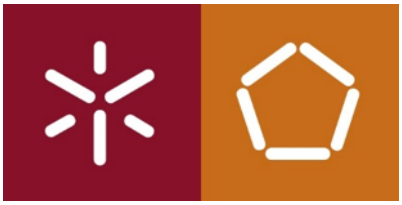

Universidade do Minho

Processamento de Linguagens (3º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Mestrado Integrado em Engenharia Informática

Realizado por
GRUPO 38
Filipa Alves dos Santos, a83631
Luís Miguel Arieira Ramos, a83930
Rui Alves dos Santos, a67656

28 de Junho de 2020

Resumo

Este relatório vai abordar todo o processo da realização do segundo trabalho prático da unidade curricular de Processamento de Linguagens do 3º ano do Mestrado Integrado em Engenharia Informática da Universidade do Minho. Este trabalho aborda principalmente a criação de filtros Flex, com uso de expressões regulares estudadas nas aulas prática, bem como a criação de gramáticas independentes de contexto (GIC), com ficheiros Yacc.

Conteúdo

1 Introdução

1.1 Enquadramento e Contextualização

Este trabalho surgiu no âmbito da unidade curricular de Processamento de Linguagens, do 3º ano do Mestrado Integrado em Engenharia Informática, sendo este o segundo (e último) da disciplina. Este projeto aborda, em primeiro lugar, a criação de um filtro flex, usando a própria notação flex e C para produzir o resultado pretendido, como estudado nas aulas práticas. Como método complementar, foi também pedido uma gramática (GIC) que traduzisse a linguagem tratada, através de um ficheiro Yacc.

Dos vários enunciados disponibilizados, como o nosso grupo é o nº 38, foi-nos atribuído o terceiro ($38\%6 + 1 = 3$): **Conversor de Pug para HTML**. No resto desta introdução e relatório, iremos aprofundar mais o problema descrito neste enunciado em específico.

1.2 Problema e Objetivo

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

- Maior experiência com o sistema operativo Linux bem como a programação em C;
- Praticar a escrita de GIC, isto é, gramáticas independentes de contexto, de acordo com o domínio do problema;
- Pôr em prática o uso expressões regulares (ERs), que servem para delinear partes de um certo documento que seguem um específico padrão;
- Utilizar pares lex/yacc para a criação de geradores de compiladores;

O problema específico descrito neste enunciado cumpre todos estes objetivos. É pedido um programa que, dado um ficheiro em formato Pug (ou pelo menos, um subconjunto da linguagem Pug), converte-o para HTML.

1.3 Resultados ou Contributos

Quanto ao resultado deste trabalho, saímos satisfeitos com o que obtivemos. O nosso programa é capaz de ler e converter quase a totalidade do exemplo fornecido pela equipa docente, abordando assim um grande subconjunto da linguagem Pug.

Conseguimos generalizar o nosso código para incluir diferentes partes de Pug, como tags com atributos, blocos dentro de tags, entre outras que serão explicadas em maior promenor no resto do relatório.

1.4 Estrutura do Relatório

Iremos explorar cada tópico mais aprofundadamente no resto deste relatório:

- Análise e Especificação: abordamos o problema do enunciado que escolhemos de maneira mais informal e específica, enunciando os objetivos a cumprir e explicamos os dados, pedidos e relações do programa;
- Conceção/desenho da Resolução: explicamos em maior detalhe as estruturas de dados e os algoritmos usados na nossa solução;
- Codificação e Testes: apresentamos os nossos resultados e testes que realizamos para garantir a qualidade do nosso código;
- Conclusão: avaliação final do nosso trabalho.

2 Análise e Especificação

2.1 Descrição informal do problema

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

Ao observarmos o problema dado, definimos as seguintes tarefas:

- Criar uma gramática de modo a representar um subconjunto da linguagem Pug, começando por casos mais simples até estarmos satisfeitos com o resultado;
- Utilizar o ficheiro flex para ler o input do nosso programa e devolver os tokens necessários à nossa gramática;
- No ficheiro yacc, de acordo com os tokens recebidos, ir criando o nosso ficheiro HTML correspondente;

2.2 Especificação do Requisitos

2.2.1 Dados

Em termos de dados observados logo à partida, detetamos algumas caracteríscas no texto de entrada que iríamos ter de filtrar no ficheiro flex e tratar no ficheiro yacc:

- tags simples: começam com o seu nome e tudo o que vem indentado a partir de uma certa tag, pertence a ela. Por exemplo, se tivermos uma tag `head` em Pug, todo o conteúdo que começa com mais espaços que a própria tag, imediatamente a seguir a ela, irá ficar dentro de `<head>` até `</head>` no HTML.
- tags com atributos: seguem a mesma lógica que as tags simples mas, a seguir aos seus nomes, vêm os atributos que as definem com os respetivos valores, dentro de parênteses. Como exemplo, temos `head(atributo="valor")` que irá ficar `<head atributo="valor">`;
- tags com texto na linha: certas tags abrem e fecham na própria linha, contendo texto no seu interior. Neste caso, por exemplo, `h1 Título` é convertido para `<h1> Título </h1>`;
- bloco de uma tag: quando surge um ponto final depois de uma tag (como `script.`), significa que tudo o que estará indentado a seguir a ela será um bloco de texto, pertencente a ela;
- omissão do nome da tag 'div': como a `div` é uma tag muito utilizada, também tratamos o caso de quando uma frase começa por `#` para abreviar a tag `div`, com os seus atributos de seguida. Por exemplo `container.col` passa para `<div> class="col"id="container"` até `</div>`;
- outros: também decidiu-se tratar de outros pequenos pormenores como quando temos uma tag seguida de um sinal de igual em Pug, o texto a seguir não é suposto passar para o HTML. Logo, `title= Olá` ficará apenas `<title> </title>`. Também se tratou do caso específico do `doctype`.

2.2.2 Pedidos

Em termos dos pedidos, estes estão bem explícitos no enunciado deste problema. Primeiramente, é pedido que escolhamos e apresentemos o subconjunto de linguagem Pug que iremos tratar (isto é explicado na secção Dados). De seguida, é solicitado um programa que funcione como um processador que valida ficheiros Pug (que utilizam apenas linguagem do subconjunto definido) e cria os ficheiros HTML correspondentes.

2.2.3 Relações

Expressões Regulares

Neste trabalho prático, o ficheiro flex teve como função ler o ficheiro Pug e enviar tokens para o ficheiro yacc. Os algoritmos associados a cada uma das expressões serão explicados com maior detalhe na seguinte parte do relatório. Nesta iremos apenas abordar o assunto de forma mais geral.

Em primeiro lugar, temos a expressão regular para ler o doctype no início do ficheiro Pug, que é apenas o texto `doctype` seguido dos seus possíveis atributos (`.*`).

```
doctype.*
```

De seguida, lemos os espaços no início de cada frase com a simples expressão `[ ]*`. Esta expressão regular é importante pois a indentação do texto (os espaços antes de cada linha) define quando é que as tags começam e acabam.

```
[ ]*
```

A expressão é utilizada para o caso específico de quando tratamos as abreviaturas da tag `div`. Já a segunda expressão corresponde ao `BEGIN id`, que vai ficar com conteúdo a seguir à , até chegar ao fim ao final da linha ou até algum ponto final. Ou seja, se receber `container.col`, apanhará o `container` apenas.

```
#
<id>[^\.\n]+
```

Neste bloco de expressões todas dentro do `BEGIN class` vêm na sequência do `id` explicado anteriormente. A primeira é para retornar ao início e a segunda é para o caso de encontrarmos apenas um ponto final, que indicará o início de um bloco de texto. Voltando ao mesmo exemplo, a terceira serve para apanharmos `col` de `container.col` e a última para ler os pontos finais.

```
<class>\n
<class>\.\n
<class>[^\.\n]+
<class>\.
```

A expressão seguinta apanha todas a tags, isto é, uma sequência de letras e/ou números `[a-zA-Z0-9]+` seguida de um ponto final, um parênteses, um ponto ou uma mudança de linha.

```
[a-zA-Z0-9]+/[ (=\n\.]
```

A seguir de capturar o nome da tag, entra no `BEGIN tag` onde analisa o restante texto imediatamente a seguir que pode corresponder a cada uma das expressões seguintes.

```
<tag>\(
<tag>\)
<tag>=.*
<tag>\n
<tag>[ ].*
<tag>\.\n
```

No caso de corresponder à primeira expressão do `BEGIN tag`, irá entrar no `BEGIN atributos` onde vai capturar tudo o que corresponde aos atributos, isto é, o testo todo até ao parênteses fechar.

```
<atributos>.*/[)]
```

Já no caso de corresponder à última expressão do `BEGIN tag`, irá entrar no `BEGIN texto` que vai tratar do casos das tags com blocos de texto dentro delas. A segunda é para verificar os espaços (se vão de acordo com a lógica do Pug) e a primeira é simplesmente para detetar todo o texto.

```
<texto>[^ ].*/\n
<texto>[ ]*
```

Por fim, temos estas expressões mais básicas que servem para, respetivamente, encontrar o final do ficheiro, encontrar e ignorar caracteres e encontrar tudo o que não tenha sido mencionado para ser possivel mandar uma mensagem de erro.

```
<<EOF>>
[\n\t\r ]
.
```

Gramática

No ficheiro yacc, foi feita a gramática correspondente ao subconjunto da linguagem Pug uilizado. Iremos explicar nesta parte a nossa lógica por detrás da estrutura escolhida. O yacc também guarda uma string que vai sendo construída recursiva de acordo com os tokens recebidos e no final escreve-a no ficheiro HTML.

Começamos por definir o programa, neste caso o Pug, como sendo uma `ListaLinhas`.

```
Prog        : ListaLinhas
            ;
```

Cada `ListaLinhas` pode ser apenas uma `Linha` ou, recursivamente, várias linhas `ListaLinhas Linha`.

```
ListaLinhas: ListaLinhas Linha
           | Linha
           ;
```

Cada `Linha` começa sempre por espaços (`Espacos`) exeto quando se trata do `DOCTYPE`, que se encontra sempre encostado no início do ficheiro. Já o último caso corresponde aos blocos de texto. As primeiras 4 expressões correspondem aos tipos diferentes de combinações que existem, sendo a `Tag` o nome/atributos da tag, o `conteudo` é texto na própria linha e a `closetagg` é apenas um token utilizado para que o yacc saiba quando fechar cada tag.

```
Linha       : Espacos Tag ListaLinhas closetagg
            | Espacos Tag closetagg
            | Espacos Tag conteudo ListaLinhas closetagg
            | Espacos Tag conteudo closetagg
            | DOCTYPE
            | Espacos string
            ;
```

O `Espacos` pode equivaler número qualquer de espaços, zero incluído.

```
Espacos     : espacos
            |
            ;
```

O `Tag` aborda nas duas primeiras definições o caso da tag ser simples (`tagg`) ou de ser uma tag com atributos (`tagg '(' atribute ')'`). Além disso, também engloba o caso da abreviatura da tag `div`, que vem com um id associado e pode potencialmente ter outros atributos também

```
Tag         : tagg
            | tagg '(' atribute ')'
            | '#' idd
            | '#' idd '.' Special
            ;
```

Aqui também há recursividade para o caso do `Special` porque a tag `div` pode ter vários valores na class, sempre separados por pontos (`classs`).

```
Special     : Special '.' classs
            | classs
            ;
```

3 Conceção/desenho da Resolução

3.1 Estruturas de Dados

Em termos de estruturas de dados, usamos maioritariamente inteiros `int`, como suporte para algumas das nossas funcionalidades:

- Começamos por criar dois inteiros `int nEspacos` e `int nEspacosAtual` para guardar o número de espaços, referente à identação, da linha anteiror e da linha atual;
- Para saber identificar se estamos numa linha de texto, ou numa linha começada por uma tag, recorremos à criação de um inteiro `int block`, em que `block=1` corresponde a texto e `block=0` corresponde a tag;
- De maneira a determinar quantas tags teriam de ser fechadas, antes de se iniciar a nova tag, foi criado um inteiro `int back`, que servirá de contador;
- Semelhante ao inteiro anterior, criou-se também o inteiro `int spaces`, de modo a saber quando era necessário retornar espaços.
- Finalmente, de modo a saber quando terminar o programa, sem que não fique nenhuma tag por fechar, criou-se o inteiro `int termina`.

3.2 Algoritmos

Como já explicado anteriormente, optou-se por fazer uma divisão do ficheiro em 2 partes: `tags` e `texto`.

Começando pela parte mais geral, no inicio de cada iteração foram colocadas quatro condições, cada uma tendo uma função diferente. A primeira condição `back==0 && termina==1`, serve para quado o programa estiver a terminar, se feche mais uma tag (antes de terminar) que corresponde à primeira tag aberta. A segunda condição `back>0` é utilizada para fechar um determinado número de tags. Já a terceira condição, `spaces>0` serve para retornar os espaços correspondentes a determinada linha. A quarta e última condição `termina==1` foram utilizadas para terminar o programa.

```
1  {if (back==0&&termina==1)    {back--; return closetagg;} }
2  {if (back>0)                 {back--; return closetagg;} }
3  {if (spaces>0)               {spaces--; return espacos;} }
4  {if (termina==1)             {return 0;} }
```

Terminadas essas condições, temos um caso especial, que é o `doctype` sendo esta uma palavra reservada. Irá retornar o tipo do ficheiro ao yacc.

```
1 doctype.*                  {yylval.s=strdup(yytext+8); return DOCTYPE;}
```

Para cada linha, a primeira operação é a contagem de espaços (identação). Assim, conseguimos saber se temos de fechar tags anteriores, ou se a atual está incluida na anterior. Estas operações fazem-se recorrendo aos inteiros criados anteriormente.

Avançando para a `tags`, estas continham vários tipo. Podiam começar com letras/numeros, ou com o caracter `#`.

```
1 #                            {BEGIN id; return yytext[0];}
2
3
4 [a-zA-Z0-9]+/[ (=\n\.]       {if (nEspacosAtual==0)  {
5                                    yyerror("Erro : Uma tag
6                                    necessita de indenta  o");
```

```
                                }
                                yylval.s = strdup(yytext);
                                nEspacosAtual=0;
                                BEGIN tag;
                                return tagg;}
```

Caso se tratasse deste último caso, então dariamos início à decifragem do id e da class da respetiva tag.

```
<id>[^\.\n]+                    {BEGIN class;
                                 yylval.s = strdup(yytext);
                                 return idd;}


<class>\n                       {BEGIN INITIAL;}
<class>\.\n                     {BEGIN texto; block=1;}
<class>[^\.\n]+                 {yylval.s = strdup(yytext); return classs;}
<class>\.                       {return yytext[0];}
```

Se fosse o primeiro caso, então teriamos de verificar se esse conjunto de letras era seguido de um '(' contendo atributos; se era seguido de '=' ignorando o valor que estivesse diante desse carater.

Se em qualquer um destes casos (exceto este último referido) a tag fosse seguida de um espaço (' '), então essa tag tem conteúdo.

Existe também um caso especial, denominado de 'block in a tag', representado por um '.' no final da tag. Quando isto ocorre, significa que pelo menos a linha seguinte é obrigatoriamente texto. Texto esse que só terminará quando o número de espaços da linha (identação) em questão, for inferior à tag que continha o '.' no final.

```
<tag>\(                         {BEGIN atributos; return yytext[0];}
<tag>\)                         {return yytext[0];}
<tag>=.*                        {BEGIN INITIAL;}
<tag>\n                         {BEGIN INITIAL;}
<tag>[ ].*                      {yylval.s = strdup(yytext+1);
                                 return conteudo;}
<tag>\.\n                       {BEGIN texto; block=1;}


<atributos>.*/[)]               {yylval.s = strdup(yytext);
                                 BEGIN tag;
                                 return atribute;}
```

Assim sendo avançamos para o `texto`. Este acontece na situação referida anteriormente. Para cada linha verificamos a identação (número de espaços). Se este for menor que a identação da tag à qual "ativou"o bloco de texto, então esse mesmo bloco será terminado, correspondendo esta mesma linha já a uma tag. Caso contrário, continuará a ser considerado texto. De notar que, caso a identação da primeira linha do bloco de texto seja inferior ou igual à identação da tag que iniciou o bloco, ocorrerá um erro e o utilizador será informado.

```
<texto>[^ ].*/\n                {yylval.s = strdup(yytext);
                                 nEspacosAtual=0;
                                 BEGIN INITIAL;
                                 return string;}

<texto>[ ]*                     {nEspacosAtual=strlen(yytext);
                                 if (nEspacosAtual<=nEspacos) {
```

```
                                yyerror("Erro : Dentro de um bloco
                                de texto    necess rio  indenta   o");}
                             yylval.s = strdup(yytext);
                             return espacos;
                            }
```

Por último, temos o END-OF-FILE usado para determinar quando o ficheiro termina e também para determinar quantas tags terão de ser fechadas.

```
<<EOF>>                     {diferenca=-nEspacos;
                             while(diferenca<-2) {diferenca+=2;back++;}
                             termina=1;
                             return closetagg;
                            }
```

Na parte do ficheiro yacc foram criadas 3 funções:

```
void yyerror(char *s){
    extern int yylineno;
    extern char* yytext;
    fprintf(stderr, "linha %d: %s (%s)\n", yylineno, s, yytext);
}

void writeFile(char* linha) {
    FILE* f = fopen("HTML.html","w");
    fprintf(f,"%s",linha);
    fclose(f);
}

char* limpaTag(char* s) {
  char* f=strdup(s);
  int i=0;
  for(i;f[i]!=' ' && f[i]!='\0';i++);
  f[i]='\0';
  return f;
}
```

A primeira função é utilizada para imprimir na consola do utilizador uma mensagem de erro, incluindo assim a linha onde ocorre o erro e a mensagem.

A segunda serve para escrever no ficheiro html todo o código lido e convertido para HTML.

Por fim, a terceira, serve para , no caso de receber uma tag, por exemplo, `script type="text"`, este irá recolher só a parte no nome da tag, ficando entao apenas, `script`.

4 Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

A implementação inicial foi relativamente fácil e simples. Fomos lendo o input do ficheiro através do flex e retornando informação para o yacc. Os problemas surgiram quando foi necessário contabilizar o número de espaços no início de cada linha, ou seja, a identação de cada uma, de modo a determinar quando era necessário fechar tags. Isso levou-nos a outro problema que foi dar return dessas **closetags**, o número de vezes necessário. De notar que também, antes de retornar o espaços da respetiva linha, teriamos de retornar os **closetags** primeiro. Para resolver estes problemas recorremos às tais condições referidas no início da secção 3.2.

Em termos de decicões relevantes tomadas, começamos por definir que haveriam 3 tipos de tags, sendo estas tag com atributos, sem atributos e uma tag especial iniciada pelo carater . Qualquer uma desta poderia conter também conteúdo, assim como qualquer uma destas poderá dar inicio a um bloco de texto. Definimos também que **doctype** era uma palavra reservada.

Por último, também decidimos tratar de alguns erros que poderiam acontecer com um template menos correto. Se uma tag, sem ser a inicial, não tiver identação, o utilizador é informado do erro. Quando uma tag inicia um bloco de texto, a identação da linha seguinte será obrigatoriamente maior do que a identação da tag que inicia esse mesmo bloco, caso contrário, o utilizador é informado do erro.

4.2 Testes realizados e Resultados

Para testar a viabilidade da nossa solução, fizemos testes com exemplos de código pug que variassem nas partes que desejássemos verificar. Como tal, usamos exemplos com diferentes tags e identações. Iremos agora listar os testes que pretendemos fazer com cada exemplo de código pug (todos os pugs estarão disponíveis do zip enviado com o resto do trabalho):

- **pug1.txt** - Exemplo simples que engloba todo o subconjunto escolhido pelo grupo (diferentes tags);
- **pug2.txt** - Exemplo simples que engloba todo o subconjunto escolhido pelo grupo (bloco de texto correto);
- **pug3.txt** - Exemplo que demonstra o erro ao utilizador quando uma tag não tem a identação necessária(bloco de texto);
- **pug4.txt** - Exemplo que demonstra o erro ao utilizador quando uma tag não tem a identação nenhuma;
- **pug5.txt** - Exemplo completo que engloba todo o subconjunto escolhido pelo grupo e que contém um grande número de diferentes identações.

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

5 Conclusão

Em suma, neste segundo trabalho, o grupo considera que aprofundou muito o seu conhecimento sobre as ferramentas usadas na unidade curricular. Foi dada novamente a oportunidade de aprender a trabalhar com expressões regulares e desenvolver processadores de linguagens regulares, e também, aprofundar os conhecimentos da linguagem Flex bem como gramáticas GIC. Por fim, com estes dois trabalhos terminados, o objetivo será implementar o conhecimento sobre Processamento de Linguagens que obtivemos durante este semestre e possivelmente aplicá-lo em problemas de programação futuros.

A Código do Programa

Ficheiro lex

```
%{
#include "y.tab.h"
#include <math.h>
void yyerror(char*);


int nEspacos=0;
int nEspacosAtual=-1;
int block=0;
int diferenca=0;
int back=0;
int spaces=0;
int termina=0;


%}

%option noyywrap yylineno
%x tag texto atributos class id

%%
  {if (back==0&&termina==1)   {back--; return closetagg;} }
  {if (back>0)                {back--; return closetagg;} }
  {if (spaces>0)              {spaces--; return espacos;} }
  {if (termina==1)            {return 0;} }


doctype.*                  {yylval.s = strdup(yytext+8); return DOCTYPE
   ;}

[ ]*                      {nEspacosAtual=strlen(yytext);
                           diferenca=nEspacosAtual-nEspacos;
                           if (block==1 && nEspacosAtual>nEspacos)  {
   BEGIN texto;}
                           if (block==1 && nEspacosAtual<=nEspacos) {
   block=0;}
                           if (block==0) nEspacos=nEspacosAtual;
                           yylval.s = strdup(yytext);
                           if(diferenca<=0){
                             while(diferenca<=-2) {diferenca+=2;back++;}
                             spaces++;
                             return closetagg;
                           }
                           return espacos;
                          }


#                           {BEGIN id; return yytext[0];}


<id>[^\.\n]+                {BEGIN class;
                             yylval.s = strdup(yytext);
                             return idd;}

```

```
52
53  <class>\n                   {BEGIN INITIAL;}
54  <class>\.\n                 {BEGIN texto; block=1;}
55  <class>[^\.\n]+             {yylval.s = strdup(yytext); return classs;}
56  <class>\.                   {return yytext[0];}
57
58
59
60  [a-zA-Z0-9]+/[ (=\n\.]      {if (nEspacosAtual==0) {yyerror("Erro : Uma
        tag necessita de indenta  o");}
61                               yylval.s = strdup(yytext);
62                               nEspacosAtual=0;
63                               BEGIN tag;
64                               return tagg;}
65
66
67
68  <tag>\(                     {BEGIN atributos; return yytext[0];}
69  <tag>\)                     {return yytext[0];}
70  <tag>=.*                    {BEGIN INITIAL;}
71  <tag>\n                     {BEGIN INITIAL;}
72  <tag>[ ].*                  {yylval.s = strdup(yytext+1);return conteudo
        ;}
73  <tag>\.\n                   {BEGIN texto; block=1;}
74
75
76
77  <atributos>.*/[)]           {yylval.s = strdup(yytext);
78                               BEGIN tag;
79                               return atribute;}
80
81
82  <texto>[^ ].*/\n            {yylval.s = strdup(yytext);
83                               nEspacosAtual=0;
84                               BEGIN INITIAL;
85                               return string;}
86
87  <texto>[ ]*                 {nEspacosAtual=strlen(yytext);
88                               if (nEspacosAtual<=nEspacos) {yyerror("Erro
        : Dentro de um bloco de texto    necess rio indenta  o");}
89                               yylval.s = strdup(yytext);
90                               return espacos;
91                              }
92
93
94
95  <<EOF>>                     {diferenca=-nEspacos;
96                               while(diferenca<-2) {diferenca+=2;back++;}
97                               termina=1;
98                               return closetagg;
99                              }
100
101
102
103 [\n\t\r ]                 ;
104
105
106 .                          {yyerror("Caracter Invalido");}
```

```
107
108 %%
```

Ficheiro yacc

```
1  %{
2  #define _GNU_SOURCE
3
4
5    int yylex();
6    void yyerror(char*);
7    void writeFile(char*);
8    char* limpaTag(char*);
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <math.h>
14
15
16 %}
17
18 %union{
19      char c;
20      int n;
21      char* s;
22 }
23
24 %token  DOCTYPE string tagg conteudo atribute idd classs closetagg
       espacos
25 %type <s>  string DOCTYPE ListaLinhas tagg Special Linha conteudo
       atribute idd classs closetagg Tag espacos Espacos
26
27
28 %%
29
30 Prog        : ListaLinhas                              {writeFile(
       1);;ListaLinhas : ListaLinhasLinhaasprintf(, Linha= strdup(1);}
31           ;
32
33
34 Linha       : Espacos Tag ListaLinhas closetagg                {asprintf
       (&,"%s<%s>\n%s\n%s</%s>",1,2,3,1,limpaTag(2));|EspacosTagclosetaggasprintf(
       , EspacosTagconteudoListaLinhasclosetaggasprintf(, EspacosTagconteudoclosetaggasprintf(
       , DOCTYPEasprintf(, ≪!DOCTYPE|Espacosstringasprintf(,"; Espacos : espacos= strdup(1)
       ;}
35           |                                          {=strdup("");}
36           ;
37
38
39 Tag         : tagg                                     {=strdup(
       1);|tagg'('atribute')'asprintf(,''iddasprintf(,"divid = ¨|''idd'.'Specialasprintf(
       ,"divclass = ¨;Special : Special'.'classsasprintf(, classs= strdup(1);}
40           ;
41
42
43
44 %%
45
```

```
46 int main(){
47     yyparse();
48     return 0;
49 }
50 
51 void yyerror(char *s){
52     extern int yylineno;
53     extern char* yytext;
54     fprintf(stderr, "linha %d: %s (%s)\n", yylineno, s, yytext);
55 }
56 
57 void writeFile(char* linha) {
58     FILE* f = fopen("HTML.html","w");
59     fprintf(f,"%s",linha);
60     fclose(f);
61 }
62 
63 char* limpaTag(char* s) {
64   char* f=strdup(s);
65   int i=0;
66   for(i;f[i]!=' ' && f[i]!='\0';i++);
67   f[i]='\0';
68   return f;
69 }
```

A.1 Executar código

Para correr o código basta escrever `make` na consola, sendo que criamos um Makefile com todos os comandos necessários. Para executar um dos ficheiros pug, é simplesmetes necessário correr o executável com o nome desse ficheiro (.\$pugtohtml ficheiropug$)