Relatório do Interpretador CQL

Processamento de Linguagens

Engenharia Sistemas Informáticos - 2º ano

Docente

Óscar Rafael da Silva Ferreira Ribeiro

Grupo 24

Vasco Gomes nº 27955

Gabriel Fortes nº 27976

Diogo Caldas nº 27951

Ano Letivo: 2024/2025

Índice

Introdução

No âmbito da UC (Unidade Curricular) de Processamento de Linguagens, no 2º ano da Licenciatura em Engenharia de Sistemas Informáticos, este trabalho prático focou-se no desenvolvimento de um interpretador para a linguagem CQL (*Comma Query Language*), uma linguagem consulta de dados em ficheiros CSV (*Comma Separated Values*).

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), através da linguagem de programação Python, e da biblioteca PLY (*Lex-Yacc*).

Interpretador CQL

Gramática concreta e Reconhecedor Léxico

Como primeira fase da realização do projeto foi necessário especificar a gramática concreta da linguagem de entrada e, de forma a reconhecer e processar os símbolos terminais identificados na mesma, desenvolver um reconhecedor léxico.

Para o desenvolver foi utilizada a biblioteca *PLY*, e foi construído no ficheiro lexer.py.

Nas figuras abaixo estão demonstrados os tokens da linguagem, da qual fazem parte a sua lista (ID, STRING, etc.) e as palavras definidas com reservadas (SELECT, IMPORT, etc.), que também incluem operadores e pontuação, e as suas regras para expressões regulares.

Reconhecedor Sintático

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.

Para isso acontecer é necessário que, além da gramática, a construção dos comandos através de um reconhecedor sintático, que está no ficheiro parser.py. Os comandos estão organizados por grupos funcionais:

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>;

Execução de Queries

SELECT * FROM <tabela>;

SELECT <coluna1>, <coluna2> FROM <tabela>;

SELECT * FROM <tabela> WHERE <condição>;

... LIMIT <n>

Criação de Novas Tabelas

CREATE TABLE <nova_tabela> SELECT ...;

CREATE TABLE <nova_tabela> FROM <tabela1> JOIN <tabela2> USING(<coluna>);

Procedimentos

PROCEDURE <nome> DO ... END

CALL <nome>;

Comentários

-- comentário {- comentário -}

Tratamento de erros

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

A análise sintática foi implementada com a biblioteca ply.yacc.

Esta etapa é responsável por verificar se a sequência de tokens que é fornecida pelo reconhecedor léxico e segue as regras 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.

Estrutura de Produção

O ponto de entrada é a regra:

```
def p_program(self, p):
    'program : statement_list'
    p[0] = p[1]
```

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, entre outros.

```
def p_select_where(self, p):
    'select_where : SELECT STAR FROM ID WHERE condition SEMICOLON'
    p[0] = ('select_where', p[4], [p[6]])
```

Operadores e Condições

Foi implementada uma regra para operadores de comparação:

```
def p_operator(self, p):
    '''operator : EQUALS
                | NOT_EQUAL
                | LESS_THAN
                | GREATER_THAN
                | LESS_EQUAL
                | GREATER_EQUAL'''
    p[0] = p[1]
```

E outra para expressar condições compostas:

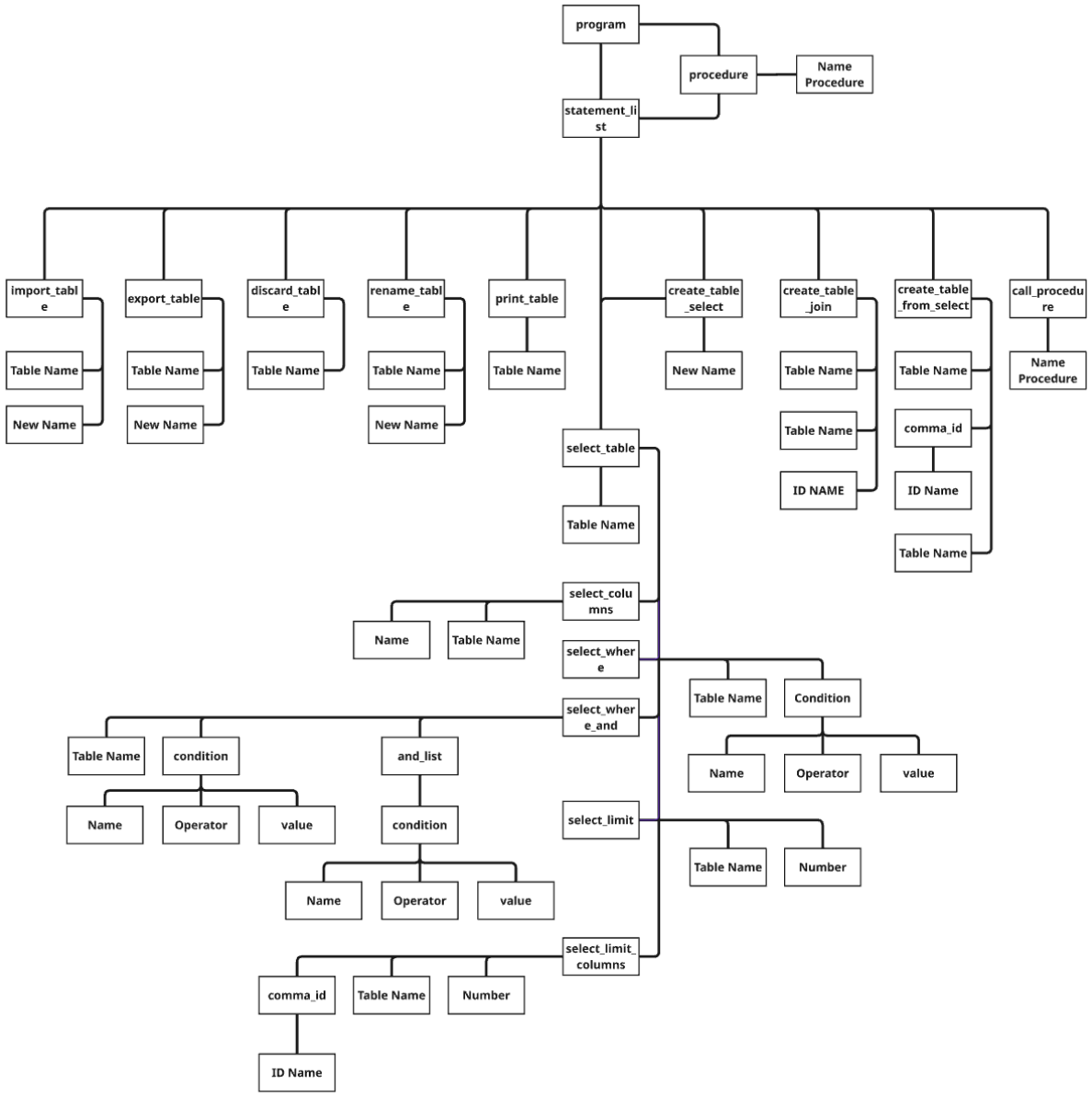
```
def p_and_list(self, p):
    '''and_list : AND condition
                | and_list AND condition'''
```

Tratamento de Erros

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

```
def p_error(self, p):
    if p:
        print(f"Erro de sintaxe na linha {p.lineno}, perto de '{p.value}'")
    else:
        print("Erro de sintaxe no fim do input")
```

Árvore de Sintaxe Abstrata



Interpretador

A execução é feita por um interpretador, no ficheiro interpreter.py , que percorre a Árvore Abstrata e aplica o comportamento correspondente a cada instrução. A lógica está definida em métodos específicos para cada tipo de comando.

Funcionamento

Por fim, no ficheiro main.py, podemos usar duas funcionalidades de entrada.

A primeira lê um ficheiro .fca (funcional do Cávado e do Ave) que contém uma sequência de comandos reconhecidos pelo programa.

Caso nenhum ficheiro seja introduzido, o programa funcionará no terminal onde os comandos devem ser lidos à medida que o utilizador os vai inserindo.

```
import sys
from interpreter import Interpreter

class main:

    interpreter = Interpreter()

    if len(sys.argv) == 2:
            try:
                with open(sys.argv[1], "r") as file:
                    contents = file.read()
                    resultado = interpreter.start(contents)
            except Exception as e:
                print(e)
    else:
        for expr in iter(lambda: input(">> "), ""):
            try:
                if (expr.strip() == "quit"):
                    break
                resultado = interpreter.start(expr)
            except Exception as e:
                print(e)
```

Testes e validações

Para testar o funcionamento correto do projeto desenvolvido, foram implementados três ficheiros de testes: test_lexer.py, test_parser.py e test_interpreter.py . Cada um desses ficheiros foca-se numa componente distinta do sistema: o analisador léxico, o analisador sintático e o interpretador da linguagem CQL (Comma Query Language), respetivamente.

Testes ao lexer

Objetivos:

Garantir que os tokens definidos são corretamente reconhecidos;
Validar o suporte a comentários com uma ou várias linhas

Exemplos testados:

```
IMPORT TABLE estacoes FROM "estacoes.csv";
-- Comentario teste\nSELECT Id FROM estacoes;
{- Comentario teste -}\nEXPORT TABLE estacoes AS "station.csv";
```

Resultados esperados:

Tokens corretamente identificados para cada comando;
Comentários ignorados pelo lexer corretamente.

Testes ao parser

Objetivos:

Garantir o reconhecimento correto da estrutura gramatical;
Confirmar que erros de sintaxe são corretamente identificados.

Exemplos testados:

```
IMPORT TABLE obs FROM "observacoes.csv";
SELECT IntensidadeVentoKM, Temperatura FROM obs;
CREATE TABLE Tempmaior SELECT * FROM obs WHERE Temperatura > 15;
PROCEDURE atualizar DO CREATE TABLE Tempmaior SELECT * FROM obs WHERE Temperatura > 20 ; END
```

Resultados esperados:

ASTs corretamente geradas, representando a estrutura lógica dos comandos;
Mensagens de erro claras para entradas inválidas.

Testes ao interpreter

Objetivos:

Garantir que os comandos executam corretamente sobre os dados CSV;
Validar a manipulação de tabelas em memória;
Confirmar que as operações de criação, seleção, exportação e remoção funcionam como esperado.

Exemplos testados:

```
IMPORT TABLE observacoes FROM "examples/observacoes.csv";
SELECT Temperatura FROM observacoes LIMIT 2;
CREATE TABLE Temp SELECT Temperatura FROM observacoes LIMIT 2;
PROCEDURE Teste DO CREATE TABLE Testetemp SELECT * FROM observacoes WHERE Temperatura > 22; END;
CALL Teste;
```

Resultados esperados:

Tabelas importadas e manipuladas corretamente;
Resultados impressos e exportados com base nas queries;
Procedimentos armazenados e invocados com sucesso.

Conclusão

Após a conclusão deste trabalho prático, acreditamos que atingimos os objetivos que pretendíamos, apesar da escassez de funcionalidades adicionais.

Porém acreditamos que foi um bom trabalho que envolveu os conceitos que adquirimos ao longo do semestre.

Para concluir, este projeto permitiu utilizar os conhecimentos abordados durante a Unidade Curricular, desde Analisadores Léxicos/Sintáticos, até ao código necessário para uma linguagem funcionar além de ter um maior cuidado com cada parte do código pois, num projeto como este, cada erro tem um impacto gigante no resto da linguagem.

Bibliografia

Código disponibilizado no moodle;

Python.org;

https://www.dabeaz.com/ply/ply.html.