



UNIVERSIDADE DO MINHO

PROCESSAMENTO DE LINGUAGENS (3º ANO DE CURSO)

---

## Trabalho Prático 1

---

### RELATÓRIO DE DESENVOLVIMENTO

---

Mestrado Integrado em Engenharia Informática

*Realizado por*

GRUPO 38

Filipa Alves dos Santos, a83631

Luís Miguel Arieira Ramos, a83930

Rui Alves dos Santos, a67656

5 de Abril de 2020

## **Resumo**

Este relatório vai abordar todo o processo da realização do primeiro trabalho prático da unidade curricular de Processamento de Linguagens do 3º ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Este trabalho aborda principalmente a criação de filtros Flex, com uso de expressões regulares estudadas nas aulas práticas.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
1.1	Enquadramento e Contextualização . . . . .	2
1.2	Problema e Objetivo . . . . .	2
1.3	Resultados ou Contributos . . . . .	2
1.4	Estrutura do Relatório . . . . .	2
<b>2</b>	<b>Análise e Especificação</b>	<b>4</b>
2.1	Descrição informal do problema . . . . .	4
2.2	Especificação do Requisitos . . . . .	4
2.2.1	Dados . . . . .	4
2.2.2	Pedidos . . . . .	4
2.2.3	Relações . . . . .	4
<b>3</b>	<b>Conceção/desenho da Resolução</b>	<b>7</b>
3.1	Estruturas de Dados . . . . .	7
3.2	Algoritmos . . . . .	7
<b>4</b>	<b>Codificação e Testes</b>	<b>10</b>
4.1	Alternativas, Decisões e Problemas de Implementação . . . . .	10
4.2	Testes realizados e Resultados . . . . .	10
<b>5</b>	<b>Conclusão</b>	<b>11</b>
<b>A</b>	<b>Código do Programa</b>	<b>12</b>
A.1	Executar código . . . . .	15

# 1 Introdução

## 1.1 Enquadramento e Contextualização

Este trabalho surgiu no âmbito da unidade curricular de Processamento de Linguagens, do 3º ano do Mestrado Integrado em Engenharia Informática, sendo este o primeiro de vários trabalhos práticos desta disciplina. Este projeto aborda maioritariamente a criação de um filtro flex, usando a própria notação flex e C, para produzir o resultado pretendido, como estudado nas aulas práticas.

Dos vários enunciados disponibilizados, decidimos optar pelo primeiro - o **Template multi-file** - logo, no resto desta introdução e relatório, iremos aprofundar mais o problema descrito neste enunciado em específico.

## 1.2 Problema e Objetivo

Os principais objetivos deste primeiro trabalho prático são logo dados no início do documento fornecido aos alunos. Os principais pontos expostos são os seguintes, por nossas palavras:

- Maior experiência com o sistema operativo Linux;
- Utilizar este gerador de analisadores léxicos (como filtros de texto, p.e.) como um método complementar à programação (em C), conseguindo assim explorar este ato de programar de uma maneira mais simples e direta;
- Pôr em prática o uso expressões regulares (ERs), que servem para delinear partes de um certo documento que seguem um específico padrão;
- Atribuir uma função/ação diferente a cada uma destas expressões regulares de modo a conseguirmos transformar o texto original num resultado pretendido.

O problema específico descrito neste enunciado cumpre todos estes objetivos. Queremos generalizar todo o software que tem como solução várias pastas, como uma pasta de exemplos, e vários ficheiros, como o README e o Makefile. Assim, pretendemos criar um programa "**mkfromtemplate**" que, dado um nome e um ficheiro descrição (o template) com tudo o que se deseja gerar, crie todas as pastas e ficheiros nas suas respetivas diretorias e com o conteúdo desejado.

## 1.3 Resultados ou Contributos

Quanto ao resultado deste trabalho, saímos satisfeitos com o que obtivemos. O nosso programa é capaz de ler diversos templates, não só o fornecido pela equipa docente, e cria o output com todo o conteúdo pedido.

Basicamente, conseguimos generalizar o nosso código para incluir diferentes situações, desde que no formato indicado, que serão explicadas em maior promenor no resto do relatório.

## 1.4 Estrutura do Relatório

Após esta introdução, iremos explorar cada tópico mais aprofundadamente no resto deste relatório:

- Análise e Especificação: abordamos o problema do enunciado que escolhemos de maneira mais informal e específica, enunciando os objetivos a cumprir e explicamos os dados, pedidos e relações do programa;
- Conceção/desenho da Resolução: explicamos em maior detalhe as estruturas de dados e os algoritmos usados na nossa solução;



- Codificação e Testes: apresentamos os nossos resultados e testes que realizamos para garantir a qualidade do nosso código;
- Conclusão: avaliação final do nosso trabalho.

## 2 Análise e Especificação

### 2.1 Descrição informal do problema

Depois de explicarmos formalmente o que o problema pede na introdução, vamos agora aprofundar o nosso raciocínio imediato do que era para fazer depois de lermos o enunciado.

Ao observarmos o template dado, definimos as seguintes tarefas:

- Dividir o nosso programa de acordo com as diversas partes do ficheiro: meta-dados, tree e, por fim, o conteúdo de cada ficheiro;
- Colocar os meta-dados quando são referenciados nos ficheiros e no caso do nome do programa em específico, temos que conseguir criar ficheiros e pastas com esse mesmo nome;
- Criar, através da tree dada, as pastas e os ficheiros nas diretorias de acordo com o formato utilizado;
- Por fim, temos que escrever nos ficheiros, referenciados na parte final do template, o texto imediatamente a seguir ao seu nome.

### 2.2 Especificação do Requisitos

#### 2.2.1 Dados

Em termos de dados observados logo à partida, detetamos algumas características no texto de entrada que iríamos ter de filtrar:

- As secções do programa começam por === (isto é, 3 sinais de igual seguidos de texto na mesma linha);
- Os meta-dados estão na forma de texto seguido de dois-pontos (como "email:", por exemplo);
- Já para a tree, entendemos que os hífens estão relacionados com o nível de profundidade na cadeia de diretorias (isto é, nenhum hífen significa que a pasta/ficheiro está na root, um hífen significa que está dentro da pasta anterior sem nenhum hífen, e assim sucessivamente para os níveis mais profundos) e que as pastas e os ficheiros se distinguiam pela barra / que as pastas tinham após o seu nome. Também notamos que, em alguns nomes dos ficheiros e pastas, o formato %name% era utilizado;
- Por último, nos ficheiros, a única parte que nos pareceu ser importante filtrar eram os meta-dados referenciados no meio do texto com o formato %dado%.

#### 2.2.2 Pedidos

Em termos dos pedidos, estes estão bem explícitos no final do enunciado deste problema: como resultado final, era pedido que no final da execução os ficheiros e diretorias descritas em tree fossem criados, com os conteúdos definidos nos templates de ficheiro e as variáveis substituídas.

#### 2.2.3 Relações

As relações, no caso deste trabalho, fizeram-se através de expressões regulares para o mapeamento e das respetivas funções para a parte da transformação em si. Os algoritmos destas transformações serão explicados com maior detalhe na seguinte parte do relatório logo nesta iremos abordar o assunto de forma mais geral.

Para que separarmos as diferentes secções do nosso programa, utilizamos **BEGINS**, um para a parte meta, um para a tree e outro para a escrita nos ficheiros:

```
1 <*>===\ meta\n\n          BEGIN META ;
2 <*>===\ tree\n\n          BEGIN TREE ;
3 <*>=                        BEGIN FICH ;
```

De seguida, lemos os tipos meta-dados do ficheiro identificados por uma sequência de caracteres (.) seguida do símbolo de dois pontos (:), bem como o valor de cada um dos meta-dados (que vem imediatamente a seguir dos dois pontos), com a expressão seguinte (\ .\* \n). Todo o resto do texto é descartado com a última expressão:

```
1 <META>.*:                  { saveMetaDado(yytext,0);}
2 <META>\ .* \n              { saveMetaDado(yytext+1,1);}
3 <META>#.* \n | \n          {}
```

Entramos agora na parte da criação das pastas dadas na tree. Para os 3 níveis (root, nível 1 e nível 2) tivemos de distinguir 2 casos: o da pasta ter um nome qualquer (representado por .\*) ou ter o nome dado como argumento ao programa ({%name%}). Em qualquer um dos casos, é chamada a função **createFolder** com os argumentos necessários, exeto quando a pasta se encontra na root - nesse caso tal operação é desnecessária.

```
1 <TREE>--.\{%name%\}\ /    { createFolder(n2, n1 ,0,0);}
2 <TREE>--+.* \ /           { createFolder(n2, n1 ,3,1);}
3 <TREE>-\.{%name%\}\ /    { createFolder(n1,root,0,0);}
4 <TREE>--+.* \ /           { createFolder(n1,root,2,1);}
5 <TREE>\{%name%\}\ /       { sprintf(root,"%s/",nome); mkdir(root);}
6 <TREE>.* \ /              { sprintf(root,"%s",yytext); mkdir(root);}
7 <TREE>.\ \n               {}
```

Tal como a criação das pastas, a criação dos ficheiros vai ter um raciocínio idêntico. Mais uma vez, para os 3 níveis fizemos 2 expressões diferentes, para o caso de utilizar {%name%} como nome do ficheiro e no caso contrário (+.\*). Um promenor a notar no 1º caso seria que, para captar a extensão do ficheiro, colocamos mais um .\* depois do {%name%}. (Relembramos que todas estas funções serão abordadas na secção "Algoritmos" do relatório)

```
1 <TREE>--.\{%name%\}.*      { createFile(n1 , 11, 0, 2);}
2 <TREE>--+.*                { createFile(n1 , 3 , 1, 2);}
3 <TREE>-\.{%name%\}.*      { createFile(root, 10, 0, 2);}
4 <TREE>--+.*                { createFile(root, 2 , 1, 2);}
5 <TREE>\{%name%\}.*        { createFile(" " , 8 , 0, 3);}
6 <TREE>[^=\n].*            { createFile(" " , 0 , 1, 3);}
```

Por último, avançamos para o **BEGIN** final, que abrange a escrita em todos os ficheiros mencionados na parte final do template. As primeiras 4 expressões servem para apanhar os nomes dos ficheiros. Nas primeiras duas tratamos o caso do primeiro ficheiro, com apenas dois sinais de igual. Isto deve-se ao facto de que o 1º sinal é apanhado pelo último **BEGIN**, o correspondente à escrita nos ficheiros, logo neste caso o filtro só consegue capturar os últimos 2 . De novo, para as duas situações (quer para os 1º ficheiro ou para os restantes), tratamos o caso de ter um nome qualquer (+.\*) ou de ser um ficheiro com o nome {%name%}. Já nas próximas 4 expressões, tratamos da escrita nos ficheiros. As primeiras duas dessas servem, respetivamente, para substituir o nome ({%name%}) e os outros meta-dados ({%[^%]}+%) nos ficheiros. As outras duas expressões servem para escrever o resto do texto, sendo que a primeira em particular (.\*[=].\*) é usada para que frases com sinais de igual não sejam detetadas como um novo ficheiro e sim como texto apenas.



```
1 <FICH>==\ \{%name%\}.*\n      { openFile(11,0);}
2 <FICH>==\ +.*\n                { openFile( 3,1);}
3 <FICH>===\ \{%name%\}.*\n      { openFile(12,0);}
4 <FICH>===+.*\n                 { openFile( 4,1);}
5 <FICH>\{\{%name%\}\}           { writeFile(nome);}
6 <FICH>\{\%[^%\%\}\}+%\%\}      { writeFile(getMetaDado(yytext));}
7 <FICH>.*[=] .*                 { writeFile(yytext);}
8 <FICH>.| \n                    { writeFile(yytext);}
9 .| \n                          {}
```



## 3 Conceção/desenho da Resolução

### 3.1 Estruturas de Dados

Em termos de estruturas de dados, usamos maioritariamente arrays de `char`, apontadores e alguns `int`:

- Começamos por criar uma variável `char* nome` para guardar o nome do ficheiro dado como argumento pelo utilizador;
- Para guardar a informação acerca dos meta dados, criamos dois arrays, um para guardar o tipo do meta dado (`char* meta[]`) e o outro para guardar o correspondente valor (`char* dados[]`), que serão guardados no mesmo índice. Adicionalmente, criou-se o `int indiceMeta` para guardar o índice atual, iniciado a 0;
- De uma maneira semelhante ao ponto anterior, para guardar a informação da `tree`, optou-se por criar dois arrays também. Um contém o nome do ficheiro/pasta (`char* nomes[]`) e o outro contém a respetiva diretoria (`char* dirs[]`). Serão guardados no mesmo índice, levando assim à criação do `int indiceAtual`, iniciado a 0;
- Relativamente à criação das diretorias, foi necessário recorrer à utilização de algumas variáveis temporárias, como é o caso do `char temp` e `char temp2`. Foram também criadas as variáveis `char root[]`, `char n1[]`, `char n2[]` e `char n3[]` para guardar informação sobre os diferentes níveis das diretorias.
- Finalmente, de modo a não perder informação necessária, foi criado a variável `char diretoriaAtual[]` que, como o próprio nome indica, guarda a diretoria atual, e a variável `char nomeFicheiroAtual[]` que contém o nome do ficheiro em que se está a escrever.

### 3.2 Algoritmos

Como já explicado nas "Relações", optou-se por dividir o ficheiro 3 partes: `META`, `TREE` e `FICH`.

Começando pela parte `META`, sabendo que o texto está no formato "meta: dado", criou-se uma função denominada de `saveMetaDado(char*,int)`. O inteiro servirá para determinar o tipo de texto que está a ser passado à função. Se for 0, o texto representa um tipo de meta e caso seja 1, representa o dado/valor, guardando esta informação nos respetivos arrays globais `meta` e `dados`. Caso esta contenha comentários (linha de texto que começa por `)` ou outro tipo de texto, este será ignorado.

```
1 void saveMetaDado (char* name, int tipo) {  
2     if (tipo == 0) meta[indiceMeta] = strdup(name);  
3     else dados[indiceMeta++] = strdup(cutBarraN(name));  
4 }
```

Avançando para a `TREE`, chegou-se à conclusão que seria melhor criar duas funções, `createFolder(char*,char*,int,int)` e `createFile(char*,int,int,int)`. A primeira, como o nome indica, servirá para criar as pastas, recebendo o nível inferior, superior, um incremento e um tipo. Se o tipo for 0, então trata-se de uma pasta de nome `{%name%}`, sendo necessário substituir o valor da variável. Caso seja 1, então o nome da pasta é o conteúdo que está no `yytext` + incremento. Este incremento vai ser diferente consoante o nível (quantos mais hífens, mais texto iremos querer ignorar, p.e.). Por fim recorre-se à função `mkdir(char*)` de modo a a criar a pasta, já na diretoria certa.

```
1 void createFolder(char* nS, char* nI, int inc, int tipo) {
2     if (tipo == 0)
3         sprintf(nS,"%s/",nome);
4     else
5         sprintf(nS,"%s",yytext + inc);
6     strcpy(temp,nI);
7     mkdir(strcat(temp,nS));
8     strcpy(nS,temp);
9 }
10
11 void mkdir(char* d) {
12     char com[100];
13     sprintf(com,"mkdir %s",d);
14     system(com);
15 }
```

A segunda função servirá para criar os ficheiros, recebendo a diretoria do ficheiro, um incremento, um tipo e um tipo2. Se o tipo for 0, então trata-se de um ficheiro cujo nome contém `{%name%}`, sendo necessário substituir essa variável. Caso seja 1, então o nome do ficheiro é o conteúdo de `yytext + incremento`. Se o tipo2 for 2, então significa que este novo ficheiro está contido dentro da pasta root, a principal. Caso seja 3, então este ficará fora dela, também na root.

```
1 void createFile (char* nivel, int inc, int tipo, int tipo2) {
2     char com[100];
3
4     if (tipo == 0)
5         sprintf(temp,"%s%s",nome,yytext + inc);
6     else
7         sprintf(temp,"%s",yytext + inc);
8     nomes[indiceAtual] = strdup(temp);
9
10    if (tipo2 == 2) {
11        sprintf(temp2,"%s/%s",nivel,temp);
12        dirs[indiceAtual] = strdup(temp2);
13        sprintf(com,"%s/%s",nivel,temp);}
14    else {
15        dirs[indiceAtual] = strdup(temp);
16        sprintf(com,"%s",temp);}
17
18    indiceAtual++;
19    FILE* f = fopen(com,"a");
20    fclose(f);
21 }
```

Por fim, a **FICH** trata de dois casos: a abertura de um ficheiro e a escrita nele. Sabendo que a cada `===` se dá a mudança de ficheiro, recorreremos à criação da função `openFile(int,int)`, que recebe um incremento e um tipo. Se o tipo for 0, então o ficheiro toma como nome `{%name%}`, sendo necessário utilizar a variável. Caso seja 1, então o nome desse ficheiro está contido no `yytext + incremento`. Obtendo assim o nome do ficheiro, é necessário percorrer o array dos `nomes` até chegar à posição certa, ou seja, em que o nome que está naquela posição do array é igual ao do ficheiro. Quando chega a essa posição, obtemos então a diretoria do ficheiro, recorrendo ao array `dirs` na posição anteriormente determinada. Caso o nome do ficheiro não esteja contido no array `nomes`, a consola apresenta mensagem de erro, e o template deste ficheiro é ignorado. Depois de ter obtido a diretoria do ficheiro, guardamos esse valor na variável `diretoriaAtual`.

```
1 void openFile (int inc, int tipo) {
2     int j = 0;
3     if (tipo == 0) {
4         sprintf(temp,"%s%s", nome, yytext + inc);
5         strcpy(nomeFicheiroAtual, cutBarraN(temp));}
6     else
7         strcpy(nomeFicheiroAtual, cutBarraN(yytext + inc));
8
9     for(int i = 0; i < M; i++) {
10         if (nomes[i]!=0) {
11             if (strcmp(nomeFicheiroAtual,nomes[i]) == 0) {
12                 strcpy(diretoriaAtual,dirs[i]);
13                 j=1;
14                 break;
15             }
16         }
17     }
18     if (j==0) {
19         strcpy(diretoriaAtual,"erro");
20         printf("Erro: Ficheiro %s n o contido na 'tree'!\n",
21             nomeFicheiroAtual);
22     }
```

A partir daqui, dá-se início à escrita no ficheiro, sendo necessário a criação de uma nova função denominada de `writeFile(char*)`, que recebe uma linha de texto como argumento e imprime-a no ficheiro, cuja diretoria se encontra na variável global `diretoriaAtual`. Na escrita de um ficheiro, caso este contenha algum texto do tipo `{%xxxx%}`, então este será verificado se é um meta-dado fornecido. Caso seja, então este é substituído pelo valor do meta-dado, senão mantém o mesmo nome. Isto é feito através da função `char* getMetaDado( char* name)`.

```
1 void writeFile(char* linha) {
2     if (strcmp(diretoriaAtual,"erro")!=0) {
3         FILE* f = fopen(diretoriaAtual,"a");
4         fprintf(f,"%s",linha);
5         fclose(f);
6     }
7 }
8
9 char* getMetaDado( char* name) {
10     char* n;
11     n = strdup(name+2);
12     n[strlen(n)-2] = ':';
13     n[strlen(n)-1] = '\0';
14
15     for(int i=0; i<M; i++) {
16         if (meta[i]==0) return name;
17         if (strcmp(meta[i],n)==0) return dados[i];
18     }
19 }
```

## 4 Codificação e Testes

### 4.1 Alternativas, Decisões e Problemas de Implementação

A implementação inicial foi relativamente fácil, os problemas surgiram mais no que respeita à generalização da resolução, isto é, no processo de tornar o código mais abrangente. Decidimos otimizar a nossa resolução para, por exemplo, funcionar para templates com mais ou menos files na tree e permitir que qualquer o texto dentro dos ficheiros não fosse apanhado numa expressão regular qualquer.

Em termos de decisões relevantes tomadas, começamos por definir um nível máximo na tree, até ao nível 3. Dito isto, qualquer sequência correta (i.e., de acordo com o formato correto da tree) de ficheiros e pastas até esse nível funciona, não só a do template dado. Também decidimos permitir que quaisquer meta-dados desejados possam ser inseridos, apesar de que o nome é o único que pode ser colocado nos nomes das pastas e ficheiros.

Por último, também decidimos tratar de alguns erros que poderiam acontecer com um template menos correto. Se um meta-dado for referenciado num ficheiro mas, não for dado no próprio template, é ignorado e o programa segue normalmente. Também se existir um template de um ficheiro não existente na tree, o código trata dessa possibilidade e novamente, ignora esse pedaço do ficheiro descrição.

### 4.2 Testes realizados e Resultados

Para testar a viabilidade da nossa solução, fizemos testes com templates que variassem nas partes que desejássemos verificar. Como tal, usamos templates com trees diferentes, mais e menos ficheiros, meta-dados diferentes, etc. Iremos agora listar os testes que pretendíamos fazer com cada template (todos os templates estarão disponíveis do zip enviado com o resto do trabalho):

- **Template1.txt** - Template de teste dado pela equipa docente;
- **Template2.txt** - Template de teste mais focado no funcionamento da tree, nomeadamente na criação de pastas e ficheiros.
- **Template3.txt** - Template de teste que demonstra o "erro" dado na consola, quando existe um template de um ficheiro que não está contido na tree.
- **Template4.txt** - Template de teste com foco nos meta-dados substituindo-os quando estes são referidos nos ficheiros, através de `{%xxxx%}` mas apenas aqueles que estão contidos na meta.
- **Template5.txt** - Template de teste virado para o uso do símbolo `=` nos templates dos ficheiros, de modo a evitar conflitos com a divisão de templates já pré-definida `===`.

Concluimos assim que obtivemos bons resultados, sendo que o nosso código se comportou da maneira esperada em todos estes testes.



## 5 Conclusão

Em suma, neste primeiro trabalho, o grupo considera que aprofundou muito o seu conhecimento sobre as ferramentas usadas na unidade curricular. Foi dada a oportunidade de aprender a trabalhar com expressões regulares e desenvolver processadores de linguagens regulares, e também, aprofundar os conhecimentos da linguagem Flex. Por fim, terminado este primeiro trabalho, o objetivo será começar já a preparar para o próximo trabalho continuando assim aprofundar o conhecimento sobre Processamento de Linguagens, com outras matérias novas como yacc.

## A Código do Programa

```
1 %option noyywrap
2
3 %x META TREE FICH
4
5 %{
6     #define M 20
7     #define N 50
8
9     /*-----GLOBAL VARIABLES-----*/
10    int    indiceAtual = 0;
11    int    indiceMeta = 0;
12    char*  nome;
13    char*  meta[M], * dados[M];
14    char*  nomes[M], * dirs[M];
15    char   diretoriaAtual[N], nomeFicheiroAtual[M];
16    char   temp[M], temp2[M], root[M], n1[M], n2[M], n3[M];
17
18    /*-----FUNCTIONS-----*/
19    char*  getMetaDado (char*);
20    void   saveMetaDado (char*, int);
21    void   createFolder (char*, char*, int, int);
22    void   createFile   (char*, int, int, int);
23    void   openFile     (int,int);
24    void   writeFile    (char*);
25    void   mkdir        (char*);
26    char*  cutBarraN    (char* s);
27 %}
28
29
30 %%
31
32 <*>===\ meta\n\n          BEGIN META;
33 <*>===\ tree\n\n          BEGIN TREE;
34 <*>=                        BEGIN FICH;
35
36
37 /*-----META-----*/
38
39 <META>.*:                  { saveMetaDado(yytext,0);}
40 <META>\ .* \n              { saveMetaDado(yytext+1,1);}
41 <META>#.* \n | \n          {}
42
43
44 /*-----CREATE FOLDERS-----*/
45
46 <TREE>--.\{%name%\}\ /    { createFolder(n2, n1 ,0,0);}
47 <TREE>--+.* \ /           { createFolder(n2, n1 ,3,1);}
48 <TREE>-\.{%name%\}\ /    { createFolder(n1,root,0,0);}
49 <TREE>-+.* \ /           { createFolder(n1,root,2,1);}
50 <TREE>\{%name%\}\ /      { sprintf(root,"%s/",nome); mkdir(root);}
51 <TREE>.* \ /             { sprintf(root,"%s",yytext); mkdir(root);}
52 <TREE>.\ | \n            {}
53
54 /*-----CREATE FILES-----*/
55
```



```
56 <TREE>--.\{%name%\}.*      { createFile(n1 , 11, 0, 2);}
57 <TREE>--+. *                { createFile(n1 , 3 , 1, 2);}
58 <TREE>-. \{%name%\}.*      { createFile(root, 10, 0, 2);}
59 <TREE>-+. *                { createFile(root, 2 , 1, 2);}
60 <TREE>\{%name%\}.*         { createFile(" " , 8 , 0, 3);}
61 <TREE>[^\n]. *             { createFile(" " , 0 , 1, 3);}
62
63 /*-----EDIT FILES-----*/
64
65 <FICH>==\ \{%name%\}.*\n    { openFile(11,0);}
66 <FICH>==\ +.*\n            { openFile( 3,1);}
67 <FICH>===\ \{%name%\}.*\n    { openFile(12,0);}
68 <FICH>===+.*\n             { openFile( 4,1);}
69 <FICH>\{%name%\}           { writeFile(nome);}
70 <FICH>\{\\%[\^\\%\\]+\%\\}   { writeFile(getMetaDado(yytext));}
71 <FICH>.*[=]. *             { writeFile(yytext);}
72 <FICH>.| \n                { writeFile(yytext);}
73
74 .|\n                      {}
75
76
77 %%
78
79
80 char* getMetaDado( char* name) {
81     char* n;
82     n = strdup(name+2);
83     n[strlen(n)-2] = ':';
84     n[strlen(n)-1] = '\\0';
85
86     for(int i=0; i<M; i++) {
87         if (meta[i]==0) return name;
88         if (strcmp(meta[i],n)==0) return dados[i];
89     }
90 }
91
92
93 void saveMetaDado (char* name, int tipo) {
94     if (tipo == 0) meta[indiceMeta] = strdup(name);
95     else dados[indiceMeta++] = strdup(cutBarraN(name));
96 }
97
98
99 void createFolder(char* nS, char* nI, int inc, int tipo) {
100     if (tipo == 0)
101         sprintf(nS,"%s/",nome);
102     else
103         sprintf(nS,"%s",yytext + inc);
104     strcpy(temp,nI);
105     mkdir(strcat(temp,nS));
106     strcpy(nS,temp);
107 }
108
109
110 void createFile (char* nivel, int inc, int tipo, int tipo2) {
111     char com[100];
112
113     if (tipo == 0)
```

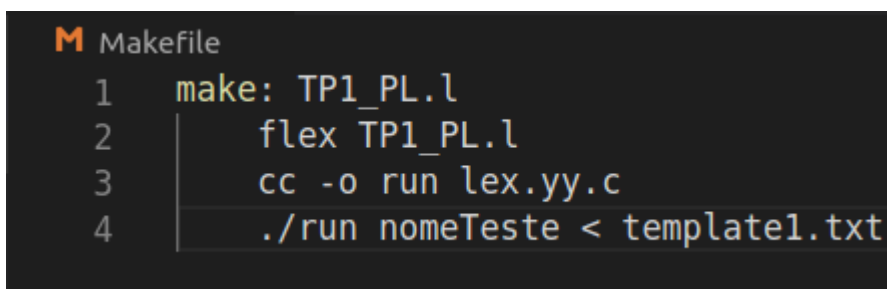
```
114     sprintf(temp,"%s%s",nome,yytext + inc);
115     else
116         sprintf(temp,"%s",yytext + inc);
117     nomes[indiceAtual] = strdup(temp);
118
119     if (tipo2 == 2) {
120         sprintf(temp2,"%s/%s",nivel,temp);
121         dirs[indiceAtual] = strdup(temp2);
122         sprintf(com,"%s/%s",nivel,temp);}
123     else {
124         dirs[indiceAtual] = strdup(temp);
125         sprintf(com,"%s",temp);}
126
127     indiceAtual++;
128     FILE* f = fopen(com,"a");
129     fclose(f);
130 }
131
132
133 void openFile (int inc, int tipo) {
134     int j = 0;
135     if (tipo == 0) {
136         sprintf(temp,"%s%s", nome, yytext + inc);
137         strcpy(nomeFicheiroAtual,cutBarraN(temp));}
138     else
139         strcpy(nomeFicheiroAtual,cutBarraN(yytext + inc));
140
141     for(int i = 0; i < M; i++) {
142         if (nomes[i]!=0) {
143             if (strcmp(nomeFicheiroAtual,nomes[i]) == 0) {
144                 strcpy(diretoriaAtual,dirs[i]);
145                 j=1;
146                 break;
147             }
148         }
149     }
150     if (j==0) {
151         strcpy(diretoriaAtual,"erro");
152         printf("Erro: Ficheiro %s n o contido na 'tree'!\n",
153             nomeFicheiroAtual);
154     }
155 }
156
157 void writeFile(char* linha) {
158     if (strcmp(diretoriaAtual,"erro")!=0) {
159         FILE* f = fopen(diretoriaAtual,"a");
160         fprintf(f,"%s",linha);
161         fclose(f);
162     }
163 }
164
165 void mkdir(char* d) {
166     char com[100];
167     sprintf(com,"mkdir %s",d);
168     system(com);
169 }
170 }
```



```
171
172
173 char* cutBarraN(char* s){
174     char* n;
175     n = strdup(s);
176     n[strlen(n)-1] = '\0';
177     return n;
178 }
179
180
181 int main(int c, char* argv[]) {
182     nome = strdup(argv[1]);
183     yylex();
184     return 0;
185 }
```

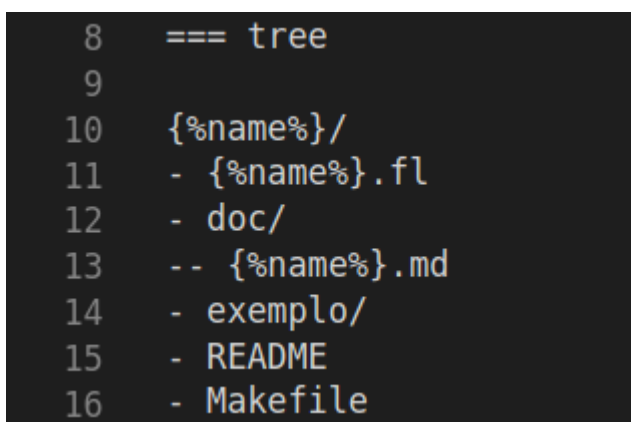
### A.1 Executar código

Para correr o código basta escrever `make` na consola, sendo que criamos um Makefile com todos os comandos necessários. Para mudar o template escolhido, é só preciso substituir onde diz "template1.txt" pelo nome de outro template fornecido. Também é possível alterar o nome dado como argumento, que neste caso é "nomeTeste".



```
M Makefile
1  make: TP1_PL.l
2      flex TP1_PL.l
3      cc -o run lex.yy.c
4      ./run nomeTeste < template1.txt
```

Como exemplo de uma possível execução do programa, apresentamos agora a tree do "template1.txt" e as pastas e ficheiros que são gerados como resultado:



```
8  === tree
9
10  {%name%}/
11  - {%name%}.fl
12  - doc/
13  -- {%name%}.md
14  - exemplo/
15  - README
16  - Makefile
```



```

  ✓ nomeTeste
    ✓ doc
      ↴ nomeTeste.md
    ✓ exemplo
    M Makefile
    ≡ nomeTeste.fl
    ≡ README
```