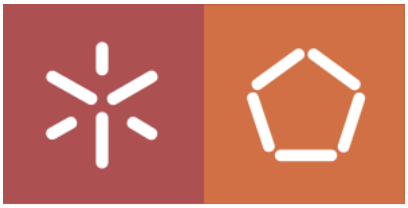

Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático 1

Template multi-file

Diogo Araújo	Diogo Nogueira	Mariana Lino
A78485	**A78957**	**A78824**

Resumo

Este trabalho prático é fruto de um desafio de projeto da Unidade Curricular de *Processamento de Linguagens* e destina-se essencialmente à prática do analisador léxico de seu nome FLex, que permite analisar e consequentemente gerar ficheiros de pequena ou grande dimensão.

Conteúdo

1 Introdução **2**

2 *Template multi-file* **3**

3 Decisões e Implementação **5**

3.1 Análise do ficheiro *template* 5

3.2 Código C - Estruturas de Dados 6

3.2.1 Definição da Estrutura 6

3.2.2 Definição das Funções 7

3.3 Código Flex - Expressões Regulares 10

3.3.1 Análise e filtragem da secção `META` 11

3.3.2 Análise e filtragem da secção `TREE` 11

3.3.3 Análise e filtragem da secção `FICHEIRO` 13

3.3.4 Análise e filtragem da secção `CONTENT` 14

4 Solução Final e Testes **15**

5 Observações Finais **16**

Capítulo 1

Introdução

O presente trabalho surge no âmbito da Unidade Curricular de Processamento de Linguagens, correspondente ao 3.º Ano do Mestrado Integrado em Engenharia Informática. O objetivo principal passa por pôr em prática o uso do interpretador léxico FLex e através dele desenvolver um programa capaz de criar a base inicial comum a um projeto de *software*.

Eis os objetivos planeados para este primeiro trabalho prático:

- Compreender o modo como o interpretador FLex opera e de que forma pode ser usado para favorecer os objetivos do projeto;
- Criar um programa de nome **makefromtemplate** que seja capaz de aceitar um nome de projeto e um ficheiro descrição de um *template multi-file*, criando os ficheiros e pastas iniciais do projeto;
- Saber analisar esse ficheiro *template* e extrair dele toda a informação necessária e útil;
- Compreender a necessidade de rotinas em C para dar resposta a todos os requisitos do projeto.

Para se alcançarem estes objetivos existirá uma fase inicial que se focará em entender/analisar todo o ficheiro *template* dado como base. Com esta análise feita e um conjunto de ideias e metedologias inicias, contextualia-ze a necessidade de existir uma estrutura de dados em C e de que forma a mesma consegue responder ao pretendido. Estando toda esta parte feita, pensa-se no código em FLex, defenindo-se todas as Expressões Regulares necessárias e estabelecendo a ligação inerente à estrutura de dados pensada.

A metodologia utilizada para este projeto passou por estudar o desenvolvimento de Expressões Regulares em FLex e a inclusão de código de linguagem C nesse mesmo ficheiro, de modo a entender a ligação entre ambos.

Capítulo 2

Template multi-file

O *template multi-file* surge como a ideia dum ficheiro que especifica toda a estrutura base de um projeto de software e que, através dum programa perfeitamente desenvolvido permite produzir uma solução esperada. Este *template* torna-se a base de todo o nosso projeto, dado que é através dele que se consegue deduzir/apurar as decisões necessárias a tomar, tanto para o código FLex como para as rotinas eventualmente inevitáveis em C.

Através da sua análise e tendo em mente a forma como o interpretador FLex funciona, pretende-se assim desenvolver um programa que através de um conjunto de definições seja capaz de criar os ficheiros e pastas iniciais pedidas para o projeto em si.

A estrutura deste ficheiro *template* deve estar devidamente definida e incluir os seguintes requisitos:

- **Metadados** – que correspondem ao **author** e **email**. Estes ficam definidos da forma como abaixo se apresenta, sendo que só dessa forma conseguem ser detetados pelo interpretador FLex.

```
=== meta

email: a78485@uminho.pt
author: Diogo Nogueira
```

- **Tree** – que especifica a estrutura de pastas e ficheiros a serem criados. Esta hierarquia deve estar definida consoante as pastas, subpastas e seus respetivos ficheiros.

```
=== tree

%name%/
- %name%.fl
- doc/
- Makefile
```

- ***Template* de cada ficheiro** – que corresponde ao conteúdo que cada ficheiro deve ter na sua constituição base.

```
=== %name%.fl

%option noyywrap yylineno
%%

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

Com estas ideias presentes, o grupo fez um levantamento inicial de requisitos, no sentido de tentar compreender aquilo que poderia vir a auxiliar na resolução do projeto:

Requisito	**Possível Resolução**
Extração do valor de **name**	Uma vez que é passado via argumento na linha de comandos a sua extração torna-se intuitiva.
Extração do valor de **email**	Tendo em conta que o ficheiro *template* deve iniciar-se com os campos referentes à parte do meta, pode-se desenvolver uma ER que o "apanhe".
Extração do valor de **author**	O mesmo pensamento aplicado anteriormente.
Interpretação da **tree**	Desde início o grupo pensou que a criação das pastas e ficheiros através das várias diretorias deveria ser feita com o auxílio de uma estrutura de dados em C.
Conteúdo do ficheiro	Se conseguirmos obter a diretoria de cada ficheiro definida na estrutura de dados, conseguimos abrir o mesmo e dessa forma obter o seu respetivo conteúdo.

Tabela 2.1: Levantamento incial de possíveis requisitos a implementar para o projeto.

Capítulo 3

Decisões e Implementação

3.1 Análise do ficheiro *template*

O ficheiro *template* é o suporte para o desenvolvimento de uma solução que satisfaça todos os requisitos nele definidos, daí a sua extrema importância na globalidade do projeto. Depois de uma análise basilar feita no Capítulo anterior, pretende-se agora estudar a fundo tudo aquilo que possa interessar saber.

Desde logo se nota que existe um padrão que deve ser respeitado e que o mesmo se encaixa em três "secções" principais:

- **Meta** – secção onde devem ser incluídos o **email** e **author** do criador do projeto. Esta informação poderá ser usada para o conteúdo de alguns dos ficheiros do mesmo.

Tendo em conta que está logo no início do ficheiro, a sua recolha é simples e pode ser armazenada em variáveis globais criadas para o efeito no próprio código FLex.

- **Tree** – secção que estipula todas as pastas e ficheiros que se pretende criar para o projeto.

Para esta secção teve de existir o cuidado de entender como é que a hierarquia dos ficheiros e pastas estava a ser representada, tendo-se verificado que estava a ser controlada/comandada pelo número de caracteres "-". É através destes que a estrutura de dados deve ser pensada e posteriormente desenvolvida.

Nota-se também uma diferença entre os "tipos" de pastas e ficheiros que podem ser declarados. Existem pastas e ficheiros cujo nome está dependente do valor do **name** – que é passado como argumento na linha de comandos. Dessa forma, sabe-se que esse **name** deve estar armazenado algures no código FLex, sob a forma de variável global, uma vez que poderá ser usado ao longo de toda a travessia pelo ficheiro *template*.

- **Ficheiros** – esta secção representa todos os ficheiros declarados na **tree** e que devem ser apropriadamente definidos.

Pelo que se nota, estes ficheiros não têm necessariamente de serem definidos pela mesma ordem que são apresentados na parte da **tree**.
Da mesma forma que se verifica na secção **tree**, existem vários tipos de ficheiros que devem ser tomados em conta. Isto é importante manter claro, visto que é através desses nomes que vamos construindo as várias diretorias.

Para o template inicialmente fornecido notamos a existência de três "tipos" de ficheiro. Quando mencionamos a palavra tipos é no sentido de dar a entender que poderão ter de existir em FLex três ERs para esta parte. Note-se que um ficheiro pode ou não ter extensão e o seu nome pode estar dependente do valor **name**, tal como acontece para as pastas.

Outro fator a ter em conta é a forma como o conteúdo de cada um destes ficheiros pode estar representado, tendo em conta que pode existir a necessidade de controlar o surgimento das variáveis **name**, **email** e ainda **author**. Assim, entende-se a importância destes "valores" serem globais em termos de código FLex, de modo a facilitar todo este controlo.

3.2 Código C - Estruturas de Dados

Os apuramentos iniciais falavam da possibilidade de criação de uma estrutura de dados que pudesse dar suporte a toda a ideia de criação de diretorias e ficheiros relacionados entre si. Depois da análise profunda do ficheiro *template*, validou-se essa necessidade, criando-se assim uma estrutura de dados simples, mas totalmente pensada para conseguir respeitar a hierarquia que a **tree** exige.

3.2.1 Definição da Estrutura

A estrutura de dados baseia-se em quatro parâmetros essenciais que permitem dar resposta a todos os requisitos.

```
#define isFile 1
#define isFolder 0

typedef struct directories {

  int lineNumber; // Ir ao último (lineNumber-1)
  int fileOrFolder;
  char* name;
  char* directory;
  struct directories* next;

} Directories;
```

Dada a simplicidade na sua criação e consequentemente preenchimento de dados, usou-se uma **Lista Ligada**. A mesma funciona por inserções no fim da lista e isto vai ao encontro da **tree** definida no *template*, que apresenta os ficheiros e pastas, um por cada linha.

Variável	Descrição da Variável
`int lineNumber`	É o parâmetro mais importante. Através dele conseguimos estabelecer um controle de até onde devemos "andar" na estrutura para obter a diretoria em que a pasta ou ficheiro devem ser criados. **Esta ideia ficará melhor definida quando for feita a análise da função responsável por criar a diretoria em si.**
`int fileOrFolder`	É o parâmetro que distingue se estamos perante um ficheiro ou uma pasta.
`char* name`	É o parâmetro que armazena o nome do ficheiro ou da pasta.
`char* directory`	É o parâmetro que armazena a diretoria onde deve ser criado o ficheiro ou pasta.

Tabela 3.1: Descrição das Variáveis da Lista Ligada.

3.2.2 Definição das Funções

As funções criadas para esta estrutura são no geral simples de compreender. A maioria delas são auxiliares, já que permitem organizar melhor o código e poupar também a repetição excessiva na parte do FLex.

```
// Auxiliar Functions: countLine, cleanName, numberIterations and inserEnd
// Principal Function: insertDirectory
int countLines(char *lineName);
char* cleanName(char *name, char *rootName);
int numberIterations(Directories *list, int lineLimit);
Directories* insertEnd(Directories *list, int lN, int fileFolder, char *n, char *d);

Directories* insertDirectory(Directories* list, int fileFolder, char *fileName,
char *rootName);
```

```
// Auxiliar Function: lastDirectory
// Principal Functions: createFolder, createFile
char* lastDirectory(Directories *list);

void createFolder(Directories *list);
void createFile(Directories *list);

// Other Functions
Directories* init();
char* getDirectory(Directories *list, char* nameFile);
void printDirectories(Directories *list);
```

Função	Descrição da Função
`int countLines`	**Função auxiliar da insertDirectory.** Conta o número de caracteres "-" que estão presentes na string **yytext** vinda do FLex.
`char* cleanName`	**Função auxiliar da insertDirectory.** Limpa uma string ao retirar todos os caracteres "-" da string **yytext** vinda do **yytext** do FLex.
`int numberIterations`	**Função auxiliar da insertDirectory.** Devolve o número de iterações que deve ser feito para se obter a diretoria relativa a um ficheiro ou pasta.
`Directories* insertEnd`	**Função auxiliar da insertDirectory.** Trata do processo de inserção de um novo elemento na lista.
`Directories* insertDirectory`	Função que insere uma nova diretoria na Lista Ligada.

Tabela 3.2: Descrição da função insertDirectory e suas auxiliares.

A tabela acima anexada faz uma listagem da função **insertDirectory** e as auxiliares à mesma, atribuindo o respetivo significado em termos de código. Estas são as funções mais importantes para o projeto, tendo em conta que é através delas que se consegue garantir uma hierarquia correta para a **tree**.

Para apurar esta concepção, considera-se necessário estudar ao detalhe a função insertDirectory, na medida em que é nela que fica definida a forma como se consegue obter sempre a diretoria correta consoante o número de caracteres "-" presentes no yytext passado pelo FLex.

1. A função começa por calcular o número de linhas existentes na string **yytext** e logo de seguida procede à limpeza desse texto, garantindo que o nome da pasta ou ficheiro se encontre preparado para ser atribuído à variável da lista.

```
int numberLines = countLines(name);
char *newName = cleanName(name, rootName);
```

2. De seguida, faz-se uma verificação do estado de armazenamento de dados. Caso não existam dados na lista, procede-se a uma inserção normal. Caso contrário, entra-se num estado de verificação mais extenso, que irá ser explicado na alínea seguinte.

```
if(list == NULL) {...}
else {
    // Do something here.
}
```

3. Assuma-se então que se está perante uma lista não vazia e que se quer agora inserir um novo elemento, garantindo que a sua diretoria vai ao encontro do que é pedido no *template*.

   Supondo que se quer inserir o elemento referente à pasta doc/ e que o estado da tree é o seguinte:

```
PL/
- PL.fl
- doc/
- exemplo/
-- doc/
```

   (a) Calcula-se o **lineLimit** para essa pasta/ficheiro, através do **numberLines** - número de caracteres -"existentes na string.

```
int lineLimit = numberLines - 1;
```

   Como existem dois caracteres -", o **lineLimit** terá o valor de 1.

(b) Calcula-se o **numberIterations** que deve ser feito para obtermos a string correta da diretoria referente a essa pasta.

```
int number = numberIterations(list, lineLimit);
```

A recorrência a esta função é de extrema importância, já que só através dela garantimos que a diretoria onde queremos inserir corresponde à do elemento mais abaixo na lista.

Note-se que já existiam a pasta **doc/** e **exemplo/** cujos **numberLines** são de 1. Com isto, garantimos que a condição do ciclo que se segue é perfeitamente respeitada e que criamos a nova pasta dentro da pasta **exemplo/** e não da pasta **doc/**.

```
while(current -> next != NULL && number != 0){
    number--;
    current = current -> next;
}
```

(c) Com estes valores calculados, conseguimos extrair a diretoria correspondente à pasta **exemplo/** e com isso criar uma variável que represente o caminho da nova pasta **doc/**.

```
char *directory;
directory = strdup(current -> directory);
strcat(directory, strdup(newName));
```

Todo este pensamento criado para a **Lista Ligada** funciona e respeita a hierarquia definida no ficheiro *template*, poupando assim a criação de estruturas mais complicadas não só de se entender mas também de manipular.

3.3 Código Flex - Expressões Regulares

Como centro deste projeto, temos o analisador léxico utilizado e ensinado nas aulas de Processamento de Linguagens, o *Flex*. Como o acrónimo indica é um *Fast Lexical Analyser* e consegue gerar e analisar bastante rápido, ficheiros de pequena ou grande dimensão.

Para a análise do template ser de forma linear e corrida foi pensado na criação de quatro `START CONDITIONS`, sendo `META`, `TREE`, `FICHEIRO` e `CONTENT`. Desta forma podíamos separar as expressões regulares de cada fase de análise e *parse* deste *template* facilitando-nos o processo e celeridade, mas também a organização e junção com o código e estruturas em C criadas para alavancar todo este processo.

3.3.1 Análise e filtragem da secção `META`

Assim a primeira expressão regular é para apanhar os caracteres literais que definem o começo da secção do *template* para os meta dados associados.

```
"=== meta" {
    BEGIN META;
}
```

Com isto, entramos na `START CONDITION` relativa à parte *meta* do *template* que irá analisar e fazer o *parse* do **email** e **author** para uma variável *char** em C para depois trocar os *tokens* futuros presentes nos conteúdos dos ficheiros para o projeto a criar.

```
<META>{
    "email: "[^\n]+ {
        email = strdup(yytext+7);
    }

    "author: "[^\n]+ {
        author = strdup(yytext+8);
        BEGIN INITIAL;
    }
    \n {;}
}
```

Ambas as expressões têm similaridades na técnica escolhida para capturar os meta dados sendo que se apanha os caracteres literais de `"email:  "` seguido por qualquer carácter exceto o carácter que define a nova linha (*newline*). Para o autor foi a mesma técnica apenas trocando a captura de caracteres literais de `"author:  "`. Como a parte dos meta dados tem apenas estas duas variáveis e um código de escrita bastante rigoroso, assumimos a não criação de sub-`START CONDITIONS` e apenas avançar o texto analisado para a frente e apanhar apenas o email e autor para as suas variáveis em C.

Por fim, foi colocado também uma regra para filtrar as linhas que ficavam no analisador e voltou-se a repor o estado para a `START CONDITION INITIAL` de forma a continuar o processo de análise e filtragem do *template*.

3.3.2 Análise e filtragem da secção `TREE`

Esta secção é o núcleo de todo este trabalho prático e obviamente demandou várias expressões regulares para as várias nuances que poderia existir no ficheiro e nesta secção em particular.

```
"=== tree" {
    BEGIN TREE;
}
```

Para o começo da START CONDITION apenas houve a análise dos caracteres literais de \=== tree" como o marco do início no *template* para a análise da estrutura da árvore.

```
<TREE>{
    ^\{"%name%"\}\/ {
        char *directory = strdup(name);
        strcat(directory, "/");

        directories = insertDirectory(directories, isFolder, strdup(directory), NULL);
        createDirectory(directories);
    }

    ^[a-zA-Z]+\/ {
        directories = insertDirectory(directories, isFolder, strdup(yytext), NULL);
        createDirectory(directories);
    }
(...)
```

Esta porção de expressões regulares assume as ideias para a criação de pastas raiz e consequentemente o início da árvore da diretoria. Assim sempre que se apanha, no início da linha, qualquer carácter de A a Z insensível a maiúsculas seguido de uma barra \/" podemos afirmar que é para a análise, filtragem e criação duma pasta raiz. Como exceção também foi criada uma expressão regular para caso a pasta raiz seja com o **name** fornecido no argumento do programa. Assim, como explicado na secção anterior, temos a criação do primeiro nodo que representa esta árvore de diretorias e por consequência a junção à lista ligada que acondicionará estes dados ao longo da análise *Flex*.

```
(...)
^\-+\ [a-zA-Z]+\.[a-z]+ {
        directories = insertDirectory(directories, isFile, strdup(yytext), NULL);
        createFile(directories);
    }

    ^\-+\ [a-zA-Z]+ {
        directories = insertDirectory(directories, isFile, strdup(yytext), NULL);
        createFile(directories);
    }

    ^\-+\ \{"%name%"\}\.[a-z]+ {
        directories = insertDirectory(directories, isFile, strdup(yytext), strdup(name));
        createFile(directories);
    }
(...)
```

Estas seguintes expressões regulares são as necessárias para filtrar e analisar todo o tipo de ficheiros que possam estar na árvore representada no *template*. Para tal, começamos por filtrar e "apanhar" todos os tracinhos que indicam o nível do ficheiro na árvore de diretorias. Assim depois basta filtrar

o nome do mesmo insensível às maiúsculas e ainda colocar um caso adicional para caso o ficheiro não tenha nenhuma extensão definida, como por exemplos ficheiros *Makefile*. Dado a natureza do trabalho prático também temos de definir uma expressão para caso o ficheiro não tenha nome definido e sim, apenas o *token* para a substituição futura. Em termos de tratamento em C todos são similares dado que o trabalho depois se remontou para a parte de estruturas e funções em C como falado na secção anterior. Aqui apenas se adiciona este ficheiro à lista ligada que está a suportar todas as informações e aciona-se a criação do ficheiro desse último nodo (com o `fopen()`).

```
(...)
===[ ] {BEGIN FICHEIRO;}

    \n  {;}
```

Por fim, temos o caso de paragem para começar a START CONDITION que irá tratar dos ficheiros e posteriormente do seu conteúdo.

3.3.3 Análise e filtragem da secção `FICHEIRO`

```
<FICHEIRO>{
    \{"%name%"\}\.[a-z]+ {
        char *nameFile = strdup(name);
        char *extension = strchr(yytext, '.');
        strcat(nameFile, extension);

        currentNameFile = nameFile;
    }
    [a-zA-Z\.]+ {
        currentNameFile = strdup(yytext);
    }

    \n {
        BEGIN CONTENT;
        currentDirectory = getDirectory(directories, strdup(currentNameFile));
        currentFile = fopen(currentDirectory, "a+");
    }

}
```

Esta `START CONDITION` apesar de ser muito simples ao nível das expressões regulares, foi essencial existir separada dado a complexidade para filtrar e analisar o nome do ficheiro e depois ir buscá-lo à lista ligada das directorias e ficheiros feita anteriormente.

Assim, caso o nome do ficheiro seja um *token*, a filtração passa por concatenar apenas ao conteúdo da variável **name** o tipo da extensão do ficheiro em si. Assim quando acontecer um *newline*, podemos assumir que o conteúdo do ficheiro irá começar, ou seja a nova `START CONDITION` e assim vamos buscar todo o caminho correto desse ficheiro à lista ligada para consequentemente abrir o ficheiro

utilizando a função predefinida em C `fopen()` já com a *flag* de *append* dado que o ficheiro já foi criado anteriormente aquando a primeira análise na secção da `TREE`.

3.3.4 Análise e filtragem da secção `CONTENT`

```
<CONTENT>{
    \{"%name%"\} {
        fwrite(name, strlen(name), 1, currentFile);
    }
    \{"%email%"\} {
        fwrite(email, strlen(email), 1, currentFile);
    }
    \{"%author%"\} {
        fwrite(author, strlen(author), 1, currentFile);
    }
    .|\n {
        fwrite(yytext, strlen(yytext), 1, currentFile);
    }
    ===[ ] {
        fclose(currentFile);
        currentDirectory = currentNameFile = "";

        BEGIN FICHEIRO;
    }
}
```

Para esta fase final entende-se que já se tem guardado numa variável local o ficheiro corrente, logo as expressões regulares são relativamente simples por causa de todo o processo e trabalho que aconteceu por detrás. A regra geral é evidentemente apanhar qualquer carácter e também o *line terminator* colocando também para filtração o carácter *newline*. Assim sempre que se apanha um destes caracteres, coloca-se no ficheiro corrente, passando assim o conteúdo para o ficheiro correto.

Como o *template* pode ter também *tokens* no conteúdo destes ficheiros, foram colocadas as expressões regulares mais restritas/literais para esses casos dos *tokens* trocando-os para os valores das variáveis globais que contêm essas informações.

Por fim, temos o caso de paragem desta `START CONDITION` igual ao caso de paragem da secção `TREE` que caso encontre `===[ ]`, ou seja três caracteres de "igual"seguido de um espaço, indica que se trata de outro ficheiro, começando aqui um ciclo até o *template* terminar.

Capítulo 4

Solução Final e Testes

Para facilitar e poupar a execução de comandos extra, desenvolveu-se uma **makefile** que aplica o comando do FLex ao ficheiro **mkfromtemplate.l** e logo de seguida corre o **gcc** do ficheiro da Estrutura de Dados com o **lex.yy.c** obtido anteriormente.

Execute-se então o comando make inicial antes de se proceder para o teste do programa em si:

```
> make
```

Executa-se o programa, fornecendo como parâmetros o name que queremos dar ao projeto bem como o *template* que deve ser seguido para a criação base do mesmo:

```
> mkfromtemplate PyLy templateNosso
```

Como solução final, obtêm-se todas as pastas, subpastas e ficheiros no seu devido local, tal como se comprova pelo template fornecido:

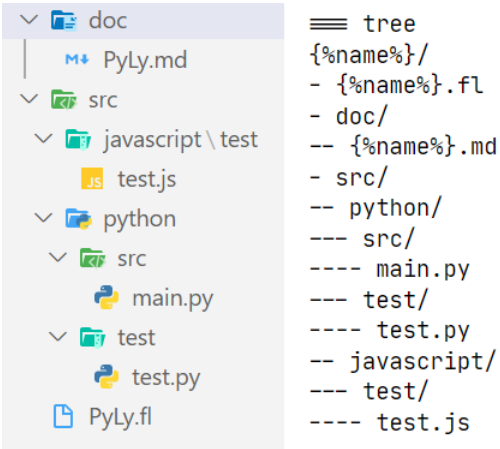

Figura 4.1: Hierarquia da pasta **PyLy** e *template* que define esta Hierarquia.

Capítulo 5

Observações Finais

Após dada como concluída toda a elaboração do trabalho com o cumprimento de todos os requisitos pré-definidos, é importante refletir o quão este trabalho ajudou a praticar, enquadrar e acima de tudo aprofundar a matéria relatiava ao *Flex*, leccionada na Unidade Curricular de Processamento de Linguagens.

Notou-se um acréscimo na capacidade de desenvolver/criar expressões regulares para a descrição de padrões de frases. Houve um desenvolvimento, a partir das mesmas, de Processadores de Linguagens Regulares, que filtram/transformam textos com base no conceito de regras de produção Condição-Ação. Criou-se também uma aprendizagem na utilização do próprio FLex em si para gerar filtros de texto em C, bem como um desenvolvimento de uma solução, para tornar todo o programa em algo mais simples, intuitivo e organizado, através de uma estrutura em C que facilita todo o código *Flex*.

A ideia base para resolver as questões deste trabalho prático passou por se interpretar adequadamente o *template* fornecido no enunciado, reconhecendo quais caracteres estavam responsáveis pela distinção das diferentes partes do programa, e quais os diferentes tipos de ficheiros/pastas descritos. Numa fase seguinte passou-se ao tratamento desse *template* com expressões regulares em *Flex* e com o auxilio de um código em C pensado de forma a facilitar todo o programa.

A principal dificuldade surgiu na criação das pastas/ficheiros nas diretorias corretas, para que fosse ao encontro da hierarquia definida no *template*. A função em C responsável pela criação das novas pastas/ficheiros na lista ligada, *insertDirectory*, teria que resolver essa questão de hierarquia. Ao íncio, tinha-se pensado em reverter a Lista Ligada, dado que assim garantiamos que a primeira condição verificada seria a esperada. No entanto, obteve-se dificuldades em executar esta ideia, tendo-se optado por utilizar uma solução mais simples - uma função auxiliar *numberIterations*, que devolve o número de iterações que deve ser feito para se obter a diretoria relativa a uma certa pasta ou ficheiro. Essa ideia tornou-se eficaz dado que se garante que a diretoria que é obtida corresponde sempre à baixa abaixo na linha, ou seja, no ficheiro *template*.

Com o resultado final do projeto e na perspetiva do grupo, pensa-se que se atingiu da melhor forma todos os objetivos propostos e sempre pensando na simplicidade e descomplicação do problema.