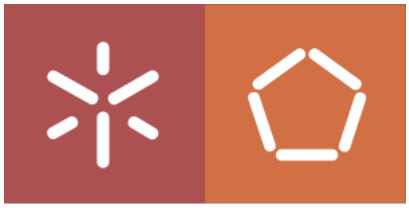

Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático 2

Desenvolvimento duma Linguagem de Domínio Específico (DSL)

Diogo Araújo	Diogo Nogueira	Mariana Lino
A78485	**A78957**	**A78824**

Resumo

Este trabalho prático é fruto de um desafio de projeto da Unidade Curricular de *Processamento de Linguagens* e destina-se essencialmente à exploração prática para criar um processador de linguagem de domínio específico, suportado por uma gramática independente de contexto.

Conteúdo

Capítulo 1

Introdução

O presente trabalho surge no âmbito da Unidade Curricular de Processamento de Linguagens, correspondente ao 3.º Ano do Mestrado Integrado em Engenharia Informática. O objetivo principal passa por pôr em desenvolvimento um processador de linguagem de domínio específico utilizando o par de gerador de compiladores `Flex/Yacc`.

Eis os objetivos planeados para este segundo trabalho prático:

- Utilizar a capacidade de escrever *gramáticas independentes de contexto* (GIC), que satisfaça a condição LR() e assim criar uma *Linguagem de Domínio Específico* (DSL);
- Desenvolver um reconhecedor léxico (utilizando a ferramenta `Flex`) e um reconhecedor sintático (utilizando a ferramenta `Yacc/Bison`) para essa linguagem;
- Construir o gerador de código que produza a resposta apropriada, associado ações semânticas de tradução às produções da gramática (na ferramenta `Yacc`), seguindo portanto o método de tradução dirigida pela sintaxe;
- Compreender a necessidade de estruturas e rotinas em C para dar resposta a todos os requisitos complexos do projeto.

Para se alcançarem estes objetivos existirá uma fase inicial que se focará em entender/analisar a documentação oficial da linguagem Pug, sobre a qual iremos escolher um *subset* bem específico para a realização duma gramática que cubra essas escolhas.

Com uma gramática bem definida, a construção dum processador de linguagens utilizando, para tal, a ajuda das ferramentas `Flex/Yacc`, que irá reconhecer e validar os documentos Pug que estejam escritos com o *sub-conjunto* escolhido para o trabalho prático. Só assim, por consequência, o processador de linguagem irá gerar o HTML correspondente.

Capítulo 2

Conversor de Pug para HTML

2.1 Subconjunto da Linguagem Pug

O presente projeto consiste na criação eficiente dum conversor da linguagem `Pug` para HTML, pensando assim numa sub-estrutura de seleções sobre as funcionalidades que o motor de conversão `Pug` proporciona.

Da mesma maneira que o código `Pug` oferece uma conversão para os *standards* HTML, o nosso processador de linguagem irá também fornecer uma panóplia gramatical possível de forma a criar uma página HTML legível para ser entendida por qualquer navegador *web* moderno.

Com esse pensamento em mente, e perante a análise da documentação `Pug` mencionada no enunciado deste projeto prática, criou-se um subconjunto de linguagem que dá resposta a:

- ***Tags*** Por padrão `Pug`, o texto no início de uma linha (ou após apenas um espaço em branco) representa uma *Tag* HTML. Para a construção do subconjunto pensou-se em tratar as *Tags* ditas "normais"e toda a parte da indentação nas *Tags* consecutivas sempre que necessário e especificado em termos de ficheiro `Pug`.

  Dentro desta ideia de *Tags* está implícita toda a parte de *Nested Content* que se detalha na subsecção que se segue e se denomina de *Inline in a Tag*.

  1. ***Indented Tags:***
     Toda a parte da indentação das *Tags* controla-se através de uma Lista Ligada completamente funcional e adaptada às várias implementações.

  2. ***Self-Closing Tags:***
     Para este tipo de *Tags* fornece-se também uma adaptabilidade da gramática para detetar *Tags Automatically Self-Closing*.

- ***Plain Text*** O *template engine* `Pug` fornece quatro formas diferentes de obter texto sem formatação, permitindo que texto não processado seja passado diretamente para o HTML de forma renderizada. Para o subconjunto escolheram-se duas dessas formas de obter conteúdo de texto, criando-se assim boas opções para a escrita em modo `Pug`.

  1. ***Inline in a Tag:***
     Com esta opção consegue-se adicionar texto sem formatação em modo *inline*. Considera-se também a possibilidade de colocar o texto sobre plicas/aspas logo após o caractér '=', tal como é especificado na secção que se segue.

  2. ***Piped Text:***
     A opção para escrita de texto em modo *Piped* torna-se útil quando se está a lidar com pedaços de texto mais extensos. Para este subconjunto considera-se apenas a possibilidade de usar em modo texto, não misturando com *Inline Tags* posteriores.

- ***Attributes*** Toda a parte dos atributos torna-se semelhante ao modo de escrita em HTML. Como escolha para o enriquecimento gramatical, existe um tratamento de apenas um ou vários atributos, separados por vírgulas ou pelo habitual espaço em branco.

Com este subconjunto da Linguagem `Pug` torna-se possível construir uma página HTML simples. É importante referir que além destas implementações está incluída toda a base necessária ao `Pug`, como por exemplo todo o controlo de espaços das várias *Tags*, sabendo-se assim através deles quando devem ser encerradas as várias *Tags* ao longo do ficheiro HTML.

Claro está também que para todo o projeto se assume um ficheiro `Pug` de entrada segundo os padrões da linguagem e sem qualquer tipo de erros de parágrafos/espaços excessivos.

2.2 Funcionamento do Subconjunto e a sua Implementação na Gramática

- ***Tags Default*:** A parte basilar do `Pug` tem como ênfase as *Tags*. Estas *Tags Default* representam na gramática aquelas *tags* em que o primeiro termo na linha é a própria *tag* e tudo após essa *tag* e um espaço é o conteúdo do texto da mesma.

Implementação	Notação em Pug	Notação em HTML

Tag Default* com (ou não) *Nested Content	`ul`
	`  li Texto de Teste 1`
	`  li Texto de Teste 2`
	`  li Texto de Teste 3`
		`<ul>`
		`  <li>Texto de Teste 1</li>`
		`  <li>Texto de Teste 2</li>`
		`  <li>Texto de Teste 3</li>`
		`</ul>`

***Tag Default* com *Nested Content* entre aspas**	`title= 'Trab.P 2 PL'`	`<title>Trab.P 2 PL</title>`

- ***Tags Attribute*:** Para aumentar o subconjunto de Linguagem pensou-se em permitir a interpretação das *Tags Attribute*, com a mesma possibilidade de uma *Tag Default*, ou seja, seguida (ou não) de *Nested Content* e ainda o *Nested Content* entre aspas logo a seguir ao caractér '='.

Implementação	Notação em Pug	Notação em HTML

Tag Attribute* com (ou não) *Nested Content	`a(href='...') Texto`
	
	`link(href="...",`
	`rel="...")`
		`<a href="...">Texto</a>`
		
		`<link href="..." rel="..." />`

***Tag Attribute* com *Nested Content* entre aspas**	`a(href='...')= 'Texto'`	`<a href='...'>Texto</a>`

No que toca à implementação deste tipo de *Tags* na gramática em si, pensou-se numa versatilidade em termos da representação do valor do atributo e da separação dos mesmos entre si. Assim sendo, a gramática encontra-se pronta para:

1. Estar opcionalmente separados com vírgulas ao invés dos espaços habitualmente usados em HTML;
2. O *value* do atributo em si pode estar entre plicas ou aspas, ficando sempre representado com aspas depois de convertido para HTML.

Note-se que a *Tag* link é uma *Tag Self-Closing* por si só. Outro reparo que pode ser feito é a existência de aspas na notação Pug, o que demonstra a adaptabilidade da linguagem criada em permitir as duas opções.

- ***Tags Self-Closing*:** As *Tags Self-Closing* são um adição muito prática em termos de linguagem, dado que permitem fechar de forma automática uma *Tag*, evitando-se assim a escrita *Tags* de fecho desnecessárias.

Implementação	Notação em Pug	Notação em HTML
***Tag Self-Closing* sem Atributos**	`foo/`	`<foo />`
***Tag Self-Closing* com Atributos**	`foo(bar='baz')/`	`<foo bar="baz" />`

Em termos de gramática, torna-se necessário existir um controle/identifiação das *Tags Default* ao nível do Analisador Sintático, dado que existem *Tags Automatically Self-Closing*. Assim sendo, a gramática encontra-se pronta para:

1. Tratar as *Tags* **img**, **link** e **meta** (que são inicialmente detetadas como *Tags Default*) como sendo *Tags Self-Closing*, fazendo-se assim de imediato um fecho automático da *Tag*;
2. Permitir que este tipo de *Tags* possam conter atributos, tal como visto anteriormente para a *Tag* do link.

Note-se que em termos de gramática este tipo de *Tags* acaba por tomar partido das *Tags Attribute*, dada a possibilidade de estes atributos existirem sob a mesma forma que a mencionada anteriormente - separados por vírgulas ou pelos habituais espaços.

- ***Tags with Piped Text*:** Relativamente ao *Plain Text*, a gramática encontra-se preparada para processar texto sem formatação de duas formas distintas - embutindo o texto logo a seguir à *Tag* (forma tratada na gramática nas *Tags Default*) ou colocando um prefixo na linha através do caractér barra vertical '|', quantas vezes forem especificadas até o texto em si terminar.

  permitindo assim que este texto não processado possa ser colocado no HTML de forma renderizada.

Implementação	Notação em Pug	Notação em HTML

Piped Text	`p`
	`  | Pipe 1,`
	`  | Pipe 2.`
		`<p>Pipe 1, pipe 2.</p>`

- ***Div Elements*:** Relativamente ao *Plain Text*, a gramática encontra-se preparada para processar texto sem formatação de duas formas distintas - embutindo o texto logo a seguir à *Tag* (forma tratada na gramática nas *Tags Default*) ou colocando um prefixo na linha através do caractér barra vertical '|', quantas vezes forem especificadas até o texto em si terminar.

  permitindo assim que este texto não processado possa ser colocado no HTML de forma renderizada.

Implementação	Notação em Pug	Notação em HTML
***Div Elements* com id e class**	`#linha.row`	`<div id="linha" class="row">` `</div>`
***Div Elements* só com id**	`#content`	`<div id="content"></div>`
***Div Elements* só com class**	`.content`	`<div class="content"></div>`

Capítulo 3

Decisões e Implementação

3.1 Criação da *gramática independente de contexto*

Com a escolha do subconjunto de Linguagem `Pug` e a sua posterior implementação em termos de gramática, cria-se logo um esboço textual daquilo que será depois transporto para o Analisador Sintático `Yacc`.

```
-------------------------------------------------------------------------------

PugFile             --> ContentPugFile

ContentPugFile      --> Tags    // Assume-se que o Pug File é um conjunto de Tags

Tags                --> Tag '\n' Tags   // As Tags encontram-se separadas pelo '\n'
                     |  Tag            // Aplica-se recursividade à direita

// Conforme se analisou e acabou por deduzir da escolha do subconjunto de linguagem,
// assume-se que existam estes 5 "tipos" de analisar as Tags

Tag                 --> TagDefault
                     |  TagAttribute
                     |  TagSelfClosing
                     |  TagPiped
                     |  TagDiv

TagDefault          --> beginTag
                     |  beginTag contentTag
                     |  beginTag '=' contentTag

TagAttribute        --> beginTag AttributeHandler
                     |  beginTag AttributeHandler contentTag
                     |  beginTag AttributeHandler '=' contentTag
```

```
TagSelfClosing      --> beginTag '/'
                     |  beginTag AttributeHandler '/'

TagPiped            --> beginTag contentPipedTag

TagDiv              --> beginDiv idDiv classDiv
                     |  beginDiv idDiv
                     |  beginDiv classDiv

// Além dos "tipos" de Tag, existem ainda Símbolos Não Terminais para tratar dos
// atributos, dado que podem ser usados também nas Tags Self-Closing

AttributeHandler    --> '(' Attributes ')'

// Toda a parte de múltiplos atributos separados por espaço ou vírgula é tratada
// no Analisador Léxico

Attributes          --> Attributes Attribute
                     |  Attribute

Attribute           --> nameAttribute valueAttribute
```

Perante a análise desta gramática e pensando já em simplificar a escrita do ficheiro `Yacc`, pode-se concluir que:

- Existem **25 Produções Diferentes** para esta GIC
- Existem **12 Símbolos Terminais:**

  `PugFile, ContentPugFile, Tags, Tag, TagDefault, TagAttribute, TagSelfClosing, TagPiped, TagDiv`

  `AttributeHandler, Attributes e Attribute`

- Existem **12 Símbolos Não Terminais:**

  `beginTag, contentTag, beginDiv, idDiv, classDiv, nameAttribute, valueAttribute`

  `'\n', '=', '/', '(' e ')'`

Para um maior entendimento do código `Yacc`, os **Símbolos Terminais** encontram-se com uma letra maiúscula no início de cada palavra, enquanto que os **Símbolos Não Terminais** possuem uma letra minúscula no início da primeira palavra.

3.2 Estruturas de Dados e Ficheiros Auxiliares em C

Tendo em conta que a gramática já se encontra perfeitamente definida, o passo seguinte será entender a forma como a Estrutura de Dados em C opera e qual a sua verdadeira utilidade em termos de conversão para HTML.

Uma vez que as Ações Semânticas criadas para GIC em `Yacc` são muito semelhantes para as várias produções, uma vez entendida a Estrutura de Dados elimina-se também a necessidade de explicar excessivamente todo o código `Yacc` em si.

Tome-se como base o seguinte ficheiro Pug e use-se os pontos como modo de contabilizar o número de espaços que delimitam a *Tag* em si. Atente-se também na definição da Estrutura ao lado:

```
html(lang='pt')
.... head
.... .... title= 'TP2 PL'
.... body
```

```
typedef struct tags {
    char *closingTag;
    int numberSpaces;
    struct tags *next;
} Tags;
```

Assuma-se agora que a inserção na Lista Ligada é sempre feita no início e não no fim e que o `char *closingTag` é a string que contém a Closing Tag com os devidos espaços. A inserção é feita assim que se deteta uma nova *Tag* no ficheiro Pug.

A ideia passará por demonstrar uma Lista Ligada inicial e o respetivo conteúdo já escrito em HTML, repetindo-se o processo para uma nova inserção na mesma e verificando assim o comportamento da estrutura.

- A Lista Ligada inicial contém 3 elementos/*Tags* que correspondem aquelas que ainda não necessitaram de ser encerradas. Dessa forma, o ficheiro HTML encontra-se no estado apresentado ao lado.

```
| .... .... </title>, 8 | -->
| .... </head>, 4 | -->
| </html>, 0 |
```

```
<html lang='pt'>
.... <head>
.... ....<title>TP2 PL
```

- Ao analisar-se o ficheiro HTML inicialmente anexado nota-se a existência da *Tag* body. É essa *Tag* que se está agora a querer tratar e se vai inserir na estrutura, fechando-se todas as *Tags* necessárias no HTML.

```
| .... </body>, 4 | --> | .... .... </title>, 8 | --> | .... </head>, 4 | -->
                        | </html>, 0 |
```

Uma vez inserida a **Closing Tag** correspondente à *Tag* `body`, a ideia passa por perceber se existirá a necessidade de fechar alguma das *Tags* anteriores.

Como é que é feito esse controlo?

1. Faz-se uso de uma função auxiliar de seu nome `newInitialTag` que recebe como parâmetros a Lista de *Tags*, a *Opening Tag* do `body` já com os devidos espaços e o número de espaços que essa *Tag* contém:

   (a) Essa função pega na Lista Ligada, ignorando o primeiro elemento, dado que corresponde à Tag que se está a tratar;

   (b) Concatena à *Opening Tag* do **body** recebida como parâmetro todas as cujos números de espaços são **maiores ou iguais ao valor inserido**;

   (c) Dessa forma, fica-se com uma *string* preparada com as *Closing Tags* corretamente colocadas para o HTML.

2. Uma vez controladas as *Tags* a fechar no momento da inserção da *Tag* `body`, eliminam-se estas *Tags* que já foram fechadas, deixando-se a *Closing Tag* do `body`.

Dessa forma, a Estrutura passará a conter os seguintes elementos e o ficheiro HTML já estará com as Closing Tags antes da Tag de abertura do `body`:

```
| .... </body>, 4 | -->
| </html>, 0 |
```

```
<html lang='pt'>
.... <head>
.... ....<title>TP2 PL</title>
.... </head>
.... <body>
```

3.3 `Flex` - Reconhecedor Léxico

O Reconhecedor Léxico torna-se fundamental para todo o *parse* da informação do ficheiro dado como *input*, dado que é ele que interpreta/analisa esse conteúdo e o envia para o Analisador Sintático para que ele possa fazer o restante do trabalho.

3.3.1 Tratamento Léxico das *Tags*

Para o tratamento das *Tags* no geral e de toda a ideia de *Nested Content* que os vários tipos de *Tag* podem conter, torna-se necessário controlar o início de cada *Tag* - **beginTag** - e o possível *Nested Content* - **contentTag**.
Este *Nested Content* pode estar também declarado sob o uso de um carácter '=' e entre plicas/aspas, sendo esta forma tratada com o uso da *Start Condition* **CONTENTTAG**.

```
-------------------------------------------------------------------------------

1       [ ]                     ;
2       [\n|\(|\)|\/|\'|\,]     return *yytext;
3       \=                      { BEGIN CONTENTTAG; return *yytext; }
4
5       ^[ ]*[a-zA-Z0-9]+       {
6                                  yylval.stringValue = strdup(yytext);
7                                  return beginTag;
8                               }
9       [ ][^\ ].*              {
6                                  yylval.stringValue = strdup(yytext+1);
6                                  return contentTag;
12                              }

-------------------------------------------------------------------------------
```

Assim, o Tratamento Léxico para as *Tags* é feito da seguinte forma:

- Sempre que se deteta o início de uma *Tag* (que se assume como sendo constituída pelo abecedário americano e/ou dígitos), devolve-se essa *string* para o lado do `Yacc`.
- Uma vez que se assume que o *Nested Content* tem de estar separado por apenas um espaço do início da *Tag*, a expressão regular apanha tudo o que vem depois disso e devolve também para o analisador `Yacc`.
- No caso do *Nested Content* estar entre as plicas/aspas, obriga-se a entrada na tal *Start Condition* **CONTENTTAG**, de modo a extrair todo o texto até à plica/aspa final.

3.3.2 Tratamento Léxico dos *Attributes*

O tratamento dos atributos ao nível do Analisador Léxico dedica-se a devolver ao Analisador Sintático não só o nome do atributo mas também o seu *value*, permitindo assim que o `Yacc` processe essa informação e atue com a ação semântica apropriada. No caso, o *value* do atributo pode estar entre plicas ou aspas, sendo sempre convertido para aspas ao ser escrito no HTML.

```
1       [ ]*[a-zA-Z]+\=/[\'|\"]         {
2                                           yylval.stringValue = strdup(yytext);
3                                           BEGIN CONTENTATTRIBUTE;
4                                           return nameAttribute;
5                                        }
```

Assim, o Tratamento Léxico para os *Attributes* é feito da seguinte forma:

- Uma vez detetado o *name* do atributo e devolvido ao Analisador Sintático, torna-se necessário absorver o *value* em si, iniciando-se então a *Start Condition* **CONTENTATTRIBUTE**.

```
1       <CONTENTATTRIBUTE>{
2           [\'|\"]                         { BEGIN ATTRIBUTETEXT; }
3           \,[ ]+|[ ]+                     { BEGIN INITIAL; }
4       }
```

- Quando se encontra a tal plica/aspa, inicia-se uma nova *Start Condition* **ATTRIBUTETEXT** para que a mesma possa obter o *value* do atributo em si.
- Dado que se assume que os vários atributos possam estar separados por vírgula ou simplesmente um espaço, assim que se reconhece um deles, força-se a sair desta *Start Condition*, uma vez que o atributo atual já foi processado e volta-se ao estado inicial para ocorrer a recursividade do próximo atributo de forma correta.

Esta segunda *Start Condition* foi então pensada para extrair o conteúdo entre plicas/aspas seguido do carácter ’,’. Dessa forma, temos três expressões regulares nesta *SC* para albergar e filtrar todos os atributos e o seu conteúdo.

```
1       <ATTRIBUTETEXT>{
2           [^\'\"]*[\'|\"]/\,          {
3                                           ...
4                                           yylval.stringValue = strdup(yytext);
5                                           BEGIN CONTENTATTRIBUTE;
6                                           return valueAttribute;
7                                       }
8           [^\'\"]*[\'|\"]/[ ]+        {
9                                           ...
10                                          yylval.stringValue = strdup(yytext);
11                                          BEGIN CONTENTATTRIBUTE;
12                                          return valueAttribute;
13                                      }
14          [^\'\"]*[\'|\"]/\)          {
15                                          ...
16                                          yylval.stringValue = strdup(yytext);
17                                          BEGIN INITIAL;
18                                          return valueAttribute;
19                                      }
20      }
```

- A primeira e segunda expressão regular reiniciam a *Start Condition* **CONTENTATTRIBUTE** mencionada anterior, dado que servem para detetar a existência de um novo atributo.
- A terceira e última serve como caso de paragem da *Start Condition* atual, já que uma vez detetado o cáracter ’)’ significa que a *Tag Attribute* atual terminou.

3.3.3 Tratamento Léxico do *Piped Text*

O *Flex* utiliza na sua base o estado **INITIAL**, logo foi utilizado isso na nossa vantagem, tendo uma expressão regular (em baixo) que consegue filtrar no *template* `Pug` o *Piped Text*, ou seja, texto dividido por linhas, com um carácter especial '`|`' como símbolo escolhido.

```
1       \n[ ]+/\|                       {
2                                           pipedText = (char*) malloc(100);
3                                           sizePipedText = 100;
4                                           BEGIN CONTENTPIPEDTAG;
5                                       }
```

Assim, o Tratamento Léxico para o *Piped Text* é feito da seguinte forma:

- Quando se deteta uma nova linha seguida dos espaços (utilizados para indentação) seguido do carácter especial '`|`', inicia-se uma variável que albergará todo o *Piped Text*.
- Para começar a extrair toda a informação começa-se uma *Start Condition* intitulada de **CONTENTPIPEDTAG** para analisar lexicamente o resto das expressões regulares para esta secção.

  Esta *Start Condition* permite que se simule uma recursividade para os vários parágrafos de *Piped Text* que possam existir.

```
1       <CONTENTPIPEDTAG>{
2           \|[ ]+                      { BEGIN PIPEDTEXT; }
3           \n[ ]+\|/[ ]+               { BEGIN PIPEDTEXT; }
4       }
```

- A *Start Condition* **CONTENTPIPEDTEXT** trata então de reencaminhar para uma nova *Start Condition* **PIPEDTEXT**.

  Esta *SC* tem a função de ir concatenando todo o texto `yytext` dos vários parágrafos *Piped* para uma variável (que é realocada sempre que necessário) `pipedText` que terá todo o texto filtrado nesta recursividade;

```
1       <PIPEDTEXT>{
2           [^\ ].*/\n[ ]+\|         {
3                                        strcat(auxPipedText, yytext);
4                                        pipedText = auxPipedText;
5                                        BEGIN CONTENTPIPEDTAG;
6                                    }
7           [^\ ].*                  {
8                                        ...
9                                        yylval.stringValue = strdup(pipedText);
10                                       BEGIN INITIAL;
11                                       return contentPipedTag;
12                                   }
13      }
```

- Esta *SC* **PIPEDTEXT** está então preparada para encontrar e funcionar de maneira recursiva filtrando o conteúdo do texto.
- Quando se detetar que não existem mais parágrafos desta forma, devolve-se para a variável `Yacc` todo o conteúdo de texto *Piped* filtrado até agora, retomando o estado inicial do `Flex`.

3.3.4 Tratamento Léxico dos Elementos `Div`

As *Divs* dão ao subconjunto de Linguagem `Pug` uma boa adição e possibilidades em termos de HTML. Dada a sua natureza, tornou-se necessário permitir a existência (ou não) do atributo **Id** e das possíveis **Classes**.

Com esse pensamento podem existir as seguintes opções para criação de uma *Div* em modo `Pug`:

- A *Div* conter apenas o valor do **Id**.
- A *Div* conter apenas o valor da **Class**, podendo esta opção ter vários valores, devidamente identificados pelo habitual carácter de ponto.
- A *Div* conter um *mix* dos dois, sendo que apenas pode existir um **Id** em qualquer *Div*.

Assim sendo, basta criar uma Expressão Regular para quando a *Div* se encontra declarada começando com o **Id** ou **Class**, uma vez que quando contêm as duas opções se trata diretamente na *SC* **IDENTIFIERDIV** que acabará por reencaminhar para a *SC* **CLASSDIV**.

```
1       ^[ ]*/\#            {
2                               classDivText = (char*) malloc(100);
3                               sizeClassDivText = 100;
4                               yylval.stringValue = strdup(yytext);
5                               BEGIN IDENTIFIERDIV;
6                               return beginDiv;
7                           }
8       ^[ ]*/\.            {
9                               classDivText = (char*) malloc(100);
10                              sizeClassDivText = 100;
11                              yylval.stringValue = strdup(yytext);
12                              BEGIN CLASSDIV;
13                              return beginDiv;
14                          }
```

Assim, o Tratamento Léxico para os elementos de uma *Div* é feito da seguinte forma:

- Quando se deteta o carácter '#' significa que um **Id** está presente, começando assim a *SC* **IDENTIFIERDIV** para extrair o seu valor;
- Quando se deteta o carácter '.' significa que a *Div* não possui um **Id**, começando-se assim de imediato a *SC* **CLASSDIV**.

```
1       <IDENTIFIERDIV>{
2           [\#][^\.]*/\.           {
3                                       ...
4                                       BEGIN CLASSDIV;
5                                       return idDiv;
6                                   }
7           [^\.\n]*                {
8                                       ...
9                                       BEGIN INITIAL;
10                                      return idDiv;
11                                  }
12      }
```

```
1       <CLASSDIV>{
2           \.[^\.\n]*/\.           {
3                                       ...
4                                       BEGIN CLASSDIV;
5                                   }
6           \.[^\.\n]*              {
7                                       ...
8                                       BEGIN INITIAL;
9                                       return classDiv;
10                                  }
11      }
```

- A *SC* **IDENTIFIERDIV** verifica se a *Div* possui valores para a **Class**. Em caso afirmativo, desencadeia a *SC* **CLASSDIV** para a extração dos restantes valores. Caso contrário, volta ao **INITIAL**;
- A *Start Condition* **CLASSDIV** deteta a existência dos vários valores para a **Class**. Quando não existirem mais, volta ao **INITIAL**.

3.4 `Yacc` - Reconhecedor Sintático com Ações Semânticas

Como falado na secção anterior, a ligação entre o gerador léxico `Flex` com o `Yacc`, existiu então a implementação da gramática falada na secção 3.1, na página 8.

Assim as ações semânticas, apesar de longas são todas similares, portanto a explicação desta fase do trabalho será limitada a um exemplo maior dado que depois as restantes produções utilizam e/ou são similares de natureza semântica.

```
1   // Verificar se consiste numa Tag 'img', 'meta' ou 'link'.
2   // Estas Tags são assim tratadas como Self Closing.
3   if(isAutoSelfClosing($1) == 1){
4      numbSpaces = countInitialSpaces($1);
5      openingTagWSpaces = tagWithSpaces($1, 1, 3, numbSpaces);
6
7      listTags = insertTag(listTags, closingTagWSpaces, numbSpaces);
8      openingTagWClosedTags = newInitialTag(listTags, openingTagWSpaces, numbSpaces);
9      listTags = removeClosedTags(listTags, numbSpaces);
10     listTags = removeFirstTag(listTags);
11
12     asprintf(&$$, "%s />", openingTagWClosedTags);
13  }
14
15  // Caso contrário, trata-se como uma Tag normal.
16  else{
17     numbSpaces = countInitialSpaces($1);
18     openingTagWSpaces = tagWithSpaces($1, 1, 2, numbSpaces);
19     closingTagWSpaces = tagWithSpaces($1, 0, 2, numbSpaces);
20
21     listTags = insertTag(listTags, closingTagWSpaces, numbSpaces);
22     openingTagWClosedTags = newInitialTag(listTags, openingTagWSpaces, numbSpaces);
23     listTags = removeClosedTags(listTags, numbSpaces);
24
25     asprintf(&$$, "%s", openingTagWClosedTags);
26  }
```

No tratamento de uma *Tag* no gerador `Yacc` foram utilizadas as capacidades da linguagem C, demonstradas e criadas na secção 3.2 na página 10 através de funções auxiliares que nos adicionaram habilidades para funcionar corretamente.

Desta maneira, e tomando-se como base o exemplo dado em `Yacc` para uma *Tag Default* sem *Nested Content*, começa-se por analisar que:

- No processo semântico de qualquer *Tag* que aparece no ficheiro `Pug` em análise, o comportamento deste processador de linguagem funciona sempre de maneira análoga:

  1. Recorrendo à função auxiliar em C `countInitialSpaces` obtém-se um valor inteiro correspondente ao número de espaços que a *Tag* em causa tem.

  2. Com a função `tagWithSpaces` consegue-se obter tanto a abertura como o fecho da *Tag* em causa, já com os espaços inicialmente concatenados. É de salientar que esta função também está preparada (através de *flags*) para responder aos vários tipos de *Tags* que o nosso processador de linguagem é capaz de entender gramaticalmente, servindo assim para o restante do projeto.

  3. Com as *Tags* de abertura, número de espaços, e *Tags* de fecho, dá-se utilização à estrutura para guardar na mesma os últimos dois parâmetros para uso futuro.

  4. Com a estrutura organizada, recorre-se a outra função auxiliar `newInitialTag` que nos retorna a real *string* de abertura a ser imprimida em HTML já com todas as *Tags* de fecho que são precisas encerrar até àquele momento, respeitando a natureza do *Nested Content*.
  5. Por fim, remove-se da estrutura todas as *Tags* que foram encerradas na string devolvida pela função auxiliar anterior, retirando a redundância da estrutura.

  É importante referir que na altura do *End of File* do ficheiro `Pug` dado como *input*, recorre-se a uma função que imprime todas as outras *Tags* que faltam encerrar, garantindo-se assim uma correta finalização do ficheiro HTML.

- Uma *Tag Default* também pode corresponder a uma *Automatically Self Closing Tag*, que é passada ao Analisador Sintático `Yacc` tal como qualquer outra *Tag Default*.

  Como se trata de uma *Tag* que é fechada imediatamente assim que é escrita em HTML, torna-se desnecessário ir buscar a uma estrutura o fecho e efetuar toda a algoritmia pensada e explicada acima neste caso.

  Note-se que para este caso existe a recorrência a uma nova função `removeFirstTag` que elimina de imediato a *Tag* que tinha sido inserida, dado que esta não será necessário fechar no decorrer do ficheiro `Pug`.

Capítulo 4

Solução Final

Para facilitar a execução de comandos extra, desenvolveu-se uma **makefile** que aplica o comando `gcc` tanto ao ficheiro que contém a estrutura em si (`structTags.c`) como ao ficheiro com as funções auxiliares. Logo de seguida faz o comando do `Flex` ao ficheiro `pugToHTML.l` e o comando `Yacc` ao ficheiro `pugtoHTML.y`. Finalmente corre o `gcc` com todos os ficheiros obtidos anteriormente.
Execute-se primeiramente o comando "make"inicial antes de se proceder para o teste do programa:

```
> make
```

Executa-se o programa, fornecendo apenas como parâmetro o ficheiro teste com código escrito na linguagem pug, que foi feito para a criação base do mesmo:

```
>  ./pugToHTML < ficheiroTeste.pug > pagina.html
```

Como solução final consegue-se obter o ficheiro HTML convertido a partir do ficheiro `Pug` inicial fornecido:

Figura 4.1: Ficheiro Teste em **pug**.

```
You, 7 minutes ago | 1 author (You)
<html lang="pt">      You, 7 minutes ago • Pagina com foto e template feito
    <head>
        <title>TP2 PL
        </title>
        <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" rel="stylesheet" integrity="sha
    </head>
    <body>
        <div class="container justify-content-center">
            <div id="titulo" class="row">
                <div class="col">
                    <h1>Pug - node template engine
                    </h1>
                </div>
            </div>
            <div id="linha" class="row">
                <div id="coluna" class="col">
                    <p>Este é o segundo <em>trabalho prático</em> de Processamento de Linguagens. O objetivo do trabalho é co
                    </p>
                </div>
                <div id="coluna2" class="col justify-content-center">
                    <img src="logo.png" />
                </div>
            </div>
            <div id="linha" class="row justify-content-center">
                <div id="coluna" class="col justify-content-center">
                    <p><b>Este trabalho foi engraçado de se fazer.</b>
                    </p>
                </div>
            </div>
        </div>
    </body>
</html>
```

Figura 4.2: Ficheiro convertido para **HTML**.

Pug - node template engine

Este é o segundo *trabalho prático* de Processamento de Linguagens. O objetivo do trabalho é converter um ficheiro pug para html.

Universidade do Minho
Escola de Engenharia

Este trabalho foi engraçado de se fazer.

Figura 4.3: Resultado visual da `página.html` gerada.

Capítulo 5

Observações Finais

Após dada como concluída toda a elaboração do trabalho com o cumprimento dos requisitos pré-definidos e correspondendo à gramática da linguagem `Pug` escolhida pelo grupo de trabalho, é importante refletir o quão o mesmo ajudou a praticar e aprofundar toda a matéria relativamente ao Analisador Léxico `Flex`, como também do Analisador Sintático `Yacc` e todo o uso das várias ações semânticas.

Notou-se uma complexidade maior relativa ao trabalho prático anterior, tanto no desenvolvimento e criação de expressões regulares em `Flex` para a descrição de padrões de frases, como no desenvolvimento de Reconhecedores Léxicos. Houve uma aprendizagem no desenvolvimento de gramáticas independentes de contexto (GIC), que satisfaçam a condição LR(), criando assim, uma Linguagem de Domínio Especifico (DSL). Assim, o grupo notou um aumento na capacidade de organização e de como criar estruturas em C da forma mais simples possível e responsiva em termos de armazenamento de conteúdo.

Primeiramente, fez-se a selecção e colecção de vários campos e pormenores da linguagem `Pug`, de forma a obter um sub-conjunto de `Pug` personalizado. De seguida fazer um ficheiro teste (um pequeno código em `Pug`) pensado em englobar todos os campos de `Pug` escolhidos nos três links, fornecidos pelos docentes no enunciado do trabalho.

A principal dificuldade foi a criação da estrutura em si, a forma como efetuar o fecho das *Tags* e como guardar tudo isto sob a forma de Lista Ligada.
Apesar de todas as dificuldades e com o resultado final do projeto bem sucedido, na perspetiva do grupo, atingiu-se da melhor forma todos os objetivos propostos e criados, pensando sempre na criação, otimização e melhoramento do problema, de forma a demonstrar uma aprendizagem sobre todos os conhecimentos adquiridos.