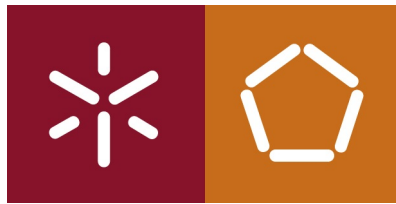


UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA



---

## Trabalho Pratico 1: Processamento de Texto em Flex

---

Processamento de Linguagens

André Soares (a67654)  
Guilherme Martins (a70782)  
Tiffany Silva (a76867)

Abril 2020

## Conteúdo

1	Resumo	2
2	Introdução	3
3	Estrutura de Dados	4
4	Metodologias de Desenvolvimento	7
5	Extras	9
6	Makefile	10
7	Conclusões gerais	11

---

# 1 Resumo

O Flex é uma ferramenta poderosa e versátil que pode ser utilizada nos mais diversos domínios. No primeiro projecto desta UC foi requisitado a elaboração de um filtro de texto recorrendo a Expressões Regulares ER e acções semânticas que realizem um processamento de uma página HTML, de modo a se extrair os dados relevantes para que estes sejam "guardados" num ficheiro JSON. Isto irá simplificar e facilitar o estudo dos comentários gerados de uma dada notícia num site de um jornal (Sol).

Assim, utilizando o Flex e através de ERs filtrou-se e transformamou-se o texto no formato que foi requerido.

---

## 2 Introdução

Apesar da análise de texto ser uma das tarefas mais comuns dos programadores, o recurso a linguagens tradicionais como Java ou C torna-a num processo demorado e complexo no que diz respeito ao desenvolvimento de analisadores léxicos. Nestas situações vem ao de cima a utilização de várias ferramentas/programas que facilitam a geração de analisadores léxicos respeitando regras de Condição-Ação. Esta abordagem é extramente vantajosa porque nos permite focar nos padrões que devem ser detetados e as acções a serem executadas.

Para a elaboração do projeto foi necessário realizar uma análise cuidada do texto do ficheiro HTML do qual se determinou quais as informações que seriam necessárias para produzir o ficheiro JSON. Após esta análise, desenvolveu-se ER's que descrevem padrões de frases e, a partir destas criou-se processadores de linguagens regulares que filtram e transformam textos com base no conceito de regras de produção de Condição-Ação. Com o auxílio de uma estrutura de dados em linguagem de programação imperativa C, guardou-se e processou-se o texto em run-time para obter um produto final no formato de um ficheiro JSON com a informação mais relevante devidamente organizada. Posteriormente, decidiu-se gerar novamente o site, desta vez utilizando os dados contidos no ficheiro JSON.

Assim sendo, ao longo deste documento explicou-se e justificou-se cada etapa do desenvolvimento deste trabalho prático.

---

### 3 Estrutura de Dados

Antes de desenvolver a estrutura de dados foi necessário analisar o problema e averiguar que dados se iriam armazenar de cada comentário. Desta forma, para cada comentário, vai-se guardar a informação relativa a:

- ID do Post
- Nome do User
- Username do user
- Data do comentário
- Timestamp do comentário
- conteúdo do comentário
- Número de likes
- Se tem ou não respostas
- Número de respostas
- Array de id's dos replies
- Se é ou não comentário Pai

Estando os dados pertinentes definidos, é necessário guardá-los para serem usados posteriormente. Para isso criou-se uma tabela de Hash, pois esta é ideal para guardar um grande volume de dados e a procura de uma dada entrada dá-se em tempo contanstante.

Sendo assim, a estrutura de Hash não passa de um array, em que cada indice tem a seguinte estrutura:

```
typedef struct comment {  
    char * id;  
    char * user;  
    char * nameUser;  
    char * date;  
    int timestamp;  
    char * comments;  
    int likes;  
    int hasReplies;  
    int numberOfReplies;  
    int *replies;
```

---

```
    int isPrinciple;  
} Comment;
```

Há variáveis cuja a sua interpretação é intuitiva como o id e o user, por exemplo. No entanto, é importante explicar algumas cuja a sua interpretação não é tão clara que são:

- int \* replies - é um vetor de inteiros que armazena os índices (da hash) das respostas a esse comentário
- int isPrinciple - representa um valor lógico : 0 - false, 1 - true. Este refere se é o comentário principal e é apenas útil para a geração do ficheiro JSON.
- variável global idHash - define a posição a inserir o próximo comentário. Esta incrementa o seu valor cada vez que é inserido um novo comentário. A ideia de ter um variável global para o id prendeu-se na necessidade de ter a ordem original dos comentários, pois esta será aproveitada aquando a inserção de uma resposta a um comentário.

```
void insertReply(Hash hp, char* nameUser, int idPost){  
    int i = idHash-1;  
    int flag = 0;  
  
    while(i>=0 && !flag){  
        if (strcmp(nameUser, hp[i]->nameUser)== 0){  
            int j = 0;  
            while(hp[i]->replies[j]!=EMPTY) j++;  
            hp[i]->replies[j] = idPost;  
            hp[i]->hasReplies = 1;  
            hp[i]->numberOfReplies++;  
            flag = 1;  
        }  
        else i--;  
    }  
}
```

Figura 1: Função que insere no comentário pai o id do filho

Esta função serve para inserir no vetor dos replies uma resposta a um dado comentário. Desta forma, quando se depara com uma resposta a um dado comentário é invocada esta função. Assim sendo, apenas é lhe passado como parâmetro o nome do user a quem respondeu e o numero do comentário atual. A função procura de ids mais altos para mais baixo até encontrar o nome correspondente. Quando encontra, insere o id da resposta no vetor de replies e incrementa ao nº de respostas.

Após ser concluído o armazenamento da informação do texto na estrutura passou-se à fase de criação e escrita do ficheiro JSON.

O ficheiro é criado e os comentários vão sendo escritos, a cada um é verificado se tem filhos e caso se confirme esses são impressos.

Para além destas funções temos ainda as típicas funções de print, criação e libertação de memória como podemos ver pelas assinaturas das funções na nossa API:

---

```
void insert(Hash hp, char * id, char * user, char* nameUser, char * date, int timestamp, char * comments, int likes){

    int key = idHash++;

    if (strcmp(hp[key]->id,EMPTYC) == 0){
        hp[key]->id = id;
        hp[key]->user = user;
        hp[key]->nameUser = nameUser;
        hp[key]->date = date;
        hp[key]->timestamp = timestamp;
        hp[key]->comments = comments;
        hp[key]->likes = likes;
    }
}
```

Figura 2: Função que insere um comentário novo na hash

```
void init(Hash hp);
void displayHash(Hash hp);
void insert(Hash hp, char * id, char * user, char*
            nameUser, char * date, int timestamp,
            char * comments, int likes);
void freeHash(Hash hp);
void insertReply(Hash hp, char* nameUser, int idPost);
void creatingJsonFile(Hash hp);
```

---

## 4 Metodologias de Desenvolvimento

Passou-se agora a descrever o funcionamento do analisador léxico desenvolvido. Iniciou-se o desenvolvimento do projecto com uma análise precisa do ficheiro html, com o objectivo de determinar os campos a capturar e a informação a processar. Tendo analisado a composição do que se presumiu ser um post, partiu-se para a criação da estrutura de hash que vai armazenar a informação, seguido da sua inicialização.

Com os campos a capturar definidos foi possível escrever as start conditions que são inicializadas à medida que o texto é processado. Desta maneira permitimos que estas se iniciem apenas quando o scanner se encontra naquela posição ou quando invocamos o BEGIN name\_startCondition.

De modo a ignorar tudo que não respeita às expressões regulares adicionamos a condição `.—\n` seguida de nenhuma acção `{;}`.

Tendo em conta a estrutura do ficheiro html, foi identificado como o início de cada post o seguinte segmento: `<li class="post"`, isto significa que todos os comentários são abertos por esta tag, neste caso, o ficheiro html terá que ter 43 vezes esse padrão correspondente aos 43 comentários presentes no ficheiro.

Iniciou-se o processamento com o estado POST que determina o início do post, a expressão regular que dá trigger ao POST e que diz respeito ao padrão acima identificado é

```
"\<li _class=\" post\""
```

Permitindo então fazer BEGIN dos estados sequencialmente:

- **POSTID**

estado onde se guarda o id do post: `"\"post-[0-9]*\""`

- **USER**

estado para a análise da informação do username, aqui é feita a distinção entre Guest e user registado, caso seja registado vamos para o estado USERNAME caso contrário é gravado "Guest" na variável username

- **USERNAME**

estado onde se guarda o username do user registado

- **BNAME e NAME**

antes de se guardar o nome do utilizador passa-se sempre por estas condições, no fim do estado NAME é sempre invocado o estado NAMEINFO

- **NAMEINFO**

estado onde se guarda o nome do user



- 
- **BDATE**  
estado onde é verificado se o comentário é uma resposta a outro e caso se confirme inicia o PARENTINFO, se não for um reply vai começar a processar a data para o estado DATE
  - **PARENTINFO**  
estado onde é feito a inserção do id do comentário "filho" no comentario pai, usando o nome do utilizador a quem respondeu este comentário.No final volta-se ao BDATE
  - **DATE**  
estado de ajuda, passa-se sempre por aqui antes de ser lida a data.
  - **DATEINFO**  
estado onde é lida a data e guardada numa variável, no final passa-se para o estado TIMESTAMP que aparece sempre depois da data
  - **TIMESTAMP**  
estado onde é lido o timestamp e guardado numa variável, depois do TIMESTAMP começa-se a processar o comentário passando para o estado BCOMMENT
  - **BCOMMENT e COMMENT**  
estados de ajuda para processar tags que aparecem sempre antes de serem efectuados comentários
  - **COMMENTTEXT**  
estado onde é lido o comentário e guardado na variável, é também feita a filtragem de tags de paragrafo, negrito, entre outras. Quando é terminado com o "`\i/div\i`" prossegue-se para o processamento dos likes do comentário no BLIKES
  - **BLIKES**  
estado de ajuda, quando se encontra a tag que marca que vão ser indicados os likes inicia-se o estado LIKES
  - **LIKES**  
leitura e gravação do numero likes, no final deste estado é feita a gravação de todos os elementos do comentário na hash, passando-se ao estado INITIAL que permite ler outro comentário e começar este processamento novamente.

Após esta captura de dados (`yylex()`), chama-se a função "creatingJsonFile" que vai então criar o ficheiro JSON, inserindo os dados de uma forma organizada apartir da estrutura hash.

---

## 5 Extras

Após a geração do ficheiro JSON, sentiu-se a necessidade de expor a informação de uma maneira mais organizada e apelativa e também para tirar partido do ficheiro JSON criado. Com esse objectivo e como aspecto extra do projecto, extraiu-se os dados do ficheiro JSON para um novo ficheiro HTML com alguns retoques visuais.

Com recurso a algum código em JavaScript, fez-se o read e o parse do ficheiro JSON através de um ficheiro `index.js` e por fim fez-se o render para HTML a partir do ficheiro `info.ejs` onde se organizou o tratamento dos dados do ficheiro JSON e o aspecto do resultado na forma de um ficheiro HTML.

De este modo, obtemos uma página mais familiar e apelativa representada abaixo.

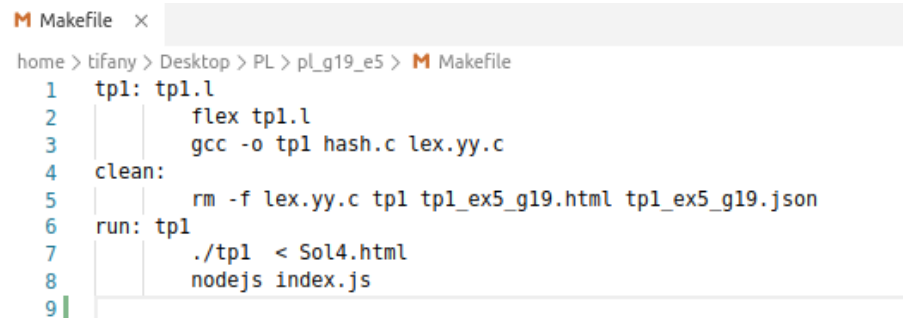


Figura 3: Output da Conversão de JSON em HTML

---

## 6 Makefile

De modo a que a correr o programa foi necessário criar uma Makefile para compilar a componente do flex com a do c. Desta forma, a makefile desenvolvida foi a seguinte:

A screenshot of a code editor window titled 'Makefile'. The editor shows a Makefile with the following content:

```
1  tp1: tp1.l
2      flex tp1.l
3      gcc -o tp1 hash.c lex.yy.c
4  clean:
5      rm -f lex.yy.c tp1 tp1_ex5_g19.html tp1_ex5_g19.json
6  run: tp1
7      ./tp1 < Sol4.html
8      nodejs index.js
9
```

The editor interface includes a tab at the top, a breadcrumb trail 'home > tifany > Desktop > PL > pl\_g19\_e5 > Makefile', and line numbers on the left margin.

Figura 4: Makefile

Assim sendo, para compilar e correr o programa basta inserir na pasta do repositório o seguinte comando:

- make run

Para limpar o executavel e os ficheiros que foram gerados é necessário correr o seguinte comando:

- make clean

É importante referir que para que o programa seja executado é necessário ter o seguinte componente instalado:

- nodejs

---

## 7 Conclusões gerais

Após a realização deste trabalho, é possível reconhecer que foram várias as dificuldades encontradas no desenvolvimento de expressões regulares bem como na geração adequada de filtros de texto para implementar a solução necessária para resolver o enunciado escolhido. As principais dificuldades prenderam-se na implementação de um mecanismo de "recursividade" que permitia processar todos os posts presentes no ficheiro html. Para além disso, também se achou complicado o processamento do campo dos comentários pois este continham promenores que tinham de ser processados especificamente.

Apesar de tudo, escolheu-se os recursos adequados e reconheceu-se que devido ao seu grande poder expressivo, versatilidade e rapidez que permite o desenvolvimento de analisadores léxicos, o FLEX é a escolha mais acertada como ferramenta de processamento de texto.

Como neste exercício foi requerido o desenvolvimento de um ficheiro json achou-se pertinente desenvolver um programa que fosse capaz de extrair a informação relevante e criar um novo html com os dados devidamente formados. Neste processo a maior dificuldade foi na extração dos dados do ficheiro json.

Conclusivamente, achou-se que a capacidade de resposta face às problemas encontrados foi à altura e que este projeto serviu para aprofundar os conhecimentos que tinham sido previamente lecionados nas aulas.