Instituto Politécnico do Cávado e do Ave
Escola Superior de Tecnologia

Trabalho Prático de Processamento de Linguagens

Autores:
Diogo Machado nº26042
Enrique Rodrigues nº28602
José Alves nº279967

Docente: Óscar Ribeiro
Data: Maio de 2025

1 Introdução

Com este trabalho prático pretendemos adquirir experiência na conceção e implementação de analisadores léxicos e sintáticos bem como a definição de ações semânticas que traduzem a linguagem de entrada. Entregamos uma solução que serve de alternativa à linguagem de interrogação de base de dados relacionais, que seja exxecutada sobre ficheiros de texto organizados num formato separado por vírgulas CSV (*Comma Separated Value*).

2 Metodologia

O interpretador foi desenvolvido em Python, utilizando as seguintes bibliotecas principais:

- **PLY (Python Lex-Yacc):** para análise léxica e sintática
- **Graphviz:** para visualização gráfica da árvore de sintaxe abstrata
- **PrettyPrinter:** para formatação de resultados em formato tabular

A estrutura do projeto foi organizada da seguinte forma:

3 Implementação

3.1 Análise Léxica

A análise léxica é realizada através de expressões regulares que identificam os diferentes tokens da nossa linguagem. O analisador léxico é implementado utilizando a biblioteca PLY, que permite definir tokens através de expressões regulares. Segue um exemplo da definição de alguns tokens:

```
#Lista de tokens
tokens = [
"ID",
"STRING",
"NUMBER",
"ASTERISK",
"COMMA",
"EQUALS",
"NOT_EQUALS",
"LESS_THAN",
"GREATER_THAN",
"LESS_EQUALS",
"GREATER_EQUALS",
"SINGLE_COMMENT",
"MULTI_COMMENT",
"SEMICOLON",
]

#Palavras reservadas
reserved = {
```

```
"import": "IMPORT",
"table": "TABLE",
"from": "FROM",
"export": "EXPORT",
"as": "AS",
"discard": "DISCARD",
"rename": "RENAME",
"print": "PRINT",
"select": "SELECT",
"where": "WHERE",
"limit": "LIMIT",
"create": "CREATE",
"join": "JOIN",
"using": "USING",
"procedure": "PROCEDURE",
"do": "DO",
"end": "END",
"call": "CALL",
"and": "AND",
}
# Regras para tokens
t_ASTERISK = r"\*"
t_COMMA = r","
t_EQUALS = r"="
t_NOT_EQUALS = r"<>"
t_LESS_THAN = r"<"
t_GREATER_THAN = r">"
t_LESS_EQUALS = r"<="
t_GREATER_EQUALS = r">="
t_SEMICOLON = r";"
```

3.2 Análise Sintática

A análise sintática é implementada usando uma gramática que define a estrutura válida dos comandos CQL. A gramática suporta os seguintes tipos de comandos:

- CREATE TABLE
- INSERT INTO
- SELECT
- DELETE
- DROP TABLE

Exemplo da definição de algumas regras gramaticais:

```
def p_create_table(p):
    '''create_table : CREATE TABLE IDENTIFIER LPAREN column_defs RPAREN'''
    p[0] = ('CREATE_TABLE', p[3], p[5])
```

```
def p_column_defs(p):
    '''column_defs : column_def
                   | column_defs COMMA column_def'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]
```

3.3 Árvore de Sintaxe Abstrata (AST)

A AST é gerada durante a análise sintática e representa a estrutura hierárquica do comando. Cada nó da árvore representa uma operação ou elemento do comando, facilitando sua interpretação e execução. A visualização da AST é realizada utilizando a biblioteca Graphviz.

Exemplo de um comando CQL e sua representação em AST:

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    name TEXT,
    email TEXT
);
```

A AST gerada para este comando é visualizada como uma árvore onde:

- O nó raiz representa o comando CREATE TABLE
- Os nós filhos representam o nome da tabela e as definições das colunas
- Cada definição de coluna contém o nome, tipo e restrições

4 Resultados

O interpretador foi testado com diversos comandos CQL, demonstrando sua capacidade de:

- Processar comandos sintaticamente corretos
- Gerar visualizações da AST
- Executar operações básicas de manipulação de dados
- Apresentar resultados em formato tabular

Exemplo de execução de um comando SELECT:

```
SELECT * FROM users WHERE id = 1;
```

Resultado apresentado em formato tabular:

```
+----+--------+------------------+
| id |  name  |      email       |
+----+--------+------------------+
| 1  | João   | joao@email.com   |
+----+--------+------------------+
```

5 Conclusão

O desenvolvimento deste interpretador CQL demonstrou a aplicação prática de conceitos fundamentais de compiladores e processamento de linguagens. O projeto pode ser expandido para suportar mais funcionalidades da linguagem CQL, tais como:

- Suporte a índices secundários
- Implementação de transações
- Otimização de consultas
- Suporte a tipos de dados mais complexos

A implementação atual serve como uma base sólida para futuras expansões e melhorias, demonstrando a viabilidade de desenvolver um interpretador completo para a linguagem CQL utilizando Python e suas bibliotecas de processamento de linguagens.