Universidade do Minho

Mestrado Integrado em Engenharia Informática

Departamento de Informática

Processamento de Linguagens

Pré-processador para LaTeX

António Sérgio Alves Costa A78296
Isabel Sofia da Costa Pereira A76550
Maria de La Salete Dias Teixeira A75281

06 de Maio de 2018

Conteúdo

1 Introdução

No âmbito da unidade curricular de Processamento de Linguagens, foi-nos proposto o desenvolvimento de um programa em *FLex* que funcione como um pré-processador para *LaTeX*. Assim, o objetivo deste trabalho é gerar um documento em *LaTeX* a partir de um documento numa linguagem com anotações simples, como o *MarkDown*.

Tendo em consideração todas as ferramentas utilizadas para formatar um texto, considerou-se essencial a migração de várias destas, como o negrito, itálico e imagens. Sendo que foi dada margem para alterações pelo docente, algumas das ferramentas processadas foram examinadas em *MarkDown* e outras desenvolvidas pelo grupo.

2 Descrição do Problema

Para o desenvolvimento deste projeto começou-se por definir as ferramentas que o grupo considerou essenciais na formatação de um texto. Assim, para além das requeridas no enunciado, em que se processou a escrita de negrito, itálico, 6 níveis de títulos (cabeçalhos) e listas de tópicos numerados e não-numerados, interpretou-se também um *tab* como um parágrafo, linhas vazias, imagens, links e a escrita de código numa linguagem específica. Todas estas ferramentas foram migradas a partir do *MarkDown*.

Para além disso, adicionou-se mais ferramentas numa linguagem desenvolvida pelo grupo, em que é possível escrever texto que não deve sofrer formatação e a execução das operações de soma, subtração, multiplicação e divisão.

De seguida, para a resolução do problema e desenvolvimento do programa, analisou-se as diferenças nas anotações entre as linguagens consideradas, dando origem à tabela apresentada.

Ferramenta	***MarkDown***	**Decisão do Grupo**	***LaTeX***
parágrafo	\t		\par
nova linha	\n		\newline
negrito	**texto**		\textbf{texto}
itálico	__texto__		\emph{texto}
lista numerada	[0-9]. texto		\begin{enumerate} \item texto \end{enumerate}
lista não-numerada	- texto		\begin{itemize} \item texto \end{itemize}
título 1	# texto		\title{texto}
título 2	## texto		\part{texto}
título 3	### texto		\chapter{texto}
título 4	#### texto		\section{texto}
título 5	##### texto		\subsection{texto}
título 6	###### texto		\subsubsection{texto}
imagem			\begin{figure}[h] \centering \includegraphics[scale=0.50]{path} \caption{legenda} \end{figure}
link	[texto](link)		\href{link}{texto}
link	<link>		\url{link}
código	‘‘‘ linguagem código ‘‘‘		\begin{minted}{linguagem} código \end{minted}
texto sem formatação		´´´ texto ´´´	texto
soma		\sum(2,4)	6
subtração		\sub(6,4)	2
multiplicação		\mul(2,3	6
divisão		\div(6,3)	2

Tabela 1: Tabela com Anotações de Fortamação de Texto.

Com as regras definidas, avançou-se para o desenvolvimento do ficheiro *FLex*, em que se apanha cada uma das ferramentas recorrendo a expressões regulares.

Para além disso, o ficheiro contém também uma *main*, que permite a inicialização do documento *LaTeX* com todos os *usepackage* necessários e com o \begin{document}, realiza o *yylex()*, para fazer parse do texto, e termina com \end{document}.

Para uma melhor perceção de como estas ferramentas devem ser manipuladas e como se podem relacionar entre si, foi desenvolvido um autómato global

que se apresenta na seguinte figura.

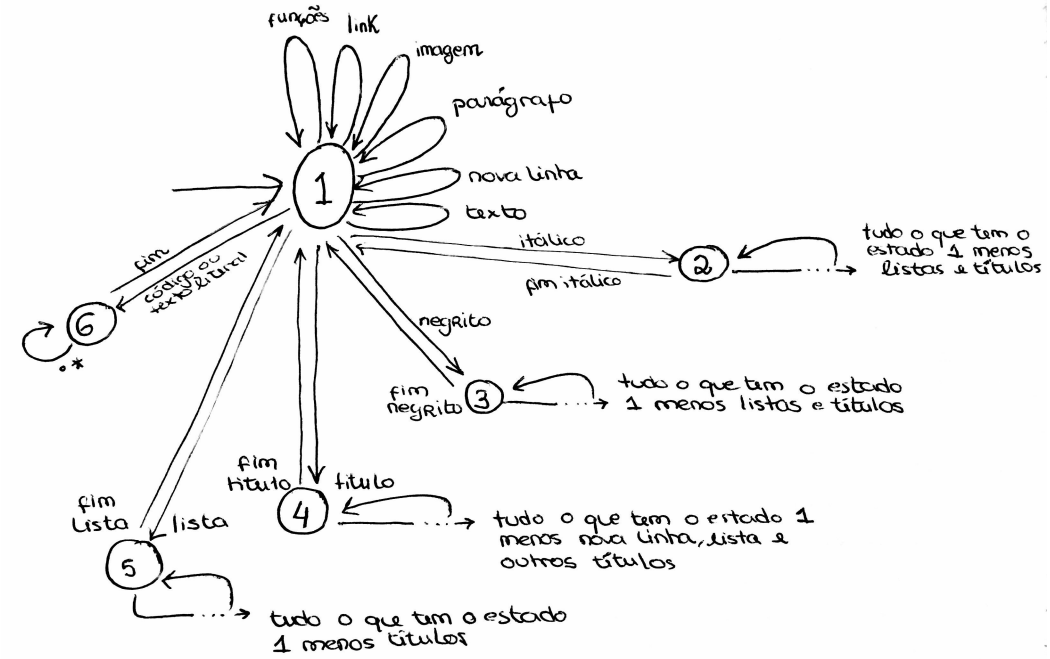

Figura 1: Autómato global

3 Implementação

Antes mesmo de codificar o programa, devido à natureza do *FLex* fazer com que se tenha que constantemente compilar o ficheiro **.l** (por exemplo, *flex processador.l*), e após isto, ter que se compilar o ficheiro *C* gerado (*lex.yy.c*), ou seja, *gcc lex.yy.c -o processador*, começou-se por criar uma *Makefile*.

O que esta *Makefile* faz é compactar os dois comandos mencionados anteriormente, portanto, para testar o programa apenas tem de escrever **make** e será compilado o ficheiro *.l* e o ficheiro *.c*.

Makefile:

```
processador : lex.yy.c
	gcc -o processador lex.yy.c -ll

lex.yy.c : processador.l
	flex processador.l
```

Ao analisar o texto percebeu-se que se teria de descobrir quando as tags eram abertas e fechadas. Este foi o primeiro problema do projeto e, para resolvê-lo, optou-se por implementar uma stack geral no sistema, o array **args**, como a seguinte:

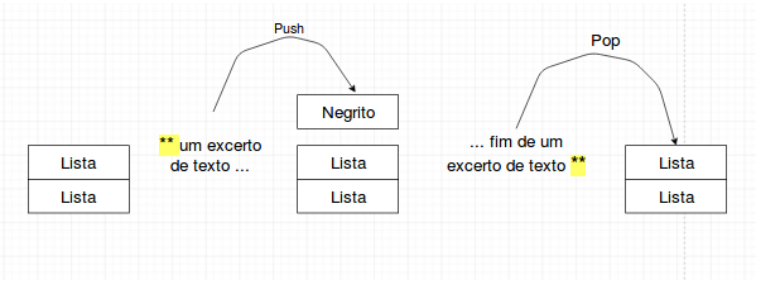

Figura 2: Desenho da stack desenvolvida

Como se pode verificar na stack anterior, sempre que se dá a inicialização de um contexto (no caso da imagem o **** que representa o negrito), adiciona-se o identificador desse contexto à stack e, a partir desse momento, o sistema encontra-se nesse contexto.

O contexto que o sistema está é sempre o do topo da stack. Por exemplo, na figura anterior o sistema encontrava-se no contexto **lista**, passou para **negrito** e retornou novamente para **lista**.

Para se remover o contexto da stack, como apresentado na figura anterior, o contexto em que o sistema se encontra tem de localizar o seu finalizador. Por exemplo, na figura anterior após o comando **** inicializou-se o contexto **negrito**, e este só termina quando encontrar o finalizador de contexto que, neste caso, é também ****.

Com isto, não só se possibilitou a oportunidade do sistema ter contextos dentro de contextos, como também obrigou o utilizador a usar a correta *syntax* da linguagem, já que os últimos contextos a serem abertos têm de ser os primeiros a ser fechados. Por exemplo, '**__ olá **__' está errado porque o último contexto a ser aberto '__' não foi o primeiro a ser fechado. A *syntax* correta seria '**__ olá __**'.

Depois de terminada a implementação desta estrutura geral começou-se a analisar os vários comandos necessários e outros comandos extras. Dentro dos comandos necessários pode-se encontrar o parágrafo, a nova linha, o negrito, o itálico, os cabeçalhos e as listas. Para além destes, decidiu-se implementar o texto sem formatação, para que fosse possível escrever highlight de código ou, por exemplo, *MarkDown* sem este se transformar em *LaTeX*. Ainda se adicionou imagens, links e, por último, operações, para se poder fazer simples operações aritméticas diretamente no ficheiro.

A implementação de todos estes contextos será explicada nos próximos tópicos do trabalho.

3.1 Parágrafo e Nova Linha

Tal como mencionado anteriormente, para o desenvolvimento de um parágrafo ou de uma nova linha é necessário expecificar duas coisas: a expressão regular que define o comando de cada um destes e a ação a executar quando se encontra tal comando, isto é, o comando correspondente em *LaTeX*.

Relativamente ao parágrafo, sabe-se que este é definido por um *tab* seguido de um espaço, sendo assim a expressão regular e a ação utilizadas foram **"\t "** e **"\par "**, correspondentemente.

Quanto à criação de uma nova linha, esta é elaborada quando se clica na tecla *ENTER*. Desta forma, a expressão regular e o comando em *LaTeX* correspondentes são ambos **"\n"**.

Para além disso, como estes comandos podem ser encontrados em vários contextos diferentes, utiliza-se a notação <*>.

Desta forma, o código em *FLex* gerado é o seguinte:

```
1 <*>"\\t "     { printf("\\par "); }
2 <*>"\n"       { printf("\n"); }
```

Como exemplo, apresenta-se de seguida um texto que utiliza os dois comandos explicitados acima.

```
Caros Alunos

No email anterior esqueci de dizer que a sala para as apresentações
do TP1 é a DI-0.02 (numeração antiga), a 1ª do lado esquerdo quem
entra no piso 0 do DI. \t Esse é o ponto de encontro/referencia; se
estiver ocupada indicaremos lá qual será a sala.
\t Até breve.
```

Tal caso, origina o seguinte código *LaTeX*.

```
Caros Alunos

No email anterior esqueci de dizer que a sala para as apresentações
do TP1 é a DI-0.02 (numeração antiga), a 1ª do lado esquerdo quem
entra no piso 0 do DI. \par Esse é o ponto de encontro/referencia; se
estiver ocupada indicaremos lá qual será a sala.
\par Até breve.
```

3.2 Negrito e Itálico

Para implementar o negrito e o itálico deparou-se com a necessidade de criar dois contextos, sendo estes o contexto **negrito** e o **italico**. Esta necessidade deveu-se ao facto de se poder incorporar itálico dentro de negrito e vice-versa, daí ser necessário saber o contexto em que se encontra.

Sabendo a anotação para estes em *MarkDown*, criou-se as expressões regulares necessárias para apanhar cada um destes tipos. No entanto, ao contrário

do *MarkDown*, que apenas considera tratar-se de negrito ou itálico quando se fecha a anotação, considerou-se que se trata do mesmo logo que é encontrado o início da anotação. Para prevenção, no caso de a anotação não ser corretamente fechada, é enviada uma mensagem de erro para o terminal. Por exemplo, no caso do negrito, quando se encontra ** pela expressão regular, inicia-se o contexto **negrito**, e processa-se todo o texto presente entre os ** ** dentro de **\textbf{}**, tal como se pode verificar no código presente a seguir.

De notar a utilização da stack para se poder retornar ao contexto anterior e a notação <*> para que estes contextos possam ser utilizados dentro de outros.

```
1  <negrito>\*\*  { printf("}");
2                   i--;
3                   BEGIN args[i-1]; }
4
5  <italico>\_\_  { printf("}");
6                   i--;
7                   BEGIN args[i-1]; }
8
9  <*>\*\*        { printf("\\textbf{");
10                  for(j=0; j<i; j++) {
11                     if(args[j]==negrito) {
12                         fprintf(stderr,"\nErro na linha
                            ↪  %d.\n",yylineno);
13                         error();
14                     }
15                  }
16                  args[i] = negrito;
17                  i++;
18                  BEGIN negrito;
19                }
20
21 <*>\_\_        { printf("\\emph{");
22                  for(j=0; j<i; j++){
23                     if(args[j]==italico) {
24                         fprintf(stderr,"\nErro na linha
                            ↪  %d.\n",yylineno);
25                         error();
26                     }
27                  }
28                  args[i] = italico;
29                  i++;
30                  BEGIN italico;
31                }
```

Para uma melhor perceção da solução implementada, são apresentados exemplos simples a seguir.

O seguinte código:

```
**negrito**
__italico__
**__composto__**
```

Passaria a:

```
\textbf{negrito}
\emph{italico}
\textbf{\emph{composto}}
```

3.3 Cabeçalhos

Para a migração de cabeçalhos houve a necessidade de criar o contexto **titulos**, pois é possível aplicar formatações dentro de cabeçalhos, como o negrito ou o itálico, sendo então o mesmo caso que no tópico anterior em que se pode ter contextos dentro de contextos. Assim, utiliza-se a stack para que seja possível retornar ao contexto anterior.

Analisando o *MarkDown*, conclui-se que existem seis tipos de títulos, sendo cada um mais interior do que o anterior. Por exemplo, considerou-se que a presença de um # seria o título principal em *LaTeX*, logo o texto presente à frente do # é inserido dentro de um \title{, e que a presença de #### seria uma secção, logo o texto à frente dos # é inserido dentro de um \par{.

Para finalizar este contexto é apenas necessário clicar na tecla *ENTER*, isto é, **"\n"** em expressão regular, imprimindo-se assim a chaveta de fecho.

Desta forma, desenvolveu-se o código *FLex* apresentado de seguida.

```
1  <titulos>"\n"  { printf("}");
2                   i--;
3                   BEGIN args[i-1]; }
4  ^"# "          { printf("\\title{");
5                   args[i] = titulos;
6                   i++;
7                   BEGIN titulos; }
8  ^"## "         { printf("\\part{");
9                   args[i] = titulos;
10                  i++;
11                  BEGIN titulos; }
12 ^"### "         { printf("\\chapter{");
13                  args[i] = titulos;
14                  i++;
15                  BEGIN titulos; }
16 ^"#### "       { printf("\\section{");
17                  args[i] = titulos;
```

```
18                  i++;
19                  BEGIN titulos; }
20  ^"##### "     { printf("\\subsection{");
21                  args[i] = titulos;
22                  i++;
23                  BEGIN titulos; }
24  ^"###### "    { printf("\\subsubsection{");
25                  args[i] = titulos;
26                  i++;
27                  BEGIN titulos; }
```

Para uma melhor perceção da solução implementada, são apresentados exemplos simples a seguir.

O seguinte código:

```
# Titulo1
## Titulo2
### Titulo3
#### Titulo4
##### Titulo5
###### Titulo6
```

Passaria a:

```
\title{Titulo1}
\part{Titulo2}
\chapter{Titulo3}
\section{Titulo4}
\subsection{Titulo5}
\subsubsection{Titulo6}
```

3.4 Listas de Tópicos Não-Numerados e Numerados

Para implementar as listas teve de se criar uma substack apenas para estas, o array **l**. Isto deveu-se ao facto de se poder ter listas dentro de listas, logo tem que se saber quando umas acabam e outras começam.

Esta estrutura foi, possivelmente, a mais difícil de codificar do projeto, já que havia sempre pequenos problemas, como por exemplo saber quando uma lista começa dentro de outra. Para tal, decidiu-se utilizar a maneira da linguagem *MarkDown*, que é a indentação. Sendo assim, sempre que se quer colocar uma lista dentro de outra tem que se utilizar mais um *tab* para cada elemento dessa lista do que a lista em que está inserida. Desta forma, uma lista só termina se o elemento a seguir apresentar menos *tab* ou se a linha começar com '\n'.

Para além disso, apesar de só haver um contexto **lista**, existem dois tipos de lista, as *numeradas* e as *não-numeradas*. Como o código iria ser o mesmo, decidiu-se juntar as duas num só contexto, no entanto é necessário distinguilas. Esta distinção é feita no início de cada elemento da lista, isto é, se a lista

começar com um '- ' significa que é uma lista não-numerada, por outro lado, se começar com um número '[0-9]. ' significa que é uma lista numerada.

Tendo em conta todos os parâmetros anteriores, o código desenvolvido foi o seguinte:

```
<lista>^(\t*)(("- ")|([0-9]+". "))
                    { //contar o número de tabs
                    for(j=0; j<strlen(yytext); j++) {
                        if(yytext[j]=='\t') tabs++;
                    }
                    //se for igual ao atual então
                    //encontramo-nos na mesma lista
                    if(tabs==currenttabs) printf("\t\\item ");
                    //se for menor, fechar listas necessárias
                    //e voltar à lista correspondente
                    else if(tabs<currenttabs) {
                        b = currenttabs-tabs;
                        currenttabs=tabs;
                        while(b){
                          if(l[k-1]==0)
                          ↪  printf("\\end{itemize}\n");
                            else printf("\\end{enumerate}\n");
                            k--; i--; b--;
                          }
                          printf("\t\\item ");
                          BEGIN args[i-1];
                      }
                      //se for maior, iniciar nova lista aninhada
                      else{
                         if(yytext[tabs]=='-') {
                           l[k] = 0;
                           k++;
                           printf("\\being{itemize}\n");
                           printf("\t\\item ");
                         }
                         else {
                           l[k] = 1;
                           k++;
                           printf("\\being{enumerate}\n");
                           printf("\t\\item ");
                         }
                          currenttabs=tabs;
                          args[i] = lista;
                          i++;
                          BEGIN lista;
                      }
```

```
                    tabs=0;
                    }

<lista>^\n
          { while(k){ //fechar todas as listas necessárias
              if(l[k-1]==0) printf("\\end{itemize}\n");
              else printf("\\end{enumerate}\n");
              k--;
              i--;
            }
            currenttabs=0;
            BEGIN 0; //voltar ao contexto inicial
          }

^(\t*)(-|[0-9]+.)" "
              { //contar o número de tabs
              for(j=0; j<strlen(yytext); j++) {
                 if(yytext[j]=='\t') tabs++;
              }
              //verificar qual lista inicializar
              if(yytext[tabs]=='-') {
                l[k] = 0;
                k++;
                printf("\\being{itemize}\n\t\\item ");
              }
              else {
                l[k] = 1;
                k++;
                printf("\\being{enumerate}\n\t\\item ");
              }
              //iniciar contexto
              if(tabs>=currenttabs) {
                currenttabs=tabs;
                args[i] = lista;
                i++;
                BEGIN lista;
              }
              tabs=0;
            }
```

Para uma melhor perceção da solução implementada, são apresentados exemplos simples a seguir.

O seguinte código:

```
1. primeiro
5. segundo
```

Passaria a:

```
\being{enumerate}
        \item primeiro
        \item segundo
\end{enumerate}
```

O seguinte código:

```
- primeiro
- segundo
```

Passaria a:

```
\being{itemize}
        \item primeiro
        \item segundo
\end{itemize}
```

O seguinte código:

```
1. primeiro
5. segundo
        - ola1
        - ola2
```

Passaria a:

```
\being{itemize}
        \item primeiro
        \item segundo
    \being{itemize}
      \item ola1
      \item ola2
    \end{itemize}
\end{itemize}
```

3.5 Texto sem Formatação

Dentro dos textos sem formatação encontram-se dois tipos de comandos: um para escrever código numa linguagem à escolha e outro para escrever texto sem este ser processado.

```
```linguagem                ´´´ texto
código                     ´´´
```
```

Ao encontrar tal padrão, o programa deve substituir esses comandos pelos correspondentes em *LaTeX*:

```
\begin{minted}{linguagem}       texto
código
\ end{minted}
```

Assim, desenvolveu-se duas expressões regulares capazes de identificar tal padrão.

```
```.+                           ´´´
```

Para finalizar utiliza-se “‘ ou ´´´ no início de uma frase que poderá conter *tabs*. Desta forma, as expressões regulares de finalização são as seguintes:

```
^[\t ]*```                     ^[\t ]*´´´
```

Como estes comandos não podem processar o texto contido dentro deles, isto é, não podem igualar o texto encontrado com nenhuma expressão regular definida no programa, criou-se o contexto **literal**. Este contexto é inicializado sempre que é encontrado um comando desta natureza e só termina com o devido comando de finalização. Dentro desse processo, o contexto tem como objetivo imprimir todo o texto que encontrar. Desta forma, este contexto apresenta-se no topo do programa para que sejam executadas apenas as expressões regulares definidas nele e não as definidas abaixo no programa.

Mais uma vez, como estes comandos podem ser encontrados em vários contextos diferentes, utiliza-se a notação <*>. Para além disso, como se inicia um contexto na sua utilização, é necessário guardar na stack que se entrou nesse contexto e ao sair, retirá-lo da stack e voltar ao contexto anterior.

```
1  <literal>^[\t ]*```        { printf("\\ end{minted}");
2                               i--;
3                               BEGIN args[i-1];
4                             }
5
6  <literal>^[\t ]*´´´        { i--;
7                               BEGIN args[i-1];
8                             }
```
```

```
9
10  <literal>.+                 { printf("%s",yytext); }
11
12  <*>```.+                    { printf("\\begin{minted}{%s}",yytext+3);
13                                args[i] = literal;
14                                i++;
15                                BEGIN literal;
16                              }
17
18  <*>```                      { args[i] = literal;
19                                i++;
20                                BEGIN literal;
21                              }
```

De seguida apresenta-se um exemplo de input.

````
```C
//código C
int main() {
    printf("hello, world");
    return 0;
}
```

    ```
Links em markdown:
[texto](http://github.com)
<http://github.com>
    ```
````

Após processado o exemplo, obtem-se:

```
\begin{minted}{C}
//código C
int main() {
    printf("hello, world");
    return 0;
}
\ end{minted}


Links em markdown:
[texto](http://github.com)
<http://github.com>
```

3.6 Links

A possibilidade de inserir links em relatórios é algo bastante importante e útil, daí se ter decidido o processamento dos mesmos.

Ao analisar o *MarkDown*, concluiu-se que existem dois tipos principais de links, sendo o primeiro uma referência de um link num excerto de texto e o segundo um link explícito, sem textos associados.

Para a migração dos dados, foi necessário desenvolver expressões regulares, que permitam "apanhar"os links corretamente em qualquer contexto (notação <*>). Então, para encontrar um url, desenvolveu-se a expressão <*>\<.+\> em que o texto presente entre <> corresponde ao url. Para encontrar um link com referência, desenvolveu-se a expressão <*>\[.+\]\(.+\). Deste modo, sabe-se que o texto presente entre os parêntesis retos corresponde ao link e o texto presente entre os parêntesis curvos corresponde à referência. Assim, foi necessária a utilização de dois arrays de *char*: **text** e **link**. O primeiro corresponde ao texto associado ao link e o segundo, tal como o nome indica, corresponde ao link em si.

Tendo em conta todos os parâmetros anteriores, o código desenvolvido foi o seguinte:

```
1  <*>\[.+\]\(.+\)    { char text[yyleng], link[yyleng];
2                       //preenchimento do texto
3                       for(b=1; yytext[b]!=']' && b<yyleng; b++) {
4                         text[b-1]=yytext[b];
5                       }
6                       text[b-1]='\0';
7                       //preenchimento do link
8                       for(j=0, b+=2; yytext[b]!=')' &&
                         ↪  b<yyleng-1; j++, b++) {
9                         link[j]=yytext[b];
10                      }
11                      link[j]='\0';
12                      //transformação para LaTeX
13                      printf("\\href{%s}{%s}",text,link); }
14
15 <*>\<.*\>        { yytext[yyleng-1]='\0'; //remoção do char '>'
16                   //remoção do '<' e transformação para LaTeX
17                   printf("\\url{%s}",yytext+1); }
```

Para uma melhor perceção da solução implementada, são apresentados exemplos simples a seguir.

O seguinte código:

```
[texto](http://github.com)
<http://www.example.com>
```

Passaria a:

```
\href{http://github.com}{Texto}
\url{http://www.example.com}
```

3.7 Imagem

As imagens são definidas pela seguinte estrutura:

```
![texto opcional](imagem.png "legenda opcional")
```

Desta forma desenvolveu-se a seguinte expressão regular capaz de identificar tal padrão: **!\[.*\]\(.+(" "\".*\")?\)**.

- ! - identifica que o comando tem de se inicializar com tal símbolo;
- \[.*\] - a seguir ao símbolo !, o comando tem dois parênteses retos, sendo o que está dentro destes opcional;
- (" "\".*\")? - é opcional um espaço seguido de informação, também opcional, apresentada entre aspas;
- \(.+\(" "\".*\")?) - a seguir aos parênteses retos, apresentam-se dois parênteses curvos sendo obrigatório conteúdo dentro destes e opcional que esse conteúdo seja seguido de um espaço com mais informação entre aspas.

Para além disso, como este comando pode ser encontrado em vários contextos diferentes, utiliza-se a notação <*****>.

Ao encontrar tal padrão, o programa deve substituir esse comando pelo correspondente em *LaTeX*:

```
\begin{figure}[h]
\centering
\includegraphics{CAMINHO}
\caption{LEGENDA}
\end{figure}
```

Desta forma, foi conveniente a criação de dois arrays: **path** e **cap**. O primeiro contem a string associada ao caminho da imagem e o segundo contem a string correspondente à legenda. O preenchimento de cada um destes arrays encontra-se explicado no código *FLex* apresentado de seguida.

```
<*>!\[.*\]\(.+(" "\".*\")?\)
                { char path[yyleng], cap[yyleng];
                //ignorar os char até ao 1º parênteses curvo
                for(b=2; yytext[b]!='(' && b<yyleng; b++);
                //apanhar os char até encontrar um espaço
                //ou um parênteses curvo a fechar
                for(j=0, b++; (yytext[b]!=')' && yytext[b]!=' ')
                              && b<yyleng; j++, b++) {
                   path[j]=yytext[b]; //construção do CAMINHO
                }
                path[j]='\0'; //estabelecer fim do array
                //se o PATH parou porque encontrou o parênteses
                //então não existe legenda, se não, costruir o CAP
                //até encontrar um parênteses curvo a fechar
                for(j=0; yytext[b]!=')' && b<yyleng; j++, b++) {
                   cap[j]=yytext[b+2]; //construção da LEGENDA
                }
                //estabelecer fim do array
                if(j) cap[j-3]='\0';
                else cap[0]='\0';
                //comando latex com as strings corretas
                printf("\\begin{figure}[h]\n");
                printf("\\centering\n");
                printf("\\includegraphics{%s}\n",path);
                printf("\\caption{%s}\n",cap);
                printf("\\end{figure}");
                }
```

De seguida apresenta-se um exemplo de input e output.

```
![](gatos.jpg "Gato pequeno")
```

```
\begin{figure}[h]
\centering
\includegraphics{gatos.jpg}
\caption{Gato pequeno}
\end{figure}
```

3.8 Operações (soma, divisão, multiplicação, subtração)

O último extra que se decidiu criar foi a possibilidade de realizar operações no código. Assim sendo, através das tags \sum, \sub, \mul e \div pode-se calcular operações de soma, subtração, multiplicação e divisão, respetivamente.

Para tal, tem que se utilizar as tags com os números separados por vírgulas entre parêntesis. A soma irá adicionar todos os números entre parêntesis ((a,b,c)=a+b+c), a multiplicação irá multiplicar todos os números sucessivamente ((a,b,c)=a*b*c), a subtração irá subtrair todos os números sucessivamente ((a,b,c)=a-b-c) e a divisão irá dividir todos os números sucessivamente ((a,b,c)=a/b/c).

Para realizar esta tarefa apenas tem que se localizar as tags e identificar os números entre parêntesis e, após isso, realizar a operação e escrever o resultado.

Para além disso, como estes comandos podem ser encontrados em vários contextos diferentes, utiliza-se a notação <*>.

Tendo em conta todos os parâmetros anteriores, o código que desenvolvido foi o seguinte:

```
1  float soma(float* buf, int vir){
2    float s = 0;
3    for(int l=0; l<vir; l++){
4      s += buf[l];
5    }
6    return s;
7  }
8
9  float divi(float* buf, int vir){
10   float s = buf[0];
11   for(int l=1; l<vir; l++){
12     s = s/buf[l];
13   }
14   return s;
15 }
16
17 float subt(float* buf, int vir){
18   float s = buf[0];
19   for(int l=1; l<vir; l++){
20     s -= buf[l];
21   }
22   return s;
23 }
24
25 float mult(float* buf, int vir){
26   float s = 1;
27   for(int l=0; l<vir; l++){
28     s *= buf[l];
29   }
```

```
30    return s;
31  }
32
33  <*>\\sum\([^)]*\)
34                { int vir=0;
35                  int letra=0;
36                  char buf[100];
37                  float num[20];
38                  memset(buf, 0, strlen(buf));
39                  for(int l=5; l<strlen(yytext); l++){
40                    if(yytext[l]==','|| yytext[l]==')') {
41                      num[vir]=atof(buf);
42                      vir++;
43                      memset(buf, 0, strlen(buf));
44                      letra=0;
45                    }
46                    else {
47                      buf[letra]=yytext[l];
48                      letra++;
49                    }
50                  }
51                  float res = soma(num, vir);
52                  printf("%f",res);
53                }
54
55  <*>\\div\([^)]*\)
56                { int vir=0;
57                  int letra=0;
58                  char buf[100];
59                  float num[20];
60                  memset(buf, 0, strlen(buf));
61                  for(int l=5; l<strlen(yytext); l++){
62                    if(yytext[l]==',' || yytext[l]==')') {
63                      num[vir]=atof(buf);
64                      vir++;
65                      memset(buf, 0, strlen(buf));
66                      letra=0;
67                    }
68                    else {
69                      buf[letra]=yytext[l];
70                      letra++;
71                    }
72                  }
73                  float res = divi(num, vir);
74                  printf("%f",res);
```

```
75              }
76
77  <*>\\mul\([^)]*\)
78              { int vir=0;
79                int letra=0;
80                char buf[100];
81                float num[20];
82                memset(buf, 0, strlen(buf));
83                for(int l=5; l<strlen(yytext); l++){
84                  if(yytext[l]==',' || yytext[l]==')') {
85                    num[vir]=atof(buf);
86                    vir++;
87                    memset(buf, 0, strlen(buf));
88                    letra=0;
89                  }
90                  else {
91                    buf[letra]=yytext[l];
92                    letra++;
93                  }
94                }
95                float res = mult(num, vir);
96                printf("%f",res);
97              }
98
99  <*>\\sub\([^)]*\)
100             { int vir=0;
101               int letra=0;
102               char buf[100];
103               float num[20];
104               memset(buf, 0, strlen(buf));
105               for(int l=5; l<strlen(yytext); l++){
106                 if(yytext[l]==',' || yytext[l]==')') {
107                   num[vir]=atof(buf);
108                   vir++;
109                   memset(buf, 0, strlen(buf));
110                   letra=0;
111                 }
112                 else {
113                   buf[letra]=yytext[l];
114                   letra++;
115                 }
116               }
117               float res = subt(num, vir);
118               printf("%f",res);
119             }
```

Para uma melhor perceção da solução implementada, são apresentados exemplos simples a seguir.

O seguinte código:

```
\sum(1,2) \div(10,5) \mul(2,5) \sub(10,5)
\mul(2,5) \sum(1,2,-5,200)
\sub(10,5)
```

Passaria a:

```
3.000000 2.000000 10.000000 5.000000
10.000000 198.000000
5.000000
```

4 Exemplos de Utilização

Para uma melhor perceção do trabalho desenvolvido, estão explícitos neste tópico alguns exemplos de aplicação do programa, onde se mostra várias das combinações de contextos possíveis. Com isto é possível demonstrar todas as funcionalidades do projeto para qualquer utilizador interessado em utilizar o mesmo. Para visualizar os casos simples de aplicação, também é possível ver outros exemplos elementares no subtópicos presentes na *Implementação*.

O seguinte código combina negrito com itálico:

```
__italico **composto**__
__**isto dá erro__**
```

Passaria a:

```
\emph{italico \textbf{composto}}
Erro na linha 2.
```

O seguinte código combina listas não-numeradas com negrito e itálico:

```
- Bem vindo
- **as __linhas__**
```

Passaria a:

```
\being{itemize}
        \item Bem vindo
        \item \textbf{as \emph{linhas}}
\end{itemize}
```

O seguinte código combina listas não-numeradas com listas numeradas e listas aninhadas:

```
1. primeiro
5. segundo
        - ola1
        - ola2
                6. terceiro
        - ola3
```

Passaria a:

```
\being{enumerate}
        \item primeiro
        \item segundo
\being{itemize}
        \item ola1
        \item ola2
\being{enumerate}
        \item terceiro
\end{enumerate}
        \item ola3
\end{itemize}
\end{enumerate}
```

O seguinte código mostra parte de dois programas:

```
```javascript
var s = "JavaScript syntax highlighting";
alert(s);
```
- ```python
        s = "Python syntax highlighting"
        print s
        ```
```

Passaria a:

```
\begin{minted}{javascript}
var s = "JavaScript syntax highlighting";
alert(s);
\ end{minted}
\begin{itemize}
        \item \begin{minted}{python}
        s = "Python syntax highlighting"
        print s
\ end{minted}
\end{itemize}
```
```

O seguinte código combina listas não-numeradas com operações numéricas:

```
- \mul(2,3,4)
- \div(10,2,5)
```

Passaria a:

```
\begin{itemize}
    \item 24.000000
    \item 1.000000
\end{itemize}
```

O seguinte código mostra o exemplo de um capítulo e de um link:

```
### Show me how it looks!
[There you go](/Screenshots).
```

Passaria a:

```
\chapter{Show me how it looks!}
\href{There you go}{/Screenshots}.
```

O seguinte código mostra um exemplo de texto sem formatação:

```
´´´
O seguinte **excerto** não deve # sofrer alterações
´´´
```

Passaria a:

```
O seguinte **excerto** não deve # sofrer alterações
```

5 Conclusão

Como neste trabalho se pretende criar um pré-processador de *LaTeX*, foi conveniente a utilização da linguagem *FLex* que permite interpretar texto e modificá-lo.

Com a aplicação desta linguagem, os nossos conhecimentos sobre a mesma foram aprofundados, nomeadamente na estrutura que a compõe, na utilização de expressões regulares, contextos e código em *C*.

No desenvolvimento do trabalho teve-se em atenção o uso de contextos, para que não fossem criados contextos desnecessários. Desta forma, apenas se adicionou contextos nos comandos que podessem conter outros dentro de si, como por exemplo uma lista que pode possuir listas aninhadas, ou até comandos mais simples como o negrito ou uma imagem. Desta forma, acreditou-se ser favorável a implementação da stack para que no fecho de um comando fosse possível voltar ao anterior.

Além disso, sabendo que o programa procura igualar o texto encontrado com as expressões regulares de cima para baixo, é de notar a atenção colocada

na ordem da utilização destas para definir uma prioridade nos comandos. Por exemplo, o contexto **literal** tem de se apresentar antes de todos os outros e de todas as outras ER para que, quando dentro deste, o programa tente igualar o texto às expressões regulares definidas neste e não às seguintes.

Como trabalho futuro tenciona-se acresentar mais funcionalidades ao processador, tais como comandos para criação de capas, comentários e modificadores de texto.