

# Trabalho Prático

## Processamento de Linguagens

Luís Pereira 27953

Tiago Ferreira 27980

Rodrigo Cruz 27971

Engenharia De Sistemas Informáticos

Maio de 2025

## Conteúdo

Introdução.....	2
Visão Geral da Linguagem CQL .....	3
Definição da Gramática .....	3
Análise Sintática e Construção da Árvore de Comandos .....	3
Tratamento de Prioridades e Precedência .....	4
Gestão de Erros Sintáticos .....	4
Código do Reconhecedor Léxico .....	4
Exemplo de execução.....	5
Configuração de tabela de dados .....	5
Execução de queries sobre tabelas de dados .....	6
Criação de novas tabelas de dados .....	7
Dificuldades .....	8
Conclusão.....	8
Bibliografia .....	9

## Índice de imagens

Figura 1 Import Table .....	5
Figura 2 Export Table.....	5
Figura 3 Discard Table.....	6
Figura 4 Select FROM observacoes Temp >22 .....	6
Figura 5 Select by ID .....	6
Figura 6 Create Table with Condition.....	7
Figura 7 Create Table with Id.....	7

## Introdução

Neste projeto foi desenvolvido um **interpretador para a linguagem CQL (Comma Query Language)**, utilizando a linguagem de programação **Python** e a biblioteca **PLY (Python Lex-Yacc)**, que permite a construção de analisadores léxicos e sintáticos. A aplicação tem como objetivo permitir a manipulação de ficheiros **CSV** através de uma linguagem própria semelhante ao SQL, possibilitando operações como importação, exportação, filtragem, junções, criação de tabelas e procedimentos.

A estrutura modular do sistema divide as responsabilidades entre diferentes ficheiros:

- `cql_lexer.py` (analisador léxico),
- `cql_parser.py` (analisador sintático),
- `tables.py` (estrutura e operações sobre as tabelas),
- `utils.py` (funções auxiliares), entre outros.

Este interpretador oferece uma forma prática e legível de consultar e transformar dados em formato de tabela.

## Visão Geral da Linguagem CQL

A linguagem CQL foi implementada através de um parser desenvolvido com a biblioteca **PLY (Python Lex-Yacc)**, que permite definir a gramática e regras de análise sintática da linguagem. O parser converte comandos escritos em CQL em estruturas internas interpretáveis pela aplicação, possibilitando a manipulação de tabelas e dados.

### Definição da Gramática

A gramática da linguagem é especificada por regras formais que definem a sintaxe dos comandos suportados. Essas regras foram implementadas como funções `p_<nome>` que descrevem, usando notação BNF, as formas válidas das instruções da linguagem, incluindo:

- **Comandos principais:** Importação, exportação, remoção, renomeação, impressão de tabelas, seleção de dados, criação de tabelas e gestão de procedimentos.
- **Consultas (SELECT):** Permitem selecionar campos específicos, filtrar dados por condições complexas (usando operadores lógicos AND, OR) e aplicar limites ao número de resultados.
- **Criação de tabelas (CREATE):** Suporta criação direta a partir de consultas, bem como junções entre tabelas existentes, com cláusulas específicas para combinar dados.
- **Procedimentos:** Definição de blocos de comandos reutilizáveis que podem ser declarados e chamados, facilitando a automação de sequências de operações.

### Análise Sintática e Construção da Árvore de Comandos

Cada regra do parser constrói uma estrutura de dados que representa o comando e seus argumentos. Por exemplo, ao reconhecer o comando **IMPORT TABLE**, a estrutura guarda o nome da tabela e o caminho do ficheiro a importar. Estas estruturas permitem que o programa interprete e execute as ações correspondentes.

## Tratamento de Prioridades e Precedência

Para garantir que expressões lógicas e condições sejam avaliadas corretamente, a implementação define precedências para operadores como AND, OR e operadores relacionais (<, <=, >, >=, =, <>). Isso assegura que as condições sejam agrupadas e interpretadas conforme a lógica esperada pelo utilizador.

## Gestão de Erros Sintáticos

O parser inclui mecanismos para detetar erros de sintaxe, emitindo mensagens com a localização do erro, o que facilita a identificação de comandos inválidos.

## Código do Reconhecedor Léxico

O reconhecedor léxico foi implementado utilizando a biblioteca **PLY Lex** em Python para identificar os símbolos terminais da linguagem CQL.

Este componente reconhece as palavras reservadas (como import, select, table), operadores relacionais (=, <, >), literais de texto, números e identificadores (nomes de tabelas e colunas).

Comentários de linha e multi-linha são ignorados, bem como espaços em branco, para garantir que apenas os tokens relevantes são processados. O lexer também mantém o controlo das linhas para facilitar a deteção de erros.

Em caso de encontrar caracteres inválidos, o lexer reporta o erro e continua a análise, tornando o processo mais robusto.

# Exemplo de execução

## Configuração de tabela de dados

### IMPORT

Inserir numa estrutura de dados em memória o conteúdo do ficheiro .csv, ficando acessível através do identificador estacoes, correspondente ao nome do ficheiro.

```
CQL> IMPORT TABLE estacoes FROM "estacoes.csv";
[DEBUG] Colunas: ['Id', 'Local', 'Coordenadas']
[DEBUG] Dados: [['E1', 'Terras de Bouro/Barral (CIM)', '[-8.31808611,41.70225278]'], ['E2', 'Graciosa / Serra das Fontes (DROTRH)', '[-28.0038,39.0672]'], ['E3', 'Olhão, EPP0', '[-7.821,37.033]'], ['E4', 'Setúbal, Areias', '[-8.89066111,38.54846667]']]
CQL>
```

Figura 1 Import Table

### EXPORT

Guarda os dados da tabela, que estão atualmente em memória, no ficheiro est.csv.

```
CQL> EXPORT TABLE estacoes as "est1.csv";
CQL>
```




















	__pycache__	15/05/2025 15:23	Pasta de ficheiros	
	.gitattributes	12/05/2025 15:01	Documento de tex...	3 KB
	.gitignore	12/05/2025 15:01	Documento de tex...	7 KB
	cql_lexer.py	11/05/2025 23:10	Python File	2 KB
	cql_parser.py	12/05/2025 00:40	Python File	4 KB
	dados_finais.csv	16/05/2025 19:35	Ficheiro CSV	1 KB
	entrada.fca	15/05/2025 15:53	Ficheiro FCA	1 KB
	est.csv	15/05/2025 15:20	Ficheiro CSV	1 KB
	est1.csv	16/05/2025 21:35	Ficheiro CSV	1 KB
	estacoes.csv	12/05/2025 00:05	Ficheiro CSV	1 KB
	estacoes_quentes.csv	15/05/2025 15:50	Ficheiro CSV	1 KB
	executor.py	15/05/2025 15:23	Python File	3 KB
	main.py	15/05/2025 15:49	Python File	3 KB
	observacoes.csv	11/05/2025 22:05	Ficheiro CSV	1 KB
	parser.out	11/05/2025 23:10	Wireshark capture ...	29 KB
	parsetab.py	11/05/2025 23:10	Python File	8 KB
	Projeto_PL.pyproj	15/05/2025 15:48	Python Project	3 KB
	tables.py	15/05/2025 15:05	Python File	6 KB
	utils.py	11/05/2025 19:44	Python File	2 KB

Figura 2 Export Table

## DISCARD

Elimina os dados da tabela produtos que estão em memória.

```
CQL> DISCARD TABLE estacoes;  
CQL> SELECT * FROM estacoes;  
Erro: Tabela 'estacoes' não encontrada  
CQL>
```

Figura 3 Discard Table

## Execução de queries sobre tabelas de dados

### Exemplo de Select com condição

Devolve todas as linhas que respondam a determinada condição. Sugere-se a possibilidade de permitir mais do que uma condição, separadas por AND.

```
CQL> SELECT * FROM observacoes WHERE Temperatura > 22;  
Id | IntensidadeVentoKM | Temperatura | Radiação | DirecaoVento | IntensidadeVento | Humidade | DataHoraObservacao  
E1 | 2.5 | 23.2 | 133.2 | NE | 0.7 | 58.0 | 1010.5  
CQL>
```

Figura 4 Select FROM observacoes Temp >22

### Exemplo de Select por Id

Devolve todas as linhas, mas apenas as colunas indicadas.

```
CQL> SELECT * FROM observacoes WHERE Temperatura > 22;  
Id | IntensidadeVentoKM | Temperatura | Radiação | DirecaoVento | IntensidadeVento | Humidade | DataHoraObservacao  
E1 | 2.5 | 23.2 | 133.2 | NE | 0.7 | 58.0 | 1010.5  
CQL> SELECT DataHoraObservacao,Id FROM observacoes;  
DataHoraObservacao | Id  
1010.5 | E1  
99.0 | E2  
96.0 | E3  
88.0 | E4  
69.0 | E5
```

Figura 5 Select by ID

## Criação de novas tabelas de dados

### Exemplo de Criação de Tabelas

Armazena numa nova tabela o resultado da query, permitindo que nas instruções seguintes se possa guardar o resultado num ficheiro.

```
CQL> CREATE TABLE mais_quentes SELECT * FROM observacoes WHERE Temperatura > 22 ;
Criando tabela 'mais_quentes' com os dados: {'columns': ['Id', 'IntensidadeVentoKM', 'Temperatura', 'Radiação', 'DirecaoVento', 'IntensidadeVento', 'Humidade', 'DataHoraObservacao'], 'data': [['E1', '2.5', '23.2', '133.2', 'NE', '0.7', '58.0', '1010.5']]}
CQL> SELECT * FROM mais_quentes;
Id | IntensidadeVentoKM | Temperatura | Radiação | DirecaoVento | IntensidadeVento | Humidade | DataHoraObservacao
E1 | 2.5 | 23.2 | 133.2 | NE | 0.7 | 58.0 | 1010.5
```

Figura 6 Create Table with Condition

Para juntar tabelas pode-se criar uma tabela. A junção de tabelas não irá permitir a seleção de colunas ou linhas, obrigando sempre à união completa de duas tabelas.

```
CQL> CREATE TABLE completo FROM estacoes JOIN observacoes USING(Id);
Realizando junção entre 'estacoes' e 'observacoes' usando 'Id'
Junção resultante (4 linhas)
Criando tabela 'completo' com os dados: {'columns': ['Id', 'Local', 'Coordenadas', 'IntensidadeVentoKM', 'Temperatura', 'Radiação', 'DirecaoVento', 'IntensidadeVento', 'Humidade', 'DataHoraObservacao'], 'data': [['E1', 'Terras de Bouro/Barral (CIM)', '[-8.31808611,41.70225278]', '2.5', '23.2', '133.2', 'NE', '0.7', '58.0', '1010.5', '2025-04-10T19:00'], ['E2', 'Graciosa / Serra das Fontes (DROTRH)', '[-28.0038,39.0672]', '15.1', '12.5', '679.6', 'E', '0.0', '4.2', '99.0', '-99.0', '2025-04-10T19:00'], ['E3', 'Olhão, EPP0', '[-7.821,37.033]', '4.0', '16.4', '0.0', 'NE', '0.0', '1.1', '96.0', '1010.5', '2025-04-10T19:00'], ['E4', 'Setúbal, Areias', '[-8.89066111,38.54846667]', '3.6', '16.8', '1.6', 'SW', '0.0', '1.0', '88.0', '1012.2', '2025-04-10T19:00']]}
CQL> SELECT * FROM completo;
Id | Local | Coordenadas | IntensidadeVentoKM | Temperatura | Radiação | DirecaoVento | IntensidadeVento | Humidade | DataHoraObservacao
E1 | Terras de Bouro/Barral (CIM) | [-8.31808611,41.70225278] | 2.5 | 23.2 | 133.2 | NE | 0.7 | 58.0 | 1010.5
E2 | Graciosa / Serra das Fontes (DROTRH) | [-28.0038,39.0672] | 15.1 | 12.5 | 679.6 | E | 0.0 | 4.2 | 99.0
E3 | Olhão, EPP0 | [-7.821,37.033] | 4.0 | 16.4 | 0.0 | NE | 0.0 | 1.1 | 96.0
E4 | Setúbal, Areias | [-8.89066111,38.54846667] | 3.6 | 16.8 | 1.6 | SW | 0.0 | 1.0 | 88.0
```

Figura 7 Create Table with Id



## Dificuldades

Ao longo do desenvolvimento da linguagem CQL, enfrentámos alguns desafios, como definir uma gramática que fosse clara e fácil de expandir, resolver conflitos entre regras do parser e garantir que os comandos funcionassem corretamente com os dados dos ficheiros CSV.

Apesar de demorado conseguimos estruturar os ficheiros por pastas.

Tentámos ainda implementar uma árvore de sintaxe abstrata (AST) utilizando classes, com o objetivo de representar os comandos de forma mais estruturada e orientada a objetos. No entanto, essa abordagem acabou por não ser concluída com sucesso, devido à integração com o parser.

Também tivemos dificuldades em implementar a funcionalidade **Procedure**.

## Conclusão

O desenvolvimento da linguagem CQL permitiu explorar, na prática, os conceitos de análise léxica e sintática, além da construção de uma linguagem específica para manipulação de dados em ficheiros CSV. Ao longo do projeto, conseguimos implementar um reconhecedor léxico e um parser funcional, capaz de interpretar comandos úteis como importação, seleção, criação de tabelas e chamadas de procedimentos.

Apesar das dificuldades encontradas, como a tentativa não concluída de implementar uma árvore de sintaxe abstrata com classes, o trabalho proporcionou uma compreensão mais profunda sobre a estrutura e execução de linguagens de programação. Foi uma experiência enriquecedora que combinou teoria e prática, no que toca a design de linguagens.

## Bibliografia

<https://www.w3schools.com/python/>

[https://www.youtube.com/watch?v=eWRfhZUzrAc&ab\\_channel=freeCodeCamp.o](https://www.youtube.com/watch?v=eWRfhZUzrAc&ab_channel=freeCodeCamp.org)  
rg