Universidade do Minho

Departamento de Informática

Mestrado em Engenharia Informática

Engenharia Gramatical

Ana João Alves PG57505

Gonçalo Brandão PG57874

Abril de 2025

Conteúdo

1 Introdução

A análise estática de código-fonte é uma componente essencial no desenvolvimento de ferramentas que visam melhorar a qualidade, segurança e eficiência de programas. No contexto da Engenharia Gramatical, este trabalho prático propõe a criação de um analisador para a Linguagem de Programação Imperativa (LPI), desenvolvida ao longo do semestre. Utilizando o parser e visitors do módulo Lark em Python, o objetivo central é processar programas escritos em LPI e gerar um relatório em HTML que detalhe aspectos críticos, como o uso correto de variáveis, a distribuição de tipos de dados, a estrutura de instruções e o aninhamento de controlos de fluxo.

O analisador deve identificar situações problemáticas, como variáveis redeclaradas ou não declaradas, além de mapear a complexidade do código por meio da contagem de instruções e estruturas de controle. A implementação exigirá a construção de uma tabela de símbolos robusta e a análise hierárquica das estruturas da linguagem, reforçando conceitos fundamentais de processamento de linguagens e boas práticas de engenharia de software. Este trabalho não apenas consolida conhecimentos teóricos, mas também demonstra a aplicação prática de ferramentas modernas na automação de tarefas de análise estática.

2 Gramática

Esta secção descreve a estrutura sintática da nossa linguagem de programação, incluindo as principais regras de formação de variáveis, operações, e estruturas de controlo. São apresentados também os elementos terminais utilizados, além de algumas considerações importantes sobre a flexibilidade e possíveis ambiguidades da linguagem.

Estrutura Geral

- **Programa (start):**
  - Composto por uma ou mais frases (*estruturas/frases*).
- **Frases (frase):**
  - Podem ser:
    - Declaração de variável (*variavel*).
    - Estrutura condicional (*frase_condicional*).
    - Operação ou atribuição (*operacao*).
    - Definição de função (*funcao*).

Componentes Principais

1. Declaração de Variável (*variavel*)

- **Sintaxe:** `<tipo> <nome> [= <valor>];`
- **Exemplos:**

```
int x = 10;
string nome;
```

- **Características:**
  - Tipos (*def_tipo*) são palavras (ex: `int`, `string`).
  - Valores podem ser números, strings, listas (`[]`), objetos ({}), ou tuplas (`()`).

2. Operações (*operacao*)

- **Sintaxe:** `<nome> = <expressão>;`
- **Exemplo:**

```
x = 5 + y * 3;
```

- Permite operações matemáticas com `+`, `-`, `*`, etc.

3. Estruturas Condicionais (*frase_condicional*)

- Inclui `if else`, `while` e `for`.
- **Sintaxe Geral:**
- **Condições Lógicas (*condicao_logica*):**
  - Comparações: `x > 5`
  - Expressões booleanas: `x AND y`
  - Variáveis simples
  - **Operadores:** `==`, `!=`, `<`, `>`, `AND`, `OR`

4. Funções (*funcao*)

- **Sintaxe:** `<tipo> <nome>() { ... }`
- **Exemplo:**

```
int soma() {
    return x + y;
}
```

Tokens (Elementos Básicos)

- **Tipos de Dados (TIPO):** Palavras como `int`, `float` (*definidas por regex*: `[A-Za-z]+`).
- **Números (NUM):** Sequências de dígitos (ex: `123`).
- **Strings/Nomes (STR):** Letras, números ou underscores (ex: `var1`, `nome`).
- **Operadores:**
  - Atribuição: `=`
  - Aritméticos: `+`, `-`, `*`, etc.
  - Relacionais: `==`, `<`, etc.
  - Lógicos: `AND`, `OR`

3 Scripts e Funcionamento

3.1 verificador_erros.py

O código implementa um interpretador semântico usando a biblioteca Lark, com foco na verificação de erros estruturais na nossa linguagem de programação. A classe VerificadorErrosInterpreter percorre a árvore sintática gerada por um parser e identifica problemas como ausência de ponto e vírgula, blocos condicionais ou funções sem conteúdo e uso incorreto de estruturas como else após while. Ele realiza verificações detalhadas em declarações de variáveis, operações, estruturas condicionais e funções. Cada erro detectado é registado com uma mensagem descritiva e a linha correspondente, se disponível. O interpretador também analisa o balanceamento de chavetas e parênteses, além de validar a presença de expressões lógicas em condições. A modularidade permite fácil extensão para novos tipos de regras. Com isso, o código promove a escrita correta e estruturada de programas.

3.2 tp_interpretador.py

Este código é um interpretador estático, sendo projetado para analisar e depurar códigos-fonte. Ele realiza a verificação de erros (como variáveis não declaradas ou não inicializadas), estruturas desbalanceadas ou uso incorreto de else. Além disso, reune informações sobre métricas detalhadas: contagem de variáveis (tipos, escopo global/local), ciclos (if, while, for) e funções definidas. Gera um relatório estruturado com problemas como variáveis não inicializadas, redundâncias ou más práticas, e integra sugestões de otimização (ex: simplificação de condições). A saída é formatada para fácil visualização em interfaces, ajudando a corrigir possíveis erros, entender a estrutura do código e melhorar eficiência. Assim, é proporcionado um ambiente para debugging e code review.

Concluíndo, integra-se a verificadores externos para análises semânticas e sintáticas, servindo como núcleo do trabalho.

3.3 Otimizacao_if_clauses.py

Este código é um otimizador de estruturas condicionais que identifica ifs aninhados passíveis de simplificação. A sua função é analisar condições encadeadas (ex: if dentro de if) e sugere combiná-las numa única verificação lógica (ex: if (A AND B)), armazenando-as em self.sugestoes. Para além disso, utiliza uma pilha (contexto_ifs) para monitorar ifs externos e verifica se o bloco interno é a única instrução do bloco externo. Extrai condições lógicas e conteúdo dos blocos, gerando mensagens explicativas com trechos de código original e otimizado, incluindo números de linha. O objetivo é melhorar a legibilidade e eficiência, reduzindo aninhamentos desnecessários, promovendo a escrita de código mais limpo e eficiente.

3.4 script_html.py

Este sistema gera um **relatório HTML interativo** voltado para análise estática de programas, organizando informações relevantes em seções visuais claras. O relatório apresenta:

- **Variáveis problemáticas**: exibe variáveis redeclaradas ou não declaradas em tabelas categorizadas.
- **Agrupamento por tipo**: organiza variáveis por tipo (`int`, `string`, etc.), mostrando seus valores e escopos.
- **Estruturas de controle**: contabiliza o uso de `if`, `while`, `for`, com detalhes sobre suas condições.
- **Sugestões de otimização**: indica formas de simplificar condições aninhadas, oferecendo exemplos de código alternativos.
- **Funções declaradas**: apresenta uma lista com tipo de retorno, nome e conteúdo das funções definidas no programa.

O relatório utiliza **CSS para um design responsivo**, com tabelas legíveis. Todo o conteúdo é dinamicamente gerado via Python, convertendo dados estruturados em `HTML` bem formatado.

4 Resultados

Nesta secção iremos apresentar os resultados do nosso sistema, baseados nos testes de código-base na pasta testes. Assim sendo, as subsecções seguintes abordam toda a informação que um utilizador da nossa linguagem terá acesso.

4.1 Listagem das variáveis do programa e scope

Como resultado apresentamos uma lista das variáveis, sendo possível ter uma visão geral dos elementos atómicos do nosso programa. Adicionalmente, é possível identificar o scope para cada uma delas e o tipo de utilização das diferentes variáveis, ou seja, para garantir a coerência ao longo do nosso programa, consideramos fundamental identificar quais as variáveis que são declaradas. Evitando possíveis erros sintáticos ou semânticos.

Figura 1: Análise das variáveis

4.2 Estruturas aninhadas

Uma vez que o nosso programa permite a utilização de ciclos aninhados, incluímos um caso de uso do mesmo e a sua análise, constituída pela condição e conteúdo condicional.

```
    if (x) {
    a = 5;
    if (y) {
        a = 3;
    }
}

if (a > 5) {
    if (c <= 10) {
        int a = 3;
    }
}


if (x > 5) {
    a = 10;
}
else {
    a = 0;
}

if (cond) {
    while (x < 10) {
        x = x + 1;
    }
}
else {
    y = 20;
}


for (int i = 0 ; i < 10; i = i + 1) {
        x = x + 1;


}
```

Figura 2: Código-fonte com ciclos aninhados

Figura 3: Análise dos ciclos aninhados

4.3 Ifs aninhados que possam ser suprimidos

Nesta subsecção verificamos situações em que ifs aninhados podem ser substituídos por um só if.

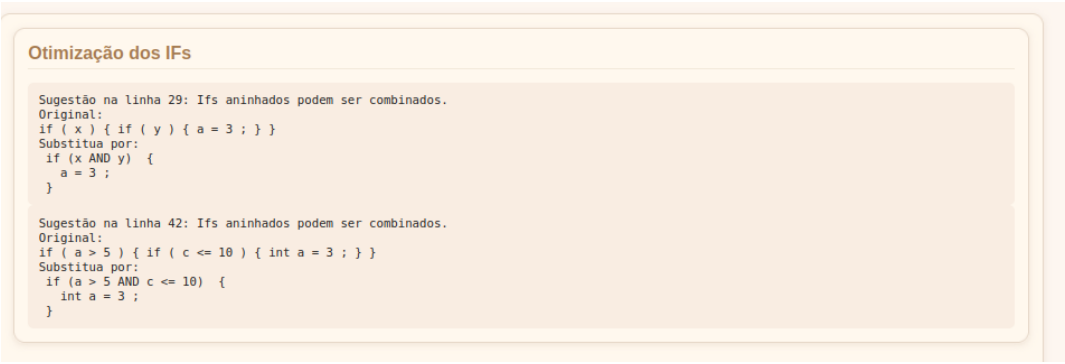

Figura 4: Otimização dos ciclos condicionais

4.4 Erros

Nesta subsecção verificamos situações em que a gramática não é respeitada, apresentando erros que devem ser corrigidos para o código ser devidamente analisado. Na nossa perspectiva, a correção de erros é essencial para ser possível obter uma análise detalhada do código.

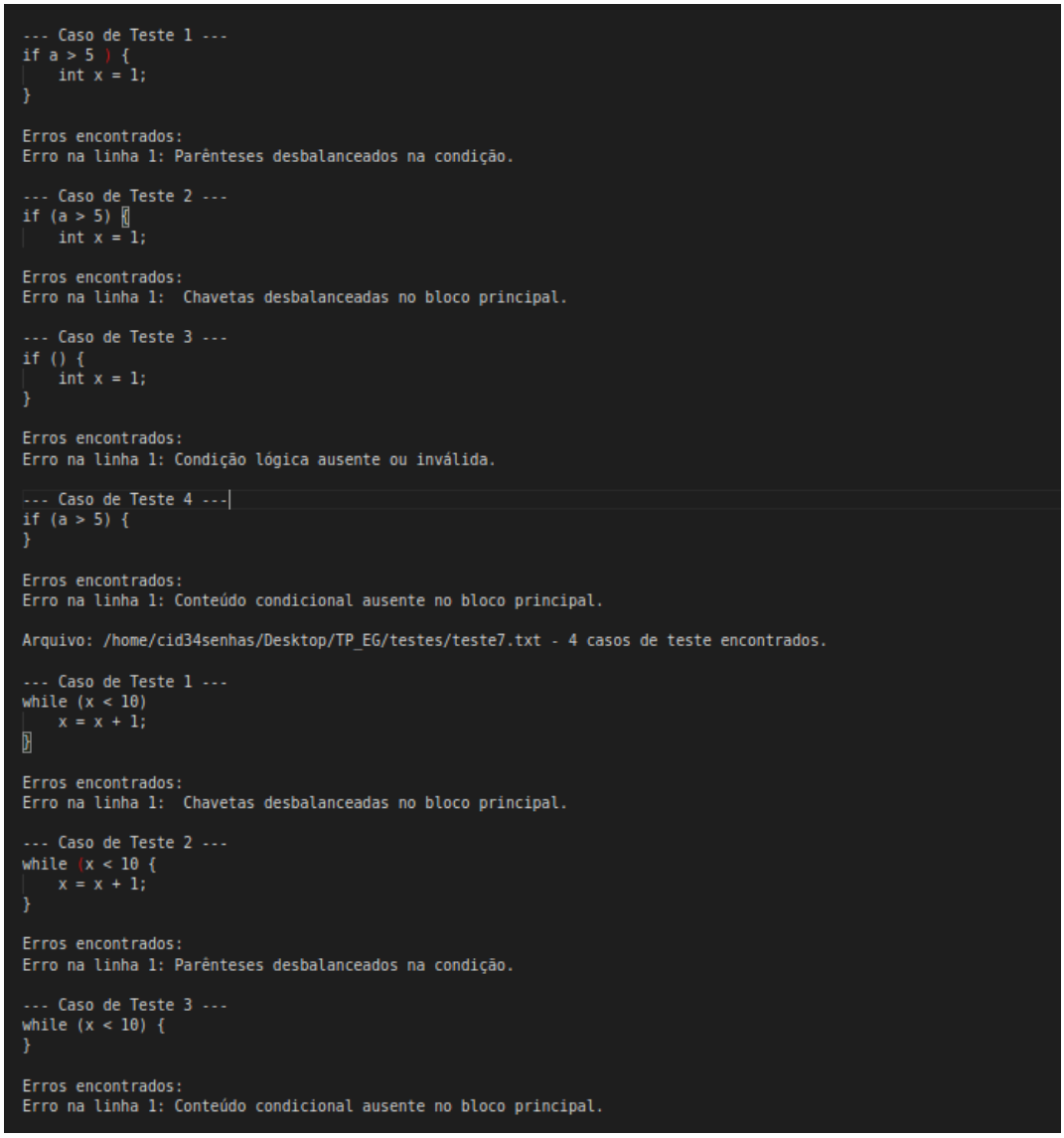

Figura 5: Análise de erros

```
--- Caso de Teste 4 ---
while () {
    x = x + 1;
}

Erros encontrados:
Erro na linha 1: Condição lógica ausente ou inválida.

Arquivo: /home/cid34senhas/Desktop/TP_EG/testes/teste8.txt - 5 casos de teste encontrados.

--- Caso de Teste 1 ---
for (int i = 0 i < 10; i = i + 1) {
    x = x + 1;
}

Erros encontrados:
Erro na linha 1: Estrutura do 'for' inválida. Use 2 ';'.

--- Caso de Teste 2 ---
for (int i = 0; i < 10; i = i + 1) {
}

Erros encontrados:
Erro na linha 1: Conteúdo condicional ausente no bloco principal.

--- Caso de Teste 3 ---
for () {
    int  x = 1 ;
}

Erros encontrados:
Erro na linha 1: Condição lógica ausente ou inválida.

--- Caso de Teste 4 ---
if (int i = 0; i < 10; i = i + 1) {
    int  x = 1 ;
}

Erros encontrados:
Erro na linha 1: Estrutura do 'for' inválida. Esperado FOR e recebeu IF.

--- Caso de Teste 5 ---
while (int i = 0; i < 10; i = i + 1) {
    int  x = 1 ;
}

Erros encontrados:
Erro na linha 1: Estrutura do 'for' inválida. Esperado FOR e recebeu WHILE.

Arquivo: /home/cid34senhas/Desktop/TP_EG/testes/teste9.txt - 1 casos de teste encontrados.

--- Caso de Teste 1 ---
if (a > 5) {
    x = 1;
}
else {

}

Erros encontrados:
Erro na linha 4: Conteúdo ausente no 'else'.
```

Figura 6: Análise de erros

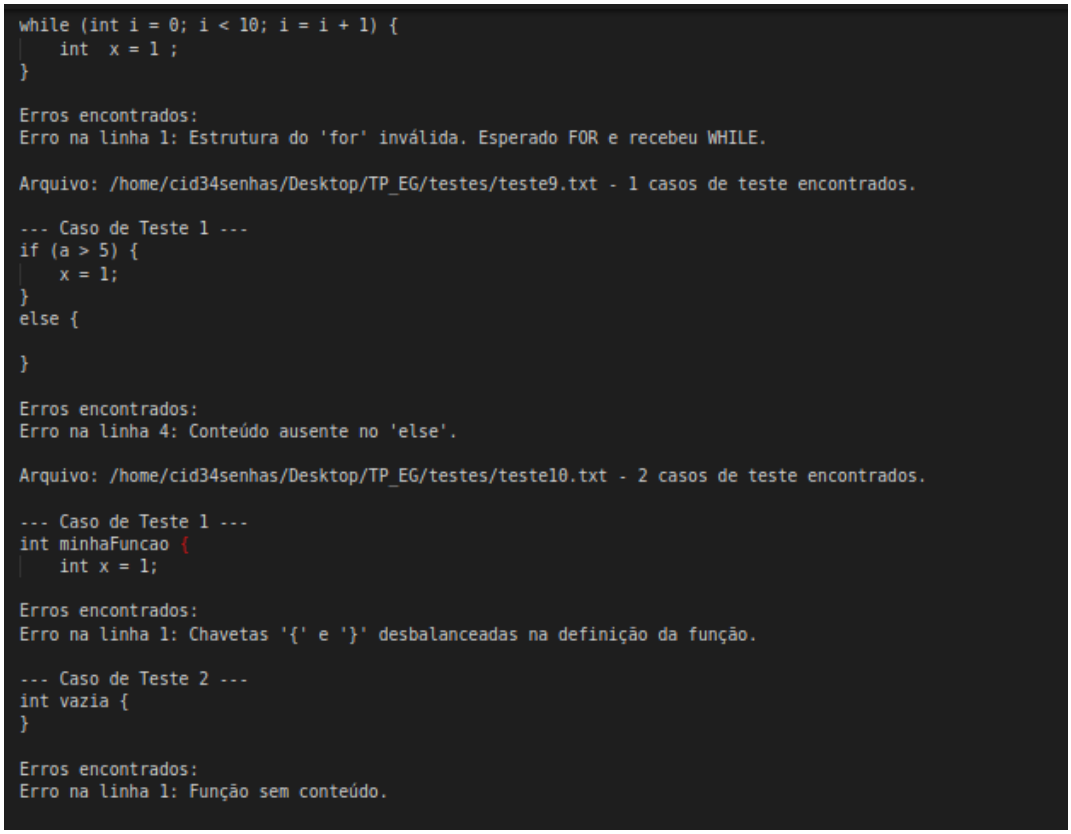

Figura 7: Análise de erros

5 Conclusão

A implementação do analisador para a Linguagem de Programação Imperativa (LPI) permitiu explorar técnicas avançadas de processamento de linguagens, destacando a importância de ferramentas como o Lark para a construção de parsers e a realização de travessias sintáticas. Através da tabela de símbolos e da estrutura de controle desenvolvidas, foi possível identificar padrões críticos no código, como variáveis mal gerenciadas e estruturas aninhadas passíveis de otimização.

Os resultados obtidos — desde a deteção de inconsistências até a geração de métricas estruturais — reforçam a relevância de análises estáticas para a melhoria contínua de código. Além disso, o trabalho evidenciou desafios práticos, como a gestão eficiente de contextos de escopo e a interpretação de aninhamentos complexos, que exigem uma abordagem sistemática. Em síntese, este projeto não apenas cumpre os requisitos técnicos propostos, mas também consolida competências em engenharia de linguagens, preparando os estudantes para enfrentar problemas reais de análise e otimização de software.