Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático nº1

Template multi-file

Grupo 15

18 de Abril de 2020

MiEI - 3ºAno - 2ºSemestre

Beatriz Rocha A84003

Filipe Guimarães A85308

Gonçalo Ferreira A84073

Conteúdo

Capítulo 1

Introdução

Este trabalho prático 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-Ação e utilizar o Flex para gerar filtros de texto em C.
Para este trabalho prático escolhemos então desenvolver o primeiro enunciado *Template multi-file*. Este passa pela criação de um programa *"mkfromtemplate"*, capaz de aceitar um nome de projeto, e um ficheiro descrição de um template-multi-file e criar os ficheiros e pastas iniciais do projecto.
Este template inclui:

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

Este relatório contém toda a nossa abordagem ao enunciado proposto.
Começamos por explicar como especificamos os padrões de frases que queremos encontrar no texto-fonte, através de expressões regulares, como identificamos as ações semânticas a realizar como reacção ao reconhecimento de cada um desses padrões. Mostramos as Estruturas de Dados globais que precisamos para armazenar temporariamente e por fim como desenvolvemos o filtro em Flex para fazer o reconhecimento dos padrões identificados e proceder à transformação pretendida.

Capítulo 2

Padrões de Frases

Para a análise deste problema temos como a primeira fase "Padrões de Frases", consiste na especificação dos *tokens* e expressões regulares (ERs) utilizadas para a sua resolução do problema proposto.

2.1 Reconhecimento de *Tokens*

Observando o *template* exemplar, pode-se observar que existem padrões diferentes que influenciam a leitura da informação seguinte. O padrão mais básico sendo "===", que origina o *token* CATEGORY, neste existem três possibilidades, que são encontrar "tree", "meta"e o nome de um ficheiro, que originam os respetivos *tokens* TREE, META, CONTENT ou VOID. Para o META são propostos ainda mais dois *tokens* EMAIL e AUTHOR, para os padrões "email:"e "author:", respetivamente.
Acrescenta-se ainda que estes *tokens* serão utilizados como inclusivos, assim tendo que especificar individualmente qual o comportamento correto de cada *token* e ordem devida.

```
%s CATEGORY META EMAIL AUTHOR TREE CONTENT VOID
```

2.2 Expressões Regulares Definições

Para a especificação das Expressões Regulares verificou-se repetição de padrões, que poderiam ser reutilizados em diversas expressões. Como tal estas são as expressões regulares declaradas nas definições, de modo a facilitar a escrita das regras.

1. `acentos \xc3[\x80-\xbf]`

   {acentos} Corresponde aos acentos.

2. `letra [a-zA-Z]|{acentos}`

   {letra} Corresponde a qualquer letra, seja acentuada ou não.

3. `file_character ([\x21-\x2E\x30-\x5B\x5D-\x7E]|{letra})`

   {file_character} Corresponde a qualquer *character* que é permitido pelo programa para um nome de ficheiro/diretoria.

4. `file_name {file_character}+`

   {file_name} Corresponde a um nome de um ficheiro.

5. `special_name {file_name}*\{%name%\}{file_name}*`

   {special_name} Corresponde a um nome especial de um ficheiro.

6. `branch_name {file_name}\/`

   {branch_name} Corresponde a um nome de uma diretoria.

7. `branch_Sname {special_name}\/`

   {branch_Sname} Corresponde a um nome especial de uma diretoria.

8. ```
   email [A-Za-z0-9_\]+((\.|\+|\-)[A-Za-z0-9_\]+)*
         \@[A-Za-z_\"]+((\.|\+|\-)[A-Za-z_\"]+)+
   ```

   {email} Corresponde a um email válido para o programa.

As oito expressões acima são as definições propostas para reutilização futura. Acompanhadas de uma breve explicação do objetivo destas expressões regulares.
   ```

2.3 Expressões Regulares Regras

As regras propostas para a resolução do problema, encontram-se enumeradas abaixo pela ordem respetiva. Encontram-se acompanhadas com uma breve explicação da sua importância.

1. `<TREE>^\=\=\=\`
2. `<CONTENT>^\=\=\=\`
3. `^\=\=\=\`
4. `<CATEGORY>meta$`
5. `<CATEGORY>tree$`
6. `<CATEGORY>{file_name}$`
7. `<META>email:\`
8. `<META>author:\`
9. `<META>\#.*$`
10. `<EMAIL>{email}$`
11. `<AUTHOR>({letra}+(\.)?(\ )?)+$`
12. `<TREE>[\-]+\`
13. `<TREE>{special_name}$`
14. `<TREE>{file_name}$`
15. `<TREE>{branch_Sname}$`
16. `<TREE>{branch_name}$`
17. `<CONTENT>\{%name%\}`
18. `<CONTENT>\{%author%\}`
19. `<CONTENT>\{%email%\}`
20. `<CONTENT>{letra}+`
21. `<CONTENT>.`
22. `<CONTENT>\n`
23. `<VOID>.|[ \t\r]`
24. `[ \t\r]`
25. `\n`
26. `.`
27. `<<EOF>>`

As regras **1, 2 e 3** têm como objetivo dar *match* ao início de um campo que irá permitir saber o começo de uma secção nova. Comprometendo a variação de CATEGORY.

As regras **3, 5 e 6** permitem especificar como será a leitura da informação dentro da secção, atribuindo assim um *token*, que irá variar para META, TREE, CONTENT e VOID.

As regras **7, 8 e 9** especificam os padrões possíveis de encontrar na secção meta, nos quais comentários começados por '#', email e autor por "email: "e "author: ", respetivamente. Para facilitar usa-se ainda as regras **10 e 11**, para o *parse* do email e autor.

A regra **12** é responsável pela contagem da profundidade na árvore de diretorias. Posteriormente as regras **13 a 16** identificam o nome da diretoria ou ficheiro.

As regras **17, 18 e 19** são responsáveis por identificar os nomes especiais que é necessário substituir. Acrescenta-se ainda que as regras **20, 21 e 22** são utilizadas para dar *match* a padrões que necessitam de tratamento específico.

As regras **23, 24 e 25** são semelhantes, pois são usadas para representar apenas o que separa as expressões que são especificadas como corretas.

A regra **26** representa o erro sintático, sendo que qualquer padrão que não especificado acima é considerado errado. A regra **27** representa o final do ficheiro.

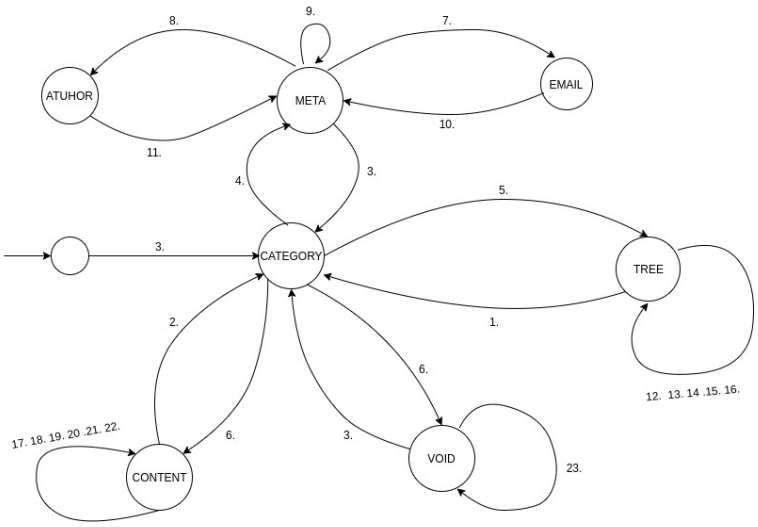

Figura 2.1: Autómato do template multi-file

Capítulo 3

Ações Semânticas

As ações semânticas têm como objetivo permitir assim controlar qual a reação às regras estabelecidas, mencionadas anteriormente no relatório. Como tal irá ser usado a numeração utilizada em "Expressões Regulares Regras", seguido do pseudo código que representa a ação.

- **Regra 1** Ao sair da secção tree e o começo de outra secção.

```
{
    informar que a tree está correta;
    construir a tree com recurso a system calls;
    BEGIN CATEGORY;
}
```

- **Regra 2** Ao sair da secção de um ficheiro e o começo de outra secção.

```
{
    fechar o file pointer desse ficheiro;
    yyout = stdout;
    BEGIN CATEGORY;
}
```

- **Regra 3** Começo de uma secção.

```
{ BEGIN CATEGORY; }
```

- **Regra 4** Encontrar a secção "meta".

```
{
    IF (já existe email e autor) mensagem de erro e
    terminar parse;
    ELSE informa começo de parse de meta;
    BEGIN META;
}
```

- **Regra 5** Encontrar a secção "tree".

```
{
    IF (já deu parse a alguma tree) mensagem de erro e
    terminar parse;
    ELSE informa começo de parse de tree;
    BEGIN TREE;
}
```

- **Regra 6** Encontrar uma secção de um ficheiro.

```
{
    IF (já deu parse a uma tree) mensagem de erro e
    terminar parse;
    IF (não existe nome na tree) {
        mensagem de erro;
        BEGIN VOID;
    }
    ELSE{
        IF (nome representa um ficheiro){
            mensagem de sucesso;
            BEGIN CONTENT;
        }
        ELSE {
            mensagem de erro;
            BEGIN VOID;
        }
    }
    BEGIN CATEGORY;
}
```

- **Regra 7** Padrão para encontrar email.

```
{ BEGIN EMAIL; }
```

- **Regra 8** Padrão para encontrar autor.

```
{ BEGIN AUTHOR; }
```

- **Regra 9** Padrão para um comentário.

```
{;}
```

- **Regras 10 e 11** Encontrar um email e autor, respetivamente.

```
{
    IF (já existe algum email/autor) {
        mensagem de erro;
        termina parse;
    }
    adicionar email/autor;
    BEGIN META;
}
```

- **Regra 12**

```
{ contagem do número de '-' seguidos; }
```

- **Regras 13, 14, 15 e 16** Estas regras são semelhantes apenas variando o tipo entre ficheiro e diretoria, como também alterando os nomes especiais nos casos respetivos.

```
{
    verificação se o ficheiro/diretoria está numa
    profundidade sintaticamente correta;
    atualização de profundidade;
    IF (ficheiro/diretoria nome já existe na árvore){
        mensagem de erro;
        terminar parse;
    }
    ELSE{
        cosntruir diretoria completa;
        adicionar à arvore com o tipo da diretoria
        ficheiro/diretoria;
    }
}
```

- **Regras 17, 18 e 19** Regras dos nomes especiais, que são escritos como ficheiros.

```
{ escrever para o yyout os nomes especiais em vez
do que os representa; }
```

- **Regras 20 e 21** Escrita para yyout.

```
{ ECHO; }
```

- **Regra 22** Contagem global de nova linha e escrita para o yyout.

```
{ incrementar o número de linhas; ECHO;}
```

- **Regras 23 e 24** Caracteres que não são pertinentes para o *parse* deste ficheiro.

```
{;}
```

- **Regra 25** Contagem global de nova linha;

```
{ incrementar o número de linhas; }
```

- **Regra 26** Erro sintático.

```
{ informar erro sintático; terminar parse; }
```

- **Regra 27** Fim do ficheiro de leitura.

```
{ informar que chegou o fim do ficheiro; terminar parse; }
```

Capítulo 4

Estruturas de Dados

Após uma análise cuidada dos requerimentos do programa, chegou-se à conclusão que será necessário utilizar variáveis globais. Serão um elemento fundamental no controlo de erros mais sofisticados, como também a armazenar estruturas de dados relevantes. Para facilitar este processo foi usada também a *glib* com recurso às estruturas dinâmicas e suas *API's*.

4.1 Controlo do Número de Linhas

Para o controlo de linhas já lidas pelo programa é usado um inteiro. É inicializado como representado abaixo, este inteiro é usado para informação adicional nas mensagens retornadas ao utilizador.

```
int line = 1;
```

4.2 Controlo de Nomes Especiais

Como sabemos existem três nomes especiais, que podem ser referenciados no decorrer do programa, exigindo variáveis para facilitar o acesso aos mesmos. Utilizou-se "char *"para cada um deles, tendo em conta a sua natureza. Inicializados como demonstrado no excerto de código abaixo.

```
char * input_name = NULL;
char * email = NULL;
char * author = NULL;
```

4.3 Validação da Árvore

A validação da árvore tem como auxílio dois inteiros, que guardam a profundidade do último ficheiro/diretoria e a profundidade atual a que se encontra. Com base nestes dois valores é possível compará-los e obter as respostas necessárias para a validação. A inicialização destas variáveis é efetuada como representado abaixo.

```
int last_branch = 0;
int branch = 0;
```

4.4 Construção de Diretorias

Construir as diretorias completas apresentou um novo obstáculo, pois é necessário reter o progresso relativo à posição atual na árvore. Para tal a estrutura abaixo é utilizada, na qual contém a profundidade e o nome da diretoria/ficheiro associada.

```
struct directory_st{
    int depth;
    char * dir_name;
};
typedef struct directory_st * DIRECTORY;
```

No entanto a construção não é possível sem a utilização de um *array* dinâmico que armazena o estado da diretoria atual.

```
GPtrArray * this_directory = NULL;
```

Após a inserção da informação necessária para conter a diretoria total, é ainda usada outra estrutura disponibilizada pela *glib*. A estrutura **GString** é utilizada para concatenar todos os fragmentos da diretoria.

4.5 Armazenamento das Diretorias

Por fim o armazenamento das diretorias completas, ou seja, da árvore completa e final. Para cada ficherio/diretoria é usada a estrutura abaixo, com as variáveis *name*, *dir* e *type*, respetivamente nome único, diretoria completa e tipo. O tipo pode tomar dois valores 'f' ou 'd', ou seja, ficheiro ou diretoria.

```
struct full_dir_st{
    char * name;
    char * dir;
    char type;
};
typedef struct full_dir_st * FULL_DIR;
```

Dado a estrutura para armazenar os dados de cada elemento da árvore. Utiliza-se um *array* dinâmico para guardar estes elementos por ordem de inserção, garantindo que não haverá dependências ao percorrer e criar a árvore.

```
GPtrArray * final_directories = NULL;
```

Capítulo 5

Filtro de Texto

Com tudo o que mencionamos anteriormente e definidas as estruturas de dados que precisamos criamos então o nosso filtro de texto que se encontra no ***Anexo 1*** deste documento.

Capítulo 6

Conclusão

Como já esperavamos no início, este trabalho prático ajudou-nos a melhorar os nossos conhecimentos na construção de filtros Flex bem como a nossa capacidade de produção de Expressões regulares. Aspetos apresentados durante as aulas teóricas e práticas. Achamos que os objetivos propostos foram todos alcançados.

Capítulo 7

Anexos

7.1 Anexo 1 - Código FLex

```
%{
    #include <stdio.h>
    #include <fcntl.h>
    #include <sys/stat.h>
    #include "glib.h"
    #include "gmodule.h"

    extern char * asprintf();

    int line = 1;

    int last_branch = 0;
    int branch = 0;

    char * input_name = NULL;
    char * email = NULL;
    char * author = NULL;

    GPtrArray * this_directory = NULL;
    GPtrArray * final_directories = NULL;


    struct full_dir_st{
        char * name;
        char * dir;
        char type;
    };

    typedef struct full_dir_st * FULL_DIR;

    FULL_DIR init_full_dir(char * n, char * d, char t);
```

```
    int duplicated_dir(char * n);

    int get_index_of_full_dir(char * n);

    void create_tree(gpointer data,gpointer user_data);



    struct directory_st{
        int depth;
        char * dir_name;
    };

    typedef struct directory_st * DIRECTORY;

    char * full_path(char * name);


    char * swap_token_for_name(char * s, char * tok, char * name);

%}

%s CATEGORY META EMAIL AUTHOR TREE CONTENT VOID

acentos \xc3[\x80-\xbf]
letra [a-zA-Z]|{acentos}

file_character ([\x21-\x2E\x30-\x5B\x5D-\x7E]|{letra})

file_name {file_character}+
special_name {file_name}*\{%name%\}{file_name}*

branch_name {file_name}\/
branch_Sname {special_name}\/

email [A-Za-z0-9\_\]+((\.|\+|\-)[A-Za-z0-9\_\]+)*\@
      [A-Za-z\_\"]+((\.|\+|\-)[A-Za-z\_\"]+)+

%%

<TREE>^\=\=\=\  {
    puts("The tree is correct, building tree. ");
    BEGIN CATEGORY;
    g_ptr_array_foreach(final_directories,create_tree,NULL);
}

<CONTENT>^\=\=\=\  {
    BEGIN CATEGORY;
    fclose(yyout);
    yyout = stdout;
```

```
}

^\=\=\=\  {
    BEGIN CATEGORY;
}


<CATEGORY>meta$ {
    if (email != NULL || author != NULL) {
        puts("The \"=== meta\" can only be written once.");
        return 0;
    }
    puts("Parsing meta:");
    BEGIN META;
}

<CATEGORY>tree$ {
    if (this_directory != NULL || final_directories != NULL) {
        puts("The \"=== tree\" can only be written once.");
        return 0;
    }
    this_directory = g_ptr_array_new();
    final_directories = g_ptr_array_new();
    puts("Parsing tree:");
    BEGIN TREE;
}

<CATEGORY>{file_name}$ {
    if (this_directory == NULL || final_directories == NULL
        || email == NULL || author == NULL) {
        printf("The \"=== %s\" must come after \"=== tree\"",yytext);
        printf(" and \"=== metta\"");
        printf(".\n");
        return 0;
    }

    int i;
    i = get_index_of_full_dir(yytext);

    if (i < 0) {
        printf("Invalid name file: %s (line %d)\n",yytext,line);
        BEGIN VOID;
    }
    else {
        FULL_DIR fdr = (FULL_DIR) g_ptr_array_index(final_directories,i);

        if (fdr->type != 'f') {
            printf("It's a directory: %s (line %d)\n",yytext,line);
            BEGIN VOID;
        }
```

```
        else {
            yyout = fopen(fdr->dir,"a+");
            printf("Valid file loading: %s (line %d)\n",yytext,line);
            BEGIN CONTENT;
        }
    }
}

<META>email:\  BEGIN EMAIL;

<META>author:\  BEGIN AUTHOR;

<META>\#.*$ ;

<EMAIL>{email}$ {
    if (email != NULL) {
        puts("You can only assign one email.");
        return 0;
    }

    printf("EMAIL ---> %s\n",yytext);
    email = strdup(yytext);
    BEGIN META;
}

<AUTHOR>({letra}+(\.)?(\ )?)+$ {
    if (author != NULL) {
        puts("You can only assign one author.");
        return 0;
    }

    printf("AUTHOR --> %s\n",yytext);
    author = strdup(yytext);
    BEGIN META;
}


<TREE>[\-]+\  {
    for(int i = 0; yytext[i] == '-'; i++)
        branch++;
}

<TREE>{special_name}$ {
    if(branch > last_branch) {
        printf("Error in branch for \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    if (branch < last_branch) last_branch = branch;
```

```
    char * updated_name = swap_token_for_name(yytext,"{\%name\%}",input_name);
    if (!duplicated_dir(yytext)) {
        printf("Duplicated directory \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    char * dir = full_path(updated_name);
    g_ptr_array_add(final_directories,init_full_dir(yytext,dir,'f'));

    branch = 0;
}

<TREE>{file_name}$ {
    if(branch > last_branch) {
        printf("Error in branch for \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    if(branch < last_branch) last_branch = branch;

    if (!duplicated_dir(yytext)) {
        printf("Duplicated directory \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    char* dir = full_path(yytext);

    g_ptr_array_add(final_directories,init_full_dir(yytext,dir,'f'));

    branch = 0;
}

<TREE>{branch_Sname}$ {
    if(branch > last_branch) {
        printf("Error in branch for \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    last_branch = branch + 1;

    char * name = strndup(yytext,yyleng-1);
    char * updated_Sbranch = swap_token_for_name(yytext,"{\%name\%}",input_name);
    if (!duplicated_dir(name)) {
        printf("Duplicated directory \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    char * dir = full_path(updated_Sbranch);
    g_ptr_array_add(final_directories,init_full_dir(name,dir,'d'));
```

```
    branch = 0;
}

<TREE>{branch_name}$ {
    if(branch > last_branch) {
        printf("Error in branch for \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    last_branch = branch + 1;

    char * name = strndup(yytext,yyleng-1);
    if (!duplicated_dir(name)) {
        printf("Duplicated directory \"%s\" (line %d)\n",yytext,line);
        return 0;
    }

    char* dir = full_path(yytext);
    g_ptr_array_add(final_directories,init_full_dir(name,dir,'d'));

    branch = 0;
}


<CONTENT>\{%name%\} fprintf(yyout,"%s",input_name);

<CONTENT>\{%author%\} fprintf(yyout,"%s",author);

<CONTENT>\{%email%\} fprintf(yyout,"%s",email);

<CONTENT>{letra}+ ECHO;

<CONTENT>. ECHO;

<CONTENT>\n {line++; ECHO;}

<VOID>.|[ \t\r] ;

    /* one character match */
[ \t\r] ;

\n line++;

. {printf("Syntatic error. (\"%s\" line %d)",yytext,line); return 0;}

<<EOF>> {puts("EOF"); return 0;}

%%
```

```
int main(int argc, char ** argv){
    if(argc>2){
        input_name = strdup(argv[1]);
        yyin = fopen(argv[2],"r");
    }
    yylex();
    return 1;
}


    // FULL DIRECTORY SECTION

    FULL_DIR init_full_dir(char * n, char * d, char t){
        FULL_DIR fdir = malloc(sizeof(struct full_dir_st));
        fdir->name = strdup(n);
        fdir->dir = strdup(d);
        fdir->type = t;
        return fdir;
    }

    gboolean equal_name_dir(gconstpointer a, gconstpointer b){
        FULL_DIR fdir = (FULL_DIR) a;
        char* name = (char*) b;
        if(!strcmp(fdir->name,name)){
            return TRUE;
        }
        return FALSE;
    }

    int duplicated_dir(char * n){
        int i = -1;
        g_ptr_array_find_with_equal_func(final_directories,n,equal_name_dir,&i);
        if (i != -1) return 0;
        else return 1;
    }

    int get_index_of_full_dir(char * n){
        int i = -1;
        g_ptr_array_find_with_equal_func(final_directories,n,equal_name_dir,&i);
        return i;
    }

    void create_tree(gpointer data,gpointer user_data){
        FULL_DIR f = (FULL_DIR) data;
        char * cmd;

        switch(f->type) {
            case 'f':
                asprintf(&cmd,"touch %s",f->dir);
```

```
            system(cmd);
            break;

        case 'd':
            mkdir(f->dir,0755);
            break;

        default:
            break;
    }
}

// DIRECTORY SECTION

DIRECTORY init_directory(int b, char * d){
    DIRECTORY dir = malloc(sizeof(struct directory_st));
    dir->depth = b;
    dir->dir_name = strdup(d);
    return dir;
}

void get_directory(gpointer data, gpointer user_data){
    DIRECTORY dir = (DIRECTORY) data;
    GString * s = (GString*) user_data;
    if(branch >= 0){
        g_string_append(s,dir->dir_name);
        branch--;
    }
}

char * full_path(char * name){
    DIRECTORY ptr = init_directory(branch,name);
    g_ptr_array_insert(this_directory,ptr->depth,ptr);

    GString * s = g_string_new(NULL);
    g_ptr_array_foreach(this_directory,get_directory,s);
    return strdup(g_string_free(s,FALSE));
}


char * swap_token_for_name(char * s, char * tok, char * name){
    char * t = strstr(s,tok);
    char * r = malloc(strlen(s)-strlen(tok)+strlen(name));
    sprintf(r,"%s%s%s",strndup(s,t-s),name,
            strndup(t+strlen(tok),t-s+strlen(tok)));
    return r;
}
```