Universidade Federal do Piauí
Disciplina: Compiladores
Professor: Raimundo Santos Moura

Grupo:

- Fernando Augusto Vieira Rosa Martins
- Carlos Eduardo Mendes de Oliveira
- José Pires Gayoso Almendra Freitas Neto

Atividade de Participação 4
(Reconhecedor de comandos SQL)

Objetivo : Implementar um parser descendente recursivo para reconhecer os 10 principais comandos de SQL.
Recursos Utilizados : Python e biblioteca re.

1) Código e explicação

```
1   import re
```

A biblioteca re é uma biblioteca padrão do Python que fornece suporte para operações com expressões regulares. O termo "re" é uma abreviação de "regular expression" (expressão regular).A biblioteca re permite trabalhar com padrões de texto complexos e realizar operações de busca, correspondência e manipulação de strings com base nesses padrões.

Ao importar a biblioteca podemos utilizar suas funcionalidades para manipular strings de forma mais avançada e flexível.Ela é utilizada para fazer o reconhecimento dos padrões de comandos SQL por meio das expressões regulares permitindo verificar se uma determinada string corresponde a um padrão específico e, se corresponder, extrair as informações relevantes usando os grupos de captura.

```
# Função para obter o próximo token da entrada
def get_next_token(comando):
    comando = comando.strip()

    # Expressões regulares para identificar os tokens
    padrao_create_database = r"^CREATE\s+DATABASE\s+(\w+);$"
    padrao_use = r"^USE\s+(\w+);$"
    padrao_create_table = r"^CREATE\s+TABLE\s+(\w+)\s+\(((?:\w+\s+\w+\s*(?:,\s*)?)*)\);$"
    padrao_insert_into = r"^INSERT\s+INTO\s+(\w+)\s+\(((?:\w+\s*(?:,\s*)?)*)\)\s+VALUES\s+\(((?:[\w\s\"]+\s*(?:,\s*)?
    padrao_select_from = r"^SELECT\s+(\*|(?:\w+\s*(?:,\s*)?)*)\s+FROM\s+(\w+)(?:\s+ORDER\s+BY\s+(\w+))?(?:\s+WHERE\s+
    padrao_update_set_where = r"^UPDATE\s+(\w+)\s+SET\s+(\w+)\s+=\s+([\w\s\"]+)\s+WHERE\s+(\w+)\s+=\s+([\w\s\"]+);$"
    padrao_delete_from_where = r"^DELETE\s+FROM\s+(\w+)\s+WHERE\s+(\w+)\s+=\s+([\w\s\"]+);$"
    padrao_truncate_table = r"^TRUNCATE\s+TABLE\s+(\w+);$"
```

```
# Verificar padrões para cada comando
if re.match(padrao_create_database, comando):
    return "CREATE DATABASE"
elif re.match(padrao_use, comando):
    return "USE"
elif re.match(padrao_create_table, comando):
    return "CREATE TABLE"
elif re.match(padrao_insert_into, comando):
    return "INSERT INTO"
elif re.match(padrao_select_from, comando):
    return "SELECT"
elif re.match(padrao_update_set_where, comando):
    return "UPDATE"
elif re.match(padrao_delete_from_where, comando):
    return "DELETE"
elif re.match(padrao_truncate_table, comando):
    return "TRUNCATE TABLE"
else:
    return "INVALID"
```

Após a importação do módulo re, é definido a função “get_next_token()”. Ela é responsável por receber um comando SQL como entrada e determinar o próximo token do comando.

Ela utiliza expressões regulares para definir padrões que correspondem aos diferentes tipos de principais comandos SQL, que são : “CREATE DATABASE”, “USE”, “CREATE TABLE”, “INSERT INTO”, “SELECT”, “UPDATE”, “DELETE”, “TRUNCATE TABLE”.

A função verifica se o comando corresponde a algum dos padrões usando a função “re.match()”. Se houver correspondência entre o padrão e o comando inserido pelo usuário, a função retorna o token correspondente ao tipo de comando SQL identificado.

Note que, no início da definição da função, o comando inserido pelo usuário recebe o método “.strip()”. Esta função é utilizada apenas para remover espaços em branco desnecessários no início e no fim do comando, antes de se iniciar o processo de correspondência.

```
# Função para analisar o comando SQL
def parse_command(comando):
    token = get_next_token(comando)

    if token == "CREATE DATABASE":
        match = re.match(r"^CREATE\s+DATABASE\s+(\w+);$", comando)
        if match:
            database_name = match.group(1)
            print(f"Criando o banco de dados '{database_name}'")
        else:
            print("Comando inválido")

    elif token == "USE":
        match = re.match(r"^USE\s+(\w+);$", comando)
        if match:
            database_name = match.group(1)
            print(f"Usando o banco de dados '{database_name}'")
        else:
            print("Comando inválido")

    elif token == "CREATE TABLE":
        match = re.match(r"^CREATE\s+TABLE\s+(\w+)\s+\(((?:\w+\s+\w+\s*(?:,\s*)?)*)\);$", comando)
        if match:
            table_name = match.group(1)
            column_definitions = match.group(2).split(", ")
            print(f"Criando a tabela '{table_name}' com as colunas:")
            for column_definition in column_definitions:
                print(f"- {column_definition}")
        else:
            print("Comando inválido")
```

```
elif token == "INSERT INTO":
    match = re.match(r"^INSERT\s+INTO\s+(\w+)\s+\(((?:\w+\s*(?:,\s*)?)*)\)\s+VALUES\s+\(((?:[\w\s\"]+\s*(?:,\s*)
    if match:
        table_name = match.group(1)
        columns = match.group(2).split(", ")
        values = match.group(3).split(", ")
        print(f"Inserindo dados na tabela '{table_name}':")
        for column, value in zip(columns, values):
            print(f"- {column}: {value}")
    else:
        print("Comando inválido")

elif token == "SELECT":
    match = re.match(r"^SELECT\s+(\*|(?:\w+\s*(?:,\s*)?)*)\s+FROM\s+(\w+)(?:\s+ORDER\s+BY\s+(\w+))?(?:\s+WHERE\s
    if match:
        select_columns = match.group(1)
        table_name = match.group(2)
        order_by_column = match.group(3)
        where_column = match.group(4)
        where_value = match.group(5)
        print("Executando comando SELECT:")
        if select_columns == "*":
            print(f"Selecionar todas as colunas da tabela '{table_name}'")
        else:
            selected_columns = select_columns.split(", ")
            print(f"Selecionar as colunas:")
            for column in selected_columns:
                print(f"- {column}")
        print(f"Da tabela '{table_name}'")
        if order_by_column:
            print(f"Ordenar por coluna: {order_by_column}")
```

```
        if where_column and where_value:
            print(f"Condição WHERE: {where_column} = {where_value}")
    else:
        print("Comando inválido")

elif token == "UPDATE":
    match = re.match(r"^UPDATE\s+(\w+)\s+SET\s+(\w+)\s+=\s+([\w\s\"]+)\s+WHERE\s+(\w+)\s+=\s+([\w\s\"]+);$", coma
    if match:
        table_name = match.group(1)
        set_column = match.group(2)
        set_value = match.group(3)
        where_column = match.group(4)
        where_value = match.group(5)
        print(f"Executando comando UPDATE na tabela '{table_name}':")
        print(f"Definir coluna '{set_column}' como '{set_value}'")
        print(f"Condição WHERE: {where_column} = {where_value}")
    else:
        print("Comando inválido")

elif token == "DELETE":
    match = re.match(r"^DELETE\s+FROM\s+(\w+)\s+WHERE\s+(\w+)\s+=\s+([\w\s\"]+);$", comando)
    if match:
        table_name = match.group(1)
        where_column = match.group(2)
        where_value = match.group(3)
        print(f"Executando comando DELETE na tabela '{table_name}':")
        print(f"Condição WHERE: {where_column} = {where_value}")
    else:
        print("Comando inválido")
```

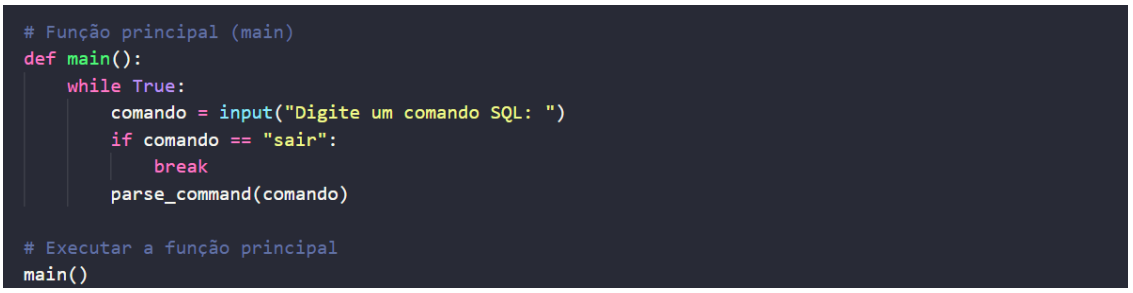
```
elif token == "TRUNCATE TABLE":
    match = re.match(r"^TRUNCATE\s+TABLE\s+(\w+);$", comando)
    if match:
        table_name = match.group(1)
        print(f"Executando comando TRUNCATE TABLE na tabela '{table_name}'")
    else:
        print("Comando inválido")

else:
    print("Comando inválido")
```

Em seguida, é definido a função” parse_command()”. Esta função analisa o comando SQL fornecido e executa a ação apropriada com base no tipo de comando.

A função inicia recebendo o comando SQL como entrada. Ela então chama a função “get_next_command()” para obter o tipo de comando correspondente. Em seguida, ela usa uma estrutura condicional (if-elif-else) para determinar qual ação deve ser executada baseado no token retornado.

Para cada tipo de comando, a função utiliza expressões regulares para extrair informações relevantes do comando, como o nome do banco de dados, nome da tabela, colunas, valores, condições WHERE, etc.

Por fim, usando estas informações como base, a função “parse_command()” exibe mensagens informativas sobre o comando executado, fornecendo detalhes como o nome do banco de dados criado, o nome da tabela, as colunas envolvidas, os valores inseridos, etc.



```
# Função principal (main)
def main():
    while True:
        comando = input("Digite um comando SQL: ")
        if comando == "sair":
            break
        parse_command(comando)

# Executar a função principal
main()
```

Por último, tem-se a função “main()”, responsável por iniciar a execução do programa. Ela mantém uma estrutura de repetição que pede constantemente a inserção de novos comandos para serem tratados pelo código. Caso o usuário digite a palavra “sair”, o código é automaticamente finalizado.

2) Saídas no Terminal

OBS: Os exemplos utilizados por nosso grupo estarão em um arquivo .txt chamado de “exemplos.txt”. No exemplo que será exposto a seguir, o exemplo utilizado foi este:

CREATE DATABASE Empresa;
USE Empresa;
CREATE TABLE Funcionario (nome string, idade int, CPF int);
INSERT INTO Funcionario (nome, idade, CPF) VALUES ("fernando", 25, 02263563885);
SELECT * FROM Funcionario;
SELECT nome, CPF FROM Funcionario;
SELECT * FROM Funcionario ORDER BY idade;
SELECT * FROM Funcionario WHERE idade = 25;
UPDATE Funcionario SET nome = "carlos" WHERE nome = "fernando";
DELETE FROM Funcionario WHERE nome = "carlos";
TRUNCATE TABLE Funcionario;

```
Digite um comando SQL: CREATE DATABASE Empresa;
Criando o banco de dados 'Empresa'
Digite um comando SQL: USE Empresa;
Usando o banco de dados 'Empresa'
Digite um comando SQL: CREATE TABLE Funcionario (nome string, idade int, CPF int);
Criando a tabela 'Funcionario' com as colunas:
- nome string
- idade int
- CPF int
Digite um comando SQL: INSERT INTO Funcionario (nome, idade, CPF) VALUES ("fernando", 25, 02263563885);
Inserindo dados na tabela 'Funcionario':
- nome: "fernando"
- idade: 25
- CPF: 02263563885
Digite um comando SQL: SELECT * FROM Funcionario;
Executando comando SELECT:
Selecionar todas as colunas da tabela 'Funcionario'
Da tabela 'Funcionario'
Digite um comando SQL: SELECT nome, CPF FROM Funcionario;
Executando comando SELECT:
Selecionar as colunas:
- nome
- CPF
Da tabela 'Funcionario'
Digite um comando SQL: SELECT * FROM Funcionario ORDER BY idade;
Executando comando SELECT:
Selecionar todas as colunas da tabela 'Funcionario'
Da tabela 'Funcionario'
Ordenar por coluna: idade
Digite um comando SQL: SELECT * FROM Funcionario WHERE idade = 25;
Executando comando SELECT:
Selecionar todas as colunas da tabela 'Funcionario'
Da tabela 'Funcionario'
Condição WHERE: idade = 25
Digite um comando SQL: UPDATE Funcionario SET nome = "carlos" WHERE nome = "fernando";
Executando comando UPDATE na tabela 'Funcionario':
Definir coluna 'nome' como '"carlos"'
```

```
Condição WHERE: nome = "fernando"
Digite um comando SQL: DELETE FROM Funcionario WHERE nome = "carlos";
Executando comando DELETE na tabela 'Funcionario':
Condição WHERE: nome = "carlos"
Digite um comando SQL: TRUNCATE TABLE Funcionario;
Executando comando TRUNCATE TABLE na tabela 'Funcionario'
Digite um comando SQL: sair
```

Após a inserção do comando “sair”, a aplicação é finalizada.

OBS: Para o reconhecimento ser realmente efetivado, é necessário que o usuário digite os comandos SQL no seu formato correto (definido na descrição da atividade). Qualquer erro na digitação dos comandos ou na ordem em que são digitados será gerado um erro de “Comando Inválido”. Segue o exemplo a seguir:

```
Digite um comando SQL: CREATE DATABASE Empresa;
Criando o banco de dados 'Empresa'
Digite um comando SQL: USE Empresa;
Usando o banco de dados 'Empresa'
Digite um comando SQL: CREATE TABLE Funcionario (nome string, idade int, CPF int
Comando inválido
Digite um comando SQL:
```

Neste caso, o comando SQL “CREATE TABLE” foi digitado de forma errada (faltou finalizar os parênteses e o “;”), ocasionando no retorno do print “Comando Inválido”. Após o print, retorna-se ao usuário uma nova possibilidade de input.