

## 16 Fase 1 - Projeto Prático

Este trabalho pode ser realizado em grupos de até 4 alunos. **Grupos com mais de 4 membros terão o trabalho anulado.** Leia todo este texto antes de começar e siga o seguinte código de ética: *você pode discutir as questões com colegas, professores e amigos, consultar livros da disciplina, bibliotecas virtuais ou físicas, e a Internet em geral, em qualquer idioma. Você pode usar qualquer ferramenta de Inteligência Artificial para dúvidas, mas não para fazer o trabalho no seu lugar. O trabalho é seu e deve ser realizado por você. Plágio resultará na anulação do trabalho.*

Os trabalhos entregues serão avaliados por uma ferramenta de Inteligência Artificial, que verificará a originalidade do texto e a autoria do código. O trabalho deve ser entregue em um repositório público no **GitHub**.

### 16.1 Objetivo

Pesquisar e praticar conceitos de analisador léxico para desenvolver um programa em **Python, C,** ou **\*\*\*\*C++\*\*\*\*** que processe expressões aritméticas em notação polonesa reversa (RPN), conforme definida neste texto, a partir de um arquivo de texto, utilizando **máquinas de estado finito** (FSMs) implementadas obrigatoriamente com funções. O programa deve executar as expressões em um ambiente de teste (ex.: o notebook do aluno) e em um Arduino Uno, ou Mega. O seu trabalho será gerar o código Assembly, compatível com a arquitetura do Arduino Uno, ou Mega. Um guia de como compilar e executar o código Assembly está incluído na seção [Chapter 17](#) no final deste documento.

### 16.2 Descrição do Trabalho

O objetivo é desenvolver um programa capaz de:

1. Ler um arquivo de texto contendo expressões aritméticas em **Escritas RPN**, segundo o formato especificado neste documento, com uma expressão por linha. Este arquivo contém o código do programa que será analisado pelo analisador léxico.
2. Analisar as expressões usando um analisador léxico baseado em Autômatos Finitos Determinísticos, com estados implementados por funções.
3. Transformar as expressões em um texto contendo o código Assembly para o Arduino ([uno|mega](#)), utilizando as operações aritméticas e comandos especiais especificados. As operações serão realizadas no Arduino.
4. Executar as expressões em um Arduino (Uno|Mega), de forma que os resultados possam ser acompanhados (display, leds ou serial).
5. Hospedar o código, arquivos de teste e documentação em um repositório público no **GitHub**.

#### 16.2.1 Características Especiais

As expressões devem ser escritas em notação RPN, no formato **(A B op)**, no qual **A** e **B** são números reais, os operandos, e **op** é um operador aritmético entre os listados neste documento. O programa deve suportar operações aritméticas básicas listadas neste documento, comandos especiais para

manipulação de memória, e deve ser capaz de lidar com expressões aninhadas sem limites de aninhamento. Segundo a seguinte sintaxe:

- Considerando que **A** e **B** são números reais, e usando o ponto como separador decimal, (ex.: 3.14), teremos:
- **Operadores suportados na Fase 1:**
  - Adição: **+** (ex.: **(A B +)**);
  - Subtração: **-** (ex.: **(A B -)**);
  - Multiplicação: **\*** (ex.: **(A B \*)**);
  - Divisão real: **/** (ex.: **(A B /)**);
  - Divisão inteira: **/** (ex.: **(A B /)** para inteiros);
  - Resto da divisão inteira: **%** (ex.: **(A B %)**);
  - Potenciação: **^** (ex.: **(A B ^)**, onde **B** é um inteiro positivo);
- Todas as operações (exceto divisão inteira e resto) usam números reais em formato de meia precisão (16 bits, IEEE 754), com duas casas decimais. A página [Os desafios da norma IEEE 754](#) contém informações relevantes sobre a norma IEEE 754.
- Expressões podem ser aninhadas sem limite, por exemplo:
  - **(A (C D \*) +)**: Soma **A** ao produto de **C** e **D**;
  - **((A B \*) (D E \*) /)**: Divide o produto de **A** e **B** pelo produto de **D** e **E**.
  - **((A B +) (C D \*) /)**: Divide a soma de **A** e **B** pelo produto de **C** e **D**.
- A ordem de precedência das operações segue a ordem de precedência usual em matemática.

### 16.2.2 Comandos Especiais

A linguagem que estamos criando inclui comandos especiais para manipulação de memória e resultados:

- **(N RES)**: Retorna o resultado da expressão **N** linhas anteriores (**N** é um inteiro não negativo).
- **(V MEM)**: Armazena o valor real **V** em uma memória chamada **MEM**.
- **(MEM)**: Retorna o valor armazenado em **MEM**. Se a memória não foi inicializada, retorna 0.0.
- Cada arquivo de texto, código fonte da linguagem que estamos criando, representa um escopo independente de memória.
- **MEM** pode ser qualquer conjunto de letras maiúsculas, tal como **MEM**, **VAR**, **X**, etc.
- **RES** é uma *keyword* da linguagem que estamos criando. A única *keyword* da linguagem nesta fase.

## 16.3 Analisador Léxico com Autômatos Finitos Determinístico

1. O analisador léxico deve ser implementado usando Autômatos Finitos Determinísticos, com **cada estado representado por uma função**. Qualquer forma diferente de implementação provocará o zeramento do trabalho.
2. O Autômato Finito Determinístico deve reconhecer *tokens* válidos: números reais (duas casas decimais), operadores (+, -, \*, /, %, ^), comandos especiais (RES, MEM), e parênteses.
3. Funções de teste específicas devem ser criadas para validar o analisador léxico, cobrindo:
  - Entradas válidas (ex.: (3.14 2.0 +), (5 RES), (10.5 CONTADOR));
  - Entradas inválidas (ex.: (3.14 2.0 &), números malformados como 3.14.5, 3,45 ou parênteses desbalanceados).

## 16.4 Arquivos de Teste

---

- Fornecer **mínimo de 3 arquivos de texto**, cada um com **pelo menos 10 linhas** de expressões segundo as especificações deste documento.
- Cada arquivo deve incluir todas as operações (+, -, \*, /, %, ^) e comandos especiais (N RES), (V MEM), (MEM).
- Os arquivos devem estar no mesmo diretório do código-fonte e ser processados via argumento de linha de comando (ex.: ./NomeDoSeuPrograma teste1.txt).
- O programa não deve incluir menu ou qualquer seleção interativa de arquivos.

## 16.5 Hospedagem no GitHub

---

- O projeto deve ser hospedado em um repositório público no **GitHub**. Se o professor não indicar qual repositório deve ser usado, o repositório deve ser criado por um dos alunos do grupo.
- O repositório deve conter:
  - Código-fonte do programa;
  - Arquivos de teste (mínimo 3);
  - Funções de teste para o analisador léxico;
  - Última versão do Código Assembly para Arduino gerado pelo programa;
  - Documentação (ex.: README.md) explicando como compilar, executar e testar o programa. Contendo o nome da instituição, disciplina, professor e nome dos alunos do grupo, em ordem alfabética seguido do usuário deste aluno no github.
- O repositório deve ser organizado com *commits* claros, as contribuições de cada um dos alunos devem estar registradas na forma de *pull requests*.

## 16.6 Requisitos do Código

---

1. As primeiras linhas do código devem conter:

- Nomes dos integrantes do grupo, em ordem alfabética seguidos do usuário deste aluno no github.
  - Nome do grupo no ambiente virtual de aprendizagem (Canvas).
2. O programa deve receber o nome do arquivo de teste como argumento na linha de comando.
  3. O código deve ser escrito em **Python, C**, ou **\*\*\*\*C++\*\*\*\***. Com as funções nomeadas como está explicitado na [Section 16.7](#).
  4. A última versão do código Assembly gerado para o Arduino deve ser funcional e incluído no repositório.
  5. A versão em Assembly deve ser exatamente o mesmo algoritmo explicitado em cada texto de teste de forma que o resultados das expressões seja obtido pelo cálculo realizado no Arduino.

## 16.7 Divisão de Tarefas para a Fase 1

Para resolver o problema de processamento da expressões definidas nos arquivos de teste da fase 1, o trabalho será dividido entre até quatro alunos, trabalhando independentemente, na mesma sala, ou de forma remota. Cada aluno será responsável por uma parte específica do sistema, com interfaces claras para facilitar a integração. Abaixo está uma sugestão da divisão das tarefas, considerando as funções solicitadas: `parseExpressao`, `executarExpressao`, `gerarAssembly`, e `exibirResultados`.

### Warning

**Nota:** As tarefas podem ser divididas da forma que cada grupo achar mais conveniente, desde que as funções e interfaces sejam respeitadas.

### Warning

**Nota:** O vetor de *tokens* gerado pelo Analisador Léxico deve ser salvo em txt para uso nas próximas fases do projeto. Cabe ao grupo decidir o formato que será usado para salvar os *tokens*. Apenas os *tokens* referentes a última execução do código do analisador léxico devem ser salvos.

### 16.7.1 Aluno 1: Função `parseExpressao` e Analisador Léxico com Autômato Finito Determinístico

#### Responsabilidades:

- Implementar `parseExpressao(std::string linha, std::vector<std::string>& _tokens_)` (ou equivalente em Python/C) para analisar uma linha de expressão RPN e extrair *tokens*.
- Implementar o analisador léxico usando Autômatos Finitos Determinísticos (AFDs), com cada estado como uma função (ex.: `estadoNumero`, `estadoOperador`, `estadoParenteses`).
- Validar *tokens*:
  - Números reais (ex.: `3.14`) usando ponto como separador decimal;
  - Operadores (`+`, `-`, `*`, `/`, `%`, `^`);
  - Comandos especiais (`RES`, `MEM`) e parênteses;
  - Detectar erros como números malformados (ex.: `3.14.5`), parênteses desbalanceados ou operadores inválidos;

- Criar funções de teste para o analisador léxico, cobrindo entradas válidas e inválidas.

#### Tarefas Específicas:

- Escrever `parseExpressao` para dividir a linha em *tokens* usando um Autômato Finito Determinístico;
- Validar *tokens*;
- Testar o Autômato Finito Determinístico com entradas diversificadas `(3.14 2.0 +)`, `(5 RES)`, `(3.14.5 2.0 +)` (inválido);
- Criar um método, antes do Autômato Finito Determinístico, para lidar com parênteses aninhados.

#### Interface:

- Recebe uma linha de texto e retorna um vetor de *tokens*;
- Fornece *tokens* válidos para `executarExpressao`.

### 16.7.2 Aluno 2: Função `executarExpressao` e Gerenciamento de Memória

#### Responsabilidades:

- Implementar `executarExpressao(const std::vector<std::string>& _tokens_, std::vector<float>& resultados, float& memoria)` para executar uma expressão RPN;
- Gerenciar a memória `MEM` para comandos `(V MEM)` e `(MEM)`;
- Manter um histórico de resultados para suportar `(N RES)`;
- Criar funções de teste para validar a execução de expressões e comandos especiais.

#### Tarefas Específicas:

- Usar uma pilha para avaliar expressões RPN (ex.: em `C++`: `std::stack<float>`);
- Implementar operações `(+, -, *, /, %, ^)` com precisão de 16 bits (IEEE 754);
- Tratar divisão inteira e resto separadamente;
- Testar com expressões como `(3.14 2.0 +)`, `((1.5 2.0 *) (3.0 4.0 *) /)`, `(5.0 MEM)`, `(2 RES)`;
- Verificar erros como divisão por zero ou `N` inválido em `(N RES)`.

#### Interface:

- Recebe *tokens* de `parseExpressao` e atualiza `resultados` e `memoria`;
- Fornece resultados para `exibirResultados` e Assembly.

### 16.7.3 Aluno 3: Função `gerarAssembly` e Leitura de Arquivo

#### Responsabilidades:

- Implementar `gerarAssembly(const std::vector<std::string>& _tokens_, std::string& codigoAssembly)` para gerar código Assembly para Arduino;
- Implementar `lerArquivo(std::string nomeArquivo, std::vector<std::string>& linhas)` para ler o arquivo de entrada;

- Criar funções de teste para validar a leitura de arquivos e a geração de Assembly; Lembre-se o Assembly deve conter todas as operações do texto de teste.
- Alertar se o arquivo tiver linhas malformadas ou exceder limites.

**Tarefas Específicas:**

- Ler o arquivo de *tokens* linha por linha, ignorando linhas vazias;
- Gerar Assembly AVR para operações RPN e comandos especiais, usando registradores e instruções do Arduino considerando os modelos adequados a esta fase;
- Testar com arquivos contendo 10 linhas, ou mais, incluindo expressões aninhadas e comandos especiais;
- Verificar erros de abertura de arquivo e exibir mensagens claras.

**Interface:**

- `lerArquivo` fornece linhas para `parseExpressao`;
- `gerarAssembly` produz código Assembly para Arduino.

## 16.7.4 Aluno 4: Função `exibirResultados`, Interface do Usuário e Testes

**Responsabilidades:**

- Implementar `exibirResultados(const std::vector<float>& resultados)` para exibir os resultados das expressões;
- Implementar `exibirMenu()` e gerenciar a interface no `main`, incluindo leitura do argumento de linha de comando;
- Corrigir problemas de entrada (ex.: em **C++**:  
`std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');`);
- Criar funções de teste para validar a saída e o comportamento do programa completo.

**Tarefas Específicas:**

- Exibir resultados com formato claro (ex.: uma casa decimal para números reais);
- Implementar o `main` para chamar `lerArquivo`, `parseExpressao`, `executarExpressao`, e `exibirResultados`;
- Testar com arquivos de teste fornecidos, verificando saídas para expressões simples e complexas;
- Testar o FSM com comandos especiais como `(V MEM)` e `(MEM)`;

**Interface:**

- Usa resultados de `executarExpressao` para exibir saídas;
- Gerencia a execução do programa via argumento de linha de comando.

## 16.8 Considerações para Integração



- **Interfaces:** concordar com assinaturas das funções e formatos de dados (ex.: vetor de *tokens*, resultados em *float*);
- **Depuração:** testar cada parte isoladamente, simulando entradas/saídas;
- **Passos de Integração:**
  1. Copiar *main* e *exibirMenu* do Aluno 4;
  2. Inserir *lerArquivo* e *gerarAssembly* do Aluno 3;
  3. Adicionar *executarExpressao* do Aluno 2;
  4. Incluir *parseExpressao* do Aluno 1;
- **Resolução de Conflitos:** discutir problemas imediatamente na sala, ou de forma remota.
- **Depuração Final:** Testar o programa com os 3 arquivos de teste, verificando expressões, comandos especiais, e saída Assembly.

## 16.9 Avaliação

---

O trabalho será avaliado antes da prova de autoria, com os seguintes critérios:

### 1. Cálculos e Funcionalidades (70%):

- Implementação completa de todas as operações RPN e comandos especiais.
  - Cada operação não implementada reduz 10% dos 70%.
  - Falha na divisão inteira reduz 50% dos 70%.
- Analisador léxico com Autômato Finito funcional e testado.
  - Uso de Expressões Regulares reduz a nota deste item em 90%.

### 2. Organização e Legibilidade do Código (15%):

- Código claro, comentado e bem estruturado.
- Repositório GitHub organizado, com README claro.

### 3. Robustez (15%):

- Tratamento de erros em expressões complexas e entradas inválidas.
- Testes do analisador léxico cobrindo todos os casos.

## 16.10 Prova de Autoria

---

- Um aluno do grupo será sorteado, por um sistema disponível *online* para responder uma pergunta em uma lista de 10 perguntas;
- Falha na resposta reduz 35% da nota obtida na Avaliação do Projeto.

## 16.11 Entrega

- O repositório GitHub deve conter:
  - Código-fonte (Python, C, ou **C++**);
  - Três arquivos de teste com expressões RPN;
  - Funções de teste para o analisador léxico;
  - Código Assembly para Arduino da última execução do analisador léxico;
  - Arquivo de texto contendo os *tokens* gerados na última execução do analisador léxico;
  - README com instruções de compilação, execução e testes;
- O programa deve ser executado com o comando `./NomeDoSeuPrograma.cpp`.

## 17 Guia de Compilação e Execução do Código Assembly

### 17.1 Toolchain Assembly para Arduino (Linha de Comando)

O conjunto de ferramentas (toolchain) padrão, gratuito e de código aberto para a arquitetura AVR (usada no Arduino Uno/Mega) é o **AVR-GCC Toolchain** e o **AVRDUDE**. Este guia mostra como compilar um arquivo Assembly (`.s`) e enviá-lo para a placa via linha de comando, sem usar a IDE do Arduino.

#### Warning

**Nota:** O plugin platformIO do VsCode deve simplificar muito este processo. Cabe ao grupo instalar e testar este plugin na máquina que usará para fazer a prova de autoria. **Não ser capaz de gerar o código Assembly por qualquer motivo pode resultar em zeramento da tarefa.**

Você não precisará fazer nada manualmente na linha de comando. O PlatformIO gerenciara toda a cadeia de ferramentas (compilador, linker, uploader) para você, bastando clicar nos botões de Compilar e Gravar.

### 17.2 1. Ferramentas Necessárias

Você precisa instalar:

- `avr-gcc`: compilador/montador para AVR.
- `avr-libc`: biblioteca C padrão para AVR (necessária mesmo em código assembly puro porque fornece símbolos de inicialização como `__do_copy_data`, `__do_clear_bss`, etc.).
- `avrdude`: programa de upload para o microcontrolador.

### 17.3 2. Instalação

#### 17.3.1 Linux (Debian/Ubuntu)

```
sudo apt update
```



```
sudo apt install gcc-avr binutils-avr avr-libc avrdude
```

**Observação:** Para outras distribuições, use o gerenciador de pacotes correspondente (ex: `dnf` para Fedora ou `pacman` para Arch).

## 17.3.2 Windows

### 17.3.2.1 Opção 1: Microchip AVR-GCC Toolchain (Recomendado)

1. Faça o download do "AVR 8-bit Toolchain" do site da [Microchip](#).
2. Instale o `.exe` e adicione a pasta `bin` da toolchain à variável de ambiente `PATH` (ex: `C:\Program Files (x86)\Atmel\Studio\7.0\toolchain\avr8\avr8-gnu-toolchain\bin`).
3. Instale `avrdude` via [Chocolatey](#) ou [Scoop](#):

```
choco install avrdude
```

### 17.3.2.2 Opção 2: WSL (Windows Subsystem for Linux)

1. Instale o WSL e uma distribuição Linux (ex: Ubuntu) via Microsoft Store.
2. Siga as mesmas instruções de instalação para Linux dentro do ambiente WSL.

## 17.4 3. O Processo de Compilação e Upload

Assumindo que você tenha um arquivo chamado `meu_codigo.s`.

### 17.4.1 Passos Principais

1. **Montar e Linkar:** Converter o código Assembly (`.s`) em um arquivo executável no formato ELF (`.elf`).
2. **Extrair o HEX:** Converter o arquivo `.elf` para o formato Intel HEX (`.hex`).
3. **Fazer o Upload:** Enviar o arquivo `.hex` para a memória flash do Arduino usando `avrdude`.

### 17.4.2 Exemplo Completo para Arduino UNO

#### 17.4.2.1 Informações do Hardware

- **MCU:** `atmega328p`
- **Programador:** `arduino`
- **Baud Rate:** `115200`
- **Porta Serial:** `/dev/ttyACM0` (Linux) ou `COM3` (Windows)

```
# Nome do arquivo de entrada (sem extensão)
FILENAME=meu_codigo
```

```
# MCU e parâmetros do avrdude
```

```
MCU=atmega328p
AVRDUDE_PARTNO=m328p
AVRDUDE_PROGRAMMER=arduino
AVRDUDE_PORT=COM3 # Mude para sua porta. Ex: /dev/ttyACM0 no Linux
AVRDUDE_BAUDRATE=115200

# 1. Montar e Linkar (Assembly -> ELF)
# Usamos avr-gcc como front-end. Ele chamará o montador (avr-as) e o linker (avr-ld)
avr-gcc -mmcu=$MCU -o $FILENAME.elf $FILENAME.s

# 2. Extrair o arquivo .hex (ELF -> HEX)
# -O ihex: formato de saída Intel HEX
# -R .eeprom: remove a seção de dados da EEPROM do arquivo de saída
avr-objcopy -O ihex -R .eeprom $FILENAME.elf $FILENAME.hex

# 3. Fazer o Upload para o Arduino UNO
avrdude -c $AVRDUDE_PROGRAMMER -p $AVRDUDE_PARTNO -P $AVRDUDE_PORT -b $AVRDUDE_BAUDRA
```

### 17.4.2.2 Exemplo Completo para Arduino MEGA

### 17.4.3 Informações do Hardware:

- **MCU:** atmega2560
- **Programador:** wiring
- **Baud Rate:** 115200
- **Porta Serial:** /dev/ttyACM0 (Linux) ou COM4 (Windows)

```
# Nome do arquivo de entrada (sem extensão)
FILENAME=meu_codigo_mega

# MCU e parâmetros do avrdude
MCU=atmega2560
AVRDUDE_PARTNO=m2560
AVRDUDE_PROGRAMMER=wiring
AVRDUDE_PORT=COM4 # Mude para sua porta. Ex: /dev/ttyACM0 no Linux
AVRDUDE_BAUDRATE=115200

# 1. Montar e Linkar (Assembly -> ELF)
avr-gcc -mmcu=$MCU -o $FILENAME.elf $FILENAME.s

# 2. Extrair o arquivo .hex (ELF -> HEX)
avr-objcopy -O ihex -R .eeprom $FILENAME.elf $FILENAME.hex

# 3. Fazer o Upload para o Arduino MEGA
avrdude -c $AVRDUDE_PROGRAMMER -p $AVRDUDE_PARTNO -P $AVRDUDE_PORT -b $AVRDUDE_BAUDRA
```

## 17.5 Observações Importantes

### 1. Diferença entre `avr-gcc` e `avrdude`:

- `avr-gcc` usa `-mmcu=atmega328p`.
- `avrdude` usa `-p m328p`.

### 2. Bootloader:

- Pressione o botão **Reset** na placa antes de executar `avrdude` para garantir que o bootloader esteja ativo.

### 3. Exemplo Prático:

```
// meu_codigo.s
.device atmega328p
.org 0x0000
rjmp reset


reset:
    ldi r16, 0xFF
    out 0x24, r16        ; DDRB = 0xFF (PB7..PB0 como saída)
    ldi r16, 0x00
    out 0x25, r16        ; PORTB = 0x00 (todos apagados)

loop:
    out 0x25, r16        ; PORTB = valor atual
    ldi r16, 0x01
    lsr r16              ; desloca bit para baixo
    rjmp loop
```

Compile e envie:

```
avr-gcc -mmcu=atmega328p -o led.elf led.s
avr-objcopy -O ihex -R .eeprom led.elf led.hex
avrdude -c arduino -p m328p -P COM3 -b 115200 -U flash:w:led.hex
```

Copyright © 2025 Frank de Alcantara

 Edit this  
page

Report an  
issue