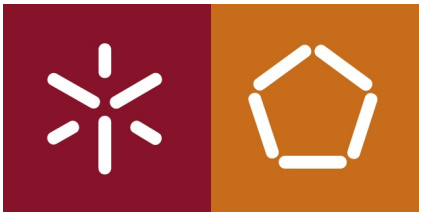

Universidade do Minho
Mestrado Integrado em Engenharia Informática
3ºano - 2º Semestre

Processamento de Linguagens

Trabalho Prático Nº.1 (FLex)

Grupo 33



a83732 – Gonçalo Rodrigues Pinto
a84197 – João Pedro Araújo Parente
a84829 – José Nuno Martins da Costa

5 de Abril de 2020

Conteúdo

1 Introdução **3**

2 Descrição do Problema **3**

3 Padrões de frases a encontrar no texto fonte, através de ERs **5**

4 Acções semânticas a realizar após o reconhecimento dos padrões **9**

5 Estruturas de Dados utilizadas **12**

6 Filtro de texto desenvolvido **14**

7 Conclusão **18**

Lista de Figuras

1 Template-multi-file para um projecto geral de um filtro flex. . 4
2 Representação da estrutura adicional 'dir'. 12
3 Representação da estrutura adicional Dicionário. 13
4 Print do código 1. 14
5 Print do código 2. 15
6 Main do ficheiro flex, é de notar que adicionamos um argumento extra opcional que é o path, se o utilizador quiser pode especificar o path para onde que o projecto criado, caso contrário, cria-o no directório actual. 16
7 Ficheiro Makefile utilizado para compilar e executar o programa mkfromtemplate. 16
8 Print screen do comando tree para a pasta criada pelo programa mkfromtemplate para o template da figura 1. 17
9 Conteúdo do ficheiro Makefile gerado pelo programa mkfromtemplate para o template da figura 1. 17

1 Introdução

No 2º semestre do 3º ano do Curso de Engenharia Informática da Universidade do Minho, existe uma unidade curricular denominada por Processamento de Linguagens, que tem como objectivo ajudar os estudantes a especificar linguagens de domínio específico através de gramáticas e/ou expressões regulares, a desenvolver processadores para essas linguagens, transformar qualquer formato textual num outro formato e especificar e implementar "front-ends" e "back-ends" para qualquer tipo de aplicação.

O presente trabalho tem como principais objectivos aumentar a experiência de uso do ambiente Linux e de algumas ferramentas de apoio à programação, aumentar a capacidade de escrever Expressões Regulares (ER) para descrição de padrões de frases, 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-Acção e por fim utilizar o Flex para gerar filtros de texto em C.

No presente trabalho é composto por diferentes enunciados cabendo a cada grupo escolher aquele que considerava mais interessante, mais desafiante e que gerasse mais motivação em trabalhar, desta forma o nosso grupo escolheu o enunciado número um cuja tema é **Template multi-file** porque julgamos ser o mais cativante e o que nos fornecesse mais vantagens num uso futuro para outro projecto.

2 Descrição do Problema

Para várias projectos de software, é habitual soluções envolvendo vários ficheiros, várias pastas. Exemplo: um ficheiro, uma makefile, um manual, uma pasta de exemplos, etc.

Pretende-se criar um programa "mkfromtemplate", capaz de aceitar um nome de projecto, e um ficheiro descrição de um template-multi-file e que crie os ficheiros e pastas iniciais do projecto.

O template inclui:

- metadados (author, email) a substituir nos elementos seguintes
- tree (estrutura de directorias e ficheiros a criar)
- template da cada ficheiro

O metadado "name"vai ser processado via argumento de linha de comando.

```
=== 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/
- README
- 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;
}
=== README

FIXME: descrição sumária do filtro
```

Figura 1: Template-multi-file para um projecto geral de um filtro flex.

Modo de executar o programa: *mkfromtemplate name template*

Como resultado da execução serão criados os ficheiros e directorias descritos em tree, com os conteúdos definidos nos templates de ficheiro, e as variáveis substituídas.

3 Padrões de frases a encontrar no texto fonte, através de ERs

Um template-multi-file para um projecto geral de um filtro Flex é composto por três partes que se distinguem uns dos outros pela presença dos caracteres '===', um dos partes é designado *meta*, este permite indicar os metadados a substituir nas partes seguintes, uma outra parte é denominado por *tree*, este é uma estrutura de directorias e ficheiros a criar e por fim o última parte com o *nome de ficheiro* onde a informação encontra-se abaixo do mesmo até ao próxima parte encontrado.

De forma a distinguir os diferentes padrões e guardar a informação de cada padrão decidimos criar estados dependentes do contexto esquerdo (designados em Flex por Start Conditions (SC)) para se conseguir identificar e guardar o texto que surja após uma marca de abertura e até se encontrar uma marca de fecho. Tal mudança de contexto, ou de estado de reconhecimento, é forçosa pois em cada parte do trabalho temos Expressões Regulares iguais mas queremos que realizem acções diferentes. Dessa forma (recorrendo a SC) a solução fica elegante e clara.

Assim sendo declaramos 6 SC para captar os diferentes padrões acima referidos:

1. meta : para identificar a parte do documento onde temos informação meta;
2. email : que é basicamente um sub-estado do meta,para identificar-mos o email;
3. autor: para substituir nas próximas oque é basicamente um sub-estado do meta,para identificar-mos o autor;
4. tree: para identificar o padrão tree;
5. namefile: para identificar o nome do ficheiro;
6. ficheiro: para guardar a informação do ficheiro respectivo;

Como foi dito previamente a marca "===" permite uma nova parte do documento template desta forma as Expressões Regulares que permitem tal são:

```
^[ ]*===[ ]*(?i:meta)
^[ ]*===[ ]*(?i:tree)
^[ ]*===[ ]*
```

É de notar que como anteriormente foi dito o template-multi-file recebido pode ser dividido em 3 partes, a parte do meta, tree e o template de cada ficheiro, sendo assim a ordem das 3 expressões regulares acima é relevante pois a 3ª expressão regular apresentada engloba as duas acima, contudo só se pretende que esta aconteça quando os outros dois casos acima não se verifiquem. As ERs apresentadas previnem caso as marcas meta e tree sejam maiúsculas ou minúsculas como também para a existência de vários espaços entre a marca "==="e as palavras.

Para armazenar a informação da marca meta decidimos utilizar as seguintes ERs para identificar o email e o autor, é de realçar que estas tem que surgir após a marca "=== meta":

```
<meta>^[ ]*(?i:email)[ ]*:[ ]*
<meta>^[ ]*(?i:author)[ ]*:[ ]*
```

Após a identificação das marcas anteriores aproveitou-se as vantagens das SC para guardar a informação das variáveis respondentes:

```
<email>[^ \n]+
<autor>[^ \n]+
```

Consequentemente, para identificar as estruturas de directorias e ficheiros a criar utilizamos as seguintes expressões regulares:

```
<tree>\n/[ ]*===[ ]*

<tree>^[ ]*\{\%name\%\}\/

<tree>\{\%name\%\}\.[a-zA-Z]+

<tree>\-+

<tree>\n

<tree>[^\/\n ]+/\/

<tree>[^\/\n ]+
```

As diferentes ERs tem em conta a quantidade de traços antes do nome da directoria ou ficheiro a criar o que equivale ao seu posicionamento dentro das diferentes directorias encontradas como também o caso de criar uma directoria com o nome do projecto recebido por parâmetro e caso o nome do projecto recebido seja também um ficheiro com determinada terminação. Por fim, as últimas expressões garantem que as directorias/ficheiros a criar não possuem determinados caracteres.

Posteriormente, para ficar a conhecer o nome do ficheiro identificado pela marca "==="onde preveniu-se os casos de ser o nome projecto recebido ou outro nome para isso utilizou-se as seguintes expressões regulares:

```
<namefile>[a-zA-Z0-9.\-_]+

<namefile>\{\%name\%\}\.[a-zA-Z]+
```

E por fim as expressões regulares abaixo apresentadas permitem identificar o conteúdo a escrever nos ficheiros:

```
<ficheiro>^[ ]*===[ ]*
<ficheiro>\{\%name\%\}
<ficheiro>\{\%author\%\}
<ficheiro>\{\%email\%\}
<ficheiro>\{
<ficheiro>\=
<ficheiro>[^{\=]*
```

As 3 últimas expressões regulares podem parecer um pouco invulgares, contudo estas serviram para escrever dentro do ficheiro. O que seria de se esperar era escrever uma expressão regular mais simples e intuitiva como .* que nos daria qualquer caractere zero ou mais vezes, o que levaria a encontrar o texto todo, contudo existem algumas palavras reservadas, tal como o caso de '{%name%}', que quando aparece pretende-se que seja substituída por algo mais concreto para isso a última regra captura todos caracteres menos o '{' que é o caractere por qual as palavras reservadas começam e o caractere '=' que pode ser o inicio de '===' que significa o inicio de uma nova parte no template. No entanto podem existir esses dentro do ficheiro, por isso mesmo temos a antepenúltima e penúltima regra.

4 Acções semânticas a realizar após o reconhecimento dos padrões

As expressões regulares apresentadas anteriormente pretendem de forma concisa e flexível identificar cadeias de caracteres do nosso interesse, contudo falta apresentar e descrever as acções semânticas realizadas mediante a identificação através de ERs.

Inicialmente após a identificação das marcas inicializou-se as diferentes Start Conditions correspondentes. Respeitando, a ordem apresentada acima de identificação das diferentes ERs concretizou-se da seguinte forma:

```
^[ ]*===[ ]*(?i:meta)          {BEGIN meta;}
^[ ]*===[ ]*(?i:tree)          {BEGIN tree;}
^[ ]*===[ ]*                   {BEGIN namefile;}
```

De modo análogo, para identificação da marca email e autor realizou-se a seguinte acção de inicializar:

```
<meta>^[ ]*(?i:email)[ ]*:[ ]*       {BEGIN email;}
<meta>^[ ]*(?i:author)[ ]*:[ ]*      {BEGIN autor;}
```

Após identificar a marca email e autor decidiu-se guardar a informação presente após a marca em duas variáveis, uma para o email e outra para o autor pois estas variáveis irão ser necessárias para efectuar a substituição quando existe referencia de tal, desta forma:

```
<email>[^ \n]+     {var_email=strdup(yytext); BEGIN meta;}
<autor>[^ \n]+     {var_autor=strdup(yytext); BEGIN INITIAL;}
```

Posteriormente, pretendeu-se guardar a estrutura de directorias e ficheiros a criar, partindo das expressões regulares descritas acima para esta etapa efectuou-se as seguintes acções semânticas que utilizam funções que posteriormente neste relatório serão explicadas, assim sendo:

```
<tree>\n/[ ]*===[ ]*            {BEGIN INITIAL;}

<tree>^[ ]*\{\%name\%\}\/       {inseredir(nome_projeto); criapasta();}

<tree>\{\%name\%\}\.[a-zA-Z]+   {char *result = malloc(strlen(nome_projeto) + strlen(yytext+8) + 1); strcpy(result, nome_projeto);
                                    strcat(result, yytext+8);inseredir(result); criaficheiro();}

<tree>\-+                       {point_dirs+=strlen(yytext);}

<tree>\n                        {point_dirs=0;}

<tree>[^\/\n ]+/\/              {inseredir(yytext);criapasta();}

<tree>[^\/\n ]+                 {inseredir(yytext);criaficheiro();}
```

De seguida, pretendeu-se ficar a conhecer o template de cada ficheiro para isso foi necessário conhecer o nome do mesmo utilizando a SC "namefile"permitindo assim abrir o ficheiro para posteriormente inserir a informação presente no template.

```
<namefile>\{\%name\%\}\.[a-zA-Z]+   {char *result = malloc(strlen(nome_projeto) + strlen(yytext+8) + 1);
                                        strcpy(result, nome_projeto); strcat(result, yytext+8);
                                        if (abreficheiro(result)==1) BEGIN ficheiro; else BEGIN INITIAL;}

<namefile>[a-zA-Z0-9.\-_]+          { if (abreficheiro(yytext)==1) BEGIN ficheiro; else BEGIN INITIAL;}
```

Por fim, após identificação do ficheiro onde se pretende guardar a informação procedeu-se com o auxilio das ERs (acima apresentadas) filtrar o conteúdo do template e executar as respectivas acções que permitem efectuar a escrita do documento em causa. Em suma, as acções realizadas foram as seguintes:

<ficheiro>^[ ]*===[ ]*	{fclose(fp);fp=NULL;BEGIN namefile;}
<ficheiro>\{\%name\%\}	{escreveficheiro(nome_projeto);}
<ficheiro>\{\%autor\%\}	{escreveficheiro(var_autor);}
<ficheiro>\{\%email\%\}	{escreveficheiro(var_email);}
<ficheiro>\{	{escreveficheiro(yytext);}
<ficheiro>\=	{escreveficheiro(yytext);}
<ficheiro>[^{\=]*	{escreveficheiro(yytext);}

5 Estruturas de Dados utilizadas

No presente trabalho criou-se duas estruturas de dados para o auxilio de processamento dos dados e respectivo armazenamento dos mesmos.

A primeira estrutura utilizada com a designação *'dir'* auxiliou-nos a construir e representar os directórios dentro do programa desenvolvido. Esta estrutura *'dir'* é basicamente um array de strings, onde a primeira posição representa o primeiro nível do directório, a segunda posição o segundo nível, etc. À medida que efectuamos a interpretação da tree de directórios que nos é fornecido no template-multi-file preenchemos esta estrutura e ao mesmo tempo efectuamos a respectiva criação do ficheiro ou pasta composto por todas as strings dentro da estrutura *'dir'*. Abaixo podemos ver uma pequena representação da estrutura que criamos.

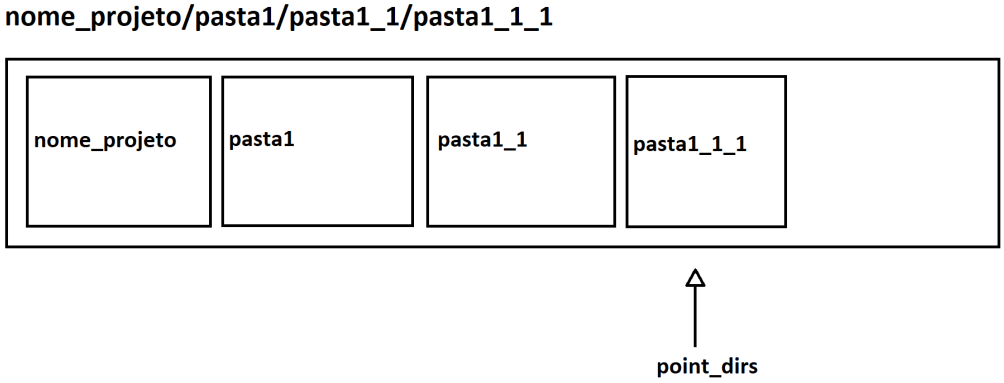

Figura 2: Representação da estrutura adicional 'dir'.

Na última parte do template-multi-file pode conter o nome do ficheiro e a referente informação a escrever dentro do mesmo, para tal foi necessário a capacidade de apenas sabendo o nome obter o seu path para o podermos abrir e escrever para ele. A estrutura anterior resolve-nos o problema da criação dos ficheiros/pastas, mas como esta está em constante evolução durante a execução do programa e apenas consegue guardar um directório de cada vez, não serve para o nosso novo problema.

Portanto criamos uma estrutura nova a qual chamamos dicionário que vai guardar para cada ficheiro uma chave que é o nome de ficheiro e o valor que é o seu path completo, ainda criamos algumas funções auxiliares tais como 'adicionar_dicionario','existe_dicionario',etc, que nos dão basicamente uma pequena API para melhor acesso ao dicionário. Em termos de C, o dicionário é um apontador para um array onde cada posição do array tem outro apontador para um array de strings de duas posições sendo a primeira posição a chave e a segunda o path completo.

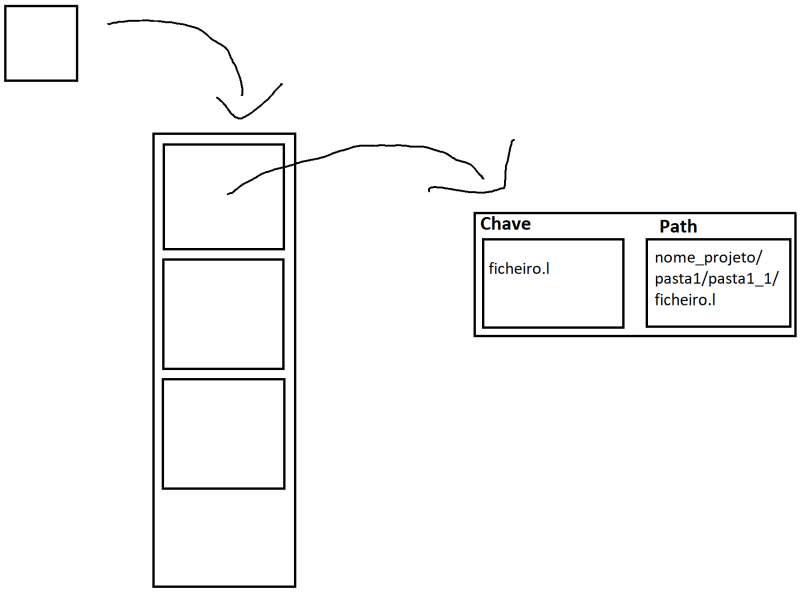

Figura 3: Representação da estrutura adicional Dicionário.

6 Filtro de texto desenvolvido

```
%{
#include <sys/stat.h>
#include <stdio.h>

char* nome_projeto;
char*var_email="";
char*var_autor="";
char*path_inicial="";

//ficheiro aberto
FILE * fp;

// temp para construir paths
char** dir;
int size_dirs=1;
int point_dirs=0;


// guarda os paths , nome file -> path completo 1
char *** dicionario_ficheiros;
int size_dicionario=1;
int pointer_dicionario=0;

// dado uma chave e  p paht completo adiciona ao dicionario
void adiciona_dicionario(char* chave, char * path_completo){

    if (pointer_dicionario>=size_dicionario){
        size_dicionario*=2;
        dicionario_ficheiros= (char ***) realloc(dicionario_ficheiros, sizeof(char *) * size_dirs);
    }

    dicionario_ficheiros[pointer_dicionario]= (char **) malloc( sizeof(char *) * 2);

    dicionario_ficheiros[pointer_dicionario][0]=strdup(chave);
    dicionario_ficheiros[pointer_dicionario][1]=strdup(path_completo);
    pointer_dicionario++;
}
// retorna o indice se existir ou -1 se nao existir
int existe_dicionario(char * chave){

    //printf("Conteudo Dicionario:\n ");
    for(int i=0;i<pointer_dicionario;i++){
        if (strcmp(chave,dicionario_ficheiros[i][0])==0) return i;
    }

    return -1;
}
// obtem path apartir do indice
char * get_path( int indice){

    if (indice>=0 && indice <=pointer_dicionario)
        return dicionario_ficheiros[indice][1];

    return NULL;
}
// cria a pasta juntado todas as strings(nivel de diretorias) que estao no array global dir
void criapasta(){

    if (point_dirs<size_dirs){


        int size=1+strlen(path_inicial); // null terminator
        for(int i=0;i<point_dirs+1;i++){
            size+=strlen(dir[i])+1;
        }

        char *result = malloc(sizeof(char)*size);
        strcat(result,path_inicial);

        for(int i=0;i<point_dirs+1;i++){
            strcat(result,dir[i]);
            strcat(result,"/");
        }

        result[strlen(result)-1]='\0';

        int err = mkdir(result,0777);

    }
}
```

Figura 4: Print do código 1.

```
// cria o ficheiro juntado todas as strings(nivel de diretorias) que estao no array global dir
void criaficheiro(){

    if (point_dirs<size_dirs){

        int size=1+strlen(path_inicial); // null terminator + touch + path inicial
        for(int i=0;i<point_dirs+1;i++){
            size+=strlen(dir[i])+1;
        }

        char *result = malloc(sizeof(char)*size);
        strcat(result,path_inicial);

        for(int i=0;i<point_dirs+1;i++){
            strcat(result,dir[i]);
            strcat(result,"/");
        }
        result[strlen(result)-1]='\0';

        FILE* fp = fopen (result, "w");
        if (fp!=NULL){
            fclose(fp);
            adiciona_dicionario(dir[point_dirs],result);
        }
        else printf("Detetado erro a criar ficheiro");
    }
}

// duplica o tamanho do array global dir
void resize(){
    size_dirs*=2;
    dir=realloc(dir,sizeof(char*) * size_dirs);
}
```

```
// insere no array dir a string
void inseredir(char * str){
    if (size_dirs*2>point_dirs){
        if (point_dirs>= size_dirs) resize();
        dir[point_dirs]=strdup(str);
    }
    else printf("Error:\n   Um ou mais pastas/ficheiros não foram criados devido a sintaxe incorreta na árvore");
}

// retorna 1 se abriu o ficheiro para o fp 0 se nao conseguiu
int abreficheiro(char * chave){
    int r=existe_dicionario(chave);

    if (r!=-1){
        fp=fopen(get_path(r),"a");
        if (fp!=NULL){
            return 1;
        }
    }

    return 0;
}

// escreve para o ficheiro que esta aberto
void escreveficheiro(char * str){
    if (fp!=NULL)
        fwrite(str,sizeof(char),strlen(str),fp);
}

%}
```

```
%x meta email autor tree   namefile ficheiro

Abrev ER
%%
^[ ]*===[ ]*(?i:meta)                   {BEGIN meta;}
^[ ]*===[ ]*(?i:tree)                   {BEGIN tree;}
^[ ]*===[ ]*                            {BEGIN namefile;}


<meta>^[ ]*(?i:email)[ ]*:[ ]*          {BEGIN email;}
<meta>^[ ]*(?i:author)[ ]*:[ ]*         {BEGIN autor;}

<email>[^ \n]+                          {var_email=strdup(yytext);BEGIN meta;}
<autor>[^ \n]+                          {var_autor=strdup(yytext);BEGIN INITIAL;}



<tree>\n/[ ]*===[ ]*                    {BEGIN INITIAL;}

<tree>^[ ]*\{\%name\%\}\/               {inseredir(nome_projeto); criapasta();}
<tree>\{\%name\%\}\.[a-zA-Z]+           {char *result = malloc(strlen(nome_projeto) + strlen(yytext+8) + 1);
                                        strcpy(result, nome_projeto);strcat(result, yytext+8);
                                        inseredir(result); criaficheiro();}
<tree>\-+                               {point_dirs+=strlen(yytext);}
<tree>\n                                {point_dirs=0;}
<tree>[^\/\n ]+/\/                      {inseredir(yytext);criapasta();}
<tree>[^\/\n ]+                         {inseredir(yytext);criaficheiro();}
```

```
<namefile>\{\%name\%\}\.[a-zA-Z]+       {char *result = malloc(strlen(nome_projeto) + strlen(yytext+8) + 1);
                                        strcpy(result, nome_projeto);
                                        strcat(result, yytext+8);
                                        if (abreficheiro(result)==1) BEGIN ficheiro; else BEGIN INITIAL;}


<namefile>[a-zA-Z0-9.\-_]+              { if (abreficheiro(yytext)==1) BEGIN ficheiro; else BEGIN INITIAL;}



<ficheiro>^[ ]*===[ ]*                  {fclose(fp);fp=NULL;BEGIN namefile;}

<ficheiro>\{\%name\%\}                  {escreveficheiro(nome_projeto);}
<ficheiro>\{\%author\%\}                {escreveficheiro(var_autor);}
<ficheiro>\{\%email\%\}                 {escreveficheiro(var_email);}

<ficheiro>\{                            {escreveficheiro(yytext);}
<ficheiro>\=                            {escreveficheiro(yytext);}

<ficheiro>[^{\=]*                       {escreveficheiro(yytext);}


.|\n                                    {;}

%%
```


Figura 5: Print do código 2.

```c
int yywrap()
{
    return(1);
}

int main(int arg, char** args)
{
    if(arg==2  || arg==3){

        if (arg==3) path_inicial=strdup(args[2]);

        dir = (char **) malloc(sizeof(char *) * size_dirs);
        dicionario_ficheiros = (char ***) malloc(sizeof(char **) * size_dirs);

        nome_projeto=strdup(args[1]);

        yylex();

    }
    return 0;
}
```

Figura 6: Main do ficheiro flex, é de notar que adicionamos um argumento extra opcional que é o path, se o utilizador quiser pode especificar o path para onde que o projecto criado, caso contrário, cria-o no directório actual.

```
build:
    flex tp1.l
    gcc lex.yy.c -o mkfromtemplate


run1:
    ./mkfromtemplate TP1 < Templates/1

run2:
    ./mkfromtemplate TP1  "/home/jpedro/Desktop/" < Templates/2
```

Figura 7: Ficheiro Makefile utilizado para compilar e executar o programa mkfromtemplate.

```
jpedro@pop-os:~/Desktop$ tree TP1
TP1
├── doc
│   └── TP1.md
├── exemplo
├── Makefile
├── README
└── TP1.fl

2 directories, 4 files
```

Figura 8: Print screen do comando tree para a pasta criada pelo programa mkfromtemplate para o template da figura 1.

```
TP1 > M Makefile
 1
 2
 3   TP1: TP1.fl
 4       flex TP1.fl
 5       cc -o TP1 lex.yy.c
 6
 7   install: TP1
 8       cp TP1 /usr/local/bin/
 9
10
```

Figura 9: Conteúdo do ficheiro Makefile gerado pelo programa mkfromtemplate para o template da figura 1.

7 Conclusão

O presente relatório descreveu, de forma sucinta, o desenvolvimento do programa "mkfromtemplate"capaz de aceitar um nome de projecto e um ficheiro descrição de um template-multi-file que cria os ficheiros e pastas iniciais do projecto.

Após a realização deste trabalho, ficamos conscientes das potencialidades que as Expressões Regulares podem trazer ao serem bem construídas e pensadas, criam (com auxilio de ferramentas) Processadores de Linguagens Regulares capazes de filtrar e transformar textos com base no simples conceito de regras de produção Acção-Condição

Consideramos que os principais objectivos foram cumpridos.

Sentimos que a realização deste trabalho prático consolidou os nossos conhecimentos no ambiente Linux, de escrever Expressões Regulares para descrever padrões de frases como também consolidou os nossos conhecimentos na ferramenta Flex para gerar filtros de texto mais propriamente em C.