Sol2NetLang

Eduardo Lourenço da Conceição
a83870

António Manuel Carvalho Gonçalves
a85516

João Pedro Dias Fernandes
a84034

5 de Abril de 2020

Resumo

Para a cadeira de Processamento de Linguagens, do 3º ano da Licenciatura em Engenharia Informática, foi-nos proposto realizar um *parser* de HTML para JSON, baseando-nos na estrutura dos comentários presentes no jornal *Sol*, utilizando a ferramenta FLex. Neste relatório iremos expor todos os passos por que passamos para produzir o resultado final, bem como as dificuldades que tivemos e como as ultrapassamos.

Conteúdo

1 Introdução

No âmbito da cadeira de Processamento de Linguagens, foi-nos proposto um projeto para realizar um *parser* de HTML para JSON, utilizando a ferramente FLex, e utilizando o esquema de HTML utilizado pelo jornal português *Sol*.

Neste relatório demonstraremos como chegamos a este resultado final, explicitando os padrões que achamos necessários identificar para obter a informação relevante, as expressões regulares utilizadas para alcançar este fim, as ações semânticas que se associavam a estas expressões, as estruturas de dados para guardar a informação extraída e, por fim, as funções necessárias para imprimir a informação com a semântica de JSON num ficheiro. Numa última secção, apresentaremos também a *Makefile* associada ao projeto.

2 Padrões relevantes e Expressões Regulares

Para entendermos quais são as informações prinicpais que queremos retirar do documento de HTML fornecido, temos primeiro de olhar para a estrutura do JSON pedido:

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

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

Figura 1: Estrutura do ficheiro JSON

Assim, concluímos que será preciso, para cada post, retirar a seguinte informação:

- ID do comentário;
- Nome do comentador;
- Data em que o comentário foi escrito;
- Timestamp do comentário;
- Texto do Comentário;
- Número de *likes* que o comentário tem;

- Se o comentário tem, ou não, respostas;
- Número de respostas ao comentário;

O aspeto das respostas que são, por sua vez, comentários será explicado mais à frente. Assim, temos de perceber como podemos obter esta informação, ou seja, de que atributos do HTML podemos obter a informação.

2.1 Atributos HTML e respetivas Expressões Regulares

Nesta subsecção explicaremos sucintamente em que seções do HTML podemos obter as informações atrás referidas e as expressões regulares associadas às mesmas.

Antes de iniciarmos as explicações de cada expressão regular, convém notar que apenas usamos o contexto INITIAL para algo que explicaremos mais à frente e, como tal, o contexto mais utilizado ao longo do trabalho é *html*. Para além deste contexto, temos um específico para cada uma das secções. Como tal, definimos os seguintes contextos:

```
%x html sid susername scomBody sdatetime slike
```

Para evitar explicar o desnecessário nesta secção, cada contexto será usado na respetiva secção.
Para além dos conetxtos que exposemos anteriormente, criámos também as abreviaturas seguintes:

```
ada \<a[ ]+data\-action\=
ac \<a[ ]+class\=
lc \<li[ ]+class\=

spec_chars \xc3[\x80-\xbf]
letter [a-zA-Z]|{spec_chars}
name ({letter}+[, ]*)+
```

As primeiras três abreviaturas servem para simplificar a escrita da abertura de certos elementos HTML, uma vez que se repetiam várias vezes ao longo do projeto, e o segundo grupo consiste em todas as letras aceitáveis para nomes, utilizando a classe de caracteres especiais que o professor José João colocou no FAQ na plataforma BlackBoard, juntamente com a classe de todas as letras standart do ASCII. Estas duas depois juntam-se ao espaço e à vírgula para aceitar os diferentes tipos de nome de utilizador que puderão aparecer.

2.1.1 ID do Comentário

O identificador de um comentário é o primeiro atributo que retiraremos. Encontramos o ID em elementos de HTML como o seguinte:

```
<li class="post" id="post-4787074863">
```

O ID aparece noutras secções, mas não será necessário tomá-las em consideração. Em cada comentário distinto, esta secção apenas aparece uma vez. Como tal, é ideal para a identificação. Assim, temos as seguintes expressões regulares para capturar o ID:

```
<html>{lc}\"post\"
<sid>[0-9]+
```

Com a primeira expressão, quando for reconhecido algo com a estrutura

```
<li class="post"
```

passamos para o contexto próprio do ID, o *sid* e, uma vez neste, se for detetada uma sequência de números então significa que identificamos um ID.

2.1.2 Nome do comentador

Para identificar o nome do comentador, teremos de procurar um elemento de HTML com uma estrutura semelhante a esta:

```
<a data-action="profile" data-username="luisloure"
href="https://disqus.com/by/luisloure/" rel="noopener
noreferrer" target="_blank">Luis Loure</a>
```

Dentro desta *tag* de HTML, apenas precisamos de retirar o que está imediatamente a seguir ao elemento de abertura. Assim sendo, utilizamos as seguintes expressões:

```
<html>{ada}\"profile\"
<susername>\>{name}
<susername>\<\/a\>
```

Assim, uma vez que seja identificado algo do tipo

```
 <a data-action="profile"
```

mudamos de contexto para *susername*.

Neste, uma vez que identificarmos uma sequência de caracteres que estejam antecedidos por um <, sequência que já foi explicada em 2.1, identificamos o nome do comentador. Depois, a última sequência serve apenas para sair deste contexto e voltar ao *html*, uma vez que identifica uma tag de fecho do tipo *a*.

Algo que é de notar é que nem todos os comentários têm um nome de utilizador associado, alguns apenas têm o seu ID.

2.1.3 Data do comentário

Para identificar a data do comentário, é necessário encontrar algo com o estilo seguinte no HTML:

```
<a class="time-ago" data-role="relative-time"
href="https://sol.sapo.pt/artigo/685414/Titulo-Notici
a-#comment-4786068237" title="Thursday, February 6,
2020 4:49 PM">
```

Para este efeito, utilizamos as seguintes expressões:

```
<html>{ac}\"time\-ago\"
<sdatetime>title\=\"[^\"]+\"
```

Com a primeira expressão, enquanto estamos no contexto *html* se encontrarmos algo igual à string

```
<a class="time-ago"
```

Então entramos no contexto *sdatetime*. Uma vez neste contexto, podemos reparar que a data que nós queremos obter vem imediatamente a seguir a "title", como tal, a segunda expressão regular vai tomar partido disto, tomando o que vier logo a seguir, capturando qualquer caracter que não seja uma aspa até chegar ao fecho de aspas. Aqui para de capturar e voltará a mudar de contexto para *html*.

2.1.4 Timestamp do comentário

O Timestamp de um comentário corresponde a um inteiro respresentativo do número de milisegundos que passaram desde 1 de Janeiro de 1900 (ou 1970) até à data presente. Acabamos por usar 1900 por ser o que vários comentários no site *StackOverflow* aconselhavam com a biblioteca de C "time.h".

Portanto, para obtermos a timestamp, é necessário apenas utilizar a expressão que identificamos anteriormente para obter a data. Esta data depois será passada como argumento a uma função "setTimestamp"que, a partir da string devidamente formatada que lhe é dada, devolve o respetivo timestamp, sob a forma de uma string.

2.1.5 Texto do Comentário

O texto do comentário acabou por se revelar um pouco mais difícil de identificar que os outros atributos, inicialmente, devido à estrutura mais maleável que podia ter. No entanto, desenvolvemos as seguintes RegEx para identificar o corpo do comentário:

```
<html>\<div\>
<scomBody>\<\/\div\>
<scomBody>\<(\/?)([a-z]{1,2})(\/?)\>
<scomBody>[^\<]+
```

Será de notar que o corpo do comentário começa pela abertura de uma tag *div*, o que é identificado pela primeira expressão, sendo que associada a ela está a mudança de contexto para *scomBody*. Logicamente, para voltar para o contexto *html* fazê-mo-lo uma vez que encontremos uma tag *div* de fecho, que é o que temos na segunda expressão.

Dentro do corpo do comentário, vamos encontrar várias tags de formatação de texto, e estas podem ser ignoradas. No entanto, para elas não apareceram no yytext resultante, temos de as identificar e filtrar, e é para isto que serve a terceira expressão. Esta identifica tags de abertura e fecho, sendo que o corpo da tag pode ser constituído por uma ou duas letras.

Por fim, tudo o que nos sobrar entre as tags será parte do corpo do comentário, logo a última expressão aparece para capturar todo e qualquer caracter até encontrar um <.

2.1.6 Número de *likes*

Para identificar a parte do ficheiro HTML que representa os *likes* de um post, temos que encontrar algo com a estrutura seguinte:

```
<a class="vote-up count-0" data-...>
```

Aqui só nos interessa o número que vem imediatamente a seguir a "count". Isto pode ser feito apenas com uma expressão, mas para manter a lógica de mudanças de contexto que até agora tivemos, desenvolvemos as seguintes expressões:

```
<html>{ac}\"vote\-up
<slike>[0-9]+
```

Assim, quando encontramos a seguinte string:

```
<a class="vote-up
```

entramos no contexto *slike* e, uma vez neste contexto, se encontrarmos uma sequência de caracteres, encontramos o número de *likes*, expresso na segunda expressão. Uma vez que esta operação é concluída, voltamos ao contexto *html*.

2.1.7 Respostas ao Comentário

As respostas aos comentários, e atributos relacionados, têm as seguintes expressões associadas:

```
<html>\-role\=\"children\">
<html>Show\ more\ replies
```

A primeira expressão procura uma string do género

```
-role="children">
```

Esta string simboliza o início de uma secção do HTML onde irão aparecer os comentários de resposta a outro comentário. Noutra secção mais à frente explicaremos melhor como isto se sucederá.

A segunda expressão procura a string "Show more replies", que representa o final de um comentário, seja ele uma resposta ou não. Tal como a primeira expressão, explicaremos a importânica desta mais tarde.

2.1.8 Outras Expressões

Um ficheiro HTML, para apresentar uma formatação correta para um website como o do jornal *Sol*, precisa de uma quantidade bastante grande de informação. No entanto, muita dessa informação é desnecessária para o JSON que queremos produzir. Como tal, é necessário ter algumas expressões para além das explicadas anteriormente para filtrar toda a informação que, para este projeto, seria apenas ruído.

Desta forma, temos as seguintes expressões:

```
<*>[ \n\t\r]
<*>.
```

O que estas expressões fazem é simplesmente filtrar todas as mudanças de linha e espaços brancos não solicitados, e todos os caracteres que não fizeram match com nenhuma expressão anterior.

Ainda existe uma outra expressão:

```
<INITIAL>[0-9]+
```

Esta expressão serve para detetar um número que surge no início do documento, que corresponde ao total de comentários que o documento tem. A necessidade deste número será explicada mais à frente. Uma vez que este número é identificado, o contexto é mudado pela primeira vez para *html* e não volta a ser mudado para *INITIAL*.

3 Estruturas de Dados

Uma vez que a estrutura do documento é vasta e complexa, optamos por guardar a informação de cada comentário em estruturas de dados. Assim, no final estas poderiam ser impressas com a estrura de um documento JSON mais facilmente.

Para este efeito, criamos duas *structs*: *Comment* e *Stack*. Ambas as estruturas têm o seu corpo definido no ficheiro "aux.c"e apontador definido em "aux.h". Comecemos por explicar a *struct Comment*.

3.1 Comment

A estrutura *comment* servirá para guardar a informação de cada comentário. Assim sendo, tem a seguinte declaração:

```
struct comment{
    char* id;
    char* name;
    char* comBody;
    int   upvote;
    int   hasReplies;
    char* date;
    char* timestamp;
    int   numberOfReplies;
    struct comment** replies;
};
```

Como podemos ver, cada um dos atributos necessários para o JSON estão guardados nesta estrutura. Cada comentário guarda os seus componentes sob a forma de um apontador para *char* ou como um inteiro. Quanto às repostas a cada comentário, estas são, por sua vez, guardadas num *array* dinâmico da própria estrutura.

Associada a esta *struct* estão métodos de construtor vazio e parametrizado, adição de *replies* e de libertação de memória.

3.2 Stack

Ao longo da leitura do ficheiro, há certas informações que serão necessárias. Por exemplo, imaginemos que nos encontramos num comentário, como é que sabemos se esse comentário era uma repostas a outro comentário? Pois, como solução a este problema, desenvolvemos uma *stack*, que nos permite guardar todos os comentários no ficheiro original, bem como resolver o problema exposto.

A declaração da *struct Stack* é a seguinte:

```
struct Stack {
     int top;
     int capacity;
     int bottom;
     Comment* cms;
};
```

Nesta struct, a variável *cms* corresponde a um *array* dinâmico de *Comment*, ou seja, *struct comment**, onde serão guardados os comentários que não são resposta a um outro comentário. A variável *capacity* representa a capacidade máxima da stack, e serve de controlo. Esta variável é igual ao número que encontramos no início do documento HTML que descrevemos anteriormente. Com este número, garantimos que o número máximo de comentários da *stack* corresponde ao número real de comentários no documento. A variável *top* é o equivalente ao *stack pointer*, e o *bottom* é o *base pointer*. O *top* corresponde à posição do comentário mais recentemente adicionado à stack, e o *bottom* corresponde ao fundo mais recente da *stack*, ou seja, o último comentário completo (com todas as suas respostas).

4 Ações Semânticas

Para cada grupo de expressões que apontamos anteriormente existe um grupo correspondente de ações semânticas. Nesta secção explicaremos cada uma delas, associando-as às estruturas de dados que temos, para explicar exatamente o funcionamente do mesmo.

Como já falamos anteriormente de mudanças de contexto e onde elas ocorrem, ignoraremos as mesmas nesta secção.

Algo que temos de ter em conta é que, no início do programa, são declaradas várias variáveis:

```
char* id_curr = NULL;
char* name_curr = NULL;
char* date_curr = NULL;
char* time_curr = NULL;
char* combody_curr = NULL;
char* likes_curr = NULL;

Comment curr_comment;

stack stackCms;
int num_coments = -1;
```

Estas variáveis serão importantes para guardar a informação do comentário que se está a processar no momento, sendo que apenas é processado um de cada vez.

4.1 ID do Comentário

O ID de um comentário é o primeiro atributo que se encontra de cada comentário. Assim, quando se encontra o ID, guarda-se o mesmo, na variável *id_curr* , e inicializa-se a variável *combody_curr*.

```
combody_curr = strdup("");
id_curr = strdup(yytext);
```

4.2 Nome do comentador

Quando encontramos o nome do utilizador que fez o comentário, guardamos esse nome em *name_curr*. De notar que a string que é guardada em *yytext* começa com o abrir das aspas, logo guarda-se apenas o que vem a seguir à primeira posição.

```
name_curr = strdup(yytext+1);
```

4.3 Data e Timestamp do Comentário

Quando se encontra a string da data, também se encontra o necessário para o Timestamp. Como tal, as ações para os dois são feitas no mesmo bloco. Como foi até agora, estas ações correspondem a guardar o necessário na respetiva variável, neste caso, *date_curr* e *time_curr*.

A string guardada em *yytext* começa com "title=", logo, é necessário guardar apenas o que aparece a partir da sétima "casa"do *yytext*, que é o que é guardado em *date_curr*. Em *time_curr*, no entanto, vai ser guardado o output de *getTimestamp()*, uma função auxiliar que devolve o timestamp a que uma data com o devido formato corresponde (neste caso, o formato da data é o devolvido pelo HTML). A definição de *getTimestamp()* é a seguinte:

```
char* getTimestamp(char* date){
    struct tm temp = {0};
    char* target = (char*)malloc(sizeof(char*));
    char* token = strtok(date, " ");

    token = strtok(NULL, ", "); //mês

    //Estrutura switch para obter um inteiro a partir do mês

    token = strtok(NULL, ", ");//dia
    temp.tm_mday = atoi(token);
    token = strtok(NULL, ", ");//ano
    temp.tm_year = atoi(token)-1900;
    token = strtok(NULL, ":");//hora
    temp.tm_hour = atoi(token);
```

```
        token = strtok(NULL, " ");//minuto
        temp.tm_min = atoi(token);
        token = strtok(NULL, " \""); //AM or PM
        if(!strcmp(token, "PM")) temp.tm_hour = (temp.tm_hour+12)%24;
        sprintf(target, "%u", (unsigned)mktime(&temp));
        return target;
}
```

Como podemos ver, esta função toma partido das estruturas e métodos fornecidos pela biblioteca "time.h"para obter o Timestamp.

Assim, o corpo das ações semânticas associadas à data e Timestamp são as seguintes:

```
    date_curr = strdup(yytext+6);
    time_curr = getTimestamp(yytext+6);
```

4.4 Texto do Comentário

O texto do comentário é um pouco mais complicado. Uma vez que não se encontra só numa string mas sim em várias, porque o texto do comentário pode ter várias *tags* relacionadas com texto em itálico ou a negrito, é preciso capturar várias strings, concatená-las e só depois guardá-las no *array combody_curr*. Sendo assim:

```
        combody_curr = (char*)realloc(combody_curr, strlen(combody_curr) +
strlen(yytext) + 1);
        char tmp[strlen(combody_curr) + strlen(yytext) + 1];
        strcpy(tmp, combody_curr); strcat(tmp, yytext);
        combody_curr = strdup(tmp);
```

Como podemos ver, alocamos a memória necessária para o *combody_curr*, fazemos uma string temporária para guardar o corpo do comentário (isto é necessário devido à tipagem da função *strcat()*, importada de "string.h"), copiamos o que já tínhamos até agora da mensagem para esta string, concatenamos com o novo conteúdo e depois copiamos o conteúdo total para o *combody_curr* outra vez.

4.5 Número de *likes*

Guardar a informação relativa a um "like"funciona como o nome ou o ID. Uma vez que temos a string *yytext()* com o *like*, simplesmente copiámo-la para likes_curr.

```
    likes_curr=strdup(yytext);
```

4.6 Respostas ao Comentário

As respostas a um comentário são processadas como qualquer outro cometário, mas o que difere é a forma como usa a stack.

Quando se identifica o início de uma resposta ao comentário atual, inicializa-se a variável *curr_comment*, com todos os valores que foram recolhidos até ao momento. Aqui, verificamos se o *top* é igual ao *bottom* (com a função *isEmpty()* e, no caso de não ser, quer dizer que o comentário que estamos a ver é uma resposta, e adicionamos o comentário como resposta ao comentário no topo da *stack*, e depois fazemos *push* deste novo comentário na *stack*, incrementando o *top*. Depois, apagamos os valores nas variáveis *curr_comment* e em *name_curr* (isto para os casos em que o comentário não tem um nome, o que acontece, e, se isto não fosse feito, o próximo comentário ficaria com um valor inválido). Tudo isto é feito quando chegamos a algo do género -rol="children, como referimos anteriormente.

```
    curr_comment = newComment(id_curr, name_curr, combody_curr, atoi(likes_curr), date_curr,
time_curr);
    if(!(isEmpty(stackCms))) {addReply(peek(stackCms), curr_comment);
    push(stackCms, curr_comment);
    curr_comment = NULL;
    name_curr = NULL;
```

Quando encontramos "Show more replies", como está na segunda expressão que explicamos anteriormente, verificamos se o *top* e o *bottom* são iguais. No caso de serem, incrementamos o *bottom*, e fazemos *pop()* da *stack*, ficando o último novo comentário sem "pai"na *stack*, e os seus "filhos, as respostas, são apagadas. Desta forma, conseguimos manter a *stack* limpa dos comentários de resposta, mantendo apenas os que não são respostas, que, por sua vez, têm os comentários resposta guardados na variável *replies*.

```
        if(checkTop(stackCms) == checkBottom(stackCms)) incBottom(stackCms);
        pop(stackCms);
```

4.7 Outros

Para além das principais ações semânticas que o nosso filtro de texto tem que foram explanadas anteriormente, também existem outras.

Quando encontramos o número no contexto *INITIAL*, inicializamos a *stack*, colocando esse número como a capacidade da *stack*.

```
        num_coments = atoi(yytext);
        stackCms = createStack(num_coments);
        combody_curr = strdup("");
        name_curr = strdup("");
```

Por fim, quando se encontram outros caracteres não reconhecidos pelas outras regras ou um \t, \n, \r ou outra forma de *whitespace*, estes são ignorados, ou seja, a ação semântica é vazia.

5 Ficheiro JSON

Uma vez que o ficheiro HTML foi devidamente processado pela função *yylex()* e as estruturas estão preenchidas, precisamos de passar as mesmas para um ficheiro JSON para concluir o programa. Como tal, na função *yywrap()*, chamamos a função *parseJSON()*. A esta função será passada a *stack* e o devido *parsing* será efetuado.

Assim, apresentamos a definição de *parseJSON()*:

```
void parseJSON(stack st){
    FILE* fp = fopen("sol_comments.json", "w");
     if(fp != NULL){
         fprintf(fp, "\"commentThread\":[");
         for(int i = 0; i < st->bottom; i++){
         fprintf(fp, "\n\t{\n");
         parseJSONAux((st->cms)[i], fp, 1);
         if(!st->cms[i+1]) fprintf(fp, "\n\t}\n");
         else fprintf(fp, "\n\t},\n");
     }
     fprintf(fp, "]");
     fclose(fp);
    }
}
```

Para além desta, temos também a função *parseJSONAux()*, que processa os comentários individualmente. A sua definição:

```
void parseJSONAux(Comment c, FILE* fp, int depth){
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"id\":\"%s\",\n", c->id);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"user\":\"%s\",\n", c->name);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"date\":%s,\n", c->date);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"timestamp\":%s,\n", c->timestamp);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"commentText\":\"%s\",\n", c->comBody);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"likes\":%d,\n", c->upvote);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
```

```
    if(c->hasReplies == 0) fprintf(fp, "\"hasReplies\":TRUE,\n");
    else fprintf(fp, "\"hasReplies\":FALSE,\n");
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"numberOfReplies\":%d,\n", c->numberOfReplies);
    for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "\"replies\":[", c->id);
    for(int i = 0; i < c->numberOfReplies; i++){
            fprintf(fp, "\n");
            for(int i = 0; i < depth+2; i++) fprintf(fp, "\t");
            fprintf(fp, "{\n");
            parseJSONAux((c->replies)[i], fp, depth+1);
            fprintf(fp, "\n");
            for(int i = 0; i < depth+2; i++) fprintf(fp, "\t");
            if(i == (c->numberOfReplies)-1) fprintf(fp, "}\n");
            else fprintf(fp, "},\n");
    }
    if(c->numberOfReplies != 0) for(int i = 0; i < depth+1; i++) fprintf(fp, "\t");
    fprintf(fp, "]");
}
```

A variável *depth* que é passada à função auxiliar serve para sabermos o nível de profundidade de indentação em que estamos. Um comentário que não é resposta a nada está ao nível 0, uma resposta a esse comentário está no nível 1, etc. Isto é necessário para sabermos o número de *tabs* em cada linha, e é por isto que, antes de qualquer impressão no documento, fazemos um ciclo *for* apenas para imprimir "\t".

Como podemos ver, a função auxiliar é uma função recursiva, que apenas para quando chega a um comentário sem respostas.

Invariavelmente, o ficheiro *output* do programa será o ficheiro "sol_comments.json", que é criado durante a execução.

A título de demonstração, colocamos aqui um excerto do documento JSON produzido pelo programa, para os primeiros comentários do ficheiro "Sol4.html".

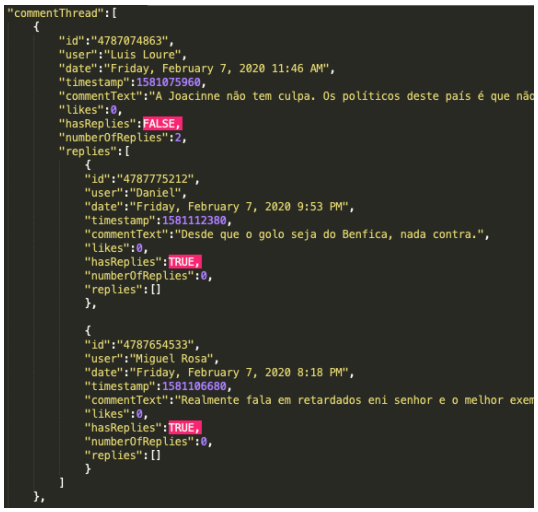

Figura 2: Excerto de "sol_comments"

6 Makefile

O ficheiro *Makefile* associado ao projeto é bastante simples e serve para podermos correr todo o programa com apenas um comando. Como tal, o conteúdo do ficheiro é o seguinte:

```
run: json.exe Sol4.html
	cat Sol4.html | ./json.exe

json.exe: lex.yy.c aux.o
	gcc -o json.exe lex.yy.c aux.o -ll

aux.o: aux.c aux.h
	gcc -c aux.c aux.h

lex.yy.c: Sol2NetLang.l
	flex Sol2NetLang.l

Sol4.html:
	wget https://natura.di.uminho.pt/~jj/pl-20/TP1/Sol4.html
clean:
	rm *.o
	rm *.html
	rm json.exe
	rm sol_comments.json
	rm *.gch
```

Figura 3: Makefile

Com esta *Makefile*, podemos correr o programa apenas com o comando "make run". Na eventualidade do ficheiro "Sol4.html"não existir, é efetuado o download do mesmo a partir do link fornecido no enunciado utilizando o comando *wget*, de um pacote de utilidades UNIX. Se quisermos eliminar os ficheiros que foram criados pela execução, podemos utilizar o comando "make clean".

7 Conclusão

Com este trabalho pudemos aprender a utilizar melhor a ferramenta FLex, que utilizamos nas aulas de Processamento de Linguagens, e certas ferramentas do C. Para além disto, também nos permitiu aprender um pouco sobre a estrutura de um ficheiro HTML e de um ficheiro JSON, o que nos vai ser bastante útil no futuro.

Tivemos algumas dificuldades ao longo do projeto, sendo que a maior esteve relacionada com a memória e com as estruturas de dados, o que nos obrigou a alterar a nossa aproximação às mesmas várias vezes.

No entanto, acreditamos que, no final, conseguimos realizar o que nos foi proposto: um Filtro de Texto, utilizando a ferramenta FLex, para passar de HTML para JSON. Como tal, consideramos que o nosso trabalho foi, ao todo, um sucesso.