git>

## **Linguagem de programação Brazuca**

A linguagem de programação criada para esse projeto chama-se Brazuca, inspirada pela linguagem de programação Portugol e Python, as instruções de um código Brazuca são escritas em Português do Brasil, aceita caracteres com acentuação e possui uma estrutura semelhante a Python, compilar código em Brazuca gera código em Python. O compilador da linguagem Brazuca foi escrito em Python.

### **2.1. Estrutura Básica**

Brazuca aceita três tipos de variáveis, números inteiros, números com casa decimal e cadeia de caracteres (strings), possui uma única estrutura de controle, se então (if, else) e duas estruturas de repetição, enquanto e paracada (while, for), a tabela 1 a seguir mostra todas as palavras reservas em sua sintaxe correta.

| *Tabela 1 – Palavras Reservadas* | |
| --- | --- |
| **Palavra reservada** | **Função** |
| inteiro | Declara variáveis do tipo inteiro (1, 512, 6718, etc.). |
| decimal | Declara variáveis do tipo decimal (0.5, 1.5, 3.14, etc). |
| palavra | Declara variáveis do tipo palavra (“isso é uma palavra”). |
| se | Sinalizar uma condição de controle. |
| então | Sinalizar o código a ser executado caso uma condição seja atendida. |
| senão | Usada em conjunto com a instrução se, executa uma ou mais instruções caso a condição em ‘se’ seja falsa. |
| enquanto | Inicia uma estrutura de repetição. |
| paracada | Inicia uma estrutura de repetição usando um objeto iterável. |
| em | Usada em conjunto com o paracada para indicar uma variável que contenha um ou mais elemento, como uma palavra. |
| imprima | Usada para imprimir texto na tela. |
| leia | Usada para ler o teclado. |

### **2.2. Variáveis**

As variáveis podem ser declaradas da seguinte forma:

TIPO ID = DADO

Os seguintes exemplos são declarações válidas:

palavra msg1 = "Digite um número inteiro"

inteiro i = -10

decimal d = 3.14

### **2.3. Estrutura de controle se então**

Uma condição se então é expressa da seguinte forma:

se *expressão* então

*instrução*

A expressão esperada é uma que pode ser avaliada como booleana, ou seja, pode ser verdadeira ou falsa. A próxima linha abaixo da estrutura se então será executada caso a expressão seja avaliada como verdadeira, é importante notar que as instruções devem ser iniciadas com uma tabulação para ser consideradas dentro do bloco se então mais próxima, por exemplo:

se a > b então

c = a

Para incluir instruções a serem executada caso a expressão seja falsa, use senão então, toda instrução tabulada abaixo do bloco senão então será executada, como no exemplo abaixo:

se a > b então

c = a

senão então

c = b

### **2.4. Estrutura de repetição enquanto**

Uma repetição enquanto é expressa da seguinte forma:

enquanto *expressão* então

*instrução*

Assim como a estrutura de controle se então, a estrutura enquanto espera uma expressão que possa ser avaliada como verdadeira ou falsa, executando o código tabulado mais próximo caso a expressão seja verdadeira, como no exemplo abaixo:

enquanto c < 10 então

imprima(c)

c += 1

### **2.5. Estrutura de repetição paracada**

A estrutura paracada é usada para iterar sobre objetos e variáveis iteráveis, é usada em conjunto com a palavra reservada em, da seguinte forma:

paracada *item* em *objeto* então

*instrução*

No exemplo acima, item é uma variável temporária que irá armazenar a informação de um elemento da variável objeto para cada iteração da repetição, a instrução da repetição deve estar tabulada para ser considerada.

**2.6.** **Operações matemáticas e booleanas**

É possível realizar as operações de adição, subtração, multiplicação e divisão em Brazuca, multiplicação e divisão são executadas antes de adição ou subtração, mas é possível adicionar parênteses entre uma expressão para alterar sua precedência, como no exemplo a seguir:

(3 + 4 \* 10 + -20 \* 2) / 2

Nas estruturas se/então e enquanto é possível efetuar operações booleanas utilizando os símbolos de comparação, conforme na tabela 2 abaixo:

| *Tabela 2 – Símbolos de comparação* | |
| --- | --- |
| **Símbolo** | **Função** |
| A > B | Compara se A é maior (em valor) que B. |
| C >= D | Compara se C é maior ou igual (em valor) que D. |
| E < F | Compara se E é menor (em valor) que F. |
| G <= H | Compara se G é menor ou igual (em valor) que H. |
| I == J | Compara se I é igual (em valor) a J. |

Os objetos a serem comparados precisam necessariamente ser do mesmo tipo, por exemplo, não é possível comparar variáveis do tipo palavra com variáveis do tipo inteiro.

## **Análise léxica**

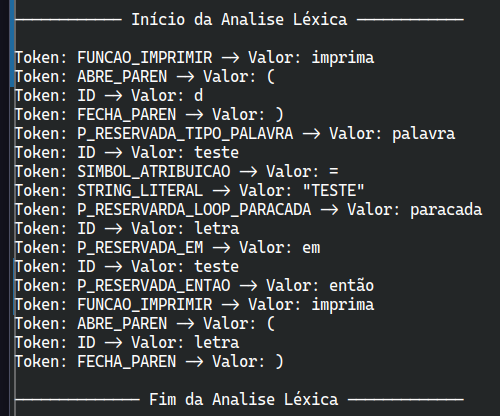
A análise léxica é o primeiro passo na compilação de um código Brazuca, todas as expressões regulares usadas para a analise léxica estão contidas no arquivo lexical\_analizer.py, a seguir uma breve descrição o conteúdo do arquivo:

reserved : Uma variável do tipo Dict que guarda todas as palavras reservadas da linguagem.

tokens: Um vetor que guarda todos os tokens da linguagem.

Variáveis e funções com prefixo t\_\*:Contém uma expressão regular usada para identificar um token.

Após analisar um arquivo, todos os tokens identificados mostrados na tela, a imagem a seguir mostra a análise léxica do arquivo de exemplo sample2.txt.



## **Análise Sintática**

A análise sintática é o segundo passo na compilação de um código Brazuca, todos os tokens gerados durante a análise léxica são analisados para verificar se estão sintaticamente corretos de acordo com as regras da gramática da linguagem. O arquivo syntax\_analyzer.py possui todas as regras de derivação para análise sintática. A seguir, todas as regras usadas para se analisar sintaticamente um código Brazuca:

Regra 0 S' -> Expressão

Regra 1 Expressão -> Declaração

Regra 2 Expressão -> Atribuição

Regra 3 Expressão -> Controle

Regra 4 Expressão -> Loop

Regra 5 Expressão -> Função

Regra 6 Expressão -> nova\_linha

Regra 7 Controle -> P\_RESERVADA\_SE Expressão P\_RESERVADA\_ENTAO

Regra 8 Controle -> P\_RESERVADA\_SENAO P\_RESERVADA\_ENTAO

Regra 9 Loop -> P\_RESERVARDA\_LOOP\_ENQUANTO Expressão P\_RESERVADA\_ENTAO

Regra 10 Loop -> P\_RESERVARDA\_LOOP\_PARACADA ID P\_RESERVADA\_EM Expressão P\_RESERVADA\_ENTAO

Regra 11 Expressão -> COMENTARIO

Regra 12 Declaração -> P\_RESERVADA\_TIPO\_INTEIRO ID SIMBOL\_ATRIBUICAO NUM\_INTEIRO

Regra 13 Declaração -> P\_RESERVADA\_TIPO\_DECIMAL ID SIMBOL\_ATRIBUICAO NUM\_DECIMAL

Regra 14 Declaração -> P\_RESERVADA\_TIPO\_PALAVRA ID SIMBOL\_ATRIBUICAO STRING\_LITERAL

Regra 15 Declaração -> P\_RESERVADA\_TIPO\_PALAVRA ID SIMBOL\_ATRIBUICAO ID ABRE\_PAREN ID FECHA\_PAREN

Regra 16 Declaração -> P\_RESERVADA\_TIPO\_INTEIRO ID SIMBOL\_ATRIBUICAO ID ABRE\_PAREN STRING\_LITERAL FECHA\_PAREN

Regra 17 Declaração -> P\_RESERVADA\_TIPO\_INTEIRO ID SIMBOL\_ATRIBUICAO ID ABRE\_PAREN FECHA\_PAREN

Regra 18 Declaração -> P\_RESERVADA\_TIPO\_PALAVRA ID SIMBOL\_ATRIBUICAO Função

Regra 19 Declaração -> P\_RESERVADA\_TIPO\_INTEIRO ID SIMBOL\_ATRIBUICAO Função

Regra 20 Declaração -> P\_RESERVADA\_TIPO\_DECIMAL ID SIMBOL\_ATRIBUICAO Função

Regra 21 Declaração -> P\_RESERVADA\_TIPO\_DECIMAL ID SIMBOL\_ATRIBUICAO Termo

Regra 22 Declaração -> P\_RESERVADA\_TIPO\_INTEIRO ID SIMBOL\_ATRIBUICAO Termo

Regra 23 Função -> FUNCAO\_IMPRIMIR ABRE\_PAREN Fator FECHA\_PAREN

Regra 24 Função -> FUNCAO\_LER ABRE\_PAREN Fator FECHA\_PAREN

Regra 25 Função -> ID ABRE\_PAREN Termo FECHA\_PAREN

Regra 26 Função -> ID ABRE\_PAREN FECHA\_PAREN

Regra 27 Atribuição -> ID SIMBOL\_ATRIBUICAO Termo

Regra 28 Atribuição -> ID ACUMULADOR Primitivo

Regra 29 Atribuição -> ID REDUTOR Primitivo

Regra 30 Expressão -> Expressão SIMBOL\_ADICAO Termo

Regra 31 Expressão -> Expressão SIMBOL\_SUBTRACAO Termo

Regra 32 Expressão -> Termo

Regra 33 Termo -> Termo SIMBOL\_MULTIPLICACAO Fator

Regra 34 Termo -> Termo SIMBOL\_DIVISAO Fator

Regra 35 Termo -> Termo MAIORQUE Fator

Regra 36 Termo -> Termo MAIOROUIGUAL Fator

Regra 37 Termo -> Termo MENORQUE Fator

Regra 38 Termo -> Termo MENOROUIGUAL Fator

Regra 39 Termo -> Termo COMPARAR\_VALOR Fator

Regra 40 Termo -> Fator

Regra 41 Fator -> Primitivo

Regra 42 Primitivo -> NUM\_INTEIRO

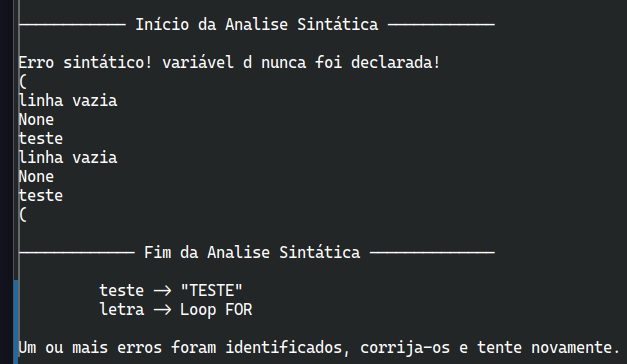
Regra 43 Primitivo -> NUM\_DECIMAL

Regra 44 Primitivo -> STRING\_LITERAL

Regra 45 Fator -> ID

Regra 46 nova\_linha -> NOVA\_LINHA

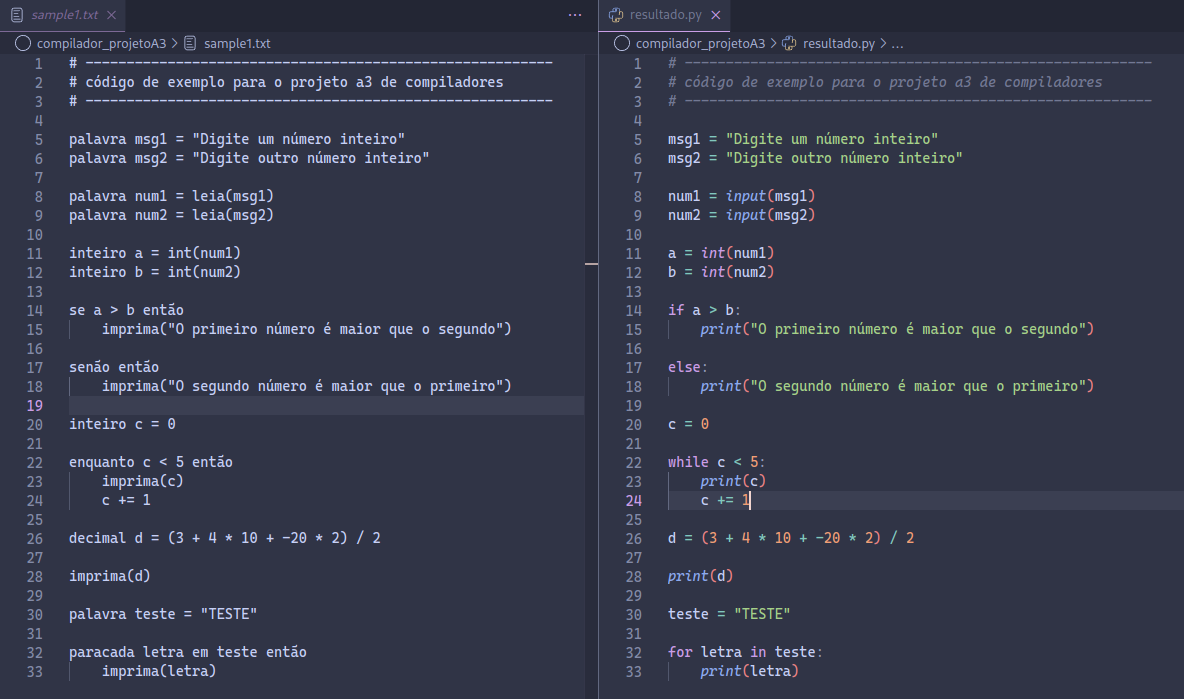
Regra 47 Fator -> ABRE\_PAREN Expressão FECHA\_PAREN



A imagem acima contém as informações impressas na tela durante a análise sintática do arquivo de exemplo sample2.txt. Se a análise sintática do arquivo não encontrar erros, irá criar o arquivo parser.out, que contém informações detalhadas de toda a análise sintática do arquivo.

1. **Geração do código-objeto Python**

O arquivo compilador.py reúne o analisar léxico e sintático da linguagem Brazuca e solicita o nome de um arquivo (precisa estar na mesma pasta do projeto) para a análise, e caso nenhum erro sintático seja detectado, traduz o código Brazuca para Python no arquivo resultado.py, a imagem abaixo mostra o código Brazuca a esquerda e sua tradução em Python a direita:



## **Conclusão**

O processo de desenvolvimento da linguagem Brazuca envolveu a definição de um alfabeto, no qual foi utilizado o alfabeto brasileiro. Foi desenvolvido então as expressões regulares para a geração dos tokens. Em seguida foi desenvolvida as regras de derivação para a analisar a sintaxe do código. Por fim, foi desenvolvido o tradutor, que transforma código em Brazuca para código válido em Python.

1. **Referências**

**PLY (python Lex-Yacc)**. Disponível em: <https://www.dabeaz.com/ply/ply.html>. Acesso em: 28 nov. 2023.