Relatório do trabalho da disciplina de Processamento de Linguagens

Documentação do trabalho prático 2

Roberto Filipe Manso Barreto - 21123

Henrique Monteiro Cartucho - 21122

Licenciatura de Engenharia de Sistemas Informáticos

Janeiro de 2022

**Índice**

[1. Introdução 1](#_Toc92571808)

[1.1. Estrutura do documento 1](#_Toc92571809)

[1.2. Resumo 1](#_Toc92571810)

[*2.* *Lexer* 2](#_Toc92571811)

[3. Gestão das tabelas 3](#_Toc92571812)

[4. Gramática 3](#_Toc92571813)

[4.1. LogicGrammar 3](#_Toc92571814)

[4.1.1. Interpretação de comandos simples 3](#_Toc92571815)

[4.2. LogicEval 6](#_Toc92571816)

[5. Logic 8](#_Toc92571817)

[Conclusão 9](#_Toc92571818)

# Introdução

## Estrutura do documento

Este documento refere-se à descrição e documentação do trabalho prático realizado pelos alunos Roberto Filipe Manso Barreto e Henrique Monteiro Cartucho da turma de licenciatura de engenharia de sistemas informáticos. Este documento contém a descrição do pensamento e do desenvolvimento do problema e dificuldades do mesmo.

## Resumo

Inicialmente foi proposto pelo professor da disciplina 2 enunciados dos quais seria necessário escolher um destes, foi então escolhido o enunciado B, com este enunciado foi iniciado um projeto cujo objetivo consiste em desenvolver um interpretador para a linguagem TQL. Este projeto tem como grande objetivo explorar expressões regulares, *lexer*, e a capacidade de produzir gramáticas logicas utilizando a biblioteca *ply* na linguagem *python*.

# *Lexer*

Para desenvolver o *lexer* foi então criado o ficheiro *logic\_lexer,* neste ficheiro foi criada a classe LogicLexer, nesta classe foram criados tokens para todos os comandos de TQL, acrescentando alguns de iniciativa do grupo, sendo estes comments, INSERT e ATTRIBUTES.

Para realizar as expressões foram criadas funções respetivas para estes tokens:

* t\_command → Este método tem como função reconhecer comandos de TQL como SELECT, SHOW, DELETE, entre outros, para isso foram colocados todos os comandos neste *token* e atribuído o seu valor ao tipo do *token*.
* t\_comments → Este método tem como função reconhecer comentários e ignorar estes realizando um *pass* no final da função.
  + Expressão → ‘--.\*\n’
  + Esta expressão reconhece qualquer caracter que esteja sucedendo os caracteres “—” até encontrar um caracter new line
* t\_string → Este método tem como função reconhecer strings que estejam entre o caracter ‘”’
  + Expressão → ‘”[^\.]+\.[^”]+”’
  + Esta expressão reconhecer qualquer caracter quer exista até encontrar um ponto que esteja sucedendo o caracter ‘”’, ou seja o nome do ficheiro, depois lê a extensão do ficheiro sendo esta qualquer valor até encontrar o caracter ‘”’
* t\_nr → Este método tem como função reconhecer números convertendo diretamente para um float
  + Expressão → ‘[0-9]+(.[0-9]+)?’
  + Esta expressão reconhece qualquer numero ‘[0-9]+’ estando este ou não seguido de uma virgula flutuante ‘(.[0-9]+)?’
* t\_str → Este método tem como função reconhecer qualquer tipo de palavra
  + Expressão → ‘[a-zA-Z0-9\_]+|\\*’
  + Esta expressão reconhece qualquer palavra contenha esta letras e ‘\_’, ou então reconhece o caracter ‘\*’ utilizado para selecionar todas as colunas de uma tabela
* t\_comparator → Este método tem como função reconhecer caracteres utilizados para comparações
  + Expressão → ‘<|>|=|>=|!=|<>’

Para ignorar *new lines, tabs,* aspas e espaços brancos que não são de interesse para o programa foi utilizado o token *t\_ignore e atribuído a string ‘ \”\n\t’,*  ignorando assim todos os caracteres desnecessários.

# Gestão das tabelas

A gestão de tabelas do projeto é necessária, desta forma para resolver este problema foi criada a classe Tables, esta classe contém as tabelas criadas e carregadas e contém todos os métodos necessários para a gestão destas tabelas, sendo estes métodos de carregar a partir de um ficheiro, guardar em ficheiros, mostrar tabelas ou selecionar tabelas entre outros.

# Gramática

De forma a ser possível interpretar comandos é necessário criar uma gramática capaz de reconhecer os comandos escritos indicando também quando acontecem problemas e quando não acontecem problemas notificando o utilizador para que este consiga também entender os erros que pode cometer durante a escrita de comandos.

As classes *LogicGrammar* e *LogicEval* vêm então complementar-se com o objetivo de interpretar os comandos escritos pelo utilizador e avaliar estes logicamente efetuando de seguida todo o processo que estes comandos indicam.

## LogicGrammar

Esta classe tem como principal objetivo interpretar os comandos que o utilizador coloca no interpretador, deste modo sempre que um comando é interpretado, um dicionário é criado com todas as informações necessárias para a realização da avaliação deste comando.

### Interpretação de comandos simples

Para a interpretação de comandos simples como carregar tabelas, demonstrar tabelas, eliminar tabelas e guardar tabelas em ficheiros foram criadas as regras:

* S' -> code
  + O axioma desta gramática é a regra code visto que pode ser necessário como por exemplo no caso do comando procedure que pode conter várias linhas de comandos dentro do mesmo
* code -> S | S ; | code S ;
  + A regra code deriva em comando sem ponto e virgula, comando com ponto e virgula ou então uma o code até agora registado mais S, criando assim uma lista caso se verifique o último caso, alcançando assim código de múltiplos comandos.
  + Visto que caso exista apenas um comando não existe necessidade de utilizar o ponto e virgula foi criada então esta opção permitindo então escrever comandos com ou sem ponto e virgula. Mas o ponto e virgula é obrigatório quando se trata de múltiplos comandos sequenciais
* S -> command | sel\_queries
  + A regra S deriva em command ou sel\_queries, command é utilizado para todos os comandos que não envolvam selects como comando principal, sel\_queries é utilizado para todos os comandos que são queries que envolvem selects.
  + A separação de comandos foi criada pois existem comandos que necessitam do resultado de comandos selects para serem efetuados sendo estes por exemplo a criação de tabelas a partir de um select realizado. Com isso foi necessário criar a distinção entre os dois tipos de comandos
* command -> PROCEDURE str DO code END | LOAD TABLE A FROM A |

CREATE TABLE A FROM A | DISCARD TABLE A | SHOW TABLE A | SAVE TABLE A AS A | CREATE TABLE A FROM sel\_queries | CREATE TABLE A FROM A JOIN J | CALL str |

CREATE TABLE A ATTRIBUTES a\_list | INSERT a\_list INTO A

* + A regra command deriva em todas as possibilidades de comandos reconhecidos pelo interpretador, esta regra foi criada explicitamente com o nome dos comandos visto que assim não é possível realizar comandos que não teriam sentido de serem executados
  + De forma a conseguir enviar as informações obtidas foi utilizada uma árvore abstrata de sintaxe onde é colocado o comando a efetuar, a lista dos argumentos a enviar, estes argumentos nem sempre se deseja efetuar a avaliação logo estes são colocados em um dicionário, caso sejam uma lista é colocada a chave lista e caso sejam uma string é colocada a chave string
  + Para permitir a seleção de vários atributos foi necessário utilizar recursividade para isso foi criada a regra a\_list que devolve uma lista de atributos, esta lista pode conter vários ou apenas 1 atributo, uma vez que não de deseja efetuar a avaliação destes atributos pois apenas são indicativos de atributos, estes são colocados em um dicionário com a chave str pois apenas contém nomes de atributos.
  + É necessário também permitir criar tabelas a partir de comandos selects e joins, desta forma foi então decidido pelo grupo que assim como SQL qualquer tipo de query sempre devolve uma tabela, sendo assim quando é necessário uma query de seleção para criar uma tabela, é referenciada a regra sel\_queries e o comando de seleção obtido pela regra é colocado nos argumentos diretamente de modo a esta ser avaliada e ser substituída pela tabela retornada na sua avaliação. Já quando é necessário um comando join a regra J é referenciada e a lista devolvida por esta é colocada nos argumentos envolvida em um dicionário com a chave join, pois não se pretende avaliar o nome da tabela e do atributo a juntar.
* sel\_queries -> SELECT a\_list FROM A lim | SELECT a\_list FROM A WHERE cond\_list lim
  + A regra sel\_queries deriva em todas as possibilidades de comandos selects, sendo estes comandos simples selects a uma tabela completa como até selects a apenas algumas colunas, com condições e com um limite de linhas a mostrar.
  + Para ser possível obter as diferentes colunas que o utilizador decide mostrar é referida a regra a\_list que devolve uma lista de strings, sendo assim é possível obter a lista de colunas que o utilizador deseja mostrar. Visto que não se deseja avaliar a lista de colunas a mostrar, esta é envolvida em um dicionário com a chave list.
  + De forma a ser possível colocar um limite no número de linhas a ser mostradas foi criada a regra lim que devolve ou uma string vazia ou o número de linhas a mostrar, este valor é colocado como valor da chave lim da árvore abstrata de sintaxe.
  + A utilização de condições nas queries do tipo select deve ser também permitida, desta forma foi criada a regra cond\_list que devolve uma lista de listas contendo todas as comparações necessárias, visto que não se deseja avaliar esta lista, então esta é envolvida em um dicionário com a chave list.
* A -> str | nr | string
  + A regra A deriva nos tokens str, nr ou string, estas são as possibilidades de escrita de nomes de tabelas, colunas e nomes de ficheiros
* E -> A comparator A
  + A regra E deriva em regra A, token comparator e regra A, desta forma é possível escrever comparações colocando o nome da coluna a comparar, o simbolo comparador e por fim o valor a comparar, retornando uma lista com estes 3 valores.
* cond\_list -> E | cond\_list AND E
  + A regra cond\_list deriva na regra E ou na própria regra cond\_list seguido de um AND e denovo a regra E
  + Esta regra retorna de forma regressiva uma lista de comparações através da segunda derivação ou uma simples comparação através da primeira derivação
  + O retorno desta regra é por fim uma lista de listas com cada lista sendo a comparação com 3 valores logo cada lista dentro da lista contém 3 valores.
* lim -> € | LIMIT nr
  + A regra lim deriva em € ou token LIMIT e token nr, desta forma ou pode não ter valor nenhum retornando uma string vazia, ou então pode ter LIMIT e um número, conseguindo assim retornar o LIMIT desejado caso este seja colocado.
* a\_list -> A | a\_list , A
  + A regra a\_list deriva na regra A ou na regra a\_list, seguida do literal ‘,’ seguida da regra A denovo, conseguindo assim obter o valor retornado na regra A caso derive na primeira opção, caso derive na segunda opção então de forma recursiva é construída uma lista de valores obtidos pela regra A separados por virgulas
* J -> A USING ( A )
  + A regra J deriva na regra A seguida do token USING, q de seguida contem o literal ‘(‘ seguido da regra A e por fim o literal ‘)’ denovo. Desta forma é possível obter o valor retornado na primeira regra A que equivale ao nome da tabela a realizar o join, é também possível obter o nome do atributo a utilizar para realizar o join, retornando assim estes dois valores em uma lista

# LogicEval

Para avaliar a árvore abstrata de sintaxe obtida a partir da classe LogicGrammar, foi necessário criar a classe LogicEval, esta classe contém um método chamado evaluate que recebe a árvore abstrata de sintaxe e caso esta seja um dicionário, este será avaliado pelo método *\_eval\_operator,*  caso esta arvore seja do tipo float ou do tipo string o valor é retornado e por fim caso a árvore seja do tipo lista, ou seja a árvore é um conjunto de comandos, então cada elemento da lista deve ser avaliado utilizando o método evaluate e passando por parâmetro o elemento da lista

Para avaliar cada operador e realizar as operações corretas no mesmo foi criado o método *\_eval\_operator,*  este método recebe a arvore de sintaxe abstrata, caso esta ast contenha a chave list ou a chave str, é retornado o valor da ast nessas chaves, caso esta ast contenha a chave join significa que o comando a executar é um comando join, logo este comando contém args, então é necessário obter os argumentos em uma lista, para isso é utilizado list comprehension e cada argumento no valor da ast na chave args é então avaliado através da função evaluate, de seguida é realizado o método*\_join* passando os argumentos por parametro.

Caso a ast contenha a chave command então significa que esta ast é uma ast de um comando completo, logo é necessário obter todos os argumentos como anteriormente mencionado, de seguida é necessário verificar se este comando contém a chave lim e caso contenha é necessário acrescentar à lista o limite de linhas a apresentar, depois é necessário verificar se existe a chave code na ast, pois caso seja um comando procedure quer dizer que é possível existirem várias linhas de código dentro do mesmo e estas devem ser também acrescentadas aos argumentos do comando, por fim é verificado se o comando existe no dicionário de comandos, caso exista então a função respetiva é executada passando por parâmetro os argumentos obtidos.

Para ser possível manter uma grande quantidade de comandos e a fácil adaptação de novo código ao projeto foi criado um dicionário de comandos, este dicionário contém o nome do comando como chave e como valor contém a função que é necessário executar para executar o processo lógico deste comando. Cada função destes comandos recebe por parâmetro os argumentos do mesmo, estes são depois tratados para isso cada comando tem uma função especifica a realizar:

* LOAD → Este comando utiliza o método *load* da classe *Tables* para carregar uma tabela a partir de um ficheiro, para isso esta função recebe o argumento na posição 0 que é o nome da tabela a receber os valores e o argumento na posição 1 que é o nome do ficheiro a carregar, por sua vez esta função abre o ficheiro em modo de leitura e acrescenta ao dicionário de tabelas na chave do nome da tabela um novo dicionário, este dicionários contém as chaves header, que recebe a lista de atributos contidos na primeira linha do ficheiro, e a chave row que recebe uma lista de listas dos valores contidos nas restantes linhas sendo cada linha uma lista
* DISCARD → Este comando utiliza o método *discard* da classe *Tables* para eliminar uma tabela, este método recebe por parametro o nome da tabela e por sua vez elimina a tabela correspondente a esse nome através do método pop
* SAVE → Este comando utiliza o método *save* da classe *Tables* para guardar uma tabela em um ficheiro, esta função recebe por parametro o nome da tabela e o nome do ficheiro, de seguida abre o ficheiro em modo de escrita e escreve o header da tabela em questão que obtém através do nome da tabela e da chave header, por fim é escrito cada linha da tabela em cada linha do ficheiro terminando e fechando o ficheiro
* SHOW → Este comando utiliza o método *show* da classe *Tables* com o parametro isTablename como verdadeiro, este método recebe por parametro a tabela a mostrar e pode receber como opcional o limite de linhas a imprimir e se o parametro tabela é um nome de tabela ou uma tabela, neste caso esta função imprime primeiramente o valor da chave header da tabela em questão, de seguida imprime todas as linhas contidas no valor da chave row da mesma.
* SELECT → Este comando executa a função *\_select* da classe LogicEval*,* esta função recebe por parametro os argumentos do comando, de seguida é verificado se o elemento na posição 2 da lista é do tipo lista, caso seja, significa que existe uma lista de condições para este select, caso contrário, significa que é apenas um select simples a uma tabela logo os valores dos argumentos são passados para variáveis e por si para a função select\_table da classe Tables. Caso existam então condições no select é necessário primeiramente verificar se não existe uma lista vazia, de seguida é necessário verificar se existem 3 elementos em todas as condições e apenas se nenhum desses se verificar é possível executar o método select\_table e passado por parametro as colunas, tabela, o limite de linhas e as condições pelo parametro opcional.
  + A função select\_tables tem em atenção as colunas em questão, pois pode ser necessário mostrar todas as colunas ou apenas algumas, de seguida imprime a tabela tendo em conta as condições, as colunas desejadas e o número de linhas necessárias
* CREATE → Este comando executa a função *\_create* da classe LogicEval, este método recebe os argumentos do comando por parametro e verifica se o último valor dos argumentos é to tipo lista ou não, caso seja, significa que estamos perante um create de uma tabela vazia sem nenhum valor, recebendo então apenas as colunas da tabela e o valor, chamando por fim o método *create\_empty\_table* da classe Tables que cria uma tabela sem linhas aprensentando apenas o nome da tabela e os atributos desta. Caso o ultimo valor não seja uma lista então significa que estamos perante um create de uma tabela que caso o valor da tabela seja uma string significa que se está a receber o nome de uma tabela logo é necessário executar o método *create\_from\_table* da classe Tables, que duplica uma tabela para tabela, caso contrário significa que se recebeu ou o resultado de um join de tabelas ou o resultado de um select e se quer criar uma tabela a partir desse resultado chamando então o método *create\_from\_select*  da classe Tables.
* PROCEDURE → Este comando executa a função *\_procedure*  da classe LogicEval, este método recebe por parametro os argumentos do comando, de seguida guarda estes argumentos em duas variáveis a variável name que guarda o nome do procedimento e a variável code que guarda a lista de comandos a executar com esse procedimento, estando estes comandos não avaliados pois apenas se avaliam estes comandos quando o procedimento é executado, por fim estes valores são guardados no dicionário procedures da classe LogicEval na chave nome do procedimento tendo como valor a variável code. Conseguindo assim então guardar o procedure criado.
* CALL → Este comando executa o método *\_call* da classe LogicEval, este método recebe por parametro os argumentos do comando, de seguida este obtem o nome do procedimento a executar através dos argumentos, é verificado se o procedimento existe nos procedimentos guardados e por fim caso este exista o código do procedimento é avaliado sendo este então executado conseguindo assim executar o procedimento
* INSERT → Este comando executa o método *\_insert* da classe LogicEval, este método recebe por parametro os argumentos do comando, de seguida este obtem o nome da tabela e os valores a inserir, depois é verificado se a tabela existe, se existem mais valores do que atributos na tabela ou menos valores do que atributos na tabela, caso nenhum deste se verifique é realizado então o insert de valores na tabela através do método insert da classe Tables, que recebe o nome da tabela e os valores a inserir e insere os mesmos no valor da chave row na tabela desejada.

# Logic

Esta classe está encarregue de verificar se primeiramente o programa foi executado com mais do que um argumento, caso seja significa que será necessário carregar o código a partir de um ficheiro indicado pelo utilizador no segundo argumento, este código é então de seguida avaliado. Caso contrário significa que o utilizador vai realizar os comandos um por um na linha de comandos e será necessário avaliar cada um dos comandos realizados pelo utilizador.

# Conclusão

Em suma este trabalho prático permitiu consolidar os conhecimentos obtidos na disciplina de Processamento de Linguagens, através do desenvolvimento de um interpretador da linguagem TQL, este permitiu também o grupo explorar e implementar comandos que os mesmos decidiram colocar amais de forma a conseguir uma melhor compreensão e consolidação destes conhecimentos.