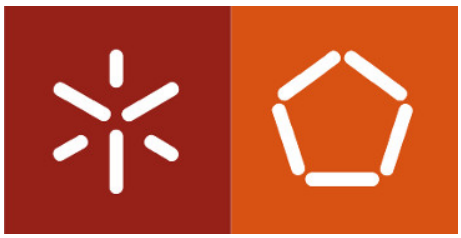
Processamento de Linguagens

Trabalho Prático 1

5 de Abril de 2020

Grupo nr.	36
a83899	André Morais
a85954	Luís Ribeiro
a84783	Pedro Rodrigues



Mestrado Integrado em Engenharia Informática
Universidade do Minho

Conteúdo

1 Introdução **2**
1.1 Contexto . . . 2
1.2 Problema . . . 2
1.3 Objetivos . . . 2

2 Análise e Especificação **4**
2.1 Requisitos: . . . 4

3 Concepção e Codificação da Resolução **4**

4 Testes e Resultados **11**
4.1 Teste 1 - Flex . . . 12
4.2 Teste 2 - C . . . 13
4.3 Teste 3 - Check Bugs . . . 14

5 Conclusão **15**

A Código do Programa **16**
A.1 vars.fl . . . 16
A.2 filtro.fl . . . 18
A.3 template.c . . . 22
A.4 Template Flex . . . 24
A.5 Template C . . . 26
A.6 Template para procurar bugs em meta e tree . . . 28

1 Introdução

1.1 Contexto

Este relatório foi produzido em conformidade com a UC de **Processamento de Linguagens**, correspondente ao segundo semestre do terceiro ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho.

1.2 Problema

Para vários projetos de desenvolvimento, é habitual haver soluções envolvendo vários ficheiros e várias pastas, como `Makefile`, `README` ou uma paste de exemplos.

Pretende-se, então, criar um programa **mkfromtemplate**, capaz de aceitar um nome de projeto e um ficheiro de descrição (template) e que crie os ficheiros e as pastas do peojeto, bem como escreva o conteúdo dos ficheiros pretendido.

O template deverá ser constituído por:

- metadados (author e email)
- tree (estrutura de diretorias e ficheiros a criar)
- template de cada ficheiro

Um exemplo do template pode ser encontrado em A.4.

1.3 Objetivos

Este projeto tem como principais objetivos:

- 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-Ação*;
- utilizar o Flex para gerar *filtros* de texto em C.

2 Análise e Especificação

Após uma análise ao problema, consegue-se identificar uma série de requisitos necessários para a resolução do problema.

2.1 Requisitos:

- Alterar nome das variáveis no ficheiro template;
- Identificar os diferentes tipos de dados do template (meta, tree e conteúdo de cada ficheiro);
- Guardar em memória, numa estrutura de dados adequada, a informação acerca da estrutura do projeto (*tree* de ficheiros e pastas).
- Utilizar a estrutura de dados e a informação guardada para criar a estrutura final.

3 Concepção e Codificação da Resolução

Observando o problema e os requisitos, precisamos, primeiramente, de alterar o nome das variáveis. Para isto foi utilizado um filtro flex especialemente para alterar o nome destas variáveis. O processo é simples, é criada uma HashTable, usando as bibliotecas da glib, onde é guardada a informação de cada metadado, da forma (Chave, Valor).

Exemplo:

`author: Pedro` → gera um par (auhtor, Pedro).

```
<META>^[a-zA-Z0-9_]+/:[ ]        {key=strdup(yytext);BEGIN VALUE;}
<VALUE>[^\n]*      {value=strdup(yytext+2);addToTable(key,value);BEGIN META;}
```

Para tal, era necessário diferenciar o grupo `METADADOS` dos restantes grupos (onde serão substituídas as variáveis). O mais sensato é utilizar uma *start condition* como mostramos a seguir:

```
<*>\{%/[a-zA-Z0-9_]+%\}     {BEGIN SUBS;}
^===[ ]meta                 {initializeHash();BEGIN META;}
<SUBS>{
[a-zA-Z0-9_]+/%\}           {printf(getValue(yytext));}
%\}                         {BEGIN TF;}
}
<META>^===[ ]tree           {ECHO;BEGIN TF;}
```

Com este filtro, obteremos o nosso template com todas as variáveis alteradas para o seu respetivo nome e sem o grupo `METADADOS`, uma vez que este é inútil nesta fase do processo.

Agora, temos um ficheiro totalmente útil para começar a estruturar o projeto. O primeiro passo é descobrir como identificar os diferentes grupos do ficheiro template (tree de ficheiros e pastas e conteúdo de ficheiros).

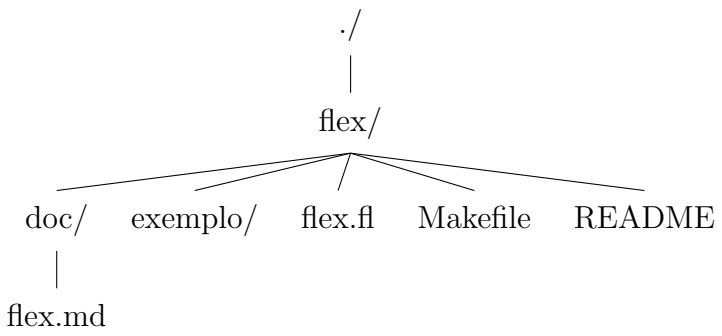
Tal como no requisito anterior, o mais sensato a utilizar é *start conditions* e identificar a expressão regular que apanhe o inicio e fim de cada grupo:

```
^===[  ]tree                {initializeTree();BEGIN TREE;}
<TREE>{
[^\n\\\*\?\<\>\|\/]+\/$     {insertDir(yytext);BEGIN PROF;}
[^\n\\\*\?\<\>\|\/]+        {insertFile(yytext);BEGIN PROF;}
.|\n                        {;}
}
<PROF>{
^===[  ][^\n]+              {makeTree();file=strdup(yytext+4);BEGIN FILES;}
^-+[  ]                     {nprof=strlen(yytext)-1;BEGIN TREE;}
.|\n                        {;}
}
<FILES>{
^===[ ][^\n]+               {file=strdup(yytext+4);}
^[^\n]*                     {writef(yytext,file);}
^\n                         {writef("",file);}
.|\n                        {;}
}
.|\n                        {;}
```

Como se pode reparar, é apresentada uma *start condition* que não era espectável: **PROF**. Esta é necessária para calcular a profundidade de um ficheiro na árvore.

Com estes pedaços de código, resolve-se o problema de identificar os grupos distintos presentes no ficheiro. No entanto, é preciso iterar sobre a informação presente em cada grupo.

Respeitando o que foi enunciado nos requisitos, é necessário decidir a estrutura de dados que mais se adequa para guardar a informação da localização de cada ficheiro, e, tal como o nome do grupo (*tree*) indica, a melhor estrutura de dados a ser usada é uma árvore, mais concretamente uma `N-ARY TREE`, ou seja, uma árvore em que cada nodo pode ter N nodos. Assim, para o ficheiro template de exemplo apresentado e com nome de projeto *flex*, a árvore seria representada da seguinte forma:



Após uma análise no mundo da internet, encontra-se uma implementação deste tipo de árvores apresentada pela `GLIB`, que pode ser usada para representar este tipo de problema.

Depois de obtida uma implementação desta árvore, é trivial que o próximo passo é identificar o tipo de nodo que será adicionado (ficheiro ou diretoria) e guardar o nome na profundidade certa da árvore.

```
GNode* addToTree(char* nome, int file, GNode* parent){
    Node* new = malloc(sizeof(struct node));
    new->descricao = strdup(nome);
    new->file = file;
    GNode* node;
    if (parent==NULL){
        node = g_node_new(new);
        g_node_append(dirs,node);
    }
    else{
        node = g_node_new(new);
        g_node_append(parent,node);
```

```
    }
    return node;
}

void insertDir(char* nome){
    while(nprof!=prof){
        parent=parent->parent;
        prof--;
    }
    parent=addToTree(nome,0,parent);
    prof++;
    nprof=0;
}

void insertFile(char* nome){
    while(nprof!=prof){
        parent=parent->parent;
        prof--;
    }
    addToTree(nome,1,parent);
    nprof=0;
}
```

Neste algoritmo é importante salientar que *prof* representa a **profundidade da última diretoria criada** e *nprof* representa a **profundidade da diretoria ou ficheiro a ser criado**.

Estas funções escritas em C, são chamadas quando são identificadas pelas expressões regulares apresentadas no código anterior a este.

Com a árvore completa, o processo de criação de pastas e ficheiros é trivial.

```
void transformTree (GNode* dir,char* path){
    if (dir!=NULL){
        GNode* child = g_node_first_child(dir);
        char command [1024];
        while(child){
            Node* dados = child->data;
            if (dados->file) sprintf(command,"touch %s%s",path,dados->descricao);
            else sprintf(command,"mkdir %s%s",path,dados->descricao);
            system(command);
            if (!dados->file){
```

```
                sprintf(command,"%s%s",path,dados->descricao);
                transformTree(child,command);
            }
            child=child->next;
        }
    }
}


void makeTree(){
    char pwd [64];
    getcwd(pwd,sizeof(pwd));
    strcat(pwd,"/");
    transformTree(dirs,pwd);
}
```

Seguindo, novamente, a estrutura criada nos requisitos, o próximo passo é processar esta informação presente na árvore. Para tal, é necessário, primeiro, encontrar as expressões regulares que apanham o nome do ficheiro a editar.

```
^===[ ][^\n]+                                   {file=strdup(yytext+4);}
^[^\n]+                                         {writef(yytext,file);}
^\n                                             {writef("",file);}
```

Com as ERs apresentadas em cima, obtemos o nome do ficheiro em *file* e conseguimos escreeer o resultado do yytext nesse mesmo ficheiro.

```
int getFilePath (GNode* dir,char* f,char* path){
    if (dir!=NULL){
        GNode* child = g_node_first_child(dir);
        while(child){
            Node* dados = child->data;
            if (dados->file){
                if (strcmp(dados->descricao,f)==0){
                    strcat(path,f);
                    return 1;
                }
            }
            else{
```

```
                strcat(path,dados->descricao);
                if (getFilePath(child,f,path)==1) return 1;
                else{
                    int nc = strlen(path)-strlen(dados->descricao);
                    path[nc]='\0';
                }
            }
            child=child->next;
        }
    }
    return 0;
}



int writef (char* s, char* f){
    char path [1024];
    getcwd(path,sizeof(path));
    strcat(path,"/");
    if (getFilePath(dirs,f,path)==1){
        FILE *f = fopen(path,"a");
        fprintf(f,"%s\n",s);
        fclose(f);
    }
    return 0;
}
```

Com todo o processo enunciado anteriormente, obtém-se dois filtros distintos que precisam de ser encadeados. O segundo deverá processar o resultado do primeiro, que por sua vez processa o texto enviado como argumento. Uma opção é criar definir o prefixo de cada um dos filtros e criar um programa em c que junte os dois. Para simplificar as coisas, define-se o primeiro filtro de alteração de variáveis como `vars` e o responsável por criar as diretorias e ficheiros como `tmpl`. Isto faz com que sejam obtidas duas funções (**varslex()** e **tmpllex()**) que podem ser usadas para chamar os respetivos filtros em C.

O processo de junção destes dois filtros deverá seguir os seguintes passos:

Primeira fase:

1. Redirecionar stdout para ficheiro temporário (pipe)

2. Redirecionar stdin para ficheiro recebido como argumento
3. Correr o primeiro filtro para alterar as variáveis com a função `varslex()`
4. Redirecionar o stdin e stdout para os descritores originais

Segunda fase:

1. Redirecionar stdin para o ficheiro temporário (pipe)
2. Correr o segundo filtro para criar diretorias e ficheiros com a função `tmpllex()`
3. Redirecionar o stdin para o descritor original
4. Elimiar o ficheiro temporário (pipe)

4 Testes e Resultados

Para testar o programa foram criados três ficheiros template:

- Template para projeto em Flex - A.4
- Template para projeto em C - A.5
- Template à procura de bugs na definição de meta e tree - A.6

Os ficheiros de testes serão apresentados no anexo.

4.1 Teste 1 - Flex

Figura 1: Árvore de diretorias e ficheiros criados

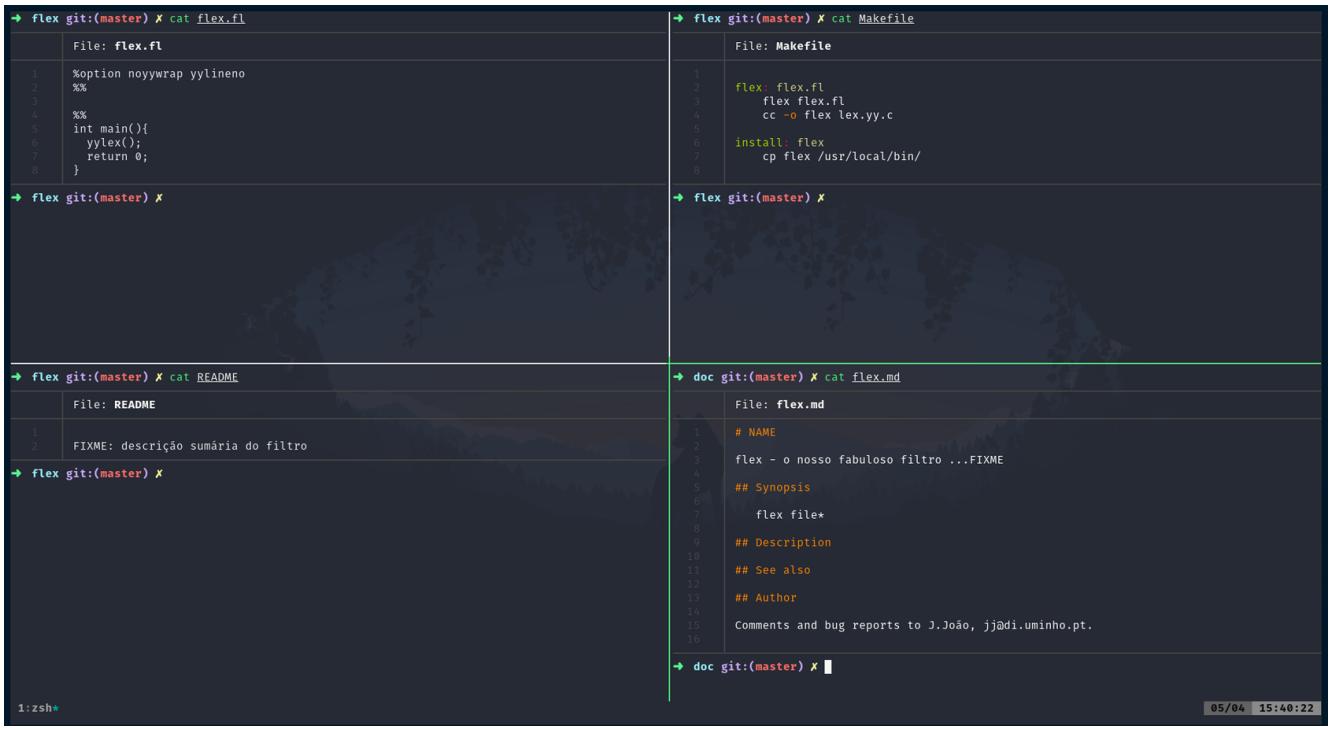


Figura 2: Conteúdo dos ficheiros criados

4.2 Teste 2 - C

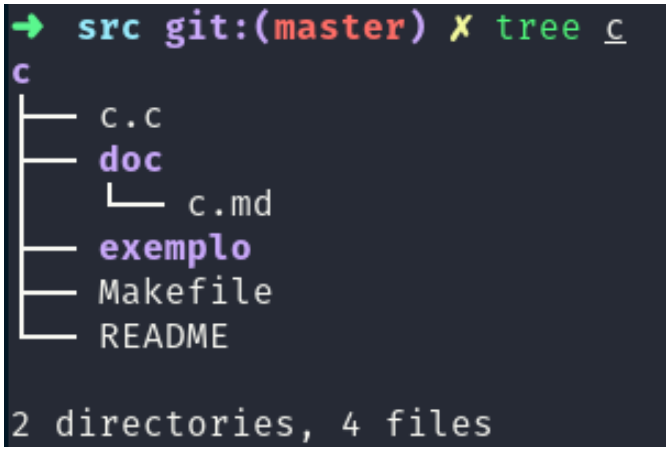


Figura 3: Árvore de diretorias e ficheiros criados

Figura 4: Conteúdo dos ficheiros criados

4.3 Teste 3 - Check Bugs

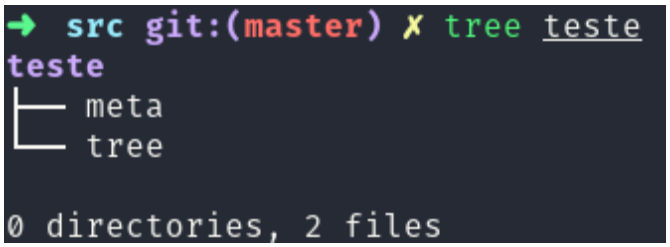


Figura 5: Árvore de diretorias e ficheiros criados

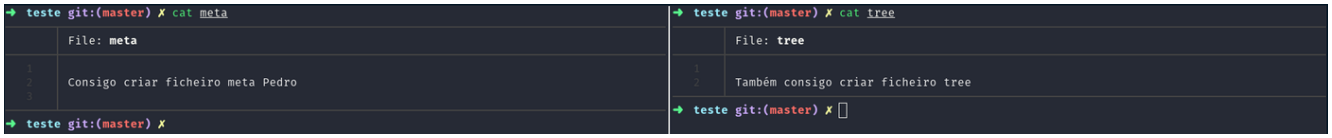


Figura 6: Conteúdo dos ficheiros criados

5 Conclusão

Depois de terminado o primeiro trabalho prático da Unidade Curricular de Processamento de Linguagens, concluímos que a realização deste permitiu-nos consolidar os conhecimentos estudados nas aulas da disciplina. Houve assim, uma aprendizagem notória relativamente ao analisador léxico (flex) e às suas funcionalidades, desde as Expressões Regulares (ERs) iniciais até ao uso de Start Conditions e look-aheads.

O uso da biblioteca Glib ajudou-nos bastante na parte da estruturação de memória, pois esta fornece estruturas de dados, tais como a GTree e a GHashTable, e funções relativas a essas estruturas.

Em jeito de conclusão, consideramos o nosso desempenho bastante satisfatório, pois foi possível responder a todas as questões pedidas pelo docente e os objetivos foram concluídos.

A Código do Programa

A.1 vars.fl

```
%option noyywrap yylineno prefix="vars"

%x META VALUE SUBS TF
%{
  #include <stdio.h>
  #include <string.h>
  #include <glib.h>

  extern char* name;
  GHashTable* metadata;
  char *key,*value;

  GHashTable* initializeHash();
  void addToTable(char* k, char*v);
  char* getValue(char* k);
%}

%%

<*>\{%/[a-zA-Z0-9_]+%\}     {BEGIN SUBS;}
^===[ ]meta                 {initializeHash();BEGIN META;}
<SUBS>{
[a-zA-Z0-9_]+/%\}           {printf(getValue(yytext));}
%\}                         {BEGIN TF;}
}
<META>{
#[^\n]+                     {;}                    // comentario nos metadados
^[a-zA-Z0-9_]+/:[  ]        {key=strdup(yytext);BEGIN VALUE;}
^===[ ]tree                 {ECHO;BEGIN TF;}
.|\n                        {;}
}
<VALUE>[^\n]*               {value=strdup(yytext+2);addToTable(key,value);BEGIN META;}
```

```
%%

GHashTable* initializeHash (){
  metadata = g_hash_table_new(g_str_hash,g_str_equal);
  addToTable("name",name);
  return metadata;
}

void addToTable(char* k, char* v){
  g_hash_table_insert(metadata,k,v);
}

char* getValue(char* k){
  return g_hash_table_lookup(metadata,k);
}
```

A.2 filtro.fl

```
%option noyywrap yylineno prefix="tmpl"

%x META TREE PROF FILES
%{
#include <stdio.h>
#include <string.h>
#include <glib.h>

typedef struct node{
    char* descricao;
    int file;
}Node;

GNode *dirs,*parent;
int prof=0,nprof=0;
char *file,*pwd;

void insertFile(char* nome);
void insertDir(char* nome);
void makeTree();
void initializeTree();
int writef (char* s, char* f);
%}

%%

^===[  ]tree                        {initializeTree();BEGIN TREE;}
<TREE>{
[^\n\\\*\?\<\>\|\/]+\/$             {insertDir(yytext);BEGIN PROF;}
[^\n\\\*\?\<\>\|\/]+                {insertFile(yytext);BEGIN PROF;}
.|\n                                {;}
}
<PROF>{
^===[  ][^\n]+                      {makeTree();file=strdup(yytext+4);BEGIN FILES;}
^-+[  ]                             {nprof=strlen(yytext)-1;BEGIN TREE;}
.|\n                                {;}
```

```
}
<FILES>{
^===[ ][^\n]+                       {file=strdup(yytext+4);}
^[^\n]*                             {writef(yytext,file);}
^\n                                 {writef("",file);}
.|\n                                {;}
}
.|\n                                {;}

%%

GNode* addToTree(char* nome, int file, GNode* parent){
    Node* new = malloc(sizeof(struct node));
    new->descricao = strdup(nome);
    new->file = file;
    GNode* node;
    if (parent==NULL){
        node = g_node_new(new);
        g_node_append(dirs,node);
    }
    else{
        node = g_node_new(new);
        g_node_append(parent,node);
    }
    return node;
}

void insertDir(char* nome){
    while(nprof!=prof){
        parent=parent->parent;
        prof--;
    }
    parent=addToTree(nome,0,parent);
    prof++;
    nprof=0;
}

void insertFile(char* nome){
```

```
    while(nprof!=prof){
        parent=parent->parent;
        prof--;
    }
    addToTree(nome,1,parent);
    nprof=0;
}

void initializeTree(){
    dirs = g_node_new(NULL);
}

void transformTree (GNode* dir,char* path){
    if (dir!=NULL){
        GNode* child = g_node_first_child(dir);
        char command [1024];
        while(child){
            Node* dados = child->data;
            if (dados->file) sprintf(command,"touch %s%s",path,dados->descricao);
            else sprintf(command,"mkdir %s%s",path,dados->descricao);
            system(command);
            if (!dados->file){
                sprintf(command,"%s%s",path,dados->descricao);
                transformTree(child,command);
            }
            child=child->next;
        }
    }
}


void makeTree(){
    char pwd [64];
    getcwd(pwd,sizeof(pwd));
    strcat(pwd,"/");
    transformTree(dirs,pwd);
}
```

```
int getFilePath (GNode* dir,char* f,char* path){
    if (dir!=NULL){
        GNode* child = g_node_first_child(dir);
        while(child){
            Node* dados = child->data;
            if (dados->file){
                if (strcmp(dados->descricao,f)==0){
                    strcat(path,f);
                    return 1;
                }
            }
            else{
                strcat(path,dados->descricao);
                if (getFilePath(child,f,path)==1) return 1;
                else{
                    int nc = strlen(path)-strlen(dados->descricao);
                    path[nc]='\0';
                }
            }
            child=child->next;
        }
    }
    return 0;
}

int writef (char* s, char* f){
    char path [1024];
    getcwd(path,sizeof(path));
    strcat(path,"/");
    if (getFilePath(dirs,f,path)==1){
        FILE *f = fopen(path,"a");
        fprintf(f,"%s\n",s);
        fclose(f);
    }
    return 0;
}
```

A.3 template.c

```
#include <stdio.h>
#include <string.h>
#include <glib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

char* name;
int varslex();
int tmpllex();

int changeVariables (char* temp){
  int in = open(temp,O_RDONLY);
  int out = open("pipe",O_CREAT | O_RDWR,0666);
  printf("Changing Variables..................\n");
  int fdout = dup(1);
  dup2(out,1);
  close(out);
  int fdin = dup(0);
  dup2(in,0);
  close(in);
  varslex();
  dup2(fdout,1);
  close(fdout);
  dup2(fdin,0);
  close(fdin);
  printf("Variables changed\n");
}

int makeTemplate (){
  printf("Reading from pipe...................\n");
  int in = open("pipe",O_RDONLY);
  int fdin=dup(0);
  dup2(in,0);
  close(in);
```

```
  tmpllex();
  dup2(fdin,0);
  close(fdin);
  printf("Template made\n");
}

int main (int argc, char* argv[]){
  char* temp = strdup(argv[2]);
  name = strdup(argv[1]);
  changeVariables(temp);
  makeTemplate();
  unlink("pipe");
  return 0;
}
```

A.4 Template Flex

```
=== 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
```

A.5 Template C

```
=== meta

email: jj@di.uminho.pt
author: J.João
compiler: gcc
flags: -Wall -g -ggdb

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

=== tree

{%name%}/
- {%name%}.c
- doc/
-- {%name%}.md
- exemplo/
- README
- Makefile

=== Makefile

{%name%}: {%name%}.c
        {%compiler%} {%flags%} -o {%name%} {%name%}.c

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

clean:
  rm {%name%}

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

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

## Synopsis
```

```
   {%name%} file*

## Description

## See also

## Author

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

=== {%name%}.c
#include <stdio.h>

int main(){
  return 0;
}
=== README

FIXME: descrição sumária do filtro
```

A.6 Template para procurar bugs em meta e tree

```
=== meta

author: Pedro
email: pedro@gmail.com

=== tree

{%name%}/
- meta
- tree

=== meta

Consigo criar ficheiro meta {%author%}

=== tree

Também consigo criar ficheiro tree
```