Processamento de Linguagens
MIEI (3º ano)

Trabalho Prático nº 1 - Parte B (FLEX)
Relatório de Desenvolvimento

Ano lectivo 16/17

João Pereira (A75273)

João Martins (A68646)

Manuel Freitas (A71646)

19 de Abril de 2017

Resumo

O *FLEX* é uma ferramenta extremamente poderosa e versátil que pode ser utilizada nos mais diversos domínios. Para se demonstrar isso, desenvolveu-se dois programas que fazem uso de analisadores léxicos gerados em *FLEX*: o primeiro gera definições de funções *C* a partir de *templates*; o segundo é utilizado no contexto de um programa simples a ser executado por um *robot*. Neste último, utiliza-se um analisador léxico para fazer *parse* do resultado de um comando de *Linux*, necessário ao funcionamento do *robot*.

Conteúdo

Capítulo 1

Introdução

A análise de texto é uma das tarefas mais comuns dos programadores. No entanto, o desenvolvimento de analisadores léxicos com recurso a linguagens tradicionais tais como o *C* ou o *Java* é um processo demorado, relativamente complexo e muito susceptível a erros. Devido a isso e ao facto dos analisadores léxicos serem utilizados em praticamente todas as áreas às quais a informática é aplicada, torna-se útil o uso de ferramentas tais como o *FLEX*, que permitem fácil e rapidamente especificar que padrões devem ser reconhecidos e as ações semânticas a realizar quando esses padrões forem encontrados.

A abordagem referida é extremamente vantajosa para quem programa porque muda o foco da implementação do mecanismo de deteção de padrões propriamente dito para os padrões que devem ser detetados e as ações a serem executadas. Para além disso, o tempo necessário para desenvolver um analisador léxico diminui significativamente, assim como a complexidade e tamanho do código necessário para o produzir. Outra vantagem de utilizar esta ferramenta é a facilidade de manutenção e extensão dos analisadores léxicos, uma vez que facilmente se adicionam ou mudam os padrões reconhecidos e as acções a eles associadas.

Por ser extremamente versátil, o *FLEX* pode ser utilizado em vários domínios. Neste relatório, descreve-se o seu uso em dois domínios diferentes. No primeiro, construiu-se um processador de *templates* para a linguagem de programção *C*. No segundo, construiu-se um programa a ser executado por um *robot* que utiliza a plataforma *ROS*. Em particular, esse *robot* faz uso do *output* de um comando *Linux* para determinar o seu comportamento. Esse *output* tem de ser processado e é para esse fim que se utiliza o *FLEX*.

Estrutura do Relatório

O relatório encontra-se dividido em 4 capítulos:

1. Introdução (o capítulo atual);
2. Templates em C;
3. Programa simples a ser executado por um Robot;
4. Conclusão.

No capítulo 2, explica-se o processo e as decisões de desenvolvimento de um analisador léxico capaz de traduzir funções definidas por *templates* para funções na síntaxe da linguagem de programação *C*.

No capítulo 3, descreve-se a metodologia e as decisões de desenvolvimento de um analisador léxico utilizado por um programa em *C++* a ser executado por um *robot*, com base na *framework ROS (Robot Operating System)*. Embora o programa *C++* referido seja relativamente simples, este está assente numa plataforma relativamente complexa (*ROS*) pelo que apenas se descreve o analisador léxico utilizado pelo mesmo. No capítulo 4, termina-se o relatório com uma síntese do trabalho desenvolvido e com as respetivas conclusões.

Capítulo 2

Templates em C

2.1 Análise e Especificação

2.1.1 Descrição Informal do Problema

Por vezes, quando se programa em *C* é necessário definir várias funções cujo comportamento é muito semelhante. Isso leva a grandes porções de código repetido que por sua vez tornam o código difícil de manter, dado que qualquer alteração numa função pode fazer com que seja necessário modificar as restantes. Uma possível solução para este problema passa por definir *templates* para cada conjunto de funções que partilham o mesmo comportamento. Neste trabalho, definiu-se um formato de *templates* para funções que devolvem *strings* resultantes da concatenação de expressões passadas no corpo do *template*.

Este exercício teve como objetivo a construção de um analisador léxico capaz de interpretar *templates* embebidos em *C*, de acordo com um formato especificado na próxima subsecção, e produzir as respetivas funções em *C*.

2.1.2 Formato dos Ficheiros de Input

Os ficheiros de *input* consistem em ficheiros de código *C* que podem conter definições de funções através de *templates* no formato:

```
Nome_da_Funcao = {{ corpo do template }}
```

No corpo do *template*, podem-se usar os seguintes padrões:

- `[% variavel %]` - quando este padrão é encontrado, regista-se `variavel` como um dos argumentos da função. O valor de `variavel` deverá constar no *output* da função definida no *template*, no local onde o padrão foi detetado;
- `[% MAP f c l %]` - sempre que se encontra este padrão, regista-se `c` e `l` como argumentos da função definida pelo *template* do qual o MAP faz parte e acrescenta-se ao resultado dessa função o resultado da aplicação da função `f` a todos os elementos da lista `l`;
- `[% C expressao_em_c %]` - quando este padrão é detetado, considera-se que `expressao_em_c` é uma expressão *C* cujo valor de retorno é do tipo *string*. No resultado da função que está a ser definida irá constar o resultado da expressão em *C*.

O texto de um *template* que não concorde com nenhum destes padrões é considerado como sendo texto que deverá constar no resultado da função definida por esse *template*, sem qualquer tipo de formatação.

2.2 Compilação e Instalação

Para se compilar o programa, deve-se invocar o comando `make` numa diretoria que contenha o ficheiro *inline_templates.fl* e a seguinte *Makefile*:

```
v=inline_templates
$v: $v.fl
  flex --noyywrap $v.fl
  cc lex.yy.c -o $v
  rm lex.yy.c

install: $v
  mkdir -p ~/bin/
  mv $v ~/bin/
```

Desta forma, é gerado o executável `inline_templates`. Também se poderá utilizar o comando `make install` para instalar o programa numa pasta que conste, preferencialmente, na variável `PATH`. Por omissão, o executável é instalado na pasta `~/bin`.

2.3 Utilização

O programa deverá ser invocado da seguinte forma:

```
$ ./inline_templates < FICHEIRO_INPUT > FICHEIRO_OUTPUT
```

O ficheiro `FICHEIRO_INPUT` corresponde a código fonte *C* com *templates* embebidos e o ficheiro `FICHEIRO_OUTPUT` corresponde ao código fonte em que as definições de funções com recurso a *templates* encontradas em `FICHEIRO_INPUT` foram substituídas por funções conforme a síntaxe da linguagem *C*.

2.3.1 Exemplo de Utilização

Abaixo segue-se um exemplo de utilização do programa *inline_templates*.

Input

```
#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[]={"expressoes regulares","parsers","compiladores"};
  printf("%s\n",Fhtml("Conteudo programático", 3, a));
}
```

Output

Passando o código *C* com *templates* embebidos acima apresentado para o comando `inline_templates`, obtém-se o seguinte *output*:

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

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

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

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

2.4 Concepção e Implementação da Resolução

Passa-se agora a descrever o funcionamento do analisador léxico desenvolvido. O código na sua totalidade pode ser consultado no apêndice A.1.

Antes de quaisquer outras considerações, é necessário descrever o funcionamento geral do programa. Sempre que é detetado código *C* "normal", i.e.: código que não ocorre dentro de um *template*, este é imediatamente impresso no `stdout`. Quando é detetada a definição de uma função com recurso a um *template*, o código dessa função é armazenado em *buffers* por forma a facilitar a sua construção. Para o efeito, são utilizados três *buffers*: um para o cabeçalho da função (`buf_cabecalho_func`), outro para o corpo (`buf_corpo_func`) e outro para armazenar uma linha da função de cada vez (`buf_linha_funcao`). Desta forma, o cabeçalho e o corpo das funções são construídos separadamente, evitando cálculos na enumeração dos parâmetros nos cabeçalhos das funções. Quando se chega ao fim da definição da função, o conteúdo dos *buffers* acima referidos é formatado de forma a obedecer à sintaxe da linguagem *C* e

completar a definição da função, culminando numa função *C* bem formada. Após o processo descrito, o código da função representada pelo *template* é impresso no `stdout`.

O nome IDENT (definido como sendo a expressão regular `[a-zA-Z_][a-zA-Z0-9_]*`) descreve os identificadores que são reconhecidos. Cada identificador é constituído por uma letra ou *underscore* seguido de 0 ou mais letras, números ou *underscores*.

Quando o analisador léxico gerado começa a executar, este encontra-se no estado `INITIAL`. Neste estado, se se localizar o padrão `{IDENT}[:space:]*=[:space:]*{{`, isto é, caso se encontre um identificador seguido de 0 ou mais ocorrências de espaços e de um sinal de igual, possivelmente com espaços a seguir e, por fim, duas chavetas, considera-se que foi encontrada a definição de uma função através de um *template*. Nestes casos, entra-se no contexto `decl_funcao` e inicia-se a construção da função nos *buffers* acima referidos. Concretamente, coloca-se o tipo de retorno da função e o respetivo nome no *buffer* do cabeçalho e a declaração de variáveis no *buffer* que armazena o corpo da função.

Seguidamente, se se encontrar o padrão `\[%[:space:]*`, entra-se no estado `var` e insere-se no corpo da função a instrução para imprimir o que foi lido da linha até ao momento. Caso contrário, o conteúdo lido é posto no *buffer* de linha e quando a linha terminar, i.e. quando se encontrar um *'\n'*, insere-se no corpo da função uma instrução para imprimir tudo o que foi lido da mesma. Se eventualmente se encontrar aspas, será inserida no *buffer* de linha a sequência de escape correspondente de forma a que quando esta for impressa, o símbolo impresso seja o pretendido.

No estado `var`, quando se encontra um identificador, pode-se suceder uma de três situações:

- o identificador é `MAP`, pelo que identifica um mapeamento. Neste primeiro caso, entra-se no estado `map`. Quando um identificador é encontrado neste estado, assume-se que este corresponde ao nome da função a aplicar aos elementos da lista e entra-se no estado `map_funcao_lida`. Quando é encontrado outro identificador, considera-se que este corresponde ao nome da variável cujo valor corresponde ao comprimento da lista e entra-se no estado `map_comprimento_lido`. Por fim, neste estado, quando se encontra um identificador considera-se que este corresponde ao nome da lista. Por esta altura, o mapeamento está completamente descrito e são colocadas no corpo da função as instruções que adicionam ao output o resultado do mapeamento.
- o identificador é `C`, o que sinaliza uma expressão *C* cujo tipo é `char *`. Neste caso, entra-se no estado `codigo_c`. No estado `codigo_c`, tudo o que se encontra até ao padrão `%]` é considerado parte da expressão *C* e, por conseguinte, a instrução para adicionar ao resultado o valor da expressão *C* é posta no corpo da função;
- nenhuma das anteriores, pelo que o identificador corresponde ao nome de uma variável. Neste último caso, insere-se no corpo da função a instrução para adicionar o valor da variável à *string* devolvida.

Quando o padrão `%]` é encontrado nos estados `var`, `map_comprimento_lido` ou `codigo_c`, considera-se que as situações anteriormente descritas terminaram e volta-se ao estado `decl_funcao`.

Quando se encontra o padrão `}}`, insere-se a lista de parâmetros no cabeçalho da função e a instrução `return strdup(NOME_DO_BUFFER);` no corpo da função, em que NOME_DO_BUFFER corresponde ao nome escolhido para a variável que contém o resultado da função obtida pelo analisador léxico. Depois disto, é impresso o cabeçalho e corpo da função, obtendo-se a função em síntaxe *C* correspondente à que estava definida no *template*.

Nota: Sempre que é lido o nome de um argumento da função, verifica-se se o seu nome já foi usado anteriormente nessa função e, caso tenha sido e os tipos não coincidam, o programa termina com um erro.

Capítulo 3

Programa simples a ser executado por um Robot

3.1 Análise e Especificação

O *ROS* é uma *framework* que permite desenvolver programas para *robots*. Esta *framework* funciona como um sistema distribuído composto, essencialmente, por nodos e tópicos. Os tópicos correspondem a filas de mensagens, enquanto que os nodos correspondem a executáveis que podem ler ou publicar em tópicos. Sugere-se aos leitores mais interessados que consultem o *website* `http://wiki.ros.org/ROS/Tutorials`.

3.1.1 Descrição Informal do Problema

Para se exemplificar as funcionalidades de um *robot*, decidiu-se construir um programa que faz com que o mesmo se desloque aleatoriamente enquanto que emite *beeps*. A frequência desses *beeps* é ditada pela intensidade do sinal *Wi-Fi* da rede *eduroam*. Quanto mais intenso for o sinal da rede, maior frequência terão os *beeps*, i.e. menor será o intervalo de tempo entre os sons. Através do comando `iwlist scan INTERFACE`, consegue-se obter informações sobre as redes detetadas. Pretende-se então construir um analisador do *output* deste comando, para ser executado por um nodo do programa.

3.1.2 Formato do *input*

O *input* do analisador léxico consiste no *output* do comando `iwlist scan INTERFACE`. Abaixo, pode-se consultar um exemplo do *output* deste comando:

```
1           Cell 01 - Address: 00:60:1D:01:23:45
2                     Channel:1
3                     Frequency:2.412 GHz (Channel 1)
4                     Quality=70/70  Signal level=0 dBm
5                     Encryption key:on
6                     ESSID:"MyNetwork"
7                     Bit Rates:1 Mb/s; 2 Mb/s; 5.5 Mb/s; 11 Mb/s; 6 Mb/s
8                     9 Mb/s; 12 Mb/s; 18 Mb/s
9                     Bit Rates:24 Mb/s; 36 Mb/s; 48 Mb/s; 54 Mb/s
10                    Mode:Ad-Hoc
11                    Extra:tsf=0000000000000000
12                    Extra: Last beacon: 12980ms ago
13                    IE: Unknown: 000473313030
14                    IE: Unknown: 010882040B160C121824
```

Nota: no exemplo acima só foi encontrada uma rede. Se tivessem sido descobertas mais redes, seria apresentada uma listagem de informações como a mostrada acima, para cada rede encontrada.

3.2 Concepção e Implementação da Resolução

Dado que as API's existentes da plataforma *ROS* estão escritas em *C++*, utilizou-se a opção `%option c++` para gerar um analisador nesta linguagem a partir do *FLEX*. A classe `wifi_info` é um tipo de mensagens que pode ser publicado num tópico. Esta classe armazena a seguinte informação acerca de uma rede *Wi-Fi*:

- endereço do *Access Point* - campo `address`;
- canal em que a rede opera - campo `channel`;
- ESSID da rede - campo `essid`;
- qualidade do sinal - campo `quality`;
- força do sinal em dBm - campo `signal_level_dBm`;

Para cada rede encontrada, pretende-se criar uma mensagem com as informações referidas acima para ser posteriormente publicada num tópico. O funcionamento deste analisador léxico é significativamente mais simples do que o que foi apresentado anteriormente. No entanto, existem alguns detalhes que o grupo quer ressalvar. Para cada um dos campos da rede referidos acima existe, no *output*, o nome do campo seguido de um separador que pode ser ':' ou '=', possivelmente seguido de espaços e por fim, o valor. A acção correspondente a cada um dos padrões consiste em retirar o nome do campo e o separador e espaços, deixando apenas o valor, e inserir o valor no campo correto da mensagem.

Dado que o ESSID é o último campo de interesse apresentado no *output* do comando, quando este valor é lido, retorna-se da função devolvendo o valor `NETWORK_READ`. Por essa razão, em cada invocação da função `yylex` lê-se, no máximo, as informações de uma rede. Quando se chega ao final do ficheiro, retorna-se o valor `END` que indica que o *output* do comando já foi completamente processado.

Capítulo 4

Conclusão

O grupo considera que o *FLEX* foi a ferramenta adequada para os trabalhos discutidos, sobretudo devido à rapidez com que permite desenvolver analisadores léxicos e à sua grande versatilidade e poder expressivo. No entanto, o grupo constatou que esta ferramenta apresenta algumas falhas que obrigam à adoção de soluções pouco elegantes, principalmente no que toca à integração da mesma com a linguagem *C++*. Além disso, o grupo é da opinião que as extensões à notação das linguagens regulares como as que existem noutros programas (por exemplo, a possibilidade de usar expressões regulares de *PERL* no programa *grep*) seriam uma boa adição ao *FLEX*, uma vez que aumentariam ainda mais o seu poder expressivo. Não obstante disso, as dificuldades encontradas foram sempre superadas e os resultados obtidos foram os esperados. Por esse motivo, o grupo considera que fez um bom trabalho, tendo também demonstrado que o *FLEX* pode ser aplicado em vários domínios que não os abordados nas aulas (neste caso, mostrou-se o uso do mesmo em robótica). Acima de tudo, ficou patente a utilidade desta ferramenta e a ubiquidade das tarefas de análise léxica.

Apêndice A

Código dos Programas

A.1 Conteúdo do ficheiro `inline_templates.fl`

```
IDENT           [a-zA-Z_][a-zA-Z0-9_]*

%x decl_funcao
%x nome_funcao_encontrado
%x dentro_aspas_funcao
%x map
%x map_funcao_lida
%x map_comprimento_lido
%x var
%x codigo_c

%top{
    #include <stdio.h>
    #include <string.h>
    #include <ctype.h>

    #define TAMANHO_BUFFER 10000
    #define TAMANHO_MAX_CABECALHO 4096
    #define TAMANHO_MAX_CORPO 20000
    #define TAMANHO_MAX_LINHA_FUNCAO 4096
    #define MAX_VARIAVEIS 1000

    /* usa-se char para armazenar informação do tipo de uma var de forma a poupar memoria */
    typedef char TIPO;
    #define INT 0
    #define STRING 1
    #define LISTA_STRINGS 2

    char buf_corpo_func[TAMANHO_MAX_CORPO];
    char buf_cabecalho_func[TAMANHO_MAX_CABECALHO];
    char buf_linha_funcao[TAMANHO_MAX_LINHA_FUNCAO];
    char* nomes_variaveis[MAX_VARIAVEIS];
    TIPO tipos_variaveis[MAX_VARIAVEIS];

    /* vars para leitura de mapas */
    char* nome_funcao;
    char* var_comprimento;
    char* nome_lista;

```

```
40      int pos_buf_cabecalho = 0;
41      int pos_buf_corpo = 0;
42      int pos_buf_linha = 0;
43      int pos_variaveis = 0;
44      int insereVar(char* nome, TIPO tipo);
45
46      /* vars usadas no programa gerado */
47      #define NOME_BUFF "BUF"
48      #define NOME_VAR1 "j"
49      #define NOME_VAR2 "i"
50
51      /* macros uteis para simplificar expressoes */
52      #define ADICIONA_CABECALHO(...) pos_buf_cabecalho += sprintf(buf_cabecalho_func +
            pos_buf_cabecalho, __VA_ARGS__)
53      #define ADICIONA_CORPO(...) pos_buf_corpo += sprintf(buf_corpo_func + pos_buf_corpo,
            __VA_ARGS__);
54  }
55
56  %%
57
58  <INITIAL>{IDENT}[:space:]*=[:space:]*\{\{          {
59    /* Encontrou um template de uma funcao */
60      int i, j;
61      BEGIN(decl_funcao);
62      char nome[128];
63      for(i = 0, j=0; !isspace(yytext[i]) && yytext[i] != '='; i++)
64          nome[j++] = yytext[i];
65      nome[j] = '\0';
66      pos_buf_cabecalho = pos_buf_corpo = pos_buf_linha = pos_variaveis = 0;
67      ADICIONA_CABECALHO("char* %s(", nome);
68      ADICIONA_CORPO("\tchar "NOME_BUFF"[%d];\n\tint "NOME_VAR1"=0;\n\tint "NOME_VAR2";\n",
            TAMANHO_BUFFER);
69  }
70
71  <decl_funcao>\[%[:space:]*          {
72      BEGIN(var);
73      if(pos_buf_linha != 0) {
74        buf_linha_funcao[pos_buf_linha++] = '\0';
75        ADICIONA_CORPO("\t"NOME_VAR1" += sprintf("NOME_BUFF"+"NOME_VAR1", \"%s\");\n",
              buf_linha_funcao);
76      }
77      pos_buf_linha = 0;
78  }
79
80
81  <decl_funcao>\"          {
82      buf_linha_funcao[pos_buf_linha++] = '\\';
83      buf_linha_funcao[pos_buf_linha++] = '\"';
84  }
85
86  <decl_funcao>.          {
87      buf_linha_funcao[pos_buf_linha++] = yytext[0];
88  }
89
90  <decl_funcao>\n          {
91      buf_linha_funcao[pos_buf_linha++] = '\\';
92      buf_linha_funcao[pos_buf_linha++] = 'n';
93      buf_linha_funcao[pos_buf_linha++] = '\0';
```

```
94      ADICIONA_CORPO("\t"NOME_VAR1" += sprintf("NOME_BUFF"+"NOME_VAR1", \"%s\");\n",
            buf_linha_funcao);
95      pos_buf_linha = 0;
96  }
97
98  <var,map_comprimento_lido,codigo_c>[:space:]*%\]          { BEGIN(decl_funcao); }
99
100 <var>{IDENT}          {
101     if(!strcmp(yytext, "MAP")){
102       BEGIN(map);
103     } else if(!strcmp(yytext, "C")){
104       BEGIN(codigo_c);
105     } else {
106       insereVar(yytext, STRING);
107       ADICIONA_CORPO("\t"NOME_VAR1" += sprintf("NOME_BUFF"+"NOME_VAR1", \"%s\", %s);\n", "%%
              s", yytext);
108     }
109 }
110
111 <map>{IDENT}          {
112     BEGIN(map_funcao_lida);
113     nome_funcao = strdup(yytext);
114 }
115
116 <map_funcao_lida>{IDENT}          {
117     BEGIN(map_comprimento_lido);
118     var_comprimento = strdup(yytext);
119     insereVar(var_comprimento, INT);
120 }
121
122 <map_comprimento_lido>{IDENT}          {
123     nome_lista = yytext;
124     insereVar(nome_lista, LISTA_STRINGS);
125
126     ADICIONA_CORPO("\tfor("NOME_VAR2"=0; "NOME_VAR2"<%s; "NOME_VAR2"++)\n", var_comprimento)
            ;
127     ADICIONA_CORPO("\t\t"NOME_VAR1" += sprintf("NOME_BUFF"+"NOME_VAR1", \"%s\", %s(%s["
            NOME_VAR2"]));\n", "%%s", nome_funcao, nome_lista);
128     free(nome_funcao);
129     free(var_comprimento);
130 }
131
132 <map_funcao_lida>.          {}
133
134 <codigo_c>.*/%\]          {
135     ADICIONA_CORPO("\t"NOME_VAR1" += sprintf("NOME_BUFF"+"NOME_VAR1", \"%s\",%s);\n", "%%s",
            yytext);
136 }
137
138 <decl_funcao>\}\}          {
139     BEGIN(INITIAL);
140
141     // completa cabecalho funcao
142     int i;
143     char* tipo = NULL;
144     for(i=0; i < pos_variaveis; i++){
145       switch(tipos_variaveis[i]){
146         case INT: tipo = "int"; break;
147         case STRING: tipo = "char*"; break;
```

```
148           case LISTA_STRINGS: tipo = "char**"; break;
149         }
150         ADICIONA_CABECALHO("%s %s,",tipo,nomes_variaveis[i] );
151         free(nomes_variaveis[i]);
152       }
153
154       switch(buf_cabecalho_func[pos_buf_cabecalho-1]){
155         case '(': buf_cabecalho_func[pos_buf_cabecalho++] = ')';
156           break;
157         case ',': buf_cabecalho_func[pos_buf_cabecalho-1] = ')';
158           break;
159       }
160       buf_cabecalho_func[pos_buf_cabecalho++] = '{';
161       buf_cabecalho_func[pos_buf_cabecalho++] = '\n';
162       buf_cabecalho_func[pos_buf_cabecalho] = '\0';
163
164       // completa corpo funcao
165       if(buf_linha_funcao[0]){
166           buf_linha_funcao[pos_buf_linha] = '\0';
167           ADICIONA_CORPO("\t"NOME_VAR1" += sprintf("NOME_BUFF"+"NOME_VAR1", \"%s\");\n",
                buf_linha_funcao);
168       }
169       ADICIONA_CORPO("\treturn strdup("NOME_BUFF");\n}");
170
171       // imprime declaracao funcao
172       printf("\n%s", buf_cabecalho_func);
173       printf("%s", buf_corpo_func);
174
175 }
176
177 %%
178
179 int insereVar(char* nome, TIPO tipo){
180   int i, found=0;
181   for(i=0; i < pos_variaveis; i++){
182     if(!strcmp(nome,nomes_variaveis[i])){
183       found = 1;
184       if(tipos_variaveis[i] != tipo){
185         fprintf(stderr, "ERRO: Está a usar variáveis iguais de tipos diferentes\n");
186         exit(1);
187       }
188       break;
189     }
190   }
191   if(!found){
192     nomes_variaveis[pos_variaveis] = strdup(nome);
193     tipos_variaveis[pos_variaveis++] = tipo;
194   }
195   return found;
196 }
197
198 int main(){
199   yylex();
200 }
```

A.2 Excerto do ficheiro `wifi_info_collector.fl`

Devido ao tamanho do programa e ao facto da maior parte do código estar fora do âmbito da Unidade Curricular, optou-se por apresentar apenas o analisador léxico. No entanto, o ficheiro na sua totalidade foi disponibilizado online por um dos elementos do grupo e pode ser consultado em `https://github.com/jcp19/ros_packages/blob/master/wifi_info/src/wifi_info_collector.l`.

```
1
2  %option c++
3
4  %{
5  #include "ros/ros.h"
6  #include "wifi_info/wifi.h"
7  #include <string>
8  #include <cstring>
9  #include <sstream>
10 #include <iostream>
11 #include <stdexcept>
12 #include <stdio.h>
13
14 wifi_info::wifi * info;
15
16 // return values:
17 #define NETWORK_READ 1
18 #define END 2
19
20 %}
21
22 %%
23 Cell     {
24           ;
25         }
26
27 Address:[[:space:]]([0-9A-F]+:)+[0-9A-F]+ {
28                         std::string s(yytext);
29                         std::string address = s.erase(0, strlen("Address: "));
30                         info->address = address;
31                 }
32 Channel:[0-9]+ {
33                 std::string s(yytext);
34                 std::string schannel = s.erase(0, strlen("Channel:"));
35                 int channel = std::stoi(schannel);
36                 info->channel = channel;
37 }
38
39 ESSID:\"[^\"]+\" {
40                 std::string s(yytext);
41                 std::string essid = s.erase(0, strlen("ESSID:\""));
42                 essid.erase(essid.size()-1, essid.size());
43                 info->essid = essid;
44                 return NETWORK_READ;
45 }
46
47 Quality=[0-9]+\/[0-9]+ {
48                 std::string q(yytext);
49                 q.erase(0, strlen("Quality="));
50                 info->quality = q;
51         }
52
```

```
53 Signal[[:space:]]level=-[0-9]+ {
54                     std::string sl(yytext);
55                     sl.erase(0, strlen("Signal level="));
56                     info->signal_level_dBm = std::stoi(sl);
57 }
58
59
60 <<EOF>> { return END; }
61 .|\n {;}
62 %%
```

Apêndice B

Código *C* com *templates* embebidos e *output* do programa `inline_templates`

Neste capítulo apresenta-se exemplos de código *C* com *templates* embebidos e, para cada exemplo, apresenta-se o resultado de o passar para o programa `inline_templates`.

B.1 Exemplo com *templates* que contêm variáveis e *maps*

Abaixo, segue-se um exemplo de código *C* com *templates* disponibilizado no enunciado. Os *templates* apresentam variáveis e *maps*:

```
#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[]={"expressoes regulares","parsers","compiladores"};
  printf("%s\n",Fhtml("Conteudo programático", 3, a));
}
```

Para o exemplo apresentado, o programa `inline_templates` produz o seguinte código *C*:

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


char* Fli(char* ele){
  char BUF[10000];
  int j=0;
  int i;
```

```
9    j += sprintf(BUF+j, "<li> ");
10   j += sprintf(BUF+j, "%s", ele);
11   j += sprintf(BUF+j, " < /li>\n");
12   return strdup(BUF);
13 }
14
15
16 char* Fhtml(char* tit ,int comp,char** items){
17   char BUF[10000];
18   int j=0;
19   int i;
20   j += sprintf(BUF+j, "<html>\n");
21   j += sprintf(BUF+j, "  <head><title>");
22   j += sprintf(BUF+j, "%s", tit);
23   j += sprintf(BUF+j, "</title></head>\n");
24   j += sprintf(BUF+j, "<body>\n");
25   j += sprintf(BUF+j, "  <h1>");
26   j += sprintf(BUF+j, "%s", tit);
27   j += sprintf(BUF+j, "</h1>\n");
28   j += sprintf(BUF+j, "  <ul>");
29   for(i=0; i<comp; i++)
30     j += sprintf(BUF+j, "%s", Fli(items[i]));
31   j += sprintf(BUF+j, "</ul>\n");
32   j += sprintf(BUF+j, "</body>\n");
33   j += sprintf(BUF+j, "</html>\n");
34   return strdup(BUF);
35 }
36
37 int main(){
38   char * a[]={"expressoes regulares","parsers","compiladores"};
39   printf("%s\n",Fhtml("Conteudo programático", 3, a));
40 }
```

B.2 Exemplo de código *C* com um *template* inválido

O exemplo que se segue é bastante parecido com o anterior, no entanto, ao contrário do exemplo anterior, é inválido uma vez que a variável `tit`, inicialmente usada na linha 8 como sendo do tipo *string*, é posteriormente utilizada na linha 11 como se fosse do tipo inteiro, ao ser passada como comprimento da lista de um MAP.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  Fli={{<li> [% ele %] < /li>
5  }}
6
7  Fhtml={{<html>
8    <head><title>[% tit %]</title></head>
9  <body>
10   <h1>[% tit %]</h1>
11   <ul>[% MAP Fli tit items %]</ul>
12 </body>
13 </html>
14 }}
15
16 int main(){
17   char * a[]={"expressoes regulares","parsers","compiladores"};
18   printf("%s\n",Fhtml("Conteudo programático", 3, a));
```

```
}
```

Como este exemplo é inválido (pelas razões mencionadas no parágrafo anterior), quando se passa o código apresentado para o programa `inline_templates`, o resultado é:

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


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

ERRO: Está a usar variáveis iguais de tipos diferentes
```

Como se pode observar, foi gerado código *C* válido para a função `Fli`, uma vez que o respetivo *template* é válido. No entanto, o *template* da função `Fhtml` era inválido. Dado que este erro é crítico e impede que a função obtida possa ser compilada, o processador de texto termina e imprime a mensagem no *stderr* "ERRO: Está a usar variáveis iguais de tipos diferentes".

B.3 Exemplo de *template* com uma expressão *C*

Um exemplo de um *template* com uma expressão *C* é:

```
Fli={{<li> [% ele %] </li>
<codigo> [% C strdup(X) %] </codigo>
}}
```

Passando o código apresentado para o programa `inline_templates`, obtém-se o seguinte *output*:

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

B.4 Exemplo de código *C* com um *template* para gerar o código *boilerplate* de uma página HTML

Um exemplo de um *template* de uma função *C* que produz o chamado código *boilerplate* de uma página HTML é:

```
BoilerPlate={{<html>
  <head>
    <meta charset="UTF-8">
    <title> [% title %] </title>
    <meta name="description" content="[% description %]">
  </head>
  <body>
  </body>
}}
```

Passando o código apresentado para o programa `inline_templates`, obtém-se como resultado a função:

```
char* BoilerPlate(char* title,char* description){
    char BUF[10000];
    int j=0;
    int i;
    j += sprintf(BUF+j, "<html>\n");
    j += sprintf(BUF+j, "  <head>\n");
    j += sprintf(BUF+j, "    <meta charset=\"UTF-8\">\n");
    j += sprintf(BUF+j, "    <title> ");
    j += sprintf(BUF+j, "%s", title);
    j += sprintf(BUF+j, " </title>\n");
    j += sprintf(BUF+j, "    <meta name=\"description\" content=\"");
    j += sprintf(BUF+j, "%s", description);
    j += sprintf(BUF+j, "\">\n");
    j += sprintf(BUF+j, "  </head>\n");
    j += sprintf(BUF+j, "  <body>\n");
    j += sprintf(BUF+j, "\n");
    return strdup(BUF);
}
```

Bibliografia

[1] W. E. Vern Paxson and J. Millaway, “Lexical analysis with flex,” Outubro 2016.