

Trabalho Pratico

Processamento de Linguagens

Implementação de um Interpretador para a Linguagem CQL (Comma Query Language)"

Engenharia Sistemas Informáticos

Docente

Óscar Rafael da Silva Ferreira Ribeiro

Participantes:

Vasco Gomes nº 27955

Gabriel Fortes nº

Diogo Caldas nº

Ano Letivo: 2024/2025

**Resumo**

Este trabalho desenvolveu um interpretador em Python para a linguagem CQL, que permite consultar e manipular dados de ficheiros CSV de forma simples e intuitiva. Usámos a biblioteca PLY para processar os comandos e o módulo CSV do Python para lidar com os dados.

**Introdução**

No âmbito da disciplina de Processamento de Linguagens, este trabalho prático focou-se no desenvolvimento de um interpretador para a linguagem CQL (Comma Query Language), uma linguagem simplificada para consulta de dados em ficheiros CSV. O projeto envolveu a aplicação de conceitos fundamentais de processamento de linguagens, como análise léxica (reconhecimento de tokens) e análise sintática (validação da estrutura das consultas), utilizando a biblioteca PLY em Python.

**Objetivos Específicos**

* Implementar um lexer para identificar os elementos básicos da linguagem (palavras-chave, operadores, identificadores).
* Desenvolver um parser para validar a sintaxe dos comandos CQL
* (ex: SELECT, IMPORT TABLE).
* Criar um mecanismo de execução para processar consultas sobre dados CSV, com operações como filtragem (WHERE), projeção de colunas e junção de tabelas (JOIN).
* Garantir a modularidade do código, permitindo a extensão futura com novos comandos ou funcionalidades.

**Índice:**

**Especificação da linguagem**

A linguagem CQL (Comma Query Language) permite a manipulação e consulta de dados contidos em ficheiros CSV, por meio de uma sintaxe simples e orientada a comandos. Os comandos estão organizados por grupos funcionais:

**Descrição da linguagem: (comandos)**

1. **Configuração de Tabelas**

* IMPORT TABLE <nome> FROM "<ficheiro.csv>";
* EXPORT TABLE <nome> AS "<ficheiro.csv>";
* DISCARD TABLE <nome>;
* RENAME TABLE <nome\_antigo> <nome\_novo>;
* PRINT TABLE <nome>;

**B. Execução de Queries**

* SELECT \* FROM <tabela>;
* SELECT <coluna1>, <coluna2> FROM <tabela>;
* SELECT \* FROM <tabela> WHERE <condição>;
* ... LIMIT <n>

**C. Criação de Novas Tabelas**

* CREATE TABLE <nova\_tabela> SELECT ...;
* CREATE TABLE <nova\_tabela> FROM <tabela1> JOIN <tabela2> USING(<coluna>);

**D. Procedimentos**

* PROCEDURE <nome> DO ... END
* CALL <nome>;

**E. Comentários**

* -- comentário
* {- comentário -}

**Uma imagem com texto, captura de ecrã, Tipo de letra, software

Os conteúdos gerados por IA poderão estar incorretos.Gramática Formal: (em EBNF ou diagramas de sequencia)**

Figura 2-lista de tokens reservados e regras de expressão

Figura 1- Lista de tokens

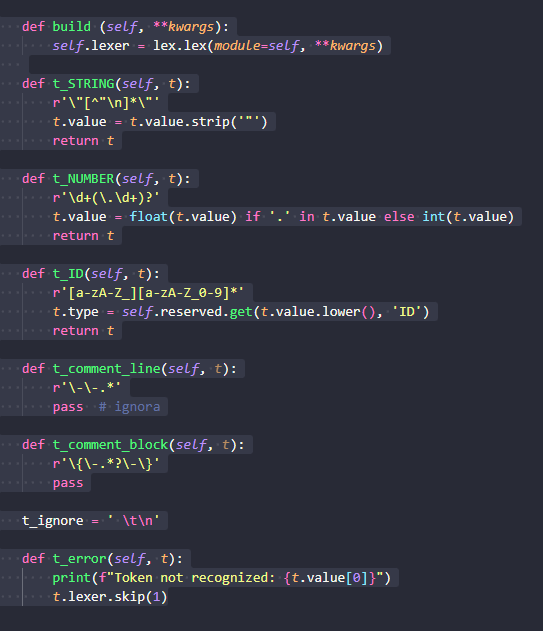
****

Figura 13-regras de expressão regulares

**Exemplos de Comandos Válidos:**

**-- Importar tabela**

IMPORT TABLE estacoes FROM "estacoes.csv";

**-- Exportar tabela**

EXPORT TABLE estacoes AS "saida.csv";

**-- Selecionar todas as colunas**

SELECT \* FROM observacoes;

**-- Selecionar colunas com condição**

SELECT Id, Temperatura FROM observacoes WHERE Temperatura > 20;

**-- Criar nova tabela com SELECT**

CREATE TABLE quentes SELECT \* FROM observacoes WHERE Temperatura > 25;

**-- Criar nova tabela com JOIN**

CREATE TABLE completo FROM estacoes JOIN observacoes USING Id;

**-- Definir procedimento**

PROCEDURE atualizar DO

CREATE TABLE resumo SELECT Id, Temperatura FROM observacoes;

END;

**-- Executar procedimento**

CALL atualizar;

**Implementação**

**Analise Léxica:**

A análise léxica da linguagem CQL foi implementada com recurso à biblioteca ply.lex, através da classe Lexer. Esta componente é responsável por identificar os **tokens léxicos** presentes nas instruções da linguagem, convertendo o texto de entrada numa sequência de símbolos que pode ser interpretada pelo analisador sintático.

**Tokens reconhecidos**

O analisador reconhece os seguintes tipos de tokens:

* **Identificadores** (ID): nomes de tabelas e colunas
* **Strings** (STRING): texto entre aspas, ex: "ficheiro.csv"
* **Números** (NUMBER): inteiros ou decimais
* **Operadores de comparação**: =, <>, <, >, <=, >=
* **Símbolos**: , (vírgula), ; (ponto e vírgula), \* (asterisco)
* **Palavras-chave**: IMPORT, TABLE, SELECT, FROM, WHERE, AND, CREATE, JOIN, USING, PROCEDURE, DO, END, CALL, EXPORT, AS, DISCARD, RENAME, PRINT, LIMIT

As palavras-chave são distinguidas dos identificadores através de uma tabela reserved.

**Regras léxicas**

As expressões regulares utilizadas permitem reconhecer cada token de forma eficiente. Por exemplo:

* **t\_STRING** reconhece strings entre aspas
* **t\_NUMBER** identifica inteiros ou decimais
* **t\_ID** identifica nomes genéricos e associa-os a palavras-chave, quando aplicável

**Comentários**

São suportados dois tipos de comentários:

* **Comentário de uma linha**: inicia com --
* **Comentário de múltiplas linhas**: delimitado por {– e –}

Estes tokens são ignorados durante a análise léxica.

**Tratamento de erros**

Tokens inválidos são ignorados e um aviso é apresentado ao utilizador, conforme definido na função **t\_error**.

**Análise Sintática:**

A análise sintática foi implementada com a biblioteca ply.yacc, através da classe Parser, que define as regras de produção da linguagem CQL. Esta etapa é responsável por verificar se a sequência de tokens fornecida pelo analisador léxico segue as regras gramaticais da linguagem.

**Organização**

O analisador sintático reconhece estruturas válidas da linguagem e constrói uma **árvore de sintaxe abstrata (AST)** que representa a instrução de forma estruturada. A gramática foi implementada com base na especificação formal definida na secção anterior (EBNF).

**Estrutura de Produção**

O ponto de entrada é a regra:

Uma imagem com Tipo de letra, texto, captura de ecrã, design

Os conteúdos gerados por IA poderão estar incorretos.

A partir daqui, as instruções são processadas por diferentes regras de produção para comandos como, **Importação/Exportação**, **Consultas**, **Criação de tabelas**, **Procedimentos**, **Chamada de procedimentos**, **Outros**:

As regras utilizam **tuplos** para construir a representação da AST, por exemplo:

Uma imagem com texto, Tipo de letra, captura de ecrã, file

Os conteúdos gerados por IA poderão estar incorretos.

**Operadores e Condições**

Foi implementada uma regra para operadores de comparação:

Uma imagem com texto, captura de ecrã, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

E outra para expressar condições compostas:

Uma imagem com texto, Tipo de letra, captura de ecrã

Os conteúdos gerados por IA poderão estar incorretos.

**Tratamento de Erros**

Erros de sintaxe são tratados pela função p\_error, que informa o utilizador sobre a linha e o conteúdo do erro:

Uma imagem com captura de ecrã, texto, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

**Arvore de Sintaxe Abstrata**:

**Semântica e Execução:**Ações Semânticas e Execução

Após a análise léxica e sintática, o programa interpreta os comandos através da associação de ações semânticas às regras de produção, construindo uma árvore de sintaxe abstrata (AST), que é posteriormente percorrida por um interpretador.

Árvore de Sintaxe Abstrata (AST)

A AST é representada por tuplos, gerados diretamente nas regras de parsing. Cada comando reconhecido pela gramática produz um tuplo com os dados relevantes.

Exemplos:

('import', 'estacoes', 'estacoes.csv')

('select\_where', 'observacoes', [('Temperatura', '>', 22)])

('create\_join', 'completo', 'estacoes', 'observacoes', 'Id')

Esta estrutura simplificada permite que os comandos sejam facilmente percorridos e executados.

**Execução dos Comandos**

A execução é feita por um **interpretador** que percorre a AST e aplica o comportamento correspondente a cada instrução. A lógica está definida em métodos específicos para cada tipo de comando.

**Procedimentos**

Os procedimentos são armazenados como listas de instruções. Quando o utilizador executa CALL <nome>, as ações dentro do bloco PROCEDURE ... DO ... END são interpretadas sequencialmente.

Extras

**Testes e validações**

Casos de teste para casa comando

Resultados obtidos/esperados

Erros identificados e corrigidos

**Testes e validações**

**Conclusão**

Dificuldades encontradas

Limitações da solução

Melhorias futuras

**Referencias**

Bibliografia

Ferramentas utilizadas

**Anexos**

Código fonte utilizado