

## 18 Fase 3 - Analisador Semântico

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.

### 18.1 1. Objetivo

Pesquisar e praticar os conceitos de analisadores léxico, sintático e semântico desenvolvendo 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 analisador semântico sobre um

parser

$LL(1)$

para a linguagem de programação simplificada que está descrita neste documento, utilizando como entrada a árvore sintática abstrata gerada na Fase 2.

## 18.2 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). Este arquivo deve estar em formato texto simples (.txt) usando apenas caracteres ASCII;
2. Utilizar a árvore sintática abstrata gerada pelo analisador sintático da Fase 2 como entrada;
3. Criar a Gramática de Atributos da linguagem descrita neste documento;
4. Implementar um analisador semântico para criar a árvore sintática abstrata atribuída.
5. Gerar um documento de análise semântica para o julgamento de tipos e outras verificações semânticas;
6. Detectar e reportar erros léxicos, sintáticos e semânticos de forma clara e informativa.

**Observação MUITO IMPORTANTE:** A partir dessa fase, todos os três analisadores serão utilizados em conjunto. Ou seja, o arquivo de teste irá passar pelo analisador léxico (Fase 1), depois pelo analisador sintático (Fase 2) e finalmente pelo analisador semântico (Fase 3). Portanto, é fundamental que o grupo utilize o mesmo formato de *tokens* definido na Fase 1 e a mesma gramática definida na Fase 2. Qualquer divergência entre as fases resultará em erros de integração e perda de pontos na avaliação.

Nesta fase, não será necessário gerar código Assembly.

### 18.2.1 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.:  $(A\ B\ -)$ );
- **Multiplificação:**  $*$  (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 onde o programa será executado para avaliação:

- 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. **Além disso, lembre-se, para esta avaliação você deve considerar apenas a arquitetura do Arduino Uno R3, Arduino Mega, ou qualquer outro processador de 8 bits.**

### 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.

### Tipos de Dados:

A linguagem suporta três tipos de dados:

- `int`: Números inteiros
- `real` (ou `float`): Números de ponto flutuante
- `booleano`: Resultado de operações relacionais (usado internamente)

**Observação:** O tipo booleano não pode ser armazenado em memórias (MEM), sendo usado apenas como resultado de expressões relacionais em estruturas de controle.

Agora que temos laços e estruturas de decisão, nossa gramática deve ser capaz de lidar com expressões que retornam valores booleanos. Portanto, você deve implementar os seguintes operadores relacionais, tanto na gramática quanto no analisador semântico:

**Operadores Relacionais** (retornam tipo booleano): -  $>$  : maior que -  $<$  : menor que  
-  $>=$  : maior ou igual -  $<=$  : menor ou igual -  $==$  : igual -  $!=$  : diferente

Todos aceitam operandos `int` ou `real` e retornam `booleano`.

**Observação:** a inclusão dos operadores relacionais implica na necessidade de atualizar a gramática, o *parser* e o analisador semântico para suportar expressões que envolvem esses operadores, garantindo a correta verificação de tipos e a geração da árvore sintática abstrata atribuída.

## 18.2.2 2.2. Comandos Especiais

A linguagem inclui os mesmos comandos especiais da fase anterior 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, apresenta um erro semântico.

**Observação:** O tratamento de MEM, mudou desde a FASE 2. Agora, MEM retorna um erro semântico se não foi inicializada e deve ser verificada no analisador semântico.

#### Regras de Escopo:

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

### 18.2.3 2.3. Tomada de Decisão e Laços

A sua gramática agora deve incluir as estruturas de tomada de decisão e laços de repetição que foram criados na Fase 2. A sintaxe dessas estruturas deve ser definida em notação pós-fixada (RPN), e você deve criar os *tokens* necessários para representá-las.

## 18.3 3. Analisador Semântico

---

O analisador semântico que você vai criar deve ser capaz de criar a árvore sintática abstrata atribuída, aplicando as regras da Gramática de Atributos que você vai definir. Além disso, ele deve realizar as seguintes verificações semânticas:

- **Julgamento de Tipos:** Verificar se as operações são realizadas entre tipos compatíveis (inteiros e reais). Por exemplo, a operação de potência  $\wedge$  deve ter um expoente inteiro;
- **Verificação de Memória:** Garantir que as memórias são inicializadas antes de serem usadas;
- **Verificação de Comandos Especiais:** Validar o uso correto dos comandos especiais **(N RES)**, **(V MEM)**, e **(MEM)**;
- **Verificação de Estruturas de Controle:** Garantir que as estruturas de tomada de decisão e laços de repetição estão corretamente formadas e que suas condições são válidas.
- **Deteção de Erros Semânticos:** Reportar erros semânticos com mensagens claras, indicando a linha do arquivo onde o erro ocorreu e a natureza do erro.
- **Geração de Relatório:** Criar um documento de análise semântica que detalhe todas as verificações realizadas, os erros encontrados, e a estrutura da árvore sintática abstrata atribuída.

### 18.3.1 3.1 Gramática de Atributos

Você deve definir a Gramática de Atributos para a linguagem descrita neste documento. A gramática deve incluir regras para todas as operações, comandos especiais, e estruturas de controle mencionadas. Cada regra de produção deve ser acompanhada por atributos que descrevem o tipo e o valor dos elementos envolvidos.

Esta gramática de atributos será composta de atributos e regras de produção e deve estar documentada em um arquivo markdown no repositório do GitHub.

### Tipos de Atributos:

- **Atributos Sintetizados:** Calculados a partir dos atributos dos vértices filhos (propagam informação de baixo para cima na árvore).
- **Atributos Herdados:** Calculados a partir dos atributos do vértice pai ou irmãos (propagam informação de cima para baixo na árvore).

### Atributos Principais:

Para cada não-terminal e terminal da gramática, você deve definir atributos como:

- **tipo**: O tipo da expressão (inteiro, real, booleano)
- **valor**: O valor calculado da expressão (quando aplicável)
- **inicializada**: Para memórias, indica se foram inicializadas
- **escopo**: Nível de escopo da variável

### Exemplos de Regras de Produção com Atributos:

A seguir, apresentamos exemplos de como as regras de produção devem ser documentadas com seus atributos e regras semânticas:

## 18.3.2 Regras Aderentes ao Documento

### 18.3.2.1 1. Adição de Inteiros

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

### 18.3.2.2 2. Adição com Promoção de Tipo

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{float}}{\Gamma \vdash e_1 + e_2 : \text{float}}$$

Ou usando a função `promover_tipo` do documento:

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 + e_2 : \text{promover\_tipo}(T_1, T_2)}$$

### 18.3.2.3 3. Estrutura Condicional (Simplificada)

$$\frac{\Gamma \vdash e_1 : \text{booleano} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T}$$

### 18.3.2.4 4. Declaração de Variável

$$\frac{\Gamma \vdash e : T' \quad T' \leq T \quad \Gamma[x \mapsto T] \vdash e_{\text{corpo}} : T_{\text{corpo}}}{\Gamma \vdash (x : T \leftarrow e; e_{\text{corpo}}) : T_{\text{corpo}}}$$

### 18.3.2.5 5. Chamada de Função

$$\frac{\text{tabela}(f) = (T_1, \dots, T_n) \rightarrow T_{ret} \quad \Gamma \vdash e_i : T'_i \quad T'_i \leq T_i \text{ para } i = 1..n}{\Gamma \vdash f(e_1, \dots, e_n) : T_{ret}}$$

#### Observações Importantes:

a. Os exemplos de regras acima usam notação infixa padrão para facilitar o entendimento. Ao implementar o analisador semântico, você deve adaptar essas regras para a notação RPN da linguagem. Por exemplo:

- Notação infixa (nas regras): `if e1 then e2 else e3`
- Notação RPN (no código): `(e1 e2 e3 IF)`

A semântica de tipagem permanece a mesma, apenas a sintaxe de superfície muda.

- b. Todas as regras devem verificar a compatibilidade de tipos antes de realizar operações.
- c. A Tabela de Símbolos deve ser atualizada sempre que uma memória for declarada ou modificada.
- d. Operações que misturam inteiros e reais devem promover o resultado para real.
- e. Todas as verificações de erro devem incluir o número da linha onde o erro ocorreu.
- f. A gramática completa deve cobrir todas as construções da linguagem descritas neste documento.

### 18.3.2.6 Exemplo de Aplicação em RPN

Para a expressão: `(5 3 +)`

Aplicação das regras: 1.  $\Gamma \vdash 5 : \text{int}$  (literal inteiro) 2.  $\Gamma \vdash 3 : \text{int}$  (literal inteiro) 3. Pela regra de adição:  $\Gamma \vdash 5 + 3 : \text{int}$

Para a expressão: `(5.0 3 +)`

Aplicação das regras: 1.  $\Gamma \vdash 5.0 : \text{float}$  (literal real) 2.  $\Gamma \vdash 3 : \text{int}$  (literal inteiro) 3. Pela regra de promoção:  $\Gamma \vdash 5.0 + 3 : \text{float}$

## 18.4 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 semânticos 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.: `./compilar teste1.txt`).

**Atenção:** não é preciso criar sistemas de testes automatizados. Contudo, é necessário incluir no readme as rotinas de testes que foram usadas para validar o funcionamento do analisador semântico.

## 18.5 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, que será o responsável por revisar e aprovar todos os pull requests antes do merge. Os demais alunos do grupo devem contribuir por meio de *pull requests*.

**Atenção:** a colaboração entre os membros do grupo deve ser feita de forma equitativa. Todos os alunos devem contribuir com o projeto e entender o funcionamento completo do código.

O repositório deve conter:

- Código-fonte do programa (Python, C, ou **C++**);
- Arquivos de teste (mínimo 3);
- Documentação completa (README.md), constando:
  - Nome da instituição de ensino, ano, disciplina, professor;
  - Integrantes do grupo em ordem alfabética;
  - Instruções para compilar, executar e depurar o programa;
  - Exemplos de uso do programa.
- Arquivo markdown com a gramática de atributos
- Arquivo markdown com árvore sintática da última execução do código.
- Arquivo markdown com os erros semânticos detectados.
- Arquivo markdown com o julgamento de tipos descrevendo as regras de dedução criadas e onde foram aplicadas na última execução do código.

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.: `RA3_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;

## 18.6 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]
```

**Observação:** os comentários acima são um exemplo para C/C++/Rust. Em Python, use `#` no lugar de `//`.

O programa deve receber o nome do arquivo de teste como argumento na linha de comando e rodar os analisadores léxico, sintático e semântico em sequência, gravando a árvore sintática abstrata atribuída e os demais arquivos de documentação em markdown. Os erros semânticos (se houver) devem ser apresentados no console.

O código deve ser escrito em Python, C, ou **C++**, com as funções nomeadas como está explicitado na Seção a seguir:

## 18.7 7. Divisão de Tarefas para a Fase 3

Para resolver o problema de análise semântica da **Fase 3**, o trabalho será dividido entre até quatro alunos. Cada aluno será responsável por uma parte específica do sistema, com interfaces claras para facilitar a integração. Porém, todos os alunos devem entender o funcionamento completo do projeto.

### 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 abstrata atribuída, além de estar na documentação, deve ser salva em JSON para facilitar a interoperabilidade com a próxima fase da disciplina. O formato deve incluir pelo menos: tipo do vértice, tipo inferido, filhos, e número da linha.

### 18.7.1 7.1. Aluno 1: Função `definirGramaticaAtributos` e Regras Semânticas

#### Responsabilidades:

- Implementar `definirGramaticaAtributos()` para especificar as regras semânticas da linguagem.
- Definir os atributos (sintetizados e herdados) para cada símbolo da gramática.
- Documentar a gramática de atributos completa em formato EBNF, detalhando as regras de produção e as ações semânticas associadas.
- Criar a estrutura de dados para a Tabela de Símbolos (ex: identificadores, tipos, escopo).

#### Tarefas Específicas:

- Escrever as regras de verificação de tipos para operadores aritméticos.
- Definir as regras para validação de escopo e uso de memórias (`MEM`).



- Especificar as ações semânticas para as estruturas de controle (decisão e laços).
- Criar funções auxiliares: `inicializarTabelaSimbolos()`, `adicionarSimbolo()`, `buscarSimbolo()`.

**Interface:**

- Entrada: Nenhuma (a gramática de atributos é fixa).
- Saída: Estrutura de dados contendo a gramática de atributos e uma Tabela de Símbolos inicializada.
- Fornece as regras semânticas e a Tabela de Símbolos para a função `analisarSemantica()`.

## 18.7.2 7.2. Aluno 2: Função `analisarSemantica` e Verificação de Tipos

**Responsabilidades:**

- Implementar `analisarSemantica(arvoreSintatica, gramaticaAtributos, tabelaSimbolos)` para percorrer a árvore sintática abstrata.
- Aplicar as regras semânticas definidas na gramática de atributos para realizar a verificação de tipos.
- Detectar e reportar erros de tipo com mensagens claras.
- Implementar a coerção de tipos quando aplicável (ex: `inteiro` para `real` em operações mistas).

**Formato de Mensagens de Erro:**

ERRO SEMÂNTICO [Linha X]: <descrição>

Contexto: <trecho relevante do código>

``

Exemplo:

```shell

ERRO SEMÂNTICO [Linha 5]: Memória 'CONTADOR' utilizada sem inicialização

Contexto: (CONTADOR)

**Tarefas Específicas:**

- Implementar um algoritmo de percurso da árvore (ex: pós-ordem).
- Validar a compatibilidade de tipos em expressões aritméticas.
- Verificar se o expoente da potenciação é um inteiro.
- Garantir que operandos de `/` e `%` são inteiros.
- Criar funções de teste específicas para validação de tipos.

**Interface:**

- Entrada: Árvore sintática abstrata (da Fase 2), gramática de atributos e Tabela de Símbolos.
- Saída: Árvore sintática com anotações de tipo ou um erro semântico.
- Fornece a árvore anotada para a função `gerarArvoreAtribuida()`.

## 18.7.3 7.3. Aluno 3: Funções `analisarSemanticaMemoria` e `analisarSemanticaControle`

**Responsabilidades:**

- Implementar `analisarSemanticaMemoria(arvoreSintatica, tabelaSimbolos)` para validar o uso de memórias.
- Implementar `analisarSemanticaControle(arvoreSintatica, tabelaSimbolos)` para validar estruturas de controle.
- Garantir que as memórias (`MEM`) são inicializadas antes de serem lidas.
- Validar o uso correto dos comandos `(N RES)` e `(V MEM)`.
- Verificar se as condições nas estruturas de controle resultam em um valor booleano (ou equivalente).
- Implementar a lógica de escopo para as variáveis.

#### Tarefas Específicas:

- Popular a Tabela de Símbolos com informações sobre a inicialização de memórias.
- Implementar verificações para garantir que `(MEM)` não seja usado antes de `(V MEM)`.
- Validar se o `N` em `(N RES)` é um inteiro não negativo e aponta para uma expressão válida.
- Verificar se as condições em estruturas de decisão e laços são expressões válidas.
- Validar o escopo correto de variáveis dentro de estruturas de controle aninhadas.
- Criar arquivos de teste com erros semânticos relacionados a memórias e controle de fluxo.

#### Interface:

- Entrada: Árvore sintática, Tabela de Símbolos.
- Saída: Lista de erros semânticos relacionados a memórias e estruturas de controle, além da Tabela de Símbolos atualizada.
- As funções `analisarSemanticaMemoria` e `analisarSemanticaControle` devem ser chamadas sequencialmente após `analisarSemantica` do Aluno 2.
- Colabora diretamente com o Aluno 2, recebendo a árvore já anotada com tipos e complementando com validações de memória e controle de fluxo.

### 18.7.4 7.4. Aluno 4: Função `gerarArvoreAtribuida`, Interface e Integração

#### Responsabilidades:

- Implementar `gerarArvoreAtribuida(arvoreAnotada)` para construir a árvore sintática abstrata atribuída final.
- Implementar a função `main()` que gerencia a execução sequencial dos analisadores (léxico, sintático e semântico).
- Gerar os relatórios em markdown: árvore atribuída, julgamento de tipos e erros semânticos.
- Coordenar a integração de todos os módulos.

#### Tarefas Específicas:

- Transformar a árvore sintática anotada em uma estrutura final (a árvore atribuída).

- Implementar a impressão da árvore em formato de texto ou JSON.
- Salvar a árvore atribuída e os relatórios nos arquivos de saída especificados.
- Implementar o `main()` para chamar as funções das Fases 1, 2 e 3 em sequência.
- Testar o sistema completo com os 3 arquivos de teste.

**Interface:**

- Entrada: Árvore sintática anotada pela análise semântica.
- Saída: Árvore sintática abstrata atribuída e arquivos de relatório.
- Gerencia a execução completa do compilador via linha de comando.

## 18.8 8. Considerações para Integração

---

**Interfaces:** Concordar com os formatos de dados (ex.: estrutura da árvore sintática, formato da Tabela de Símbolos, árvore atribuída). **Depuração:** Testar cada módulo isoladamente antes da integração final.

**Passos de Integração:**

1. Utilizar o sistema integrado das Fases 1 e 2 como base.
2. Inserir `definirGramaticaAtributos()` e a Tabela de Símbolos do Aluno 1.
3. Integrar a função `analisarSemantica()` (desenvolvida pelos Alunos 2 e 3).
4. Integrar `gerarArvoreAtribuida()` e atualizar o `main()` (do Aluno 4) para orquestrar todo o processo.
5. Realizar testes completos com os arquivos de teste, verificando a cadeia de execução.

**Resolução de Conflitos:** Discutir problemas imediatamente na sala ou de forma remota. **Depuração Final:** Testar o programa completo, validando a detecção de erros léxicos, sintáticos e semânticos, e a geração correta dos artefatos de saída.

## 18.9 9. Avaliação

---

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

### 18.9.1 9.1. Funcionalidades do Analisador (70%)

- Implementação correta da verificação de tipos: Base da nota;
- Falha na verificação de inicialização de memória: **-20%**;
- Falha na validação das estruturas de controle: **-20%**;
- Cada regra de tipo não verificada corretamente (ex: expoente inteiro): **-10%**;
- Falha na geração da árvore sintática abstrata atribuída: **-30%**;
- Gramática de atributos incompleta ou mal documentada: **-20%**.

## 18.9.2 9.2. Organização e Legibilidade do Código (15%)

- Código claro, adequadamente comentado (funções principais e lógica complexa) e bem estruturado. Na dúvida, use ferramentas de IA para melhorar a qualidade dos comentários usando como referência o Google [C++ Style Guide](#) ou o [PEP 8](#) para Python.
- README bem escrito contendo, no mínimo:
  - 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 e *pull requests*.

**Atenção:** a participação dos integrantes do grupo é fundamental para o sucesso do projeto. Esta participação será avaliada automaticamente, o desbalanceamento na participação dos integrantes do grupo resultará na redução da nota final do trabalho.

## 18.9.3 9.3. Robustez (15%)

- Tratamento de erros semânticos com mensagens claras, indicando linha e tipo de erro.
- Testes cobrindo todos os casos semânticos (válidos e inválidos).
- Geração correta de todos os arquivos de saída em markdown.

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

## 18.10 10. Prova de Autoria

- Um aluno do grupo será sorteado usando um aplicativo online disponível em <https://frankalcantara.com/sorteio.html>.
- Depois de explicar o projeto e responder as dúvidas do professor, 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 provisória que tenha sido atribuída ao projeto. Esta redução será aplicada 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.

## 18.11 11. Entrega

A entrega será um link para um repositório do GitHub contendo:

### 18.11.1 11.1. Código-fonte

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

### 18.11.2 11.2. Arquivos de Teste

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

### 18.11.3 11.3. Documentação

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

### 18.11.4 11.4. Arquivos de Saída


Arquivos markdown gerados pela execução do programa:

- A gramática de atributos da linguagem.
- O relatório de julgamento de tipos.
- O relatório de erros semânticos encontrados.
- A representação da árvore sintática abstrata atribuída de um arquivo de teste.

### 18.11.5 11.5. Formato de Execução

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

Copyright © 2025 Frank de Alcantara

 Edit this  
page

Report an  
issue