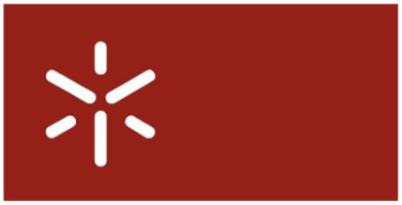

Processamento de Linguagens

Mestrado Integrado em Engenharia Informática

Trabalho Prático nº 1

FLex

Grupo XX

A85227 João Pedro Rodrigues Azevedo

A85729 Paulo Jorge da Silva Araújo

A83719 Pedro Filipe Costa Machado

Braga
Abril 2020

Conteúdo

Capítulo 1

Introdução

Este relatório é relativo ao trabalho prático da UC de Processamento de Linguagens que tem como um dos principais objetivos desenvolver, a partir de *Expressões Regulares (ER)* e processadores de linguagens regulares capazes de alterarem e transformarem textos através do conceito de regras de produção Condição-Ação.

Para tal, iremos recorrer ao *FLex* para gerar filtros de texto em C, num ambiente de trabalho *Linux* e outras ferramentas de apoio.

Assim, iremos analisar ficheiros de texto e tentar encontrar padrões de frases nos mesmos, de forma a ser possível criar *ERs*, que depois irão reagir da forma que nós estipulamos às mesmas, criando os resultados esperados.

Capítulo 2

Enunciado Escolhido

Com vista a realizar este trabalho, foram-nos propostos pela equipa docente seis enunciados e pedido, pela mesma, para escolhermos um.

Depois de algum debate entre os membros do grupo, achamos por bem escolher o **TransformadorPublico2NetLang** (enunciado 4). Neste enunciado, era nos pedido que analisássemos um ficheiro *HTML* que continha 85 comentários extraídos de uma notícia publicada na versão online do jornal "O Público".

Com a finalidade de se fazer um estudo sócio-linguístico de forma e conteúdo dos comentários que a notícia suscitou, era pretendido extrair do ficheiro *HTML* a informação relevante para análise.

De seguida, pedia-se que transformasse-mos esta informação para um formato *JSON*, como indica a figura 2.1.

```
"commentThread": [
  {
    "id": "STRING",
    "user": "STRING",
    "date": "STRING",
    "timestamp": NUMBER,
    "commentText": "STRING",
    "likes": NUMBER,
    "hasReplies": TRUE/FALSE,
    "numberOfReplies": NUMBER

    "replies": [ ]
  },.......
]
```

Figura 2.1: Formato Json

Esta escolha deveu-se ao facto de ser desafiante a procura de padrões num ficheiro de texto e ainda ser necessário a utilização de uma estrutura que consiga armazenar estes mesmo padrões em memória, para posteriormente serem utilizados com a sua respetiva finalidade.

Capítulo 3

Estratégia Adotada

Numa primeira fase, o grupo começou por analisar os comentários da página de notícias do jornal através do *browser* de Internet, tirando notas relativas ao tipo de comentários que poderiam existir. A título de exemplo, o grupo notou que os utilizadores podiam possuir caracteres especiais na sua identificação (nome de *user*).

De seguida, obtivemos o ficheiro *HTML* a partir do comando Linux *wget*, podendo começar, assim, uma análise mais detalhada ao modo do qual os comentários estavam estruturados.

Neste capítulo, iremos abordar a estrutura do ficheiro *HTML* em que trabalhamos, bem como a nossa implementação e especificação do filtro *FLex*.

3.1 Estrutura do Ficheiro HTML

O ficheiro *HTML* possui várias *tags* que o grupo definiu como relevantes no que conta à recolha de informação e, posteriormente, à definição de *ERs*. Assim, iremos indicar quais *tags* o grupo teve especial atenção e explicar qual o seu propósito.

- `<ol>`

  Esta *tag* indica que até ao final desta (`</ol>`) estariam um conjunto de comentários, representados pela *tag* `<li>`. De realçar que o ficheiro *HTML* possui uma `<ol>` principal que contém todos os comentários principais. Esta *tag* é representada por ter como etiqueta de abertura os atributos "*class*"e "*id*", como se segue no exemplo:

  ```
  <ol class="comments__list" id="approved-comments">
  ..
  </ol>
  ```

  Juntamente a este uso da *tag*, esta pode também representar uma lista de *replies* a um dado comentário. Este tipo de utilização da *tag* é identificável por apenas possuir o atributo "*class*", como se observa no seguinte exemplo:

  ```
  <ol class="comments__list">
  ..
  </ol>
  ```

- `<li>`

  A *tag* `<li>` serve para guardar todas as informações de um comentário. Dentro desta, podemos encontrar uma *tag* `<div>`, que contém outras duas *tags* `<div>`, cada uma contendo informações diferentes, ou seja a "*meta*"ou o "*content*"do comentário em causa.

  De realçar que dentro das `<div>`'s existem outras *tags* tais como: `<h5>`, `<time>`, `<p>` e, possivelmente `<ol>`, caso o respetivo comentário tenha *replies*.

  O grupo decidiu focar-se mais nestas últimas *tags*, pois são mais concretas em relação à informação que se pretende obter.

  Referir que `<li>` tem uma outra *tag*(`<form>`) que o grupo não notou grande importância para a formulação do ficheiro *Json*. Um exemplo do uso desta *tag* é o seguinte:

```
<li class="comment" data-comment-id="06..">
    <div class="comment__inner">
        <div class="comment__meta">..</div>
        <div class="comment__content">..</div>
    </div>
    ..
</li>
```

  Neste exemplo, verificámos que o valor do atributo "*data-comment-id*"representa o *id* do comentário.

- `<h5>`

  Com o `<h5>` e, dentro desta a *tag* `<a>`, podemos obter o nome do utilizador responsável pelo comentário. Esta *tag* é a primeira dentro de um `<li>` que se demonstra fulcral para obter a informação de forma a construir o ficheiro *Json* pretendido.

  Segue-se de seguida um exemplo desta *tag*:

```
<h5 class="comment__author">
    <a href="/utilizador/perfil/.." rel="nofollow">PdellaF </a>
</h5>
```

  Assim, é possível indicar o nome do utilizador depois do caracter ">"posterior ao atributo "rel"da *tag* `<a>`. Neste caso, o utilizador tem como nome "PdellaF".

- `<time>`

  A *tag* `<time>` armazena toda a informação relativa ao momento no tempo em que a mensagem/comentário foi realizado. Dentro desta, encontra-se outra *tag* (`<a>`) na qual se encontra a data do comentário, que contém o mesmo significado que a data do `<time>`. Assim, conseguímos obter não só a data, como também o *timestamp* após alguns processamentos.

  De realçar que o grupo para obter os valores necessários relativos a esta *tag* utilizou a data do `<a>` como referência.

```
<time class="dateline.." datetime="2019-10-03T21:11:55.99">
    <a class="comment__permalink">03.10.2019 21:11</a>
</time>
```

- `<p>`

  Dentro da `<div>` relativa ao conteúdo do comentário, encontramos a *tag* `<p>` que armazena o texto do comentário, como se observa no seguinte exemplo:

```
<div class="comment__content">
    <p>
        Como vamos de Salgado, Bava, Granadeiro e ..
    </p>
</div>
```

- `<h3>`

  A *tag* `<h3>` é a primeira *tag* disponibilizada pelo ficheiro *HTML* e indica o número de comentários que o ficheiro possui.

  Embora não seja pedida para o ficheiro *Json* final, a leitura desta *tag* serviu para verificarmos se o nosso programa estava a ler o número correto de comentários.

```
<h3 class="i-comment"></i> 85 comentarios</h3>
```

3.2 Implementação e especificação do filtro FLex

Em primeiro lugar, o grupo desenvolveu uma estrutura de dados capaz de suportar os dados relevantes e, de certa forma, semelhante à estrutura de um *commentThread* do ficheiro *Json* pretendido. Assim sendo, a estrutura de dados em C, definida no ficheiro "commentThread.h"ficou da seguinte forma:

```
typedef struct commentThread
{
    char*          id;
    char*          user;
    char*          date;
    long int       timestamp;
    char*          commentText;
    int            likes;
    int            hasReplies;
    int            numberReplies;

    struct commentThread* next;

} *COMMENT_T;
```

A transformação desta estrutura para *Json* é feita de maneira relativamente simples. Caso o valor "hasReplies"seja 1, ou seja, *True*, então quer dizer que os próximos comentários serão *replies* deste mesmo, sendo que o número de *replies* é dado pelo parâmetro "numberReplies". Assim, os comentários estão inseridos de forma ordenada em relação à leitura do ficheiro *HTML*, com as respostas de cada um deles inseridas logo após o comentário principal.

De seguida, foram desenvolvidas **ERs** (Expressões Regulares) para preencher os campos da estrutura "commentThread", as quais vamos explicar de seguida o processo feito. Para tal, iremos também mostrar um diagrama para uma melhor percepção da nossa especificação do uso de **FLex** (Figura 3.1).

Este diagrama explícita como os estados definidos pelo grupo no *FLex* se comportam. Assim, para cada estado e cada ligação entre eles, vamos indicar o propósito destes. De notar que as seguintes *ERs* estão definidas no ficheiro "filter.l"do trabalho.

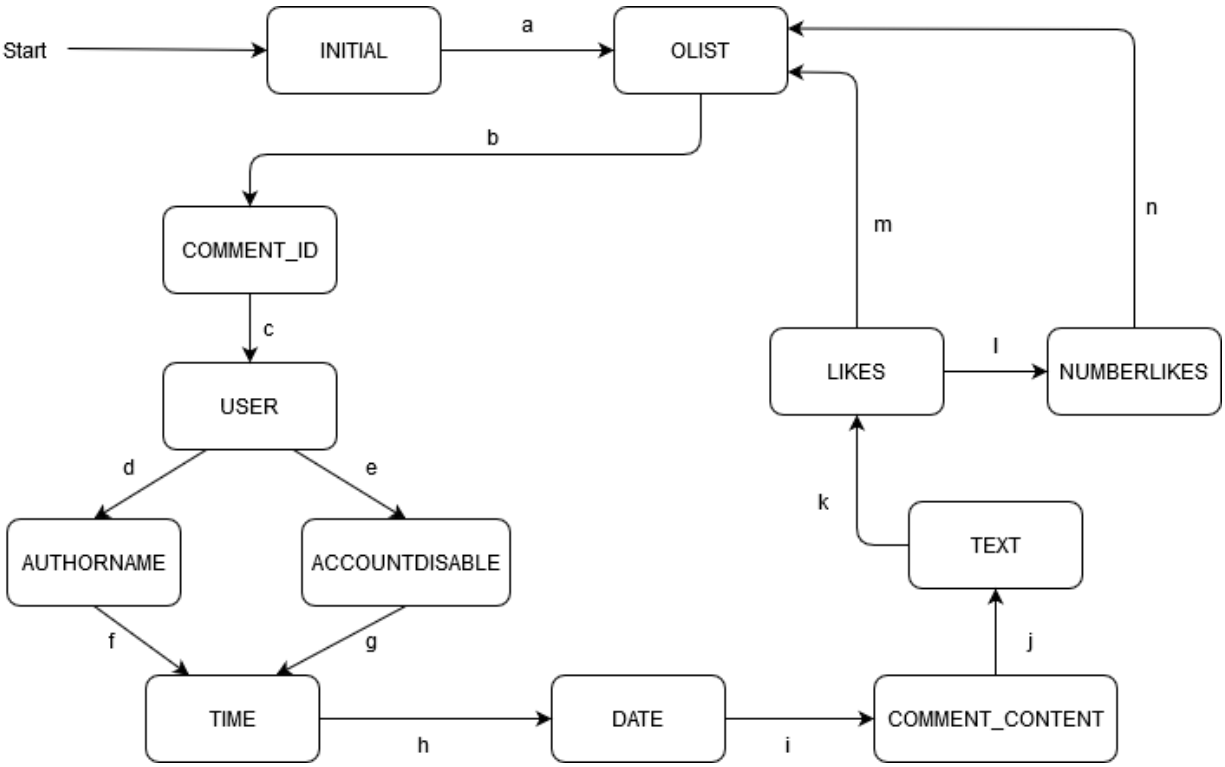

Figura 3.1: Diagrama de estados no FLex

- **Obter o número de comentários no ficheiro:**

  No diagrama observamos que o filtro começa no start e que o primeiro estado que este entra, ou seja, que dá match, é o **OLIST**. Contudo, antes de entrar neste estado o filtro executa uma *ER*, sendo esta a que lê o número de comentários que estão no ficheiro.

  Esta *ER* é a seguinte:

```
[0-9]+/[ ]+[-'a-zA-ZÀ-ÖØ-öø-ÿ]+<\/h3>\] {
        numberComments = atoi(yytext);
        }
```

  No código acima, a ER dá *match* ao conteúdo da *tag* `<h3>` que indica o número de comentários que o ficheiro *HTML* possui. Segue-se um exemplo em que a *ER* dá match:

```
<h3 class="i-comment"></i> 85 comentários</h3>
```

  Neste exemplo, depois de feito o *match*, vamos trabalhar apenas com o número antes do caractere do espaço e da palavra (aceita acentos), epara isso, utilizámos a barra "/".

  Assim, podemos armazenar o número de comentários numa variável global previamente definida, através do C, que neste caso fica a 85.

- **OLIST**

  O estado **OLIST** é iniciado quando há um "match"na seguinte *ER*.

```
\<ol(.*)class\=\"comments__list\"(.*)\>\< {
    ct = (COMMENT_T) malloc(sizeof(struct commentThread));
    BEGIN OLIST;
    }
```

  A *ER* acima apenas faz "match"uma vez durante o programa inteiro (caso o ficheiro *HTML* esteja na forma que esperada), pois depois de entrar neste estado o programa percorre um ciclo, como se percebe no diagrama da figura 3.1.

  Assim, o estado **OLIST** é iniciado na *tag* `<ol>` que possui todos os comentários e é aí inicializada a estrutura de dados em C previamente referida.

  Já no estado **OLIST**, utilizámos uma *ER* para cada início de comentário (representado pela *tag* `<li>`), inicializando algumas das suas variáveis como se observa:

```
<OLIST>li(.*)data-comment-id\=\" {
    ct -> next = (COMMENT_T) malloc(sizeof(struct commentThread));
    ct = ct->next;

    ct->timestamp = 0;
    ct->likes = 0;
    ct->hasReplies = 0;
    ct->numberReplies = 0;

    BEGIN COMMENT_ID;
    }
```

  Após o "match", é chamado o estado **COMMENT_ID** para trabalhar com o id do comentário.

Outra *ER* definida deste estado tem como objetivo fazer "match"ao início de um conjunto *replies* de um comentário, ou seja, a uma nova *tag* `<ol>`.

```
<OLIST>\<ol(.*)\"comments__list\"\>\n*\<  {
    ct->hasReplies = 1;
    p = ct;

    ct->next = (COMMENT_T) malloc(sizeof(struct commentThread));
    ct = ct->next;

    ct->timestamp = 0;
    ct->likes = 0;
    ct->hasReplies = 0;
    ct->numberReplies = 0;

    replies = 0;

    BEGIN COMMENT_ID;
    }
```

Visto que estamos a trabalhar com uma *reply* temos de indicar isso na estrutura de dados do comentário principal, ou seja, metemos o valor do "hasReplies"a 1. Para podermos futuramente alterar o valor "numberReplies"do comentário principal, temos de usar uma outra "commentThread"**p** para guardar os campos desse comentário.

De seguida, andamos com a estrutura para o próximo elemento (sendo esta agora uma *reply*), inicializar os seus valores e, por último, chamar o estado **COMMENT_ID** para obter o seu id.

Visto que o estado **OLIST** trabalha tanto com comentários principais e as suas respostas, precisamos de uma outra *ER* para verificar se as respostas de um comentário já terminaram.

```
<OLIST>\<\/ol\>   {
    if(countComments <= numberComments)
        p->numberReplies = replies;
    }
```

Assim, ocorre o "match"quando encontramos o fim da *tag*, ou seja, `</ol>`. Falta-nos apenas alterar o valor do número de respostas do comentário principal (representado pelo "p") dado pela variável definida anteriormente, "replies". Esta variável é incrementada no estado **COMMEN_ID**. De notar, que a condição no código serve para não meter dados errados no último comentário, quando a *ER* faz "match"ao `</ol>` da tag principal com todos os comentários principais.

Por último, o programa possui uma *ER* para manter o terminal limpo do texto que não deu "match"pelas *ERs* definidas neste estado.

Esta *ER* é definida várias vezes ao longo do programa e dos diferentes estados, pelo que a explicação é semelhante.

- **COMMENT_ID**

  Neste estado fazemos match ao id do comentário através de uma *ER* capaz de apanhar letras, números e o caracter '-', pertencentes ao id de um *user*. De realçar que este estado é sempre chamado pelo **OLIST**, pelo que o match deste será sempre o id do comentário.

```
<COMMENT_ID>[A-Za-z0-9-]+/\"> {
    replies++;
    ct->id = strdup(yytext);

    BEGIN USER;
    }
```

  Para além de inicializar o id do comentário e de incrementar o número de replies, este estado também chama o estado **USER**.

- **USER** O estado **USER** tem como propósito encontrar a *tag* `<h5>`, juntamente com a `<a>`. Com estas *ERs* o objetivo é fazer match na posição correta do nome do utilizador, para permitir depois chamar o estado **AUTHORNAME** ou o estado **ACCOUNTDISABLED** possibilitando armazenar o valor do nome do *user* na estrutura.

  Caso o utilizador não tenha a conta desativada, este possui a *tag* `<a>` que contém o nome deste. Chamamos assim o estado **AUTHORNAME**.

```
<USER>\<h5(.*)\>\n\<a(.*)\"> { BEGIN AUTHORNAME; }
```

  Caso não possua a *tag* `<a>`, irá então chamar o estado **ACCOUNTDISABLED**. De notar que esta *ER* não faz "match"a utilizadores sem a conta desativada, visto que o *FLex* apanha a maior expressão, que é sempre a *ER* acima.

```
<USER>\<h5(.*)\>\n[ ]+   { BEGIN ACCOUNTDISABLED; }
```

- **AUTHORNAME**

  A *ER* deste estado faz "match"com todo o tipo de texto que vem antes do fecho das duas *tags*, `</a>` e `</h5>`. Depois de obter o "match"é introduzido o valor do nome do utilizador na estrutura de dados e iniciado o estado **TIME**.

```
<AUTHORNAME>(.*)/\<\/a\>\n\<\/h5\> {
    ct->user = strdup(yytext);
    BEGIN TIME;
    }
```

- **ACCOUNTDISABLED**

  Este estado difere do anterior porque não possui a *tag* `<a>`, logo a *ER* apenas trabalha com o conteudo que está dentro do `<h5>`, sendo esta definida, então, por:

```
<ACCOUNTDISABLED>(.*)/\n[ ]+\<\/h5\> {
    ct->user = strdup(yytext);
    BEGIN TIME;
    }
```

  Assim, este estado também inicializa o campo *user* da estrutura e chama o estado **TIME**.

- **TIME**

  O estado **TIME** serve para posicionar o "match"no sítio correto para depois o estado seguinte (**DATE**) poder obter os valores da data e do *timestamp*.

  Assim, a *ER* deste estado faz "match"à *tag* `<time>` e, dentro desta a `<a>`. É nesta última que se encontram os valores pretendidos.

```
<TIME>\<time(.*)\>\n\<a(.*)\"\>    { BEGIN DATE; }
```

- **DATE**

  É neste estado que os campos da estrutura relativos à data do comentário(*date* e *timestamp*) são preenchidos.

  Assim sendo, fazemos "match"a tudo até `</a>`, visto que o estado anterior já nos garante a posição correta para obter os dados relevantes.

  De realçar que o grupo decidiu usar o valor de `<a>` tanto para o *date*, como para a obtenção do *timestamp*.

```
<DATE>(.*)/\<\/a> {
    ct->date = strdup(yytext);

    sscanf(yytext, "%d.%d.%d %d:%d", &day, &month,
                   &year, &hour, &minutes);

    struct tm t;
    time_t data;

    t.tm_year = year;
    t.tm_mon = month;
    t.tm_mday = day;
```

```
t.tm_hour = hour;
t.tm_min = minutes;
t.tm_sec = 0;
t.tm_isdst = -1;
data = mktime(&t);

ct->timestamp = (long) data;

BEGIN COMMENT_CONTENT;
}
```

De notar que para a obtenção do valor de *timestamp*, o grupo utilizou a biblioteca "time.h"do C.

De seguida, invoca-se o estado **COMMENT_CONTENT**.

- **COMMENT_CONTENT**

  O estado **COMMENT_CONTENT** propociona a encontrar o sítio correto do início do texto do comentário. Assim, percorremos a *tag* `<p>` de forma a depois chamar o estado **TEXT**.

```
<COMMENT_CONTENT>\<p\>[ \n]*  { BEGIN TEXT; }
```

- **TEXT**

  Com este estado, conseguímos obter o texto do comentário e podemos aumentar o número de comentários lidos pelo filtro, através da variável "countComments".

  De realçar que usamos a função "takeEnterOut", definida no ficheiro "commentThread.h"de modo a não haver ocorrências do caractere `\n` no texto. Esta função é necessária de forma a manter a estrutura do ficheiro *Json* correta, pois não aceita campos com este caractere.

  Por último, podemos chamar o estado **LIKES**.

```
<TEXT>[^\<]*   {
    countComments++;
    ct->commentText = strdup(takeEnterOut(yytext));

    BEGIN LIKES;
    }
```

- **LIKES**

  O estado **LIKES** serve para dar "match", através de uma *ER*, à *tag* que armazena essa informação. Contudo, o ficheiro *HTML* que nos foi entregue, este não possui tais dados.

  Mesmo assim, o grupo decidiu que o filtro devia comportar-se como se tal informação existisse. Para isso, percorremos exemplos de ficheiros *HTML* de outros enunciados e seguimos a estrutura da notícia do jornal "O Sol"do enunciado 2.5, como o seguinte:

```
<span class="updatable count" data-role="likes">0</span>
```

  Logo, construimos a seguinte *ER* que chama o estado **NUMBERLIKES** para armazenar na estrutura o número de *likes*.

```
<LIKES>(.*)\"updatable(.*)\=\"likes\"\>  { BEGIN NUMBERLIKES; }
```

  Devido a não existir o número de *likes* no *HTML*, a *ER* nunca dá "match". Pelo que esta próxima *ER* serve para chamar o estado **OLIST** para voltar a ler o próximo comentário. Para além disso, verifica também se o comentário é o primeiro, para termos uma estrutura igual ao "ct", mas com no primeiro elemento desta. Assim, a estrutura "beginCt"serve para a escrita no ficheiro *Json* desde o primeiro elemento inserido até ao último.

```
<LIKES>.|\n {
    if(isBegin)
    {
        isBegin = 0;
        beginCt = ct;
    }

    BEGIN OLIST;
    }
```

- **NUMBERLIKES**

  Este estado, embora seja inalcançável, na teoria permite obter o número de *likes*, armazená-los na estrutura e invoca também o estado **OLIST**.

```
<NUMBERLIKES>[0-9]*/\<span\>  {
    ct->likes = atoi(yytext);
    if(isBegin)
    {
        isBegin = 0;
        beginCt = ct;
    }

    BEGIN OLIST;
    }
```

Capítulo 4

Estrutura do projeto e resultados obtidos

Neste capítulo iremos abordar algumas notas e detalhes que achamos importantes para a compreensão da estrutura do nosso trabalho, bem como, uma pequena demonstração do resultado obtido pelo filtro desenvolvido.

O filtro do programa está contido no ficheiro "filter.l". Este contém todas as Expressões Regulares, bem como a invocação das funções que escrevem os dados para *Json*.

De realçar que o ficheiro "commentThread.c"possui as funções relativas à escrita dos dados da estrutura para o ficheiro *Json*, que assume o nome "comments.json"e está contido na pasta "Files".

O trabalho possui também uma *Makefile* para a compilação e execução deste, sendo que dever-se-á utilizar os comandos "make"e "make run", respetivamente.

Segue-se de seguida um breve exemplo do resultado final do ficheiro *Json*:


```
{
"commentThread": [
{
"id": "06de7129-6167-49cd-d330-08d743683e5c" ,
"user": "PdellaF " ,
"date": "03.10.2019 21:11" ,
"timestamp": 61530959460 ,
"commentText": "Do assunto e de Justiça, Abrunhosa ..",
"likes": 0,
"hasReplies": "FALSE" ,
"numberOfReplies": 0 ,
"replies": [ ]
},
{
"id": "2c5940ee-754e-41f7-d893-08d748126a85" ,
"user": "PEDROA Santos " ,
"date": "03.10.2019 19:30" ,
"timestamp": 61530953400 ,
"commentText": "Como  vamos  de Salgado, Bava, .." ,
"likes": 0,
"hasReplies": "FALSE" ,
"numberOfReplies": 0 ,
"replies": [ ]
},.. ]}
```

Capítulo 5

Conclusão

Em suma, achamos que este trabalho teve grande proveito no que toca ao assimilar de conhecimentos relativos à utilização de Expressões Regulares para descrição de padrões de frases e ao desenvolvimento de *Processadores de Linguagens Regulares* que filtram e transformam textos, tendo em base o conceito de regras de produção *Condição-Ação*.

Também possibilitou aumentar a nossa experiência relativamente ao uso do ambiente *Linux* e de outras ferramentas, como o *FLex*, o C e *Json*.

Assim, achamos que cumprimos os requisitos propostos pela equipa docente no que toca a este projeto de *FLex*.