Relatório - Trabalho Prático

Processamento de Linguagens

Alunos:

27962 – Gustavo Daniel Loureiro Marques

27977 – Igor Miguel Torres da Costa

29852 – Gustavo Da Costa Pereira

Professor: Óscar Rafael da Silva Ferreira Ribeiro

Licenciatura em Engenharia de Sistemas Informáticos

Barcelos | maio , 2025

Índice

Índice de Figuras

1. Introdução

No âmbito da unidade curricular de Processamento de Linguagens, este relatório tem como objetivo descrever o trabalho prático desenvolvido, que consiste na concessão e implementação de uma linguagem denominada CQL (*Comma Query Language*). O principal objetivo deste projeto consistiu no desenvolvimento de um sistema completo e capaz de interpretar e executar instruções escritas nesta linguagem, permitindo operações como importação, exportação, seleção e transformação de dados.

Para isso, foi necessário definir formalmente a gramática da linguagem, construir um analisador léxico e sintático com recurso à biblioteca PLY (*Python Lex-Yacc*) e desenvolver uma estrutura de execução baseada em árvores de sintaxe abstrata (AST). O sistema deveria ainda permitir a execução de comandos tanto a partir de ficheiros como em modo interativo, com suporte a procedimento reutilizáveis.

Este relatório documenta todas as fases do desenvolvimento, desde a especificação forma até à implementação e testes realizados.

2. Gramática Concreta da Linguagem de Entrada

$$\boldsymbol{G = < T, N, S, P >}$$

$\boldsymbol{literais} = [\text{'='},\text{'<'},\text{'>'},\text{';'},\text{','},\text{'*'},\text{'.'},\text{'('},\text{')'}$

$\boldsymbol{tokens} = [IMPORT, EXPORT, DISCARD, RENAME, PRINT, SELECT, FROM, WHERE$

$, LIMIT, CREATE, TABLE, JOIN, USING, PROCEDURE, CALL, DO, END, AND, AS, ID, NUMBER,$

$LE, GE, NE]$

$\boldsymbol{T = \{literais, tokens\}}$

$\boldsymbol{N} = \{programa, instrucoes, instrucao, inst\_import, inst\_export, inst\_discard,$

$inst\_rename, inst\_print, inst\_select, list\_select, list\_column, opt\_where, opt\_limit, condicao,$

$op, inst\_create, inst\_procedure, inst\_call\}$

$\boldsymbol{S = programa \rightarrow Axioma}$

$\boldsymbol{P} = \{(programa, \text{'}instrucoes\text{'}), (instrucoes, \text{'}instrucoes\ instrucao\text{'})$

$(instrucoes, \text{'}instrucao\text{'}), (instrucao, \text{'}inst\_import\ SEMICOLON\text{'}),$

$(instrucao, \text{'}inst\_export\ SEMICOLON\text{'}), (instrucao, \text{'}inst\_discard\ SEMICOLON\text{'})$,
$(instrucao, \text{'}inst\_rename\ SEMICOLON\text{'}), (instrucao, \text{'}inst\_print\ SEMICOLON\text{'}),$

$(instrucao, \text{'}inst\_select\ SEMICOLON\text{'}), (instrucao, \text{'}inst\_create\ SEMICOLON\text{'})$

$, (instrucao, \text{'}inst\_procedure\text{'}), (instrucao, \text{'}inst_{call} SEMICOLON\text{'}),$

$(inst\_import, \text{'}IMPORT\ TABLE\ ID\ FROM\ STRING\text{'}),$

$(inst\_export, \text{'}EXPORT\ TABLE\ ID\ AS\ STRING\text{'}), (inst\_discard, \text{'}DISCARD\ TABLE\ ID\text{'}),$

$(inst\_rename, \text{'}RENAME\ TABLE\ ID\ ID\text{'}), (inst\_print, \text{'}PRINT\ TABLE\ ID\text{'}),$

$(inst\_select, \text{'}SELECT\ list\_select\ FROM\ ID\ opt\_where\ opt\_limit\text{'}), (list\_select, \text{''}*\text{''}),$

$(list\_select. \text{'}list\_column\text{'}), (list\_column, \text{'}list\_column\ \text{','}\ ID\text{'}), (list\_column, \text{'}ID\text{'}),$

$(opt\_where, \text{'}WHERE\ condicao\text{'}), (opt\_where, \varepsilon), (opt\_limit, \text{'}LIMIT\ NUMBER\text{'}),$

$(opt\_limit, \text{'}\ \varepsilon\text{'}), (condicao, \text{'}ID\ op\ NUMBER\text{'}), (condicao, \text{'}condicao\ AND\ condicao\text{'}),$

$(op, \text{''} = \text{''}), (op, \text{''} <> \text{''}), (op, \text{''} < \text{''}), (op, \text{''} > \text{''}), (op, \text{''} <= \text{''}), (op, \text{''} >= \text{''}),$

$(inst\_create\ , \text{'}CREATE\ TABLE\ ID\ inst\_select\ \text{'}),$

$(inst\_create, \text{'}CREATE\ TABLE\ ID\ FROM\ ID\ JOIN\ ID\ USING\ \text{'('}\ ID\ \text{')''}),$

$(inst\_procedure, \text{'}PROCEDURE\ ID\ DO\ instrucoes\ END\text{'}), (inst\_call, \text{'}CALL\ ID\text{'})\}$

2.1. Notação BNF

$programa \rightarrow instrucoes$

$instrucoes \rightarrow instrucoes\ instrucao$
$\quad | instrucao$

$instrucao \rightarrow inst\_import\ ';'$
$\quad | inst\_export\ ';'$
$\quad | inst\_discard\ ';'$
$\quad | inst\_rename\ ';'$
$\quad | inst\_print\ ';'$
$\quad | inst\_select\ ';'$
$\quad | inst\_create\ ';'$
$\quad | inst\_procedure$
$\quad | inst\_call\ ';'$

$inst\_import \rightarrow IMPORT\ TABLE\ ID\ FROM\ STRING$

$inst\_export \rightarrow EXPORT\ TABLE\ ID\ AS\ STRING$

$inst\_discard \rightarrow DISCARD\ TABLE\ ID$

$inst\_rename \rightarrow RENAME\ TABLE\ ID\ ID$

$inst\_print \rightarrow PRINT\ TABLE\ ID$

$inst\_select \rightarrow SELECT\ list\_select\ FROM\ ID\ opt\_where\ opt\_limit$

$list\_select \rightarrow '*'$
$\quad | list\_column$

$list\_column \rightarrow list\_column\ ','\ ID$
$\quad | ID$

$opt\_where \rightarrow WHERE\ condicao$
$\quad | \varepsilon$

$opt\_limit \rightarrow LIMIT\ NUMBER$
$\quad | \varepsilon$

$condicao \rightarrow ID\ op\ NUMBER$
$\quad | condicao\ AND\ condicao$

$op \rightarrow$ " = "
| " <> "
| " < "
|" > "
| " ≤ "
| " ≥ "

$inst\_create \rightarrow CREATE\ TABLE\ ID\ inst\_select$
$\quad | CREATE\ TABLE\ ID\ FROM\ ID\ JOIN\ USING$ "(" ID ")"

$inst\_procedure \rightarrow PROCEDURE\ ID\ DO\ instrucoes\ END$

$inst\_call \rightarrow CALL\ ID$

3. Analisador Léxico

De forma a analisar o texto de entrada e convertê-lo numa sequência de *tokens* que representam unidades léxicas significativas (como palavras-chave, identificadores, operadores, etc.) é fundamental construir um reconhecedor léxico para que posteriormente a análise sintática seja funcional.

Deste modo, foi utilizada a biblioteca PLY (*Python Lex-Yacc*) para a implementação do analisador léxico, mais especificamente o módulo *ply.lex*.

3.1. Tokens e Palavras reservadas

O léxico em questão reconhece identificadores, números, strings, operadores relacionais, símbolos literais e um conjunto de palavras reservados, que correspondem a comandos específicos da linguagem.

```
# Lista de tokens
reserved = {'import': 'IMPORT','export': 'EXPORT','discard': 'DISCARD',
'rename': 'RENAME','print': 'PRINT','select': 'SELECT','from': 'FROM',
'where':   'WHERE','limit':   'LIMIT','create':   'CREATE','table':   'TABLE',
'join': 'JOIN','using': 'USING','procedure': 'PROCEDURE','call': 'CALL',
'do': 'DO','end': 'END','and': 'AND','as': 'AS'}

tokens = ['ID', 'NUMBER', 'STRING', 'LE', 'GE', 'NE'] +
list(reserved.values())

literals = ['(', ')', '=', '<', '>', ';', ',', '*', '.']
t_ignore = ' \t\r'
```

Os tokens compostos como <=, >= e <> são definidos com expressões regulares:

```
t_LE = r'<='
t_GE = r'>='
t_NE = r'<>'
```

3.2. Regras de Tokenização

Cada tipo de token é identificado por uma função iniciada por *t_*, contendo a expressão regular correspondente. A prioridade das regras segue a ordem de definição em questão.

3.2.1. Identificadores e Palavras Reservadas

A função *t_ID* reconhece nomes válidos de variáveis, e verifica se correspondem a palavras reservadas:

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value.lower(), 'ID')
    return t
```

As funções *t_STRING* e *T_NUMBER* reconhecem tipos específicos de literais usados na linguagem de entrada, no caso cadeias de caracteres e números inteiros e decimais:

```
def t_STRING(t):
    r'"([^\"]|\")*"'
    t.value = t.value[1:-1]
    return t
def t_NUMBER(t):
    r'-?\d+(\.\d+)?'
    t.value = float(t.value) if '.' in t.value else int(t.value)
    return t
```

3.2.2. Comentários e Ignorar Espaços

O analisador ignora espaços, tabulações e comentários de linha ou bloco:

```
def t_COMMENT_LINE(t):
    r'\-\-.*'
    pass
def t_COMMENT_BLOCK(t):
    r'\{\-([\s\S]*?)\-\}'
    pass
```

3.2.3. Controlo de Linhas e Erros

Já a função *t_newline* atualiza o número da linha em questão e a função *t_error* lida com caracteres inválidos:

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
def t_error(t):
    print(f"Caracter inválido: {t.value[0]}")
    t.lexer.skip(1)
```

4. Reconhecedor Sintático

De seguida, a implementação do reconhecedor sintático foi construída no ficheiro *parser.py*, utilizando a biblioteca PLY (*Python Lex-Yacc*). Este reconhecedor é responsável por verificar a estrutura gramatical das instruções, transformando-as em representações estruturadas para que possam ser posteriormente processadas.

Deste modo, baseado na gramática acima definida, funções *Pyhton* definem as regras de produção, começando cada função por *p_<nome>*, correspondendo assim a uma regra gramatical. O resultado da análise é representado sob a forma de um dicionário *Python* que descreve a operação e os seus argumentos.

4.1. Início da Gramática

O ponto de entrada da gramática é a regra definida na função abaixo descrita, indicando assim que um programa é composto por uma ou mais instruções, que podem ser executadas sequencialmente:

```
def p_programa(p):
    '''programa : instrucoes'''
    p[0] = p[1]
def p_instrucoes(p):
    '''instrucoes : instrucoes instrucao
                  | instrucao'''
    if len(p) == 3:
        p[0] = p[1] + [p[2]]
    else:
        p[0] = [p[1]]
def p_instrucao(p):
    '''instrucao : inst_import ';'
                 | inst_export ';'
                 | inst_discard ';'
                 | inst_rename ';'
                 | inst_print ';'
                 | inst_select ';'
                 | inst_create ';'
                 | inst_procedure
                 | inst_call ';' '''
    p[0] = p[1]
```

4.2. Instruções suportadas

E de acordo com as instruções suportadas no programa, cada uma se traduz numa função que converte para um dicionário *Python* a instrução em causa, com a chave *'op'* indicando a operação e '*args'* com os argumentos relevantes, facilitando a execução posterior por um interpretador.

Deste modo ficam alguns exemplos de funções definidas para os diferentes tipos de instruções suportadas:

- **Importar e Exportar Tabelas**

```
def p_inst_import(p):
    '''inst_import : IMPORT TABLE ID FROM STRING'''
    p[0] = {'op': p[1], 'args': [p[3], p[5]]}
def p_inst_export(p):
    '''inst_export : EXPORT TABLE ID AS STRING'''
    p[0] = {'op': p[1], 'args': [p[3], p[5]]}
```

- **Descartar, Renomear e Imprimir Tabelas**

```
def p_inst_discard(p):
    '''inst_discard : DISCARD TABLE ID'''
    p[0] = {'op': p[1], 'args': [p[3]]}
def p_inst_rename(p):
    '''inst_rename : RENAME TABLE ID ID'''
    p[0] = {'op': p[1], 'args': [p[3], p[4]]}
def p_inst_print(p):
    '''inst_print : PRINT TABLE ID'''
    p[0] = {'op': p[1], 'args': [p[3]]}
```

- **Procedimentos e chamamentos dos mesmos**

```
def p_inst_procedure(p):
    '''inst_procedure : PROCEDURE ID DO instrucoes END'''
    p[0] = {'op': p[1], 'args': [p[2], p[4]]}

def p_inst_call(p):
    '''inst_call : CALL ID'''
    p[0] = {'op': p[1], 'args': [p[2]]}
```

5. Árvore de sintaxe abstrata

A árvore sintaxe abstrata (AST) representa a estrutura semântica das instruções da linguagem CQL, isolando os elementos essenciais à execução dos comandos e eliminado destalhes sintáticos irrelevantes, como pontuação ou palavras-chave redundantes. A AST foi desenhada com o objetivo de ser simples de percorrer e interpretar, permitindo a avaliação dos comandos de forma sistemática.

5.1. Estrutura da AST

Cada nó da árvore é representado por um dicionário Python, com duas chaves principais:

- '*op*': identifica a operação (ex: 'IMPORT', 'CREATE', 'SELECT', 'JOIN');
- '*args*':lista dos argumentos associados à operação (ex: nomes de tabelas, colunas, condições, etc.)

5.2. Exemplo de AST

Abaixo apresentam-se um exemplo da árvore gerada para a instrução em questão e respetivo diagrama (Fig. 1 - Diagrama da árvore de sintaxe abstrata):

5.2.1. Instrução : *IMPORT TABLE:*

```
{'op': 'IMPORT', 'args': ['estacoes', 'estacoes.csv']}
```

Representa a importação da tabela estacoes a partir do ficheiro estacoes.csv

Fig. 1 - Diagrama da árvore de sintaxe abstrata

6. Gerador de Código

Deste modo procedeu-se o desenvolvimento do executor da AST, responsável por interpretar os comandos da linguagem CQL e executá-los. Esta tarefa é feita pela classe *InterpreterEval*, que analisa a árvore gerada pelo *parser* e aplica as operações correspondentes nos dados carregados.

A classe usa um dicionário de operadores para associar cada um a uma função que realiza a operação:

```
operators = {
        'PRINT': lambda args: InterpreterEval._print_table(args[0]),
        'IMPORT': lambda args: InterpreterEval._import_table(args[0],
args[1]),
        /*…*/
        'PROCEDURE': lambda args: InterpreterEval._define_procedure(args[0],
args[1]),
        'CALL': lambda args: InterpreterEval._call_procedure(args[0]),
    }
```

Sendo as funções definidas posteriormente, coincidindo com o objetivo do operador em questão, como é possível verificar nos exemplos abaixo relativos às operações de *PRINT* e *IMPORT*:

```
    @staticmethod
    def _print_table(tablename):
        if tablename in tables:
            data = tables[tablename]
            print(','.join(data['columns']))
            for row in data['rows']:
                print(','.join(map(str, row)))
        else:
            print(f"Tabela {tablename} não encontrada.")
    @staticmethod
    def _import_table(name, filename):
        tables[name] = csvm.load_csv(filename)
```

Este executar permite que a linguagem CQL tenha um comportamento funcional e útil, aplicando as instruções de forma dinâmica sobre os dados em questão.

7. Testes realizados

Com todos os passos de desenvolvimento, prosseguiu-se a fase de testes, reforçando a possibilidade de o programa ler um ficheiro passado como argumento e executar os comandos nele presente sequencialmente, ou então ser inicializado em modo interativo onde o utilizador vai colocando os comandos e executando um de cada vez.

Visto isso, realizamos um primeiro teste (Fig. 2 - Resultado Teste – Ficheiro com código) com o ficheiro “entrada.fca”, que contém os seguintes comandos:

-- Este é um comentário de uma linha

IMPORT TABLE estacoes FROM "estacoes.csv";

PRINT TABLE estacoes;

{- Comentário

Multilinha -}

SELECT Coordenadas FROM estacoes;

Obtendo o resultado abaixo colocado:

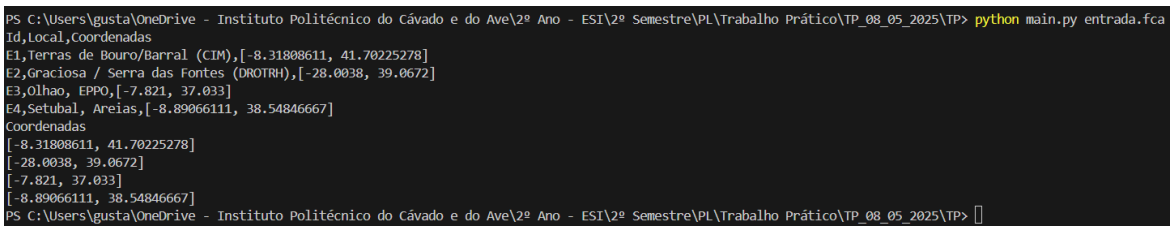

Fig. 2 - Resultado Teste – Ficheiro com código

Já no modo interativo podemos verificar a sucessão de comando executados e respetivos resultados (Fig. 3 - Resultado Teste – Configuração de Tabela de Dados , Fig. 4 - Resultado Teste - Execução de Queries, Fig. 5 - Resultados Teste - Criação de Novas Tabelas e Fig. 6 - Resultados Teste - Procedimentos):

7.1. Configuração de Tabelas de Dados

```
PS C:\Users\gusta\OneDrive - Instituto Politécnico do Cávado e do Ave\2º Ano - ESI\2º Semestre\PL\Trabalho Prático\TP_08_05_2025\TP> python main.py
Modo interativo (escreve comandos e termina com CTRL+D ou CTRL+Z):
>>> IMPORT TABLE estacoes FROM "estacoes.csv";
>>> IMPORT TABLE observacoes FROM "observacoes.csv";
>>> RENAME TABLE estacoes est;
>>> PRINT TABLE est;
Id,Local,Coordenadas
E1,Terras de Bouro/Barral (CIM),[-8.31808611, 41.70225278]
E2,Graciosa / Serra das Fontes (DROTRH),[-28.0038, 39.0672]
E3,Olhao, EPPO,[-7.821, 37.033]
E4,Setubal, Areias,[-8.89066111, 38.54846667]
>>> DISCARD TABLE est;
>>> SELECT * FROM TABLE est;
Erro de sintaxe perto de TABLE
>>> SELECT * FROM est;
Tabela est não encontrada.
```

Fig. 3 - Resultado Teste – Configuração de Tabela de Dados

7.2. Execução de *Queries*

```
PS C:\Users\gusta\OneDrive - Instituto Politécnico do Cávado e do Ave\2º Ano - ESI\2º Semestre\PL\Trabalho Prático\TP_08_05_2025\TP> python main.py
Modo interativo (escreve comandos e termina com CTRL+D ou CTRL+Z):
>>> IMPORT TABLE observacoes FROM "observacoes.csv";
>>> SELECT * FROM observacoes;
Id,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
E2,15.1,12.5,679.6,E,0.0,4.2,2025-04-10T19:00
E3,4.0,16.4,0.0,NE,0.0,1.1,2025-04-10T19:00
E4,3.6,16.8,1.6,SW,0.0,1.0,2025-04-10T19:00
>>> SELECT DataHoraObservacao,Id FROM observacoes;
DataHoraObservacao,Id
2025-04-10T19:00,E1
2025-04-10T19:00,E2
2025-04-10T19:00,E3
2025-04-10T19:00,E4
>>> SELECT * FROM observacoes WHERE Temperatura > 22;
Id,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
>>> SELECT * FROM observacoes WHERE Temperatura > 22 AND Humidade < 50;
Id,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
>>> SELECT * FROM observacoes WHERE Temperatura >= 20 AND Temperatura <= 30 LIMIT 5;
Id,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
```

Fig. 4 - Resultado Teste - Execução de Queries

7.3. Criação de Novas Tabelas

```
PS C:\Users\gusta\OneDrive - Instituto Politécnico do Cávado e do Ave\2º Ano - ESI\2º Semestre\PL\Trabalho Prático\TP_08_05_2025\TP> python main.py
Modo interativo (escreve comandos e termina com CTRL+D ou CTRL+Z):
>>> IMPORT TABLE observacoes FROM "observacoes.csv";
>>> IMPORT TABLE estacoes FROM "estacoes.csv";
>>> CREATE TABLE mais_quentes SELECT * FROM observacoes WHERE Temperatura > 22;
>>> CREATE TABLE completo FROM estacoes JOIN observacoes USING(Id);
>>> SELECT * FROM mais_quentes;
Id,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
>>> SELECT * FROM completo;
Id,Local,Coordenadas,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,Terras de Bouro/Barral (CIM),[-8.31808611, 41.70225278],2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
E2,Graciosa / Serra das Fontes (DROTRH),[-28.0038, 39.0672],15.1,12.5,679.6,E,0.0,4.2,2025-04-10T19:00
E3,Olhao, EPPO,[-7.821, 37.033],4.0,16.4,0.0,NE,0.0,1.1,2025-04-10T19:00
E4,Setubal, Areias,[-8.89066111, 38.54846667],3.6,16.8,1.6,SW,0.0,1.0,2025-04-10T19:00
```

Fig. 5 - Resultados Teste - Criação de Novas Tabelas

7.4. Procedimentos

```
PS C:\Users\gusta\OneDrive - Instituto Politécnico do Cávado e do Ave\2º Ano - ESI\2º Semestre\PL\Trabalho Prático\TP_08_05_2025\TP> python main.py
Modo interativo (escreve comandos e termina com CTRL+D ou CTRL+Z):
>>> IMPORT TABLE observacoes FROM "observacoes.csv";
>>> IMPORT TABLE estacoes FROM "estacoes.csv";
>>> PROCEDURE atualizar_observacoes DO CREATE TABLE mais_quentes SELECT * FROM observacoes WHERE Temperatura > 22; CREATE TABLE completo FROM estacoes JOIN observacoes USING (Id); END
>>> CALL atualizar_observacoes;
>>> SELECT * FROM mais_quentes;
Id,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
>>> SELECT * FROM completo;
Id,Local,Coordenadas,IntensidadeVentoKM,Temperatura,Radiacao,DirecaoVento,IntensidadeVento,Humidade,DataHoraObservacao
E1,Terras de Bouro/Barral (CIM),[-8.31808611, 41.70225278],2.5,23.2,133.2,NE,0.7,58.0,2025-04-10T19:00
E2,Graciosa / Serra das Fontes (DROTRH),[-28.0038, 39.0672],15.1,12.5,679.6,E,0.0,4.2,2025-04-10T19:00
E3,Olhao, EPPO,[-7.821, 37.033],4.0,16.4,0.0,NE,0.0,1.1,2025-04-10T19:00
E4,Setubal, Areias,[-8.89066111, 38.54846667],3.6,16.8,1.6,SW,0.0,1.0,2025-04-10T19:00
>>>
```

Fig. 6 - Resultados Teste - Procedimentos

8. Conclusão

Com o desenvolvimento da linguagem CQL foi possível atingir todos os objetivos propostos: contruir uma ferramenta funcional e capaz de interpretar instruções específicas para a manipulação de dados tabulares. Através da definição da gramática concreta e da notação BNF, bem como da implementação do analisador léxico e sintático em PLY, foi possível transformar instruções textuais em estruturas semânticas organizadas (AST), prontas para serem executadas por um interpretador.

Além disso, a criação de um executor modular permitiu garantir a funcionalidade completa da linguagem, incluindo operações básicas e avançadas como *JOIN*, *PROCEDURE*, *CALL*, e filtros condicionais.

Os testes realizados, tanto em modo ficheiro como interativo, confirmaram a robustez do sistema e a fidelidade da sua execução em relação à gramática definida. Assim, este projeto, culmina na entrega uma linguagem funcional, extensível e com potencial para aplicação em diversos contextos de análise de dados.