Processamento de Linguagens - TP1b

Carlos Pereira (A61887)
João Barreira (A73831)
Rafael Costa (A61799)

Abril 2017

Índice

1 Introdução

O presente trabalho consiste no desenvolvimento de filtros de texto recorrendo à ferramenta *Flex*. Os diferentes filtros devem ser produzidos com o recurso a *Expressões Regulares* para detetar *padrões de frases*. Para aumentar estas capacidades, foram propostos um conjunto de exercícios. Neste trabalho, optou-se por resolver o exercício seis, pois apresenta uma complexidade considerável e, para além disso, achamos interessante aumentar as capacidades da linguagem em *C* visto ser bastante utilizada na nossa área.

Ao longo deste relatório explicaremos, com detalhe, a resolução deste exercício, dando ênfase às *Expressões Regulares* e *Ações Semânticas*, bem como eventuais estruturas e variáveis auxiliares utilizadas. No final apresentaremos também o filtro de texto desenvolvido e exemplos da sua execução.

2 Inline Templates em linguagem C

Um template embedido em linguagem C contém, tipicamente, texto com várias variáveis e expressões.

De acordo com esta informação, foi desenvolvido um program em *Flex* que dado um ficheiro de template, gera, em C, um programa correspondente válido.

2.1 Expressões Regulares e Ações Semânticas

- **ˆ[a-zA-Z]+\=\{\{** – Esta expressão é usada para extrair tudo o que se encontra antes do corpo de um *template* (e.g. "Fli"e "Fhtml"na **Figura 1**) presentes no template. Desta forma, imprime-se no ficheiro resultante os nomes das funções (precedidas de "char*"), abrindo um parêntesis para o início dos argumentos e iniciando o estado "TEMPLATE".

- **<TEMPLATE>\[** – Identifica o início da codificação dos argumentos das funções (e.g. "ele"na função "Fli"da **Figura 1**). Inicia-se o estado "PARAMS".

- **<TEMPLATE>\}\}** – Identifica o fim de uma função do *template*. Quando um *template* termina procede-se à escrita de todas as variáveis passadas como parâmetro à função gerada. De seguida, procede-se à escrita do corpo da função associando, sempre que necessário, uma variável a uma certa instrução dessa função. No final, todas as variáveis auxialiares são igualdas a zero e o programa volta ao estado "INITIAL".

- **<TEMPLATE>ˆ[\t]*[a-zA-Z0-9<> /]+** – Serve para extrair uma linha de um *template*.

- **<TEMPLATE>[\t]*[a-zA-Z0-9<> /]+$** – Serva para extrair o conteúdo de uma linha de um *template* após o processamento de uma variável ou de um *map*. Neste caso, verifca-se o valor da varíavel *mapMode*. Se o valor de *mapMode* for igual a um, então isso quer dizer que se processou um *map* nesta linha. Caso contrário foi processada uma variável. Em ambos os casos armazena-se a linha de texto lida num *array*. Se o valor de *mapMode* for igual a zero, associa-se a linha processada ao nome da variável lida.

- **<TEMPLATE>[\t]*[a-zA-Z0-9<> /]+** – Serve para extrair uma linha de um *template*.

- **<PARAMS>"%"** – Identifica o início de uma variável (e.g. "tit"na **Figura 1**). Inicia-se o estado "VARS".

- **<VARS>"%]"** – Identifica o fim da codificação de uma variável. Volta-se a iniciar o estado "TEMPLATE".

- **<VARS>"MAP"** – Identifica o início da codificação de um map. Aciona-se a flag *mapMode*, guarda-se a linha de início do *map* em *mapLine* e inicia-se o estado "MAP".

- **<VARS>[a-zA-Z]+** – Serve para extrair o nome das variáveis. Um nome de uma variável apenas é guardado no *array* caso não exista.
- **<MAP>"%]"** – Identifica o fim da codificação de um map. Volta-se a iniciar o estado "TEMPLATE".
- **<MAP>[a-zA-Z]+** – Serve para extrair o nome dos parâmetros de um map.
- **.\n** – Todos os restantes carateres (incluindo \n), são escritos no ficheiro resultante.

2.2 Estrutura de Dados Globais

Neste exercício foi utilizado um *array* de strings para se armazenarem todas as linhas de texto provenientes de um *template*, bem como uma variável auxiliar para armazenar o seu tamanho. Com o uso desta estrutura de dados conseguiu-se, facilmente, armazenar todas as linhas de todos os *templates* de um ficheiro num modo sequencial.

De modo a tratar das variáveis associadas a um *template*, os nomes destas foram também guardados num *array* de *strings*. Foi também utilizada uma variável auxiliar que guarda do tamanho deste *array*.

Tendo como recurso a biblioteca *glib* usou-se uma estrutura do tipo *GTree* para se efetuar a associação entre uma linha de texto qualquer e uma variável presente nessa mesma linha.

Finalmente, para o tratamento de *maps* recorreu-se a três variáveis do tipo *string*, para armazenar o nome da função, o tamanho da lista e o nome da lista usados num *map*.

2.3 Filtro de Texto

```
%option noyywrap
%x TEMPLATE PARAMS VARS MAP

%{
   #include <glib.h>


   #define MAX_LINES 10000
   #define MAX_VARS  20


   gint comp(gconstpointer, gconstpointer);


   FILE* fp;
   int array_size = 0;
```

```
    char* array[MAX_LINES];
    char* line = NULL;
    char* var = NULL;
    GTree* tree;

    int vars_size = 0;
    char* vars[MAX_VARS];

    int mapMode = 0;
    int mapLine = 0;
    char* functionName = "";
    char* listLength = "";
    char* list = "";
%}

%%

<*>^[a-zA-Z]+\=\{\{ {
    int i = 0;

    fprintf(fp, "char* ");

    for (; i < yyleng && yytext[i] != '='; i++) {
        fprintf(fp, "%c", yytext[i]);
    }

    fprintf(fp, "(");

    BEGIN TEMPLATE;
}

<TEMPLATE>\[ {
    BEGIN PARAMS;
}

<TEMPLATE>\}\} {
    int i = 0;
    gchar* auxVar = "";

    for (; i < vars_size; i++) {
        if (i > 0) {
            fprintf(fp, ", ");
        }

        fprintf(fp, "char* ");
        fprintf(fp, "%s", vars[i]);
```

```
}

if (mapMode == 1) {
   if (i > 0) {
      fprintf(fp, ", int %s,", listLength);
   }
   else {
      fprintf(fp, "int %s,", listLength);
   }

   fprintf(fp, " char* %s[]", list);
}

fprintf(fp, ")\n{\n");
fprintf(fp, "\tchar BUF[10000];\n\tint j = 0;\n");

for (i = 0; i < array_size; i++) {
   if (mapMode == 1 && mapLine == i) {
      fprintf(fp, "\tj += sprintf(BUF + j, \"%s\");\n", array[i]);
      fprintf(fp,
                 "\tfor(int i = 0; i < %s; i++) {
                 \n\t\tj += sprintf(BUF + j, \"%%s\", %s(%s[i])); \n\t}\n
                 listLength,
                 functionName,
                 list);
      fprintf(fp, "\tj += sprintf(BUF + j, \"%s\\n\");\n", array[i + 1]);
   }
   else {
      auxVar = (gchar*)g_tree_lookup(tree, array[i]);

      if (auxVar != NULL && strcmp(auxVar, "") != 0) {
         fprintf(fp,
                    "\tj += sprintf(BUF + j, \"%s\\n\", %s);\n",
                    array[i],
                    auxVar);
      }
      else if (auxVar) {
         fprintf(fp, "\tj += sprintf(BUF + j, \"%s\\n\");\n", array[i]);
      }
   }
}

fprintf(fp, "\treturn strdup(BUF);\n}\n");

array_size = 0;
vars_size = 0;
```

```
   mapMode = 0;
   mapLine = 0;

   BEGIN INITIAL;
}

<TEMPLATE>^[ \t]*[a-zA-Z0-9<> /]+ {
   if (line == NULL) {
      line = strdup(yytext);
   }
   else {
      strcat(line, yytext);
   }
}

<TEMPLATE>[ \t]*[a-zA-Z0-9<> /]+$ {
   int auxMap = 0;

   if (line != NULL && mapMode != 1) {
      strcat(line, "%s");
      strcat(line, yytext);
      array[array_size++] = strdup(line);
   }
   else if (line != NULL && mapMode == 1) {
      auxMap = 1;

      if (var != NULL) {
         var[0] = '\0';
      }

      array[array_size++] = strdup(line);
      array[array_size++] = strdup(yytext);
   }
   else if (line == NULL) {
      if (var != NULL) {
         var[0] = '\0';
      }

      array[array_size++] = strdup(yytext);
   }

   if (auxMap != 1) {
      if (var == NULL) {
         g_tree_insert(tree, array[array_size - 1], "");
      }
      else {
```

```
            g_tree_insert(tree, array[array_size - 1], strdup(var));
        }
    }

    line = NULL;
}

<TEMPLATE>[ \t]*[a-zA-Z0-9<> /]+ {
    if (line == NULL) {
        line = strdup(yytext);
    }
    else {
        strcat(line, yytext);
    }
}

<PARAMS>"% " {
    BEGIN VARS;
}

<VARS>"%]" {
    BEGIN TEMPLATE;
}

<VARS>"MAP" {
    mapMode = 1;
    mapLine = array_size;
    BEGIN MAP;
}

<VARS>[a-zA-Z]+ {
    int i = 0;
    int existVar = 0;

    var = strdup(yytext);

    for (; i < vars_size; i++) {
        if (strcmp(var, vars[i]) == 0) {
            existVar = 1;
            break;
        }
    }

    if (existVar == 0) {
        vars[vars_size++] = strdup(var);
    }
```

```
}

<MAP>"%]" {
   BEGIN TEMPLATE;
}

<MAP>[a-zA-Z ]+ {
   functionName = strdup(strtok(yytext, " "));
   listLength = strdup(strtok(NULL, " "));
   list = strdup(strtok(NULL, " "));
}

.|\n { fprintf(fp, "%s", yytext);}

%%

gint comp(gconstpointer a, gconstpointer b)
{
   return (strcmp(a, b));
}


int main(int argc, char** argv)
{
   char* output;

   tree = g_tree_new(comp);

   if (argc == 2) {
      yyin = fopen(argv[1], "r");

      output = strtok(argv[1], ".");
      strcat(output, ".c");
      fp = fopen(output, "w");

      yylex();
   }

   fclose(fp);
   g_tree_destroy(tree);

   return 0;
}
```

2.4 Resultado da Execução

A partir do template que é apresentado na **Figura 1** – e através da execução do programa em *Flex* –, foi gerado o código em C correspondente, presente na **Figura 2**.

```
#include <stdio.h>
#include <string.h>

Fli={{<li> [% ele %] </li>
}}

Fhtml={{<html>
    <head><title>[% tit %]</title></head>
<body>
    <h1>[% tit %]</h1>
    <ul>[% MAP Fli comp items %]</ul>
</body>
</html>
}}

int main(){
    char * a[]={"expressões regulares","parsers","compiladores"};
    printf("%s\n",Fhtml("Conteudo programático", 3, a));
}
```

Figura 1 - Template embebido em linguagem C

```
#include <stdio.h>
#include <string.h>

char* Fli(char* ele)
{
    char BUF[10000];
    int j = 0;
    j += sprintf(BUF + j, "<li> %s </li>\n");
    strdup(BUF);
}


char* Fhtml(char* tit, char* tit, char* MAP, char* Fli, char* comp, char* items)
{
    char BUF[10000];
    int j = 0;
    j += sprintf(BUF + j, "<html>\n");
    j += sprintf(BUF + j, " <head><title>%s</title></head>\n");
    j += sprintf(BUF + j, "<body>\n");
    j += sprintf(BUF + j, " <h1>%s</h1>\n");
    j += sprintf(BUF + j, " <ul>%s</ul>\n");
    j += sprintf(BUF + j, "</body>\n");
    j += sprintf(BUF + j, "</html>\n");
    strdup(BUF);
}


int main(){
    char * a[]={"expressões regulares","parsers","compiladores"};
    printf("%s\n",Fhtml("Conteudo programático", 3, a));
}
```

Figura 2 - Código C resultante

A **Figura 3** apresenta outro template de exemplo e o respetivo resultado (**Figura 4**).

```
#include <stdio.h>
#include <string.h>

FraseExemplo={{ Funcao sem argumentos }}

int main(){
    printf("%s\n", FraseExemplo());
}
```

Figura 3 - Template embebido em linguagem C

```c
#include <stdio.h>
#include <string.h>

char* FraseExemplo()
{
    char BUF[10000];
    int j = 0;
    j += sprintf(BUF + j, " Funcao sem argumentos \n");
    return strdup(BUF);
}


int main(){
    printf("%s\n", FraseExemplo());
}
```

Figura 4 - Código C resultante

3 Conclusão

Com a realização deste trabalho, pudemos pôr em prática os conceitos adquiridos nas aulas teórico-práticas sobre a ferramenta *Flex*.

Percebemos a importância do uso das expressões regulares para descrever padrões de frases, de forma a podermos pegar num ficheiro e obtermos as informações que necessitamos.

Além disso, aprendemos também a desenvolver processadores de linguagem que nos permitiram, a partir das informações obtidas anteriormente, filtrar e modificar textos.

Assim sendo, em retrospetiva, achamos que a realização deste segundo trabalho prático foi bastante enriquecedora.