



UNIVERSIDADE DO MINHO

PROCESSAMENTO DE LINGUAGENS (3º ANO DE CURSO)

---

## Trabalho Prático 1

---

### RELATÓRIO DE DESENVOLVIMENTO

---

Mestrado Integrado em Engenharia Informática

***Realizado por G09:***

Hugo André Coelho Cardoso, a85006

João da Cunha e Costa, a84775

Válter Ferreira Picas Carvalho, a84464

2 de Abril de 2020

## Resumo

O primeiro trabalho prático realizado por este grupo no âmbito da unidade curricular *Processamento de Linguagens* consistiu na resolução do enunciado **Template Multi-File**, que se traduziu no desenvolvimento de um programa à base de *Flex* que gera o esqueleto de um projeto a partir de um *template*.

No presente relatório será explicada a forma como o grupo interpretou o problema e envisionsou a solução, bem como as adversidades que surgiram durante a sua implementação, de maneira a construir um programa escalável capaz de processar não apenas o exemplo dado no enunciado, como também *templates* bem mais complexos, de maneira a proporcionar mais liberdade ao utilizador final da aplicação.

Os principais objetivos deste trabalho prático passaram por aumentar a experiência de uso do ambiente Linux, bem como a capacidade de escrever *expressões regulares (ER)* para descrição de *padrões de frases*. Para além disso, serviu também para desenvolver, a partir de ERs, sistemática e automaticamente *Processadores de Linguagens Regulares*, que filtrem ou transformem textos com base no conceito de regras de produção *Condição-Ação*, usando o *Flex* para gerar *filtros de texto* em C.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
1.1	Enquadramento e Contextualização . . . . .	3
1.2	Problema e Objetivo . . . . .	3
1.3	Resultados ou Contributos . . . . .	3
1.4	Estrutura do Relatório . . . . .	4
<b>2</b>	<b>Análise e Especificação</b>	<b>5</b>
2.1	Descrição informal do problema . . . . .	5
2.2	Especificação do Requisitos . . . . .	5
2.2.1	Dados . . . . .	5
2.2.2	Pedidos . . . . .	5
2.2.3	Relações . . . . .	5
<b>3</b>	<b>Conceção/desenho da Resolução</b>	<b>7</b>
3.1	Estruturas de Dados . . . . .	7
3.1.1	Metadados . . . . .	7
3.1.2	Ficheiros . . . . .	7
3.1.3	Diretorias . . . . .	7
3.2	Algoritmos . . . . .	8
3.2.1	Recolha de metadados . . . . .	9
3.2.2	Construção da árvore . . . . .	10
3.2.3	Escrita nos ficheiros . . . . .	12
3.3	<i>Makefile</i> . . . . .	12
<b>4</b>	<b>Codificação e Testes</b>	<b>13</b>
4.1	Alternativas, Decisões e Problemas de Implementação . . . . .	13
4.1.1	Metadados . . . . .	13
4.1.2	Ficheiros . . . . .	13
4.1.3	Diretorias . . . . .	13
4.2	Testes realizados e Resultados . . . . .	14
<b>5</b>	<b>Conclusão</b>	<b>22</b>
<b>A</b>	<b>Código do Programa</b>	<b>23</b>
<b>B</b>	<b><i>Template</i> do enunciado</b>	<b>29</b>

## Lista de excertos de código

1	Estrutura do mapa criado (map.c). . . . .	7
2	Declaração do mapa de metadados. . . . .	7
3	Declaração do mapa de metadados. . . . .	7
4	Declaração do <i>array</i> de diretorias. . . . .	8
5	Filtragem das secções do ficheiro. . . . .	9
6	Processamento de metadados. . . . .	9
7	Processamento de informação irrelevante. . . . .	9
8	Processamento dos nomes de elementos da árvore (1/3). . . . .	10
9	Processamento dos nomes de elementos da árvore (2/3). . . . .	11
10	Processamento dos nomes de elementos da árvore (3/3). . . . .	11
11	Preenchimento dos ficheiros. . . . .	12
12	Declaração de strings para guardar metadados. . . . .	13
13	Declaração dos apontadores para os ficheiros. . . . .	13
14	Código do ficheiro filter.l. . . . .	23
15	Código do ficheiro map.c. . . . .	26
16	Código do ficheiro map.h. . . . .	28
17	Código da <i>Makefile</i> . . . . .	28

## Lista de Figuras

1	Árvore gerada a partir do template dos docentes. . . . .	14
2	Ficheiros gerados a partir do template dos docentes. . . . .	14
3	Árvore gerada a partir do segundo template. . . . .	16
4	Ficheiros gerados a partir do segundo template. . . . .	16
5	Árvore gerada a partir do terceiro template. . . . .	18
6	Ficheiros gerados a partir do terceiro template. . . . .	18
7	Árvore gerada a partir do quarto template. . . . .	20
8	Ficheiros gerados a partir do quarto template. . . . .	21

# 1 Introdução

*Área:* Processamento de Linguagens

No âmbito da unidade curricular *Processamento de Linguagens*, inserida no plano de estudos do Mestrado Integrado em Engenharia Informática da Universidade do Minho, foi proposta a realização de um trabalho prático focado na elaboração de filtros de texto, com recurso à ferramenta *Flex*. Para tal efeito, cada grupo de alunos foi encarregado de escolher e resolver um dos seis enunciados elaborados pelo corpo docente.

Neste relatório será abordado, em detalhe, todo o processo criativo por detrás da resolução do problema proposto no enunciado **Template Multi-file**, que o grupo decidiu resolver após a leitura e extensiva ponderação sobre todos os trabalhos propostos.

## 1.1 Enquadramento e Contextualização

Para vários projetos de software, é habitual a solução final envolver vários ficheiros e pastas. Exemplo: um ficheiro, uma *makefile*, um manual, uma pasta de exemplos, etc.

Este projeto consiste em criar um programa **mkfromtemplate** que gira o esqueleto do programa (ficheiros e pastas iniciais) através do seu nome pretendido e de um ficheiro descrição de um *template multi-file*. O *template* deve incluir:

- metadados (p.e. autor e email) a substituir nos elementos seguintes;
- *tree* (estrutura de diretorias e ficheiros a criar
- *template* de cada ficheiro.

## 1.2 Problema e Objetivo

Como foi referido acima, o *template multi-file* está dividido em três secções, pelo que é lógico que a sua resolução apresente um número equivalente de fases.

Numa etapa inicial, realiza-se a recolha e armazenamento dos metadados fornecidos, incluindo o nome do projeto que é passado como argumento ao programa, ou seja, fora do *template*. Ao longo das outras secções do ficheiro são usadas referências a estes metadados sob a forma `{%metadado%}` (p.e. `{%name%}` para referir o nome do projeto), que deverão ser substituídas pelos elementos respetivos.

De seguida, é necessário gerar a *tree* descrita, criando todas as pastas e ficheiros nas respetivas localizações. O *template* usa uma estratégia baseada em hífen para indicar a profundidade a que se encontra cada elemento da árvore.

Por fim, resta preencher cada ficheiro com a informação que lhe diz respeito, substituindo, se necessário, as tais referências a metadados. Deve haver um *template* para cada ficheiro, pois não se espera que um ficheiro de tipo *flex* tenha a mesma informação que um ficheiro de documentação.

## 1.3 Resultados ou Contributos

O grupo procurou ir além dos requisitos presentes no *template* fornecido, com o objetivo de elaborar uma solução mais abrangente e complexa. Desta maneira, o resultado obtido permite, entre outros:

- a explicitação de um número arbitrário de metadados, tantos quantos o utilizador pretenda;

- a elaboração de árvores mais complexas e ricas;
- a criação de ficheiros de qualquer tipo de extensão;
- a criação de pastas até um certo nível de profundidade;
- a substituição de referências a metadados pelos mesmos nos próprios nomes de ficheiros e pastas;
- a filtragem de certos erros e irregularidades na formatação do *template* fornecido ao programa.

## 1.4 Estrutura do Relatório

O presente relatório está dividido em cinco capítulos, sendo este o primeiro, cujo propósito é introduzir o projeto, contextualizando-o e fazendo uma breve análise do problema e objetivos, além de tocar levemente nos resultados obtidos.

No [capítulo 2](#) realiza-se a análise e especificação do projeto, descrevendo o problema de maneira mais pormenorizada e identificando os seus requisitos, distinguindo dados, pedidos e relações.

No [capítulo 3](#) é abordada em detalhe a conceção/desenho da resolução, explicando as estruturas de dados usadas e os motivos pelos quais foram escolhidas, os algoritmos elaborados e a função de todos os ficheiros que foram criados.

No [capítulo 4](#) expõem-se as alternativas e problemas de implementação que surgiram, justificando as decisões tomadas, e percorre-se os testes realizados ao programa de maneira a testar a sua operabilidade, bem como os resultados produzidos a partir dos mesmos.

Finalmente, no [capítulo 5](#), termina-se o relatório com uma síntese do que foi dito, as conclusões e o trabalho futuro.

No apêndice do relatório estão ainda disponíveis para consulta o [código](#) desenvolvido e o [template](#) do enunciado, que é referenciado várias vezes.

## 2 Análise e Especificação

### 2.1 Descrição informal do problema

Dado um ficheiro seccionado e com informação por formatar (como é o caso das etiquetas {`%metadado%`}), é necessário, em primeiro lugar, saber identificar onde começa e acaba cada secção, de maneira a não misturar os métodos usados em cada uma e incorrer em erros.

Na primeira secção, é necessário extrair o nome e respetivo valor de cada metadado e guardar esta informação numa estrutura de fácil acesso, não esquecendo o nome do projeto, que é passado ao programa de maneira diferente.

Na constituição da árvore, é fundamental saber identificar o nome de cada elemento, distinguindo se o texto filtrado é o nome final ou uma etiqueta, bem como a extensão dos ficheiros (caso a possuam), de maneira a gerar o ficheiro. É também essencial conseguir precisar a localização de cada elemento, interpretando a ordem da árvore e o sistema de hífen usados.

Por fim, é indispensável a correta associação entre cada ficheiro e o respetivo conteúdo, de maneira a não haver extravio de informação, bem como a filtragem de eventuais etiquetas de metadados.

### 2.2 Especificação do Requisitos

#### 2.2.1 Dados

Os dados do problema constituem o **conhecimento *a priori*** a partir do qual se elabora a solução e, no que toca a este enunciado, correspondem ao **texto de entrada** que é necessário filtrar e às suas características.

Como tal, pode-se afirmar que há três tipos de dados neste problema:

- Os **metadados**, fornecidos na primeira secção do *template*, que funcionam como um par <chave,valor>, sendo necessário identificar o tipo de metadado (p.e. email) e, de seguida, o seu valor (p.e. jj@di.uminho.pt);
- A **árvore** a ser gerada, especificada na segunda secção do texto de entrada, ou seja, os **ficheiros e pastas** que a constituem. Será necessário saber identificar a **rota relativa** de cada elemento da árvore, a fim de o criar, bem como aceder aos ficheiros mais tarde para os preencher com informação;
- O **template de cada ficheiro**, isto é, a informação a escrever nos ficheiros depois de os criar. Estes dados são os únicos que não precisarão de ser armazenados, visto que podem ser transcritos diretamente do texto de entrada para os ficheiros.

#### 2.2.2 Pedidos

Os pedidos correspondem aos **requisitos estabelecidos** no enunciado e os **resultados pretendidos** para o trabalho.

Neste caso, é esperado que a partir do *template* introduzido seja criada uma diretoria com todos os ficheiros e subdiretorias especificadas, e que os ficheiros sejam todos preenchidos com os respetivos templates fornecidos, de maneira a ficar com o esqueleto operacional de um novo projeto no final.

#### 2.2.3 Relações

As relações traduzem-se nas **ligações existentes entre os dados e os pedidos**, ou seja, de que maneira é que obtemos estes últimos partindo dos primeiros. No contexto deste enunciado, são os **mapeamentos ou transformações necessárias** para produzir a saída:



- Qualquer etiqueta do formato `{%...%}` que apareça no texto de entrada deva ser substituída pelo **valor do metadado** que lhe diz respeito;
- O **número de hífen**s antes do nome de um elemento da árvore deve corresponder à **profundidade** a que o mesmo se encontra na árvore.



### 3 Conceção/desenho da Resolução

O desenvolvimento da solução foi um processo iterativo, ao longo do qual o grupo mudou várias vezes de estratégia, implementando estruturas de dados e algoritmos cada vez mais eficientes. Neste capítulo, será abordada apenas a solução final, deixando a evolução do projeto para o subcapítulo 4.1, onde serão explicadas as limitações das implementações iniciais e a culminação no produto final.

#### 3.1 Estruturas de Dados

Há **três** tipos de dados que é necessário armazenar: os **metadados**, os **nomes/apontadores para os ficheiros** e os **nomes das pastas** criadas (para resolver as rotas relativas dos elementos das árvores). A sua abordagem vai ser explanada separadamente.

##### 3.1.1 Metadados

Com o objetivo de **escalar** a solução, de maneira a ser possível **filtrar mais metadados**, o grupo desenvolveu um módulo onde criou uma estrutura análoga a um **mapa <chave,valor>** e elaborou funções para realizar várias operações sobre a mesma, nomeadamente inserção e consulta.

```
1 typedef struct mapitem {
2     char* key; //chave - p.e. "author"
3     void* val; //valor - p.e. "J.Joao"
4     int type; //sinalizador para gestao posterior de memoria
5 } MI;
6
7 typedef struct map {
8     int size; //numero de pares
9     MI* items; //pares
10 } M;
```

Listing 1: Estrutura do mapa criado (map.c).

A solução final guarda os nomes de todos os metadados e respetivos valores nesta estrutura, eliminando, desta maneira, a restrição previamente existente.

```
1 struct map* meta; //mapa de metadados com par (nome,valor)
```

Listing 2: Declaração do mapa de metadados.

##### 3.1.2 Ficheiros

A gestão dos ficheiros é um processo bastante semelhante ao dos metadados. Consistiu na criação de um **mapa** para guardar os **pares <nome,apontador>** dos ficheiros. Desta maneira, torna-se possível criar e aceder a um **número arbitrário de ficheiros**, o que proporciona um **nível muito superior de liberdade** no planeamento das árvores de projetos.

```
1 struct map* files; //mapa de ficheiros com par <nome,apontador>
```

Listing 3: Declaração do mapa de metadados.

##### 3.1.3 Diretorias

Atentemos o seguinte exemplo:

```
=== tree

projeto/
- doc/
-- projeto.fl
- exemplo/
-- projeto.md
- Makefile
```

Na figura anterior, os ficheiros têm as seguintes diretorias:

- projeto.fl: projeto/doc/projeto.fl
- projeto.md: projeto/exemplo/projeto.md
- Makefile: projeto/Makefile

Através deste exemplo, é possível retirar **duas informações fulcrais** para esta parte do trabalho:

- A *Makefile* encontra-se na pasta "projeto", embora tenham aparecido outras na *tree* **depois** dessa. Logo, **não basta** usar os nomes de **todas as pastas antes de um ficheiro** para **obter a sua diretoria**. É preciso verificar a que **profundidade** ele se encontra.
- As **rotas** até aos ficheiros "projeto" são **diferentes**, embora estes se encontrem à **mesma profundidade**. Logo, é necessário ter em conta a **pasta mais recente** que foi criada em cada nível.

Com base nisto, o grupo implementou um *array* **estático** de *strings*, com **cinco posições**, para armazenar o nome da pasta **mais recentemente criada** a cada nível de profundidade, de maneira a ser possível "construir" as diretorias à medida que se percorre a árvore.

```
1 #define maxDir 5 //profundidade maxima
2 char *dir[maxDir];
```

Listing 4: Declaração do *array* de diretorias.

## 3.2 Algoritmos

A interpretação do *template* fornecido é realizada de modo **sequencial**, sendo este aberto para leitura e percorrido do início ao fim, processo durante o qual é realizada a **filtragem** pelo programa *Flex*. O ficheiro encontra-se **seccionado por divisórias** que seguem o seguinte formato:

```
=== meta
=== tree
=== <nome de ficheiro/pasta> (p.e. === {%name%}.fl)
```

Como tal, o programa guia-se por estas divisórias. Sempre que encontra uma, analisa a sua **palavra chave** e invoca o algoritmo correspondente para processar a parte seguinte do ficheiro. Desempenha esta função recorrendo a **start conditions**:

```
1 <*>==[ ]*      { if (writeTo) { free(writeTo); writeTo = NULL; }  
2                  } BEGIN DIV;  
3  
4 <DIV>meta        BEGIN META;  
5 <DIV>tree        BEGIN NAME;  
6 <DIV>"{%".+"%}"*. {writeTo = getMetaFilename(yytext);} BEGIN NEWLINES;  
7 <DIV>.+          { writeTo = strdup(yytext); } BEGIN NEWLINES;
```

Listing 5: Filtragem das secções do ficheiro.

Este código será explicado em maior detalhe nos próximos subcapítulos. Para já, é importante observar que há **três** condições diferentes passíveis de serem despoletadas:

- **META**: recolha de metadados.
- **NAME**: construção da árvore.
- **NEWLINES**: escrita para os ficheiros.

De seguida, serão explicados os algoritmos implementados em cada uma destas.

### 3.2.1 Recolha de metadados

Na primeira secção do ficheiro dá-se a **filtragem e armazenamento dos metadados**, cujo formato no *template* é **<nome do metadado>: <valor do metadado>**. Logo, a sua filtragem é bastante simples:

```
1 <META>^.+:[ ]*   { newMeta = strdup(strtok(yytext,":"));  
2                  } BEGIN SAVEMETA;  
3  
4 <SAVEMETA>.+     { mapDynAdd(newMeta,strdup(yytext),meta);  
5                  free(newMeta); } BEGIN META;
```

Listing 6: Processamento de metadados.

Como podemos observar na primeira condição, o programa começa por processar o nome do metadado, bem como o ":" e os espaços que o separam do texto que vem a seguir. O grupo programou a filtragem de um **número arbitrário de espaços** de maneira a salvaguardar eventuais irregularidades e tornar o algoritmo mais **versátil**. De seguida, extrai-se o **nome do metadado** do texto retido e guarda-se-lo numa **variável temporária** *newMeta*.

Na outra condição, que é ativada no fim da primeira, o programa apanha todo o restante texto na linha que, em teoria, é o **valor do metadado** e **adiciona o par ao mapa**, após o qual liberta a memória alocada para o nome. No fim, **desencadeia de novo a primeira condição**, originando, desta maneira, um **ciclo** que processará **todos** os metadados e só parará quando o programa encontrar a próxima divisória.

**NOTA:** O programa possui uma condição no final que **descarta** todo o texto que **não é de interesse**, isto é, que não é filtrado nas restantes condições. É esta parte do sistema que lida com os parágrafos entre as linhas de metadados e, eventualmente, a linha [neste template](#) acerca do nome do projeto.

```
1 <*>.\|\n      ;
```

Listing 7: Processamento de informação irrelevante.

### 3.2.2 Construção da árvore

Para esta secção do projeto, o grupo contemplou as várias possibilidades para nomes de ficheiros e pastas e considerou que:

- **Faz sentido** aceitar os seguintes formatos:
  - ficheiros sem extensão (p.e. Makefile);
  - ficheiros sem etiqueta de metadado com extensão (p.e. log.txt);
  - ficheiros com etiqueta de metadado com extensão (p.e. {%name%}.fl);
  - pastas sem etiqueta de metadado (p.e. doc/ e exemplo/);
  - pastas com etiqueta de metadado apenas (p.e. {%name%}/ ou {%author%}/, nomeadamente a *root*).
- Os seguintes formatos **não fazem sentido**:
  - nomes com várias etiquetas de metadados (p.e. {%name%}-{%author%}.md);
  - nomes com etiqueta de metadado e mais texto que não faça parte da extensão (p.e. 2-{%name%}.md).

Assim, foi implementado um algoritmo que abrange os formatos considerados e, por coincidência, outros que não estavam planeados no desenho da solução, nomeadamente ficheiros apenas com uma etiqueta de metadado no nome, sem extensão, e nomes com texto entre a etiqueta de metadado e a extensão (p.e. {%email%}.1.2.3.4.txt).

O algoritmo desenvolvido para este efeito vai ser explicado de seguida. Uma vez encontrada a divisória da árvore no ficheiro, o programa salta para a condição **NAME**.

```
1 <NAME>"{%".+"%}"[^\\n\\/]* { name = getMetaFilename(yytext);  
2  
3                               if (strchr(name, '/')) BEGIN HYPHENS;  
4                               else BEGIN EXTENSION; }  
5  
6 <NAME>[^\\n\\/]+ { name = strdup(yytext); } BEGIN EXTENSION;
```

Listing 8: Processamento dos nomes de elementos da árvore (1/3).

Como é possível observar no excerto supracitado, esta condição filtra o **nome** de um elemento da árvore, seja ele um **ficheiro** ou uma **pasta**. De seguida, são explicadas as **duas** ações possíveis que são tomadas pela ordem em que aparecem no excerto exibido:

1. Na linha em filtragem consta uma **etiqueta de metadado**, pelo que o programa procura encontrar a mesma e o que possa aparecer a seguir - nomeadamente uma **extensão** de ficheiro. De seguida, invoca a função *getMetaFilename* que extrai o **nome do metadado** do texto filtrado, obtém o **valor correspondente** e devolve uma *string* com o **nome final do ficheiro** (incluindo a possível extensão). Esta função pode ser consultada no [anexo A](#).
2. Caso contrário, o elemento da árvore tem um nome "normal" que não precisa de ser alterado, pelo que se filtra e guarda o mesmo.

Durante este processo, é tomada a precaução de se verificar se o **nome** do elemento da árvore **não possui** o caractere **" / "**, tanto no texto filtrado como eventualmente no conteúdo do metadado. Isto deve-se ao facto de pastas e ficheiros com esse caractere no nome **não serem suportados em ambiente Linux**, pelo que se for apanhado um nome **inválido**, a linha da árvore é **ignorada** e avança-se para a próxima.

O nome é guardado numa **variável temporária** em memória e o programa salta para a condição **EXTENSION**, onde acaba de processar a linha.

```

1 <EXTENSION>\/*. * { if (strlen(yytext) == 1) {
2
3         for (int i = hyphens; i < maxDir; i++) {
4             if (dir[i]) {
5                 free(dir[i]);
6                 dir[i] = NULL;
7             }
8         }
9         dir[hyphens] = strdup(name);
10
11         char* route = buildRoute();
12         mkdir(route, 0700);
13
14         free(route); }
15     } BEGIN HYPHENS;
16 <EXTENSION>\n { char* route = buildRoute();
17
18         FILE *fd = fopen(route, "w");
19         mapDynAdd(name, fd, files);
20
21         free(route); } BEGIN HYPHENS;

```

Listing 9: Processamento dos nomes de elementos da árvore (2/3).

Nesta condição, o programa verifica se se trata de um **ficheiro** ou de uma **pasta**, averiguando se a linha **termina** com um **"/** (pasta) ou não (ficheiro). A primeira expressão regular processa **pastas**, procurando detetar esse caractere, bem como qualquer texto que eventualmente o possa seguir.

Se **existir** texto depois do **"/**, significa que esse linha tem um nome **inválido** em sistemas Linux, pelo que é **descartado**. Caso contrário, o programa **atualiza a estrutura** exposta em 4 com os novos dados, libertando, se for o caso, o nome da pasta anterior à mesma profundidade e suas eventuais subdiretorias. De seguida, recorre a esta mesma estrutura para **construir a rota relativa** da nova pasta, na função **buildRoute**, a partir da qual a gera.

Se se tratar de um **ficheiro**, todo o seu nome **já terá sido filtrado** na condição anterior, pelo que só deve sobrar o **parágrafo** na linha (segunda expressão regular). Neste caso, o programa resolve também a rota relativa do ficheiro, com recurso à estrutura supramencionada, **criando e abrindo-o em modo de escrita**. No fim, **guarda o nome e apontador no mapa** de ficheiros.

Uma vez processada a linha na íntegra, o programa salta para a condição **HYPHENS**:

```

1 <HYPHENS>-+ [ ] * { hyphens = 0;
2                     while (yytext[hyphens] == '-') hyphens++;
3
4                     if (name) { free(name); name = NULL; }
5                     if (hyphens <= maxDir && dir[hyphens-1])
6                         BEGIN NAME;
7                     }

```

Listing 10: Processamento dos nomes de elementos da árvore (3/3).

Nesta condição, filtra-se a parte **inicial** da linha, contando o **número de hífen**s, que indicam a **profundidade** na árvore. Note-se que quando é detetada a divisória da árvore, o programa salta para a condição **NAME** em vez desta, visto que a **primeira linha** da árvore é a **root**, que se encontra a uma **profundidade 0**, pelo que **não tem hífen**s antes do nome.

Esta expressão regular realiza ainda um **processamento de espaços** análogo ao que foi explicado em 6. De seguida, o programa liberta a memória alocada para o nome na linha anterior e verifica se o elemento da linha atual está a uma **profundidade válida**, passando, nesse caso, ao tratamento do nome.

### 3.2.3 Escrita nos ficheiros

Por fim, uma vez gerada a árvore, resta **preencher** cada ficheiro de acordo com o respetivo *template*. Como tal, quando o programa encontra uma divisória com o **nome de um ficheiro**, faz a correspondência com o apontador, guardando-o numa variável temporária e dá início ao processo de escrita, como é mostrado em 5.

```
1 <NEWLINES>\n+      BEGIN TOFILE;
2
3 <TOFILE>"{%\"([^\%\\}]|\"[^\%\\}]\"|\%[^\%\\}])\"+%}" { fprintf(mapGet(writeTo,
4     files), \"%s\", (char*) mapGet(getMetaKey(yytext), meta)); }
5 <TOFILE>.\|\\n      { fprintf(mapGet(writeTo, files), \"%s\", yytext); }
```

Listing 11: Preenchimento dos ficheiros.

Em primeiro lugar, o programa passa por uma condição de **filtragem de parágrafos**, de maneira a que os ficheiros **não fiquem com linhas vazias no início**.

De seguida, dá-se começo ao processo de escrita. O **único** texto que **não se transcreve diretamente para o ficheiro** (última expressão regular) são as **etiquetas de metadados**, caso no qual o programa vai buscar o respetivo valor ao mapa e o imprime (primeira expressão regular da condição **TOFILE**).

## 3.3 Makefile

De maneira a simplificar a compilação do código e facilitar o papel do utilizador, o grupo encarregou-se de desenvolver uma *Makefile* que **compila o filtro Flex e o código C resultante**. A compilação do código envolve também a **inclusão** dos ficheiros de código C onde foi criada a estrutura dos mapas, que é necessária à correta execução do programa.

Deste modo, para executar o programa basta compilar o código com o comando *make* no terminal e correr o **executável** criado, fornecendo-lhe o nome pretendido e o *template* do projeto como argumentos, por exemplo:

```
./mkfromtemplate Exemplo templateExemplo
```

A *Makefile* dispõe também da opção **clean**, que limpa o ficheiro de código C e o executável gerados.

**NOTA:** É possível haver problemas com a compilação do código C gerado devido à não-inclusão de uma *flag* referente à biblioteca de *Flex*, que pode ser necessária em algumas máquinas. A biblioteca a importar depende do sistema em que está a ser corrido o programa e, caso surja este entrave, o utilizador deve adicionar a respetiva *flag* a este ficheiro.

## 4 Codificação e Testes

### 4.1 Alternativas, Decisões e Problemas de Implementação

Como já foi referido no capítulo anterior, a resolução do problema foi um processo iterativo, pelo que é natural que alguns aspetos da implementação inicial não tenham chegado ao produto final. De seguida, serão expostas algumas estruturas de dados alternativas pelas quais não se optou e problemas que surgiram, procurando expôr as suas limitações.

#### 4.1.1 Metadados

O **programa inicial** funcionava sob o pressuposto de que a secção de metadados do *template* possuía sempre **dois dados apenas: autor e email**. Esta implementação baseava-se no exemplo fornecido pelo corpo docente, que pode ser observado no [anexo B](#) deste relatório.

Dado que se assumia um número fixo de metadados, usava-se **apenas três strings** (tantas quantas os dados) **alocadas dinamicamente em memória** (conforme o comprimento):

```
1 char *name;  
2 char *email;  
3 char *author;
```

Listing 12: Declaração de strings para guardar metadados.

Esta estratégia foi abandonada posteriormente, de maneira a construir um programa **flexível com o número de metadados**.

#### 4.1.2 Ficheiros

O tratamento dos ficheiros teve uma evolução análoga à dos metadados. Numa fase inicial, o programa contava apenas com a existência de **três ficheiros**, conforme vinha no *template* fornecido.

Com esta abordagem, era **desnecessário** guardar o nome dos ficheiros. Bastava criá-los ao ler a *tree* e **guardar os apontadores** para os mesmos, fazendo mais tarde a correspondência ao filtrar o seu conteúdo.

```
1 FILE *MK; //Makefile  
2 FILE *MD; //{%name%}.md  
3 FILE *FL; //{%name%}.fl
```

Listing 13: Declaração dos apontadores para os ficheiros.

Esta estratégia bastante rudimentar apenas funcionava porque **se sabia a priori que ficheiros havia**. Como tal, não era escalável, pelo que serviu apenas de passo intermédio no desenvolvimento da solução, permitindo a **ambientação e experimentação** com a **génese de pastas e ficheiros em ambiente Flex**.

#### 4.1.3 Diretorias

Nesta etapa, surgiu também um problema semelhante ao dos outros dados: a possibilidade de haver um número **arbitrário** de pastas dentro de pastas, o que **impossibilita** o conhecimento *a priori* da **profundidade máxima** de uma árvore.

De facto, foi ponderada a implementação de uma estrutura mais elaborada, de maneira a **escalar** esta parte do projeto, mas o grupo decidiu **não avançar** com esta decisão por **dois** motivos:

1. Ao contrário dos outros dados, em que a chave da correspondência é uma *string*, neste caso é um **inteiro** - o **número de hífen** - que corresponde ao nome da pasta mais recente.

Graças a isto, os *arrays* provam ser uma estrutura bastante adequada para guardar este tipo de informação, dado que os nomes das pastas são armazenados nas **posições do array** que representam a sua **profundidade**. Assim, é **desnecessário** programar uma estrutura manualmente para **guardar os dois atributos num par**.

- De um ponto de vista **lógico**, é pouco provável que o esqueleto inicial de um projeto tenha pastas a uma **grande profundidade**, pelo que até a utilização de um *array dinâmico em memória* seria **contraproducente**, visto não compensar os custos de manutenção associados.

Após uma troca de correspondência eletrónica com o supervisor José João Almeida, o grupo ficou seguro de que o docente não se opunha ao estabelecimento de uma profundidade máxima. Como tal, estabeleceu-se um **limite máximo de 5** (cinco) **hífens**.

## 4.2 Testes realizados e Resultados

Foram elaborados vários *templates* com o intuito de testar todas as funcionalidades do programa e verificar que se encontrava totalmente operacional, dentro dos limites estabelecidos pelo grupo (profundidade máxima, etc). O teste mais básico realizado foi o [exemplo](#) fornecido pelos docentes no enunciado, a partir do qual o programa gera a seguinte solução:

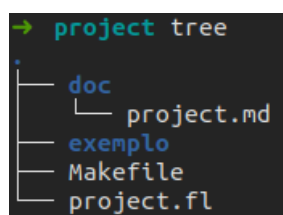
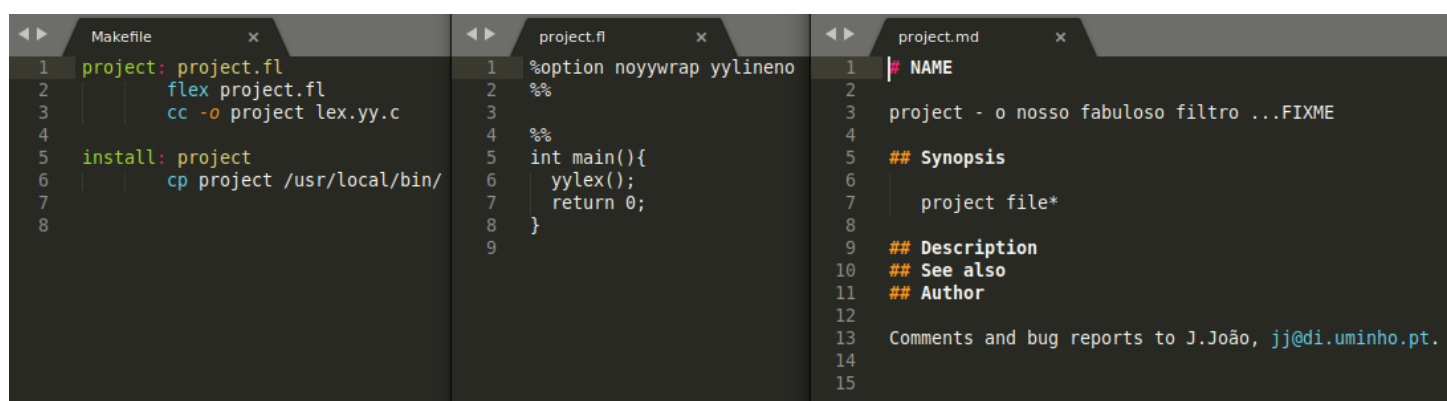


Figura 1: Árvore gerada a partir do template dos docentes.



```

Makefile
1 project: project.fl
2   flex project.fl
3   cc -o project lex.yy.c
4
5 install: project
6   cp project /usr/local/bin/
7
8
project.fl
1 %option noyywrap yylineno
2 %%
3
4 %%
5 int main(){
6   yylex();
7   return 0;
8 }
9
project.md
1 # NAME
2
3 project - o nosso fabuloso filtro ...FIXME
4
5 ## Synopsis
6
7 project file*
8
9 ## Description
10 ## See also
11 ## Author
12
13 Comments and bug reports to J.João, jj@di.uminho.pt.
14
15
  
```

Figura 2: Ficheiros gerados a partir do template dos docentes.

O segundo ficheiro de teste elaborado visa aferir a capacidade de processamento de metadados do programa, focando os seguintes pontos:

- número arbitrário de metadados;
- irrelevância de espaços antes do valor do metadado;
- irrelevância de parágrafos entre metadados;
- armazenamento dos dados no mapa e transcrição para ficheiros.



```
=== meta

email:      jj@di.uminho.pt
author: J.João
date:      23/03/2020


institution:Universidade do Minho

course: Processamento de Linguagens
# "name" é dado por argumento de linha de comando (argv[1])

=== tree
{%name%}/
- doc/
-- {%name%}.md
- Makefile

=== Makefile

{%name%}: {%name%}.fl
        flex {%name%}.fl
        cc -o {%name%} lex.yy.c

install: {%name%}
        cp {%name%} /usr/local/bin/

=== {%name%}.md
# NAME

{%name%} - o nosso fabuloso filtro ...FIXME

## Synopsis

    {%name%} file*

## Description
## See also
## Author

Comments and bug reports to {%author%}, {%email%}.

Done on {%date%}.
For the course {%course%}, lectured in {%institution%}.
```

Quando executado com este ficheiro, o programa produziu a seguinte diretoria:

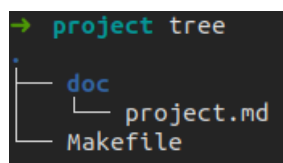


Figura 3: Árvore gerada a partir do segundo template.

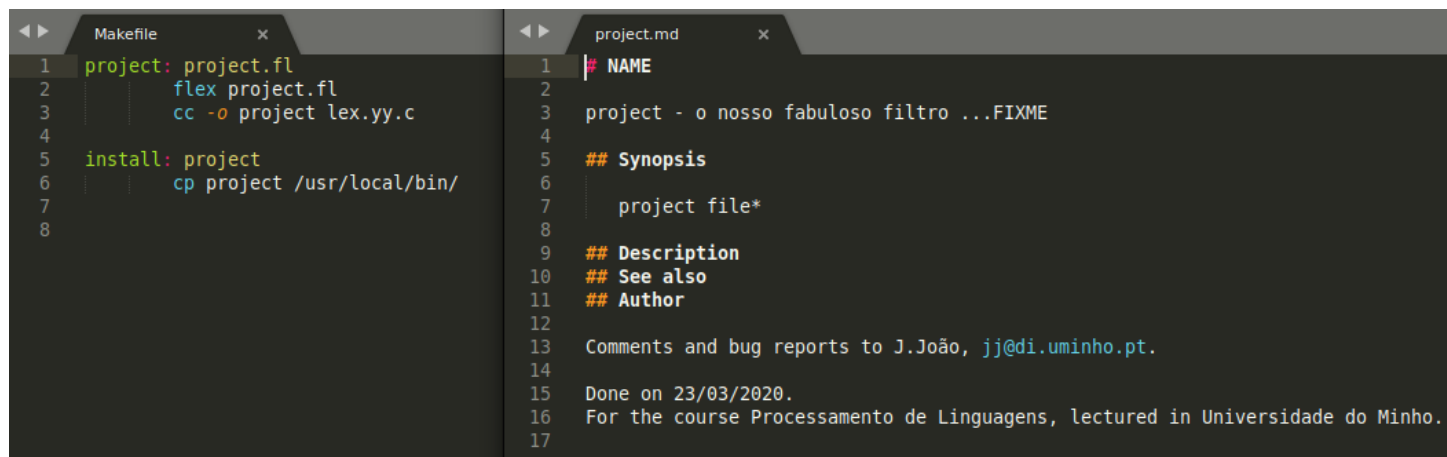


Figura 4: Ficheiros gerados a partir do segundo template.

O terceiro *template* testa ao máximo as possibilidades da árvore, verificando se algum nome, de entre os ponderados para a solução, ou erro na formatação incorre o programa em erro. Como tal, este teste tem por objetivo verificar:

- irrelevância de um número natural arbitrário de parágrafos entre os elementos da árvore;
- todos os tipos admissíveis de nomes de ficheiros;
- todos os tipos admissíveis de nomes de pastas;
- refutação de elementos inválidos:
  - elemento com '/' no nome (inválido em sistemas Linux);
  - elemento com {%...%} no nome cujo respetivo metadado possui '/';
  - elemento com profundidade errada (p.e. 4, quando a pasta atual com maior profundidade tem 2);
  - ficheiro com profundidade impossível (>5).
- profundidade máxima.

```

=== meta

email: jj@di.uminho.pt
author: J.João
date: 23/03/2020
# "name" é dado por argumento de linha de comando (argv[1])

=== tree
{%name%}/
  
```

```
- todosTiposFicheiros/
-- log.txt
-- Makefile
-- {%name%}.md
-- {%author%}.fl
-- {%name%}
-- {%author%} Jr
-- {%name%}.1.2.3.fl

- todosTiposPastas/
-- doc/
-- {%name%}/
-- {%email%}/
-- {%name%}.2/

--- ficheiro/2.txt
--- pasta/2/
--- {%date%}.fl
--- {%date%}/
---- exemplo2/
----- ficheiroInvalido
----- profundidadeInvalida

- profundidade1/
-- profundidade2/
--- profundidade3/
---- profundidade4/
----- profundidadeMax/
----- ficheiroMaxProfundidade.md

=== Makefile

{%name%}: {%name%}.fl
        flex {%name%}.fl
        cc -o {%name%} lex.yy.c

install: {%name%}
        cp {%name%} /usr/local/bin/

=== ficheiroMaxProfundidade.md
# NAME

{%name%} - o nosso fabuloso filtro ...FIXME

## Synopsis

        {%name%} file*

## Description
## See also
## Author
```

```

Comments and bug reports to {%author%}, {%email%}.

=== {%author%}.fl
%option noyywrap yylineno
%%

%%
int main(){
    yylex();
    return 0;
}

```

Como é possível deduzir a partir dos seguintes resultados, o programa conseguiu gerir corretamente todas as situações:

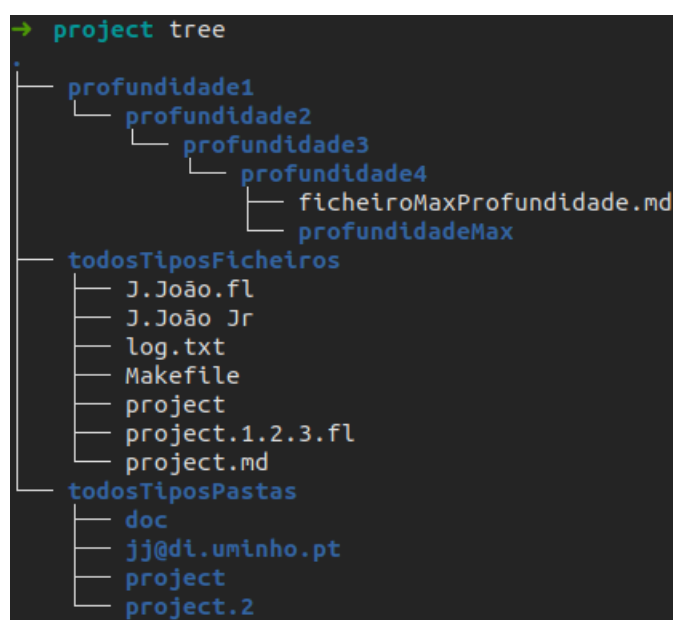


Figura 5: Árvore gerada a partir do terceiro template.

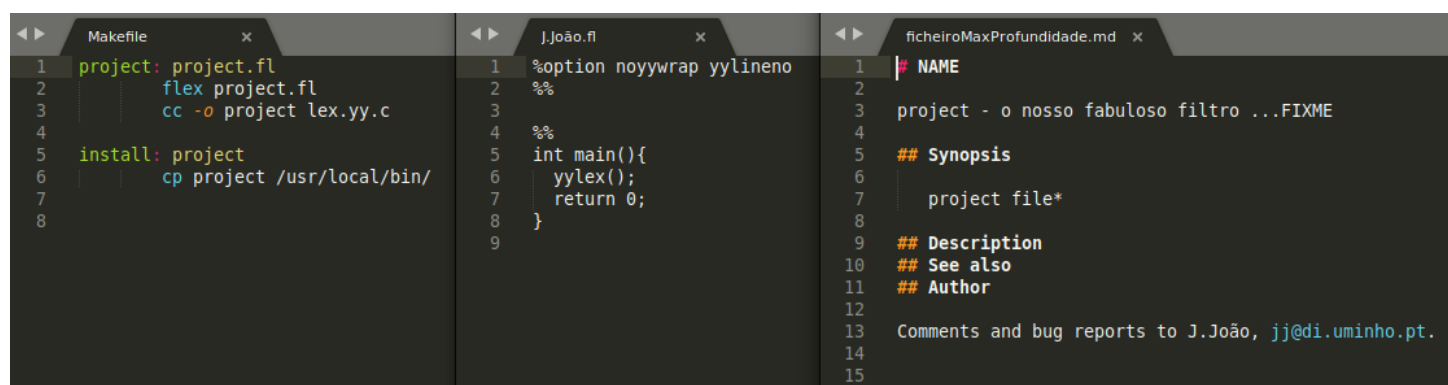


Figura 6: Ficheiros gerados a partir do terceiro template.

Por fim, elaborou-se um último ficheiro que mistura todas as funcionalidades testadas nos anteriores, procurando obter uma solução mais completa e realista. Além disto, põe também à

prova a irrelevância da ordem de escrita para os ficheiros, sendo que os seus *templates* aparecem por ordem diferente da que os ficheiros aparecem na árvore.

```
=== meta

email: jj@di.uminho.pt
author: J.João
date: 23/03/2020
institution: Universidade do Minho
course: Processamento de Linguagens
# "name" é dado por argumento de linha de comando (argv[1])

=== tree
{%name%}/
- {%name%}.fl
- doc/
-- metadata.md
-- {%name%}.md
-- doc2/
--- doc3.1/
---- doc4/
----- max.txt
--- doc3.2/
---- {%course%} for Begginers
- exemplo/
- Makefile
--- ficheiroInvalido.md

=== Makefile

{%name%}: {%name%}.fl
        flex {%name%}.fl
        cc -o {%name%} lex.yy.c

install: {%name%}
        cp {%name%} /usr/local/bin/

=== {%course%} for Begginers

Ficheiro de demonstração das capacidades do nosso programa flex!

=== {%name%}.md
# NAME

{%name%} - o nosso fabuloso filtro ...FIXME

## Synopsis

    {%name%} file*

## Description
## See also
```

```
## Author

Comments and bug reports to {%author%}, {%email%}.

Done on {%date%}.
For the course {%course%}, lectured in {%institution%}.

=== {%name%}.fl
%option noyywrap yylineno
%%

%%
int main(){
    yylex();
    return 0;
}

=== max.txt

Esta é a profundidade máxima a que pode ser criado um ficheiro!

=== metadata.md

Email: {%email%}
Author: {%author%}
Date: {%date%}
Institution: {%institution%}
Course: {%course%}
```

Foi gerada a seguinte diretoria e, com isto, o grupo concluiu que o programa estava totalmente operacional.

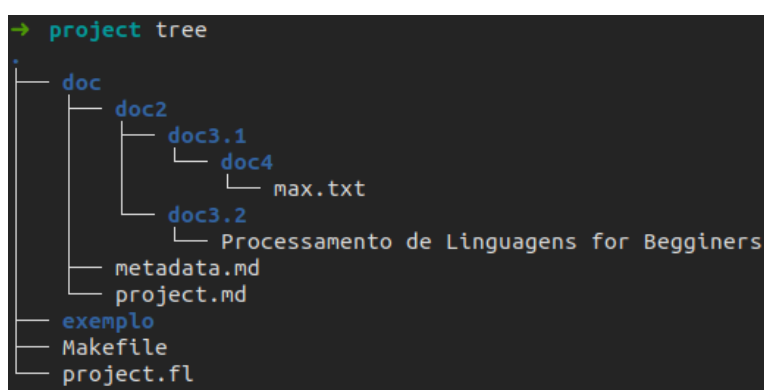
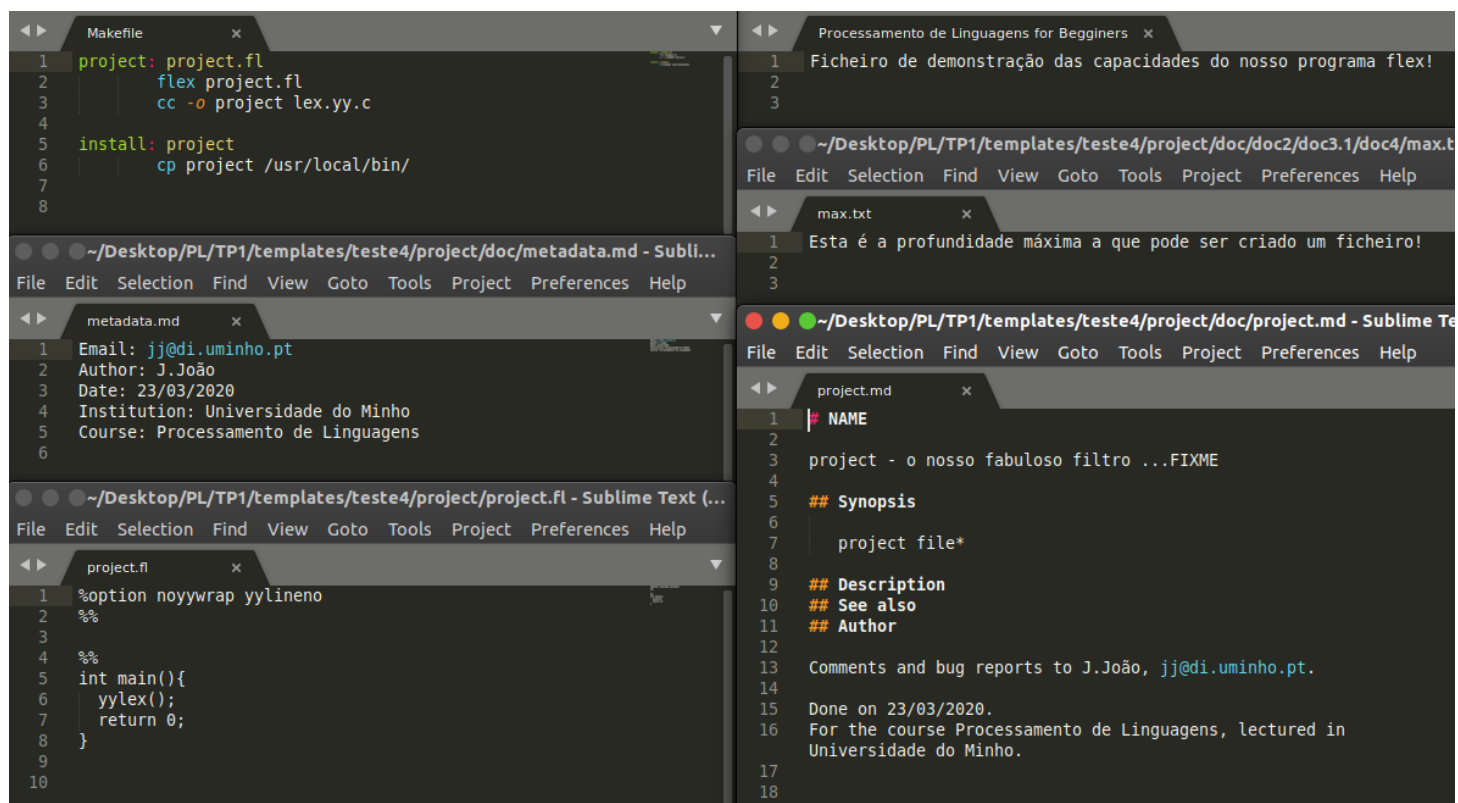


Figura 7: Árvore gerada a partir do quarto template.



```
Makefile
1 project: project.fl
2     flex project.fl
3     cc -o project lex.yy.c
4
5 install: project
6     cp project /usr/local/bin/
7
8

~/Desktop/PL/TP1/templates/teste4/project/doc/project.md - Sublime Text (...)
1 # NAME
2
3 project - o nosso fabuloso filtro ...FIXME
4
5 ## Synopsis
6
7     project file*
8
9 ## Description
10 ## See also
11 ## Author
12
13 Comments and bug reports to J.João, jj@di.uminho.pt.
14
15 Done on 23/03/2020.
16 For the course Processamento de Linguagens, lectured in
17 Universidade do Minho.
18

project.fl
1 %option noyywrap yylineno
2 %%
3
4 %%
5 int main(){
6     yylex();
7     return 0;
8 }
9
10

metadata.md
1 Email: jj@di.uminho.pt
2 Author: J.João
3 Date: 23/03/2020
4 Institution: Universidade do Minho
5 Course: Processamento de Linguagens
6

Processamento de Linguagens for Beginners
1 Ficheiro de demonstração das capacidades do nosso programa flex!
2
3
```

Figura 8: Ficheiros gerados a partir do quarto template.

## 5 Conclusão

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionada nas aulas da unidade curricular e a elucidação da verdadeira utilidade e poder de ferramentas como os filtros *Flex*, expressões regulares e regras Condição-Ação, que permitem o fácil processamento e recolha de dados com base em padrões em ficheiros extensos.

O grupo está bastante satisfeito com o resultado final e considera que não só cumpriu os requisitos propostos no enunciado, como foi para além disso e desenvolveu uma solução mais escalável e dinâmica, que proporciona uma liberdade muito superior ao utilizador, permitindo-lhe gerir mais dados e criar projetos mais complexos.

Além disso, a flexibilidade do produto final também esteve sempre em mente, pelo que se procurou implementar o tratamento de erros e irregularidades no *template* onde fosse possível, de maneira a ter em consideração o fator do erro humano, que poderia levantar bastantes problemas e inutilizar a solução.

Concluindo, foi possível obter conhecimentos valiosos através da elaboração deste trabalho prático que, de facto, já se provaram úteis em outros projetos alheios à unidade curricular, e desenvolver uma solução rica e compacta que abrange todos os requisitos propostos, facilitando o trabalho de qualquer pessoa que pretenda começar um novo projeto.



## A Código do Programa

Lista-se a seguir o código da solução desenvolvida pelo grupo. Os comentários presentes nos ficheiros originais foram removidos e a disposição do código foi ligeiramente alterada a fim de tornar o código mais compacto e legível.

```
1 %option noyywrap yylineno
2 %x DIV META SAVEMETA HYPHENS NAME EXTENSION NEWLINES TOFILE
3
4 %{
5 #include <stdio.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8 #include <unistd.h>
9 #include "map.h"
10
11 #define maxDir 5
12
13 char *newMeta;
14 char *name;
15 char *writeTo;
16 char *dir[maxDir];
17 struct map* meta;
18 struct map* files;
19 int hyphens = 0;
20
21 char* buildRoute();
22 char* getMetaKey(char* yytext);
23 char* getMetaFilename(char* yytext);
24 char* append(char* str1, char* str2, char separator);
25 %}
26
27 %%
28
29 <*>===[ ]*      { if (writeTo) { free(writeTo); writeTo = NULL; }
30                  } BEGIN DIV;
31
32 <DIV>meta        BEGIN META;
33 <DIV>tree        BEGIN NAME;
34 <DIV>"{" .+"}" .* {writeTo = getMetaFilename(yytext);}BEGIN NEWLINES;
35 <DIV>.+          { writeTo = strdup(yytext); } BEGIN NEWLINES;
36
37 <META>^\.+: [ ]* { newMeta = strdup(strtok(yytext,":"));
38                  } BEGIN SAVEMETA;
39
40 <SAVEMETA>.+     { mapDynAdd(newMeta,strdup(yytext),meta);
41                  free(newMeta); } BEGIN META;
42
43 <HYPHENS>-- [ ]* { hyphens = 0;
44                  while (yytext[hyphens] == '-') hyphens++;
45
46                  if (name) { free(name); name = NULL; }
47                  if (hyphens <= maxDir && dir[hyphens-1])
48                      BEGIN NAME;
49                  }
50
51 <NAME>"{" .+"}" [^\n\/]* { name = getMetaFilename(yytext);
52
```



```
53         if (strchr(name, '/')) BEGIN HYPHENS;
54         else BEGIN EXTENSION; }
55
56 <NAME>[^\n\/]+      { name = strdup(yytext); } BEGIN EXTENSION;
57
58 <EXTENSION>\/.*     { if (strlen(yytext) == 1) {
59
60         for (int i = hyphens; i < maxDir; i++) {
61             if (dir[i]) {
62                 free(dir[i]);
63                 dir[i] = NULL;
64             }
65         }
66         dir[hyphens] = strdup(name);
67
68         char* route = buildRoute();
69         mkdir(route, 0700);
70
71         free(route); }
72     } BEGIN HYPHENS;
73 <EXTENSION>\n       { char* route = buildRoute();
74
75         FILE *fd = fopen(route, "w");
76         mapDynAdd(name, fd, files);
77
78         free(route); } BEGIN HYPHENS;
79
80 <NEWLINES>\n+       BEGIN TOFILE;
81
82 <TOFILE>"{%"}([^\%\\}| | [^\%\\}| | \[%^\%\\}| | )+"%" { fprintf(mapGet(writeTo,
83     files), "%s", (char*) mapGet(getMetaKey(yytext), meta)); }
84
85 <TOFILE>.\|\n       { fprintf(mapGet(writeTo, files), "%s", yytext); }
86
87 <*>.\|\n           ;
88
89 %%
90
91 char* buildRoute() {
92     char* route = NULL;
93
94     for (int i = 0; i < hyphens; i++)
95         route = append(route, dir[i], '/');
96
97     return append(route, name, '/');
98 }
99
100 char* getMetaKey(char* yytext) {
101     char* key = strtok(yytext, "%");
102     return strtok(NULL, "%");
103 }
104
105 char* getMetaFilename(char* yytext) {
106     char* key = getMetaKey(strdup(yytext));
107     char* name = strdup(mapGet(key, meta));
108
109     return append(name, yytext+strlen(key)+4, '!');
```

```
110
111 char* append(char* str1, char* str2, char separator) {
112     char * new_str;
113
114     if (!str1) new_str = strdup(str2);
115     else {
116         int len = strlen(str1);
117
118         new_str = malloc(len+strlen(str2)+2);
119         new_str[0] = '\0';
120         strcat(new_str, str1);
121
122         if (separator != '!') {
123             new_str[len] = separator;
124             new_str[len+1] = '\0';
125         }
126
127         strcat(new_str, str2);
128         free(str1);
129     }
130
131     return new_str;
132 }
133
134 int main(int argc, char* argv[]) {
135     yyin = fopen(argv[2], "r");
136
137     meta = mapNew();
138     files = mapNew();
139
140     mapDynAdd("name", strdup(argv[1]), meta);
141     newMeta = name = writeTo = NULL;
142
143     for (int i = 0; i < maxDir; i++)
144         dir[i] = NULL;
145
146     yylex();
147
148     mapCloseMeta(meta);
149     mapCloseFiles(files);
150     fclose(yyin);
151
152     free(name);
153     free(writeTo);
154
155     for (int i = 0; i <= maxDir; i++)
156         if (dir[i]) free(dir[i]);
157
158     return 0;
159 }
```

Listing 14: Código do ficheiro filter.l.



```
1 #include <stdlib.h>
2 #include <string.h>
3 #include <stdio.h>
4
5 #include "map.h"
6
7 #define MAP_BY_VAL 0
8 #define MAP_BY_REF 1
9
10 typedef struct mapitem {
11     char* key;
12     void* val;
13     int type;
14 } MI;
15
16 typedef struct map {
17     int size;
18     MI* items;
19 } M;
20
21 M* mapNew() {
22     M* map;
23
24     map = malloc(sizeof(M));
25     map->size = 0;
26     map->items = NULL;
27
28     return map;
29 }
30
31 void mapAdd(char* key, void* val, M* map) {
32     char* newkey = strdup(key);
33
34     if (map->size == 0) {
35         map->items = malloc(sizeof(MI));
36     }
37     else {
38         map->items = realloc(map->items, sizeof(MI) * (map->size + 1));
39     }
40
41     (map->items + map->size)->key = newkey;
42     (map->items + map->size)->val = val;
43     (map->items + map->size++)->type = MAP_BY_VAL;
44 }
45
46 void mapDynAdd(char* key, void* val, M* map) {
47     mapAdd(key, val, map);
48     (map->items + map->size - 1)->type = MAP_BY_REF;
49 }
50
51 void* mapGet(char* key, M* map) {
52     int i;
53
54     for (i = 0; i < map->size; i++)
55     {
56         if (strcmp((map->items + i)->key, key) == 0)
57         {
58             return (map->items + i)->val;
```

```
59     }
60 }
61
62 return NULL;
63 }
64
65 void mapPrint(M* map) {
66     int i;
67
68     for (i = 0; i < map->size; i++)
69     {
70         printf("Key: %s, FD: %p\n", (map->items + i)->key, (map->items + i)
71             ->val);
72         //printf("Key: %s, Meta: %s\n", (map->items + i)->key, (char*) (map
73             ->items + i)->val);
74     }
75 }
76
77 void mapCloseMeta(M* map) {
78     int i = 0;
79
80     for(; i < map->size; i++)
81     {
82         free((map->items + i)->key);
83
84         if ((map->items + i)->type == MAP_BY_REF)
85             free((map->items + i)->val);
86     }
87
88     free(map->items);
89     free(map);
90 }
91
92 void mapCloseFiles(M* map) {
93     int i = 0;
94
95     for(; i < map->size; i++)
96     {
97         free((map->items + i)->key);
98
99         if ((map->items + i)->type == MAP_BY_REF)
100             fclose((map->items + i)->val);
101     }
102
103     free(map->items);
104     free(map);
105 }
```

Listing 15: Código do ficheiro map.c.

```
1 #ifndef MAP_H
2 #define MAP_H
3
4 struct map;
5
6 struct map* mapNew();
7 void mapAdd(char* key, void* val, struct map* map);
8 void mapDynAdd(char* key, void* val, struct map* map);
9 void* mapGet(char* key, struct map* map);
10 void mapPrint(struct map* map);
11 void mapCloseMeta(struct map* map);
12 void mapCloseFiles(struct map* map);
13
14 #endif
```

Listing 16: Código do ficheiro map.h.

```
1 make: filter.l
2   flex filter.l
3   gcc -o mkfromtemplate lex.yy.c map.c
4
5 clean:
6   rm -r mkfromtemplate lex.yy.c
```

Listing 17: Código da *Makefile*.

## B *Template* do enunciado

```
=== meta

email: jj@di.uminho.pt
author: J.João
# "name" é dado por argumento de linha de comando (argv[1])

=== tree

{%name%}/
- {%name%}.fl
- doc/
-- {%name%}.md
- exemplo/
- Makefile

=== Makefile

{%name%}: {%name%}.fl
flex {%name%}.fl
cc -o {%name%} lex.yy.c

install: {%name%}
cp {%name%} /usr/local/bin/

=== {%name%}.md
# NAME

{%name%} - o nosso fabuloso filtro ...FIXME

## Synopsis

{%name%} file*

## Description
## See also
## Author

Comments and bug reports to {%author%}, {%email%}.

=== {%name%}.fl

%option noyywrap yylineno
%%
%%
int main(){
    yylex();
    return 0;
}
```