

## 17 Fase 2 - Analisador Sintático *LL(1)*

Este trabalho pode ser realizado em grupos de até **4** alunos. Grupos com mais de 4 membros terão o trabalho anulado. Leia todo este documento 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. Com ou sem o uso de ferramentas de Inteligência Artificial. Contudo, 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 e o cumprimento de todas as regras de entrega e desenvolvimento. Para isso, o trabalho deve ser entregue em um repositório público no GitHub. O resultado desta avaliação automatizada será a nota base do trabalho, que poderá ser alterada em resultado da prova de autoria (ver seção 10, [Section 18.10](#)).

Este é um projeto “Melhorado com o uso de Ferramentas de Inteligência Artificial”. O uso de ferramentas de Inteligência Artificial não é apenas permitido, mas incentivado para tarefas específicas. No entanto, seu uso é governado por um princípio fundamental: **Você é o arquiteto e o engenheiro; a Inteligência Artificial é sua assistente. Você mantém total responsabilidade pelo código que envia.** Você poderá usar as ferramentas de Inteligência Artificial para:

- Gerar código repetitivo (*boilerplate*) e arquivos de configuração;
- Explicar conceitos complexos e documentação;
- Depurar mensagens de erro e sugerir correções;
- Escrever testes unitários e documentação;
- Otimizar e refatorar código existente;
- Fazer *brainstorming* de ideias de projetos e arquitetura.

Não é permitido:

- **Envio às Cegas:** enviar código gerado por uma Inteligência Artificial sem revisão, compreensão ou modificação é uma violação da integridade acadêmica. Você deve ser capaz de explicar qualquer código do seu projeto, linha por linha. E provocará a anulação do trabalho.
- **Contornar o Aprendizado:** usar Inteligência Artificial para completar os objetivos centrais de aprendizado por você é proibido, por exemplo, pedir a uma Inteligência Artificial para construir um componente inteiro do projeto que você tem a tarefa de aprender.
- **Violação de Licenças:** é proibido enviar código que viole as políticas de honestidade acadêmica da universidade, a licença dos dados de treinamento, ou qualquer outra restrição legal.

### 17.1 1. Objetivo

Pesquisar e praticar os conceitos de analisadores léxico e sintático para desenvolver um programa em Python, C, ou **C++**. O objetivo é complementar o material de aula, aprimorando sua formação

acadêmica e profissional por meio da construção de um *parser*

$LL(1)$

para a linguagem de programação simplificada que está descrita neste documento, utilizando como entrada o vetor de *tokens* gerado na Fase 1.

## 17.2 2. Descrição do Trabalho

---

O objetivo é desenvolver um programa capaz de:

1. Ler um arquivo de texto contendo o código-fonte de um programa escrito na linguagem especificada neste documento (uma expressão ou declaração por linha);
2. Utilizar o *string* de *tokens* (gerado por um analisador léxico, como o da Fase 1) como entrada para o analisador sintático;
3. Implementar um analisador sintático descendente recursivo do tipo  $LL(1)$  para validar a estrutura do código;
4. Gerar a árvore sintática correspondente para cada expressão válida;
5. Implementar estruturas de controle (tomada de decisão e laços) mantendo a notação polonesa reversa;
6. Detectar e reportar erros sintáticos de forma clara e informativa.

Nesta fase, não será necessário gerar código Assembly.

### 17.2.1 2.1. Características da Linguagem

As declarações continuam sendo escritas em notação polonesa reversa (RPN), no formato  $(A\ B\ op)$ , no qual  $A$  e  $B$  são operandos (números reais ou inteiros, ou referências a memória), e  $op$  é um operador aritmético, ou *keyword*, ou identificador, entre os listados neste documento.

#### Operadores Suportados:

- **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).

#### Precisão Numérica:

A precisão dos números de ponto flutuante depende da arquitetura do processador:

- Se a arquitetura for de 8 bits: use meia precisão (16 bits, IEEE 754);
- Se a arquitetura for de 16 bits: use precisão simples (32 bits, IEEE 754);

- Se a arquitetura for de 32 bits: use precisão dupla (64 bits, IEEE 754);
- Se a arquitetura for de 64 bits: use precisão quádrupla (128 bits, IEEE 754).

As operações de divisão inteira e resto são realizadas exclusivamente com números inteiros.

### Expressões Aninhadas:

Expressões podem ser aninhadas sem limite definido, por exemplo:

- $(A (C D *) +)$ : Soma  $A$  ao produto de  $C$  e  $D$ ;
- $((A B \%) (D E *) /)$ : Divide o resto de  $A$  por  $B$  pelo produto de  $D$  e  $E$ ;
- $((A B +) (C D *) |)$ : Divide (usando divisão real) a soma de  $A$  e  $B$  pelo produto de  $C$  e  $D$ .

Nestes exemplos,  $A$ ,  $B$ ,  $C$ ,  $D$ , e  $E$  podem ser números literais ou referências a memórias.

## 17.2.2 2.2. Comandos Especiais

A linguagem inclui os mesmos comandos especiais da fase anterior para manipulação de memória e resultados:

- $(N \text{ RES})$ : Retorna o resultado da expressão  $N$  linhas anteriores ( $N$  é um inteiro não negativo);
- $(V \text{ MEM})$ : Armazena o valor real  $V$  em uma memória chamada  $\text{MEM}$ ;
- $(\text{MEM})$ : Retorna o valor armazenado em  $\text{MEM}$ . Se a memória não foi inicializada, retorna 0.

### Regras de Escopo:

- Cada arquivo de texto representa um escopo de memória independente;
- $\text{MEM}$  pode ser qualquer conjunto de letras maiúsculas (ex:  $\text{MEM}$ ,  $\text{VAR}$ ,  $\text{X}$ ,  $\text{CONTADOR}$ );
- $\text{RES}$  é uma keyword da linguagem.

## 17.2.3 2.3. Novas Estruturas: Tomada de Decisão e Laços

Você deverá criar e documentar a sintaxe para estruturas de tomada de decisão e laços de repetição. A única restrição é que estas estruturas mantenham o padrão da linguagem: devem estar contidas entre parênteses e seguir a lógica de operadores pós-fixados.

## 17.3 3. Analisador Sintático com *parser* $LL(1)$

O analisador sintático deve ser implementado como um *parser* descendente recursivo do tipo  $LL(1)$ . A implementação deve incluir:

### 17.3.1 3.1. Gramática $LL(1)$

Definir o conjunto completo de regras de produção para a linguagem, incluindo:

- Expressões aritméticas em RPN;
- Comandos especiais ( $\text{RES}$ ,  $\text{MEM}$ );
- Estruturas de controle (decisão e laços);

- Tratamento de aninhamento.

### 17.3.2 3.2. Conjuntos FIRST e FOLLOW

Calcular e documentar os conjuntos **FIRST** e **FOLLOW** para cada não-terminal da gramática. Estes conjuntos devem ser incluídos na documentação do projeto.

### 17.3.3 3.3. Tabela de Análise (*parsing* Table)

Construir a tabela de análise  $LL(1)$  baseada nos conjuntos **FIRST** e **FOLLOW**. A tabela deve mapear pares (não-terminal, terminal) para regras de produção.

### 17.3.4 3.4. *parser* Descendente Recursivo

Implementar o *parser* utilizando:

- Buffer de entrada para os *tokens*;
- Pilha de análise para controle do *parsing*;
- Funções recursivas para cada não-terminal;
- Geração da árvore sintática durante o *parsing*.

### 17.3.5 3.5. Funções de Teste

Criar funções de teste específicas para validar o analisador sintático, cobrindo:

- Expressões válidas simples e aninhadas;
- Estruturas de controle válidas;
- Entradas inválidas (erros sintáticos);
- Casos extremos (aninhamento profundo, expressões vazias);

## 17.4 4. Arquivos de Teste

---

Fornecer um mínimo de **3 arquivos de texto**, cada um com pelo menos **10 linhas** de expressões segundo as especificações deste documento.

#### Requisitos dos Arquivos de Teste:

- Cada arquivo deve incluir todas as operações (+, -, \*, |, /, %, ^);
- Cada arquivo deve incluir comandos especiais ( *(N RES)* , *(V MEM)* , *(MEM)* );
- Cada arquivo deve incluir pelo menos um laço de repetição;
- Cada arquivo deve incluir pelo menos uma tomada de decisão;
- Os testes devem incluir literais inteiros, reais e o uso de memórias (variáveis);
- Incluir casos de teste com erros sintáticos para validar o tratamento de erros.

Os arquivos devem estar no mesmo diretório do código-fonte e ser processados via argumento de linha de comando (ex.: *./AnalisadorSintatico teste1.txt*).

## 17.5 5. Hospedagem no GitHub

O projeto deve ser hospedado em um repositório público no GitHub criado exclusivamente para este projeto. O repositório deve ser criado por um dos alunos do grupo.

O repositório deve conter:

- Código-fonte do programa (Python, C, ou **C++**);
- Arquivos de teste (mínimo 3);
- Funções de teste para o analisador sintático;
- Documentação completa (README.md);
- Arquivo markdown com a gramática, conjuntos **FIRST/FOLLOW**, tabela  $LL(1)$  e a árvore sintática da última execução.

O repositório deve ser organizado com *commits* claros, as contribuições de cada um dos alunos devem estar registradas na forma de pull requests.

### Important

- O nome do repositório deve ser o nome do grupo no Canvas (ex.: **RA2\_1**);
- É fundamental que cada aluno documente suas contribuições de forma clara e detalhada;
- O aluno não pode trocar o seu usuário no GitHub durante o desenvolvimento do projeto;
- O aluno não pode alterar o repositório para privado;
- O uso de *issues* no GitHub é encorajado para discutir tarefas e bugs;

## 17.6 6. Requisitos do Código

As primeiras linhas do código devem conter:

```
// Integrantes do grupo (ordem alfabética):  
// Nome Completo 1 - username1  
// Nome Completo 2 - username2  
// Nome Completo 3 - username3  
// Nome Completo 4 - username4  
//  
// Nome do grupo no Canvas: [Nome do Grupo]
```

O programa deve receber o nome do arquivo de teste como argumento na linha de comando.

O código deve ser escrito em Python, C, ou **C++**, com as funções nomeadas como está explicitado na Seção 7 [Section 18.7](#).

## 17.7 7. Divisão de Tarefas para a Fase 2

Para resolver o problema de análise sintática da **Fase 2**, 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 de divisão das tarefas, considerando as funções solicitadas: `lerTokens`, `construirGramatica`, `parsear`, e `gerarArvore`.

#### Note

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

**Nota:** A árvore sintática gerada pelo *parser* além de estar na documentação em markdown, deve ser salva em formato texto ou JSON para uso nas próximas fases do projeto. Cabe ao grupo decidir o formato que será usado para salvar a árvore.

### 17.7.1 7.1. Aluno 1: Função `construirGramatica` e Análise $LL(1)$

#### Responsabilidades:

- Implementar `construirGramatica()` para definir as regras de produção da linguagem;
- Calcular os conjuntos **FIRST** e **FOLLOW** para cada não-terminal;
- Construir a tabela de análise  $LL(1)$ ;
- Validar que a gramática é  $LL(1)$  (sem conflitos na tabela);
- Documentar a gramática completa em formato EBNF (use letras maiúsculas para não-terminais e minúsculas para terminais).

#### Tarefas Específicas:

- Escrever as regras de produção para expressões RPN, comandos especiais e estruturas de controle;
- Implementar algoritmos para calcular **FIRST** e **FOLLOW**;
- Detectar e resolver conflitos na gramática (se houver);
- Criar funções auxiliares: `calcularFirst()`, `calcularFollow()`, `construirTabelaLL1()`;
- Testar a tabela com entradas diversas para garantir determinismo.

#### Interface:

- Entrada: Nenhuma (a gramática é fixa);
- Saída: Estrutura de dados contendo gramática, **FIRST**, **FOLLOW** e tabela  $LL(1)$ ;
- Fornece tabela  $LL(1)$  para a função `parsear()`.

### 17.7.2 7.2. Aluno 2: Função `parsear` e *parser* Descendente Recursivo

#### Responsabilidades:

- Implementar `parsear(_tokens_, tabela_ll1)` para análise sintática descendente recursiva;
- Usar a tabela  $LL(1)$  para guiar o processo de *parsing*;
- Implementar a pilha de análise e controle de derivação;
- Detectar e reportar erros sintáticos com mensagens claras;
- Criar funções de teste para validar o *parser*.

**Tarefas Específicas:**

- Implementar o algoritmo de *parsing*  $LL(1)$  com pilha;
- Criar função para cada não-terminal (*parsing* descendente recursivo);
- Gerenciar o buffer de entrada de *tokens*;
- Implementar recuperação de erros básica;
- Testar com expressões válidas e inválidas:  $(3.14\ 2.0\ +)$ ,  $((A\ B\ +)\ (C\ D\ *)\ /)$ ,  $(A\ B\ +\ C)$  (erro);

**Interface:**

- Entrada: Vetor de *tokens* e tabela  $LL(1)$ ;
- Saída: Estrutura de derivação ou erro sintático;
- Fornece estrutura de derivação para `gerarArvore()`.

### 17.7.3 7.3. Aluno 3: Função `lerTokens` e Estruturas de Controle

**Responsabilidades:**

- Implementar `lerTokens(arquivo)` para ler *tokens* salvos da Fase 1;
- Definir e implementar a sintaxe para a estrutura de decisão e repetição em notação pós-fixada;
- Criar *tokens* especiais para estruturas de controle;
- Documentar a sintaxe das novas estruturas;
- Criar arquivos de teste com estruturas de controle.

**Tarefas Específicas:**

- Ler arquivo de *tokens* no formato definido pelo grupo;
- Adicionar *tokens* para as estruturas de decisão e repetição e operadores relacionais ( $>$ ,  $<$ ,  $=$ , etc.);
- Implementar validação básica de *tokens*;
- Criar exemplos;
- Testar integração com o analisador léxico da Fase 1.

**Interface:**

- Entrada: Nome do arquivo de *tokens*;
- Saída: Vetor de *tokens* estruturado;
- Fornece *tokens* para a função `parsear()`.

### 17.7.4 7.4. Aluno 4: Função `gerarArvore`, Interface e Integração

**Responsabilidades:**

- Implementar `gerarArvore(derivacao)` para construir a árvore sintática;
- Implementar a função `main()` e gerenciar a interface;
- Criar visualização da árvore sintática (texto ou gráfica);



- Coordenar a integração de todos os módulos;
- Criar funções de teste end-to-end.

**Tarefas Específicas:**

- Transformar a derivação em estrutura de árvore;
- Implementar impressão da árvore em formato legível;
- Salvar árvore em arquivo (JSON ou formato customizado);
- Implementar o `main()` para chamar `lerTokens()`, `construirGramatica()`, `parsear()`, e `gerarArvore()`;
- Testar o sistema completo com os 3 arquivos de teste.

**Interface:**

- Entrada: Estrutura de derivação do *parser*;
- Saída: Árvore sintática em formato estruturado;
- Gerencia a execução completa via linha de comando.

## 17.8 8. Considerações para Integração

---

**Interfaces:** Concordar com assinaturas das funções e formatos de dados (ex.: estrutura de *tokens*, formato da árvore).

**Depuração:** Testar cada parte isoladamente, simulando entradas/saídas.

**Passos de Integração:**

1. Copiar `main()` e interface do Aluno 4;
2. Inserir `lerTokens()` e definições de estruturas de controle do Aluno 3;
3. Adicionar `construirGramatica()` e tabela *LL*(1) do Aluno 1;
4. Incluir `parsear()` do Aluno 2;
5. Integrar `gerarArvore()` do Aluno 4.

**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, estruturas de controle, e árvore sintática.

## 17.9 9. Avaliação

---

O trabalho será pré-avaliado de forma automática e novamente durante a prova de autoria, com os seguintes critérios:

### 17.9.1 9.1. Funcionalidades do Analisador (70%)

- Entrega e validação do *string* de *tokens*: Base da nota;



- Cada operação aritmética não identificada corretamente: **-10%**;
- Falhas na definição/implementação dos laços de repetição: **-20%**;
- Falhas na definição/implementação da tomada de decisão: **-20%**;
- Se o analisador só processar números inteiros: **-50%**;
- Falha na geração da árvore sintática: **-30%**;
- Gramática não- $LL(1)$  ou com conflitos: **-20%**.

### 17.9.2 9.2. Organização e Legibilidade do Código (15%)

- Código claro, comentado e bem estruturado
- README bem escrito contendo:
  - Nome da instituição de ensino, ano, disciplina, professor;
  - Integrantes do grupo em ordem alfabética;
  - Instruções para compilar, executar e depurar;
  - Documentação da sintaxe das estruturas de controle;
- Repositório GitHub organizado com *commits* claros.

### 17.9.3 9.3. Robustez (15%)

- Tratamento de erros em expressões complexas e entradas inválidas;
- Mensagens de erro claras indicando linha e tipo de erro;
- Testes cobrindo todos os casos (válidos e inválidos);
- Recuperação básica de erros.

**Aviso:** Trabalhos identificados como cópias terão a nota zerada.

## 17.10 10. Prova de Autoria

---

- Um aluno do grupo será sorteado usando um aplicativo online;
- Depois de explicar o projeto, o aluno sorteado escolherá um número de 1 a 10, que corresponderá a uma pergunta sobre o projeto;
- A falha na resposta, ou na explicação do projeto, implicará na redução de **35%** da nota automática do projeto para todo o grupo.

#### Important

Apesar da sugestão de divisão de tarefas, todos os alunos devem entender o funcionamento completo do projeto. O aluno sorteado para a prova de autoria deve ser capaz de responder qualquer pergunta sobre qualquer parte do projeto.

## 17.11 11. Entrega

---

O repositório no GitHub deve conter:

### 17.11.1 11.1. Código-fonte

- Programa completo em Python, C, ou **C++**;
- Todas as funções especificadas implementadas.

### 17.11.2 11.2. Arquivos de Teste

- Mínimo de 3 arquivos com 10 linhas, ou mais, cada;
- Casos de teste válidos e inválidos;
- Exemplos com estruturas de controle.

### 17.11.3 11.3. Documentação

- **README.md** bem formatado com:
  - Informações institucionais;
  - Instruções de compilação e execução;
  - Sintaxe das estruturas de controle;
  - Exemplos de uso.

### 17.11.4 11.4. Arquivo de Saída


Um arquivo markdown contendo:

- O conjunto de regras de produção da gramática;
- Os conjuntos **FIRST** e **FOLLOW**;
- A Tabela de Análise  $LL(1)$ ;
- A representação da árvore sintática de um arquivo de teste.

### 17.11.5 11.5. Formato de Execução

O programa deve ser executado com: `./AnalisadorSintatico teste1.txt`.

Copyright © 2025 Frank de Alcantara

 Edit this  
page

Report an  
issue