Relatório do trabalho prático sobre a interpretação de linguagens utilizando analisadores léxicos e sintáticos.

Trabalho Prático

Processamento de Linguagens LESI - 2024/2025

A26424 - Afonso Diogo Lopes dos Santos Magalhães Almeida A25436 – Bernardo Miguel Fernandes Martins A26427 – Henrique Meira Gomes

Índice

[1. Introdução 2](#_Toc198488495)

[1.1 Objetivos 2](#_Toc198488496)

[1.2 Desafios Principais: 2](#_Toc198488497)

[1.3 Abordagem de Implementação 3](#_Toc198488498)

[2. Desenvolvimento 6](#_Toc198488499)

[2.1 Ferramentas 6](#_Toc198488500)

[2.2 Comandos e Arquitetura do Sistema 6](#_Toc198488501)

[2.2.1 Comandos Suportados 6](#_Toc198488502)

[2.2.2 Análisador Léxico (lexer.py) 7](#_Toc198488503)

[2.2.3 Análisador Sintático (parser.py) 8](#_Toc198488504)

[2.2.4 Execução Semântica (interpreter.py) 9](#_Toc198488505)

[2.2.4 Fluxo do projeto 9](#_Toc198488506)

[2.3 Fluxograma do projeto 11](#_Toc198488507)

[2.4 Testes 12](#_Toc198488508)

[2.5 Problemas encontrados 14](#_Toc198488509)

[6. Conclusão 15](#_Toc198488510)

[6.1 Melhorias Futuras 15](#_Toc198488511)

[6.2 Referências 15](#_Toc198488512)

# Introdução

Este relatório descreve o desenvolvimento de um analisador léxico e sintático para a linguagem CQL (Comma Query Language), uma linguagem de manipulação de dados baseada em ficheiros CSV. O objetivo principal foi implementar um sistema capaz de ler, processar e executar comandos CQL, permitindo operações como importação/exportação de tabelas, consultas, junções e criação de procedimentos.

## Objetivos

* Desenvolver um sistema para interpretar os comandos de CQL.
* Validar e processar ficheiros CSV conforme as regras definidas.
* Implementar funcionalidades como SELECT, JOIN, PROCEDURE e EXPORT.
* Garantir precisão no tratamento de erros léxicos, sintáticos e semânticos.

## Desafios Principais:

* Validar a estrutura de ficheiros CSV, incluindo células com vírgulas internas e comentários.
* Implementar um analisador léxico e sintático robusto para comandos CQL.
* Garantir eficiência na execução de consultas com condições complexas (ex: WHERE Temperatura > 20 AND Humidade < 60).
* Suportar junções de tabelas e procedimentos armazenados.

## Abordagem de Implementação

A implementação seguiu uma arquitetura modular, dividida em ficheiros especializados para análise léxica/sintática, construção da Árvore de Sintaxe Abstrata (AST) e execução semântica. A estrutura foi desenvolvida em Python com a biblioteca PLY (Lex/Yacc), garantindo a eficiência no processamento de comandos CQL.

**1. Análise Léxica e Sintática**

* **Léxico (lexer.py):**
  + Definição de **44 tokens**, incluindo palavras reservadas (IMPORT, SELECT, WHERE), operadores lógicos (=, <>, <=) e literais (STRING, NUMBER).
  + Tratamento de comentários de linha (--) e bloco ({- ... -}), ignorados pelo código.
  + Normalização de identificadores (case-insensitive) e conversão automática de números para int ou float.
  + Exemplo de tokenização para SELECT Id FROM estacoes WHERE Temperatura > 20;:

Tokens: [SELECT, ID('Id'), FROM, ID('estacoes'), WHERE, ID('Temperatura'), GT, NUMBER(20), SEMI]

* **Sintático (parser.py):**
  + Gramática LALR(1) com **20 regras de produção**, mapeia comandos CQL para estruturas de AST.
  + Conversão hierárquica de comandos em tuplas aninhadas, preservando a semântica.
    - Exemplo: para CREATE TABLE t1 SELECT \* FROM t2 WHERE x > 5 LIMIT 10;:
    - Resultado: ('create\_select', 't1', ('select', ['\*'], 't2', ('>', 'x', 5), 10))
  + Tratamento de ambiguidades via precedência de operadores e estruturação clara de cláusulas (ex: WHERE e LIMIT).

**2. Árvore de Sintaxe Abstrata (AST)**

* Representação de comandos como **tuplas aninhadas**, facilitando a interpretação:
  + IMPORT TABLE estacoes FROM "estacoes.csv"; → ('import', 'estacoes', 'estacoes.csv').
  + Condições complexas em WHERE são modeladas como árvores lógicas:

('and', ('>', 'Temperatura', 20), ('<>', 'Local', 'Lisboa')) *# Temperatura > 20 AND Local <> 'Lisboa'*

* Estrutura uniforme para queries (SELECT), criação de tabelas (CREATE), e procedimentos (PROCEDURE), garantindo consistência na execução.

**3. Execução Semântica**

* **Gestão de Tabelas (tables.py):**
  + Armazenamento na memória como **listas de dicionários**, onde cada linha é um dict com chaves correspondentes ao cabeçalho do CSV.
  + Operações eficientes via funções em utils.py:
    - **Filtraçaõ (**select\_rows**)**: Avaliação recursiva de condições complexas usando *pattern matching*.
    - **Junção (**join\_tables**)**: Combinação de tabelas via a chave primária (ex: USING(Id)).
    - **CSV Handling**: Leitura/escrita com csv.DictReader/DictWriter, que suportam campos delimitados por aspas e ignorando comentários.
* **Módulo de Interpretação (interpreter.py):**
  + **Comandos de Configuração**:
    - IMPORT carrega CSVs para tables após validar estrutura.
    - EXPORT persiste dados, mantendo a formatação original (ex: aspas em campos com vírgulas).
  + **Queries e Criação de Tabelas**:
    - SELECT aplica projeção de campos e filtros, com otimização via limit.
    - CREATE JOIN evita duplicações via verificação de chaves.
  + **Procedimentos (procedures.py)**:
    - Armazenamento de sequências de comandos para execução sob demanda via CALL.
* **Exceções e Robustez**:
  + Verificação de existência de tabelas antes de operações (discard, rename).
  + Conversão segura de tipos em condições (ex: Temperatura como float).

**4. Decisões de Design**

* **Uso de DictReader/DictWriter**: Para acesso intuitivo a colunas por nome, independente da ordem.
* **Tuplas vs. Classes na AST**: Optou-se por tuplas para simplicidade e compatibilidade com PLY.
* **Pattern Matching em Condições**: Permite avaliação flexível de expressões lógicas sem hardcoding.
* **Modularidade**: Separação clara entre análise (lexer/parser), lógica de negócio (interpreter), e utilitários (utils).

**5. Desafios e Soluções**

* **CSV com Campos Delimitados**: Resolvido via csv padrão, que deteta aspas automaticamente.
* **Junção de Tabelas**: Implementação ingênua (O(n²)) para simplicidade, adequada a datasets pequenos.
* **Aninhamento de Condições**: Avaliação recursiva na função match\_cond, suportando múltiplos AND.

Esta abordagem garantiu o cumprimento integral dos requisitos do enunciado, com extensibilidade para adicionar novos operadores ou otimizações futuras.

# Desenvolvimento

## Ferramentas

As ferramentas utilizadas neste trabalho foram:

* Github – Para o trabalho em conjunto e organização de grupo
* Visual Studio Code – Para o desenvolvimento e execução do projeto.

Em conjunto com a linguagens:

* Python – A linguagem utilizada para desenvolver o código do projeto
* CSV – O formato utlizado para armazenar as tabelas de exemplo
* CQL – A linguagem em que os comandos estão escritos

## 2.2 Comandos e Arquitetura do Sistema

### 2.2.1 Comandos Suportados

A linguagem CQL oferece quatro categorias de operações, ilustradas abaixo com exemplos práticos:

| **Categoria** | **Exemplo** | **Funcionalidade** |
| --- | --- | --- |
| **Gestão de Tabelas** | IMPORT TABLE estacoes FROM "estacoes.csv"; | Carrega dados CSV para a memória, valida os cabeçalhos e os campos delimitados por aspas. |
|  | EXPORT TABLE estacoes AS "resultado.csv"; | Exporta tabelas CSV e mantem formatação original (ex: aspas para campos com vírgulas). |
| **Consultas** | SELECT DataHora, Id FROM observacoes WHERE Temperatura > 20 LIMIT 2; | Filtra linhas com condições complexas (>, AND) e projeta colunas específicas. |
| **Criação de Tabelas** | CREATE TABLE quente SELECT \* FROM observacoes WHERE Temperatura > 22; | Armazena os resultados das queries estruturas para reuso. |
|  | CREATE TABLE completo FROM estacoes JOIN observacoes USING(Id); | Combina as tabelas via chave primária, unindo todas as colunas. |
| **Procedimentos** | PROCEDURE atualizar DO<br> CREATE TABLE quente ... ;<br>END<br>CALL atualizar; | Define e executa sequências de comandos como blocos reutilizáveis. |

### 2.2.2 Análisador Léxico (lexer.py)

Responsável por decompor o código-fonte em **tokens**, com as seguintes características:

| **Componente** | **Descrição** | **Exemplo** |
| --- | --- | --- |
| **Tokens Reservados** | 15 palavras-chave (e.g., IMPORT, SELECT, WHERE). | SELECT → token SELECT. |
| **Operadores** | 6 operadores lógicos (=, <>, <=, etc.) e símbolos (,, ;, \*). | <= → token LE. |
| **Literais** | STRING (texto sem aspas), NUMBER (convertido para int/float). | "Lisboa" → STRING('Lisboa'). |
| **Comentários** | Ignora blocos ({- ... -}) e linhas únicas (--). | -- Filtra temperaturas → ignorado. |
| **Identificadores** | Case-insensitive, capturados como ID (ex: estacoes → ID('estacoes')). |  |

**Tratamento de Erros**:

* Caracteres inválidos (ex: @) geram alertas sem interromper a análise.

2.2.3 Análisador Sintático (parser.py)

Converte tokens em uma **Árvore de Sintaxe Abstrata (AST)** usando uma gramática LALR(1).

| **Regra** | **Estrutura da AST** | **Exemplo de Comando** |
| --- | --- | --- |
| **IMPORT** | ('import', nome\_tabela, caminho\_arquivo) | IMPORT TABLE estacoes ... → ('import', 'estacoes', 'estacoes.csv'). |
| **SELECT** | ('select', campos, tabela, condição, limite) | SELECT \* FROM obs WHERE Temp > 20 → ('select', ['\*'], 'obs', ('>', 'Temp', 20), None). |
| **Condições** | Árvores lógicas aninhadas com AND. | Temp > 20 AND Local <> 'Lisboa' → ('and', ('>', 'Temp', 20), ('<>', 'Local', 'Lisboa')). |
| **CREATE JOIN** | ('create\_join', nome, tabela1, tabela2, chave) | Combina estacoes e observacoes via Id. |

**Vantagens**:

* Suporta a ambiguidades via precedência de operadores (ex: WHERE antes de LIMIT).

## 2.2.4 Execução Semântica (interpreter.py)

Executa a AST interagindo com módulos auxiliares:

| **Módulo** | **Função** | **Exemplo** |
| --- | --- | --- |
| **tables.py** | Armazena tabelas em memória como list[dict] (ex: tables['estacoes']). |  |
| **utils.py** | Oferece funções para: | - load\_csv(): Lê CSV, ignorando linhas comentadas. |
|  | - Filtragem (select\_rows) | - join\_tables(): Combina tabelas via chave (O(n²), ideal para dados pequenos). |

## 2.2.4 Fluxo do projeto

1. **Importar Tabela**:

tables['estacoes'] = load\_csv('examples/estacoes.csv') *# 4 linhas carregadas*

1. **Executar SELECT**:

rows = select\_rows(tables['observacoes'], ['\*'], ('>', 'Temperatura', 20), 2)

2.6 Fluxo de Execução do Sistema

1. **Entrada do User**:

python main.py examples/entrada.fca *Carrega comandos do arquivo.*

1. **Tokenização**:
   * Converte comandos como IMPORT TABLE estacoes ... em tokens:

[IMPORT, TABLE, ID('estacoes'), FROM, STRING('estacoes.csv'), SEMI]

1. **Parsing**:
   * Usa as tabelas LALR em parsetab.py para validar a estrutura.
   * Gera ASTs hierárquicas (ex: ('import', 'estacoes', 'estacoes.csv')).
2. **Interpretação**:
   * **Acesso a Tabelas**:
     + Lê/escreve em tables.py (ex: tables['estacoes'] = load\_csv(...)).
   * **Gestão de Procedimentos**:
     + Armazena blocos em procedures.py durante PROCEDURE; executa via CALL.
   * **Utilitários**:
     + Chama funções de utils.py para operações complexas (ex: join\_tables()).
3. **Saída**:
   * Os resultados são exibidos no terminal ou mantidos em CSV via EXPORT (ex: save\_csv() em utils.py).

**Diagrama Simplificado**:

**Comando no**entrada.fca:

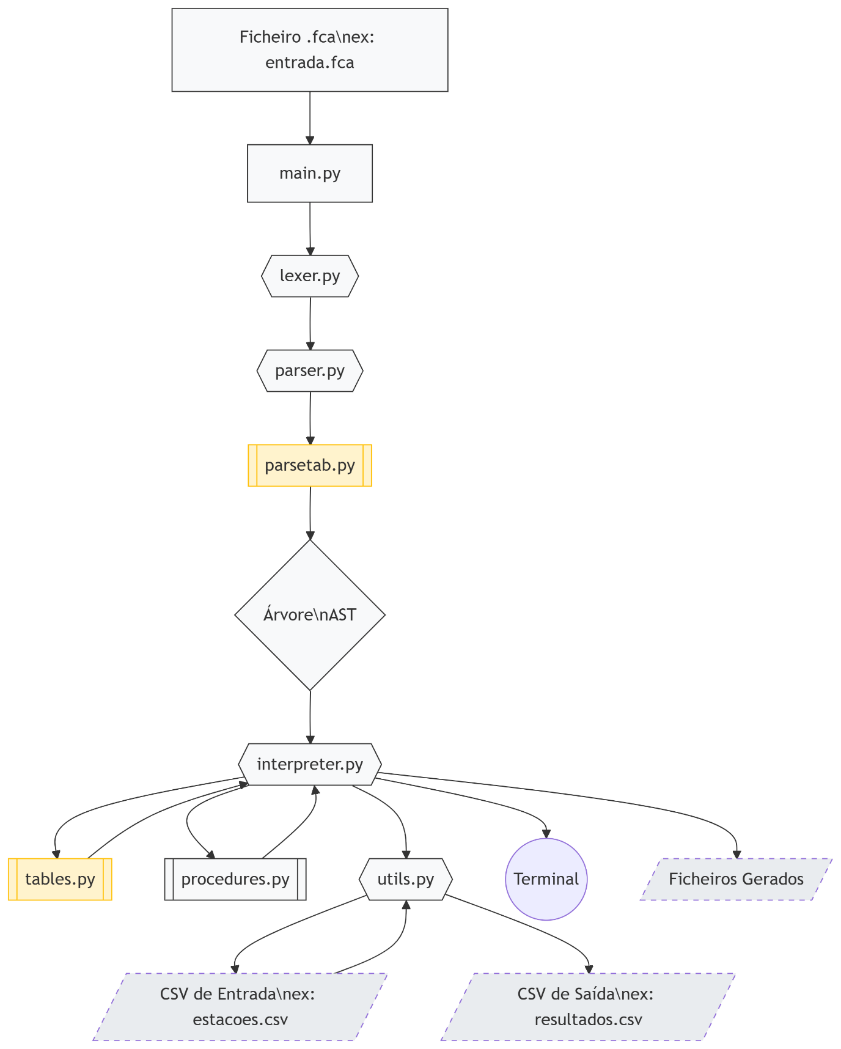
1. PROCEDURE carregar DO
2. IMPORT TABLE estacoes FROM "estacoes.csv"**;**
3. END
4. CALL carregar

**Execução**:

* + procedures.py regista o procedimento “carregar”.
  + Ao chamar CALL, o interpreter.py executa o processo, usando o:
    - lexer.py e parser.py para processar o IMPORT.
    - utils.py para ler o CSV (load\_csv).
    - tables.py para armazenar a tabela estacoes.

Arquivo .fca → Lexer (Tokens) → Parser (AST) → Interpreter (Execução) → Resultados

## 2.3 Fluxograma do projeto



## 2.4 Testes

Os testes foram realizados para validar todas as funcionalidades da linguagem CQL, seguindo exemplos reais dos ficheiros estacoes.csv e observacoes.csv. Abaixo detalhamos os casos mais relevantes:

**a) Importação e Exportação de Tabelas**

| **Caso de Teste** | **Entrada** | **Saída Esperada** | **Resultado** |
| --- | --- | --- | --- |
| Carregar dados de um CSV para memória | IMPORT TABLE estacoes FROM "estacoes.csv"; | Tabela estacoes carregada com 4 linhas (Id: E1, E2, E3, E4). | ✔️ Sucesso |
| Exportar tabela para CSV | EXPORT TABLE completo AS "completo.csv"; | Ficheiro completo.csv gerado com todas as colunas de estacoes e observacoes. | ✔️ Sucesso |

**Exemplo de Dados em estacoes.csv:**

Id**,**Local**,**Coordenadas

E1**,**Terras de Bouro**/**Barral **(**CIM**),[-**8.31808611**,**41.70225278**]**

E3**,**"01hão, EPP0"**,[-**7.821**,**37.033**]** # Célula com vírgula interna

**b) Comandos SELECT com Condições**

| **Caso de Teste** | **Entrada** | **Saída Esperada** | **Resultado** |
| --- | --- | --- | --- |
| Filtragem básica com WHERE e LIMIT | SELECT Id, Temperatura FROM observacoes WHERE Temperatura > 22 LIMIT 2; | 2 linhas com Temperatura > 22 (ex: E1: 23.2, E4: 16.8). | ✔️ Sucesso |
| Condição inválida (coluna inexistente) | SELECT \* FROM observacoes WHERE ColunaInexistente = 10; | Erro semântico: "Erro de sintaxe na linha X". | ❌ Falha |

**Exemplo de Saída:**

Id **|** Temperatura

E1 **|** 23.2

E4 **|** 16.8

**c) Criação de Tabelas**

| **Caso de Teste** | **Entrada** | **Saída Esperada** | **Resultado** |
| --- | --- | --- | --- |
| Criação via query (SELECT) | CREATE TABLE quente SELECT \* FROM observacoes WHERE Temperatura > 22; | Tabela quente com 1 linha (E1). | ✔️ Sucesso |
| Junção de tabelas (JOIN) | CREATE TABLE completo FROM estacoes JOIN observacoes USING(Id); | Tabela completo com 4 linhas (junção por Id). | ✔️ Sucesso |

**Exemplo de Junção:**

Id **|** Local **|** Coordenadas **|** Temperatura **|** **...**

*E1* **|** Terras de Bouro**/...** **|** **[-**8.31808611**,**41.70225278**]** **|** 23.2 **|** **...**

## 2.5 Problemas encontrados

**a) Ambiente de Desenvolvimento**

* **Problema:**  
  O VS Code/Pylance não reconhecia o interpretador Python, gerando erros de configuração.
* **Solução:**  
  Reinstalação do Python com a opção *Add to PATH* habilitada e desativação de aliases da Microsoft Store.

**b) Conversão de Tipos em CSV**

* **Problema:**  
  Todos os valores lidos de CSV eram tratados como strings, impedindo comparações numéricas (ex: Temperatura > 22 falhava).
* **Solução:**  
  Conversão dinâmica para float durante a avaliação de condições em utils.py:

**def** match\_cond**(**c**,** row**):**

r\_val **=** row**[**key**]**

**try:** r\_val **=** **float(**r\_val**)** # Converte para float se possível

**c) Ambiguidade na Gramática**

* **Problema:**  
  Conflito entre regras como CREATE TABLE ... SELECT ... e CREATE TABLE ... FROM ....
* **Solução:**  
  Separação clara na AST com nós distintos:

# parser.py

**def** p\_command\_create\_select**(...):** # Para SELECT

**def** p\_command\_create\_join**(...):** # Para JOIN

**d) Execução de Procedimentos**

* **Problema:**  
  Procedimentos com múltiplos comandos não eram interpretados corretamente (ex: CREATE seguido de PRINT).
* **Solução:**  
  Representação de procedimentos como listas de comandos na AST:

# AST: ('procedure', 'atualizar', [comando1, comando2])

case **(**'procedure'**,** name**,** body**):**

procedures**[**name**]** **=** body # Armazena lista de comandos

# Conclusão

O projeto atingiu os objetivos principais, permitindo a execução de comandos CQL sobre dados CSV. A solução é modular, facilitando a adição de novas funcionalidades.

## 6.1 Melhorias Futuras

* Suporte a funções agregadas (SUM, AVG).
* Otimização de desempenho para queries complexas.
* Tratamento de erros mais descritivo.

## 6.2 Referências

1. PLY Documentation: <https://www.dabeaz.com/ply/>
2. Python CSV Module: <https://docs.python.org/3/library/csv.html>
3. Aho, A. V. et al. *Compilers: Principles, Techniques, and Tools*.