Processamento de Linguagens (3º ano de Engenharia Informática)

Trabalho Prático

Relatório de Desenvolvimento

Gabriela Cunha (A97393) | Miguel Senra (A97496) | Nuno Varela (A96455)

28 de maio de 2023

Resumo

O presente relatório aborda o processo de desenvolvimento e decisões tomadas no âmbito do trabalho prático da unidade curricular de Processamento de Linguagens, que consistiu na implementação de um conversor de código Pug para código HTML.

Conteúdo

Capítulo 1

Introdução

No âmbito da unidade curricular de Processamento de Linguagens, foi-nos proposta a elaboração de um trabalho prático que tem tem como principais objetivos:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;
- desenvolver um compilador gerando código para um objetivo específico;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

O projeto é composto por vários enunciados dos quais apenas um deve ser selecionado, sendo, portanto, uma escolha livre. A escolha do grupo recaiu sobre o conversor de Pug para HTML, uma vez que o aprofundamento e a exploração desta primeira linguagem seria útil também para a unidade curricular de Engenharia Web. Deste modo, o programa desenvolvido deve aceitar um subconjunto da linguagem Pug e gerar o HTML correspondente.

Estrutura do Relatório

O presente relatório encontra-se dividido em cinco capítulos e dois apêndices. O capítulo 1, o atual, corresponde à introdução do projeto. O capítulo 2 corresponde a uma contextualização e análise detalhada do problema e à especificação dos requisitos que o conversor desenvolvido deve cumprir. Em seguida, o capítulo 3 aborda a concepção de uma solução que cumpra com os requisitos previamente definidos, com ênfase na criação da sua *tokenização* e gramática. O capítulo 4 aborda a implementação da solução, detalhando os procedimentos seguidos, todas as decisões importantes tomadas e os problemas que surgiram no desenvolvimento, juntamente com os testes desenvolvidos para o conversor e os respetivos resultados, de modo a garantir a sua correção e melhorar o seu desempenho. O último capítulo - 5 - apresenta as conclusões do projeto e um balanço final do que foi realizado, englobando o trabalho a desenvolver no futuro. Relativamente aos apêndices, o primeiro, apêndice A, apresenta os *tokens* e estados utilizados no desenvolvimento do *lexer* e o apêndice B apresenta a gramática elaborada neste projeto.

Capítulo 2

Análise e Especificação

2.1 Contextualização

O Pug é um *template engine* com alto desempenho que funciona como um intermediário entre o Node e o HTML, ou seja, em tempo de execução, o Pug substitui as variáveis por valores atuais e, em seguida, envia o texto resultante para o cliente. Consiste numa linguagem para escrever *templates* HTML de forma mais concisa e legível, fornecendo recursos não disponíveis em HTML como, por exemplo, *mixins*, *includes* e *loops*, que ajudam a reutilizar e organizar o código de maneira mais eficiente.

2.2 Descrição informal do problema

O problema consiste em converter a linguagem de *template* Pug para a linguagem de *markdown* HTML. Assim, o programa deve, a partir de um ficheiro de texto escrito em Pug, ser capaz de gerar um ficheiro de texto com código em HTML equivalente.

2.3 Especificação dos requisitos

No enunciado do projeto foi fornecido um exemplo de ficheiro com código em Pug, o qual o nosso programa deveria converter para HTML. Portanto, o nosso objetivo, foi, desde o ínicio, cumprir com as funcionalidades requeridas pelo exemplo e, posteriormente, expandir o programa de forma a poder receber outros tipos de instruções disponíveis na documentação do Pug.

Primeiramente, apresentamos os requisitos que consideramos como obrigatórios durante a realização deste trabalho.

- Implementação de *tags* com ou sem atributos e com ou sem texto;
- Implementação da sintaxe especial de definição de atributos “id“ e “class“;
- Implementação de *tags* “div“;
- Implementação de instruções condicionais;
- Implementação de blocos de texto;
- Implementação de atribuição de valores a variáveis através do símbolo “=“.

No entanto, especificamos, desde logo, alguns requisitos que consideramos importantes e alguns que nos propusemos a fazer como desafio.

- Implementação de instruções com vários atributos, podendo estes ter representações diferentes;
- Garantir a presença de um único atributo "id" por *tag*;
- Implementar a definição de variáveis em Pug;
- Implementar iterações através dos ciclos "each" e "while";
- Implementar os dois tipos de comentários - um funciona como comentário exclusivo para Pug, o outro é apresentado no resultado em HTML;
- Implementar expressões *JavaScript* na interpolação de variáveis;
- Interpolação de variáveis dentro de texto;
- Implementar condições e valores de variáveis como expressões *JavaScript*.

Capítulo 3

Concepção/Desenho da Resolução

A resolução deste conversor pode ser dividida em dois componentes: o analisador léxico e o analisador sintático.

A função essencial do analisador léxico consiste em ler o texto fonte do programa, neste caso proveniente do ficheiro de texto, e dividir o texto em unidades léxicas, chamadas de *tokens*, elementos primários de qualquer linguagem. Para além desta tarefa principal, o *lexer* pode, ainda, realizar algumas verificações básicas de sintaxe. Para efetuar diferentes tratamentos dos *tokens* em diferentes contextos, são ainda definidos diferentes estados. Os *tokens* recolhidos e estados definidos encontram-se no apêndice A.

Depois de identificados os *tokens*, procedemos à produção da gramática, onde são definidas as funções de reconhecimento de cada um dos símbolos não terminais. Na implementação do analisador sintático é, portanto, preciso ter em consideração o tipo de *parser* que será usado. Neste caso, será o ply.yacc, que é um parser LALR[Bea] (*bottom-up*), facilitando a definição da gramática na medida em que suporta prioridade e associatividade explícita de cada produção. A gramática implementada encontra-se no apéndice B. De notar que esta foi desenvolvida tendo em conta os conflitos que poderiam ocorrer, sendo que os que apareceram foram resolvidos e a solução final está isenta dos mesmos.

A implementação destes dois componentes foi um processo iterativo e que sofreu várias revisões, com vista a suportar novas funcionalidades, resolver problemas que surgiram ao longo do processo e corrigir alguns pormenores que levassem a uma melhor solução.

Capítulo 4

Codificação e Testes

4.1 Analisador léxico

De forma a organizar a análise léxica, visto que é uma parte que escala muito facilmente com a implementação de novas funcionalidades, organizamos cada um dos tipos de instruções por estados. O estado em que o programa começa é o "INITIAL". De notar que sempre que é iniciado um novo estado, quando ocorre uma mudança de linha o estado atual volta a ser o "INITIAL", à exceção do estado que utilizamos para capturar blocos de texto pertencentes a uma *tag*, que vamos abordar nesta mesma secção. Assim, utilizamos o primeiro *token* verificado para nos dirigir para a operação que pretendemos.

4.1.1 Definição de *tags* e atributos

A operação mais comum em Pug é a definição de *tags*, pelo que esta é realizada diretamente no estado "INITIAL". Uma *tag* pode conter atributos, seja na forma geral, entre parênteses, ou então através da forma simplificada (".class" e "#id"). Nesta forma simplificada, no caso de omissão do nome da *tag*, esta toma o valor de "div"[Pug]. Em seguida, apresentam-se alguns exemplos de conversão de modo a esclarecer as diferentes representações de definições de *tags* com atributos.

Pug	HTML
`a(href="google.com")`	`<a href="google.com"></a>`
`a.btn`	`<a class="btn"></a>`
`a(class="btn" href="google.com")`	`<a class="btn" href="google.com"></a>`
`a(class="btn" href="google.com").cont`	`<a class="btn cont" href="google.com"></a>`
`.content`	`<div class="content"></div>`
`a#main-link`	`a id="main-link"></a>`
`#content`	`<div id="content"></div>`
`#content.btn.cont`	`<div id="content" class="btn cont"></div>`
`a#content.btn(class="cont")`	`<a class="btn cont" id="content"></a>`

Tabela 4.1: Exemplos de conversão de *tags* com atributos.

Os atributos sem parênteses podem ser verificados diretamente, através de uma expressão regular, dando origem a dois *tokens*, "CLASS" e "ID", onde removemos o símbolo inicial ("#" ou ".") visto que este não faz parte do resultado em HTML.

Para os restantes atributos, definidos dentro de parênteses, utilizamos o estado "attributes", do tipo exclusivo, que é iniciado aquando da presença de uma abertura de parênteses. Neste estado, reconhecemos o "ATTRIBUTENAME", o *token* "ASSIGN" que representa o símbolo "=", o "ATTRIBUTESPACE" que identifica a existência de mais do que um atributo e o fecho de parênteses que nos indica que devemos voltar ao estado anterior. Para além destes *tokens*, quando nos encontramos no estado "attributes" e estamos perante a presença de aspas, entramos num novo estado "attributescont". Este novo estado recolhe o valor do atributo ao reconhecer o *token* "ATTRIBUTE" e termina na presença de aspas.

4.1.2 Indentação

Ao contrário de todos os exemplos que vimos durante as aulas, a linguagem Pug envolve a indentação, o que resultou em algumas dificuldades. A indentação é um elemento fundamental no desenvolvimento deste projeto, porque é ela que nos permite identificar blocos aninhados.

O reconhecimento dos *tokens* relativos à indentação, ou seja *token* "INDENT" e "DEDENT", foi feito com base na seguinte expressão regular `r"\n[ ]*"`. Com isto, a cada mudança de linha ocorrida avaliamos se corresponde a um "INDENT", "DEDENT" ou pode até não corresponder a nada.

Esta avaliação terá de ser feita com base nos níveis de indentação anteriores. Para tal, recorremos a uma *stack* como estrutura auxiliar, que vai guardar os níveis de indentação até ao momento.

Listing 4.1: Função que reconhece a indentação.

```
1  def t_ANY_newline(t):
2      r'\n[ ]*'
3      n = len(t.value[1:])
4      t.lexer.begin('INITIAL')
5
6      if n > t.lexer.indents[-1]:
7          t.type = 'INDENT'
8          t.value = t.value[1:]
9          t.lexer.indents.append(n)
10         return t
11
12     elif n < t.lexer.indents[-1]:
13         t.type = 'DEDENT'
14         t.lexer.indents.pop()
15         t.lexer.skip(-n-1)
16         return t
17
18     else:
19         pass
```

No reconhecimento dos *tokens* associados à indentação começamos por obter o seu tamanho e, de seguida, vamos avaliá-lo, podendo ocorrer três situações:

- Caso o tamanho da indentação seja maior à indentação do topo da *stack*, estamos perante um *token* "INDENT" e realizamos um *push* do seu tamanho na *stack*.

- Caso o tamanho da indentação seja menor à indentação presente no topo da *stack*, retornamos um *token* "DEDENT". De seguida, fazemos um *pop* na *stack* e retrocedemos em um *token*, visto que podemos estar numa situação de múltiplos "DEDENT" seguidos.
- Caso contrário, ignoramos.

4.1.3 Texto e blocos de texto

Uma *tag* pode ser seguida de texto ou, até mesmo, um bloco de texto.

No caso de uma linha de texto, este é separado dos atributos ou da *tag* através de um espaço em branco, pelo que o reconhecimento deste nos leva a um novo estado, "text", que poderá identificar texto que, neste caso, são todos os caracteres à exceção do '\n'.

Para além do Pug poder conter uma simples linha de texto dentro de uma *tag*, o mesmo pode conter blocos. Os blocos de texto são iniciados por "." aparecendo o texto nas linhas seguintes, sendo identificado pela indentação. Assim, foi necessário definir o token "DOT" que identifica esse caracter e que inicia o estado para essa funcionalidade, intitulado por "textblock". Houve em primeiro lugar, o cuidado de definir a função de reconhecimento desse *token* depois da função do *token* "CLASS" porque senão seria impossível verificar este último. A nossa primeira abordagem foi reconhecer em ambos os casos o *token* "DOT" e depois efetuar o *look-behind* do ponto no *token* "CLASS". No entanto, tínhamos depois alguns conflitos na gramática, pelo que resolvemos desta forma. O bloco de texto só é finalizado quando existe um "DEDENT" pelo que enquanto existirem "INDENT" ou o nível ficar constante, mantemo-nos dentro do estado, caso contrário voltamos ao estado inicial. Desta forma, todo o texto que for encontrado numa das linhas intermédias será considerado como, efetivamente, texto.

4.1.4 Definição de variáveis

Primeiramente, apresentamos um exemplo de uma definição de variáveis em Pug, que deve ser realizada da seguinte forma: `- var title = "Pug to HTML"`.

No reconhecimento da sequência "-var" ou "- var", sabemos que estamos na presença de uma definição de variável e, portanto, inicia-se o estado "vardefinition". Este estado recolhe o nome da variável "VAR", o *token* "ASSIGN", correspondente ao símbolo "=", e a respetiva expressão em *JavaScript* que será atribuída à variável em questão.

4.1.5 Interpolação de variáveis

A interpolação de expressões "JavaScript" em *strings* foi outro dos objetivos do grupo, pelo que teríamos de levantar os *tokens* necessários para a sua implementação.

Desta forma, se estivermos no estado "text" ou "textblock", referentes aos estados dos textos como explicado acima, e capturarmos a sequência de carateres `#{`, entramos no estado de interpolação. Neste estado, capturamos todos os carateres exceto `}`, que irão ser guardados no token "INTERPOLATION". De seguida, retornamos ao estado anterior e, para isso, recorremos ao método "pop_state()" do '*lexer* que nos permite voltar ao estado antigo sem o precisarmos de conhecer.

4.1.6 Definição de condicionais

Uma instrução condicional do tipo "if", apenas possui dois tipos de *tokens* - a palavra reservada "IF" e uma expressão a ser testada, "JSEXPRESSION". Quando é reconhecida a palavra reservada "if" , entramos no estado exclusivo "conditional" que passa a reconhecer a expressão a ser testada. De notar que esta expressão pode resumir-se ao nome de uma variável definida.

Para além da instrução condicional do tipo "if", foi também implementado o reconhecimento da instrução condicional do tipo "unless", suportada pelo Pug. Esta instrução tem um funcionamento contrário ao do "if", isto é, as ações do bloco condicional serão realizadas caso a condição apresentada seja falsa. Em relação ao reconhecimento, o processo é bastante semelhante: quando é reconhecida a palavra "unless" entramos no estado "conditional" que passa a reconhecer a expressão a ser testada, esta que, mais uma vez, pode ser simplesmente o nome de uma variável.

4.1.7 Definição de ciclos

Em relação aos métodos de iteração, foram implementados os dois ciclos disponíveis: "each" e "while".

As instruções com "each" possuem dois *tokens* associados a palavras reservadas, "EACH" e "IN". Segundo a documentação do Pug, pode-se também definir ciclos "each" através da palavra "for", sendo, por isso, importante referir que a expressão regular definida para o *token* "EACH" reconhece ambas as palavras: "for" e "each". O reconhecimento deste símbolo faz o *lexer* entrar no estado "eachiteration". Neste estado podemos validar o nome da variável que vai iterar, a palavra reservada "in" e a estrutura de dados a percorrer. No nosso trabalho apenas definimos as listas, ou seja, assim que é reconhecido o *token* "[", entramos no estado "list" que, por sua vez, reconhecerá os *tokens* "ITEM", que representa cada item da lista, "COMMA", que é a vírgula responsável por separar os elementos, e, por último, "]", que iniciará de volta o estado "eachiteration".

Quanto aos ciclos "while", assim que é reconhecida a palavra "while", inicia-se o estado "whileiteration" que reconhece a "JSEXPRESSION" a ser avaliada.

4.1.8 Comentários

Em Pug, existem comentários que devem ser mantidos no *output* em HTML e outros que se limitam apenas ao texto na primeira linguagem. Os comentários que devem aparecer no texto HTML que é renderizado partilham o mesmo formato que os comentários em *JavaScript*, isto é, iniciam com "//", enquanto que os que devem ser estritos ao Pug iniciam com "//-", ou seja, têm um hífen adicional.

Deste modo, quando reconhecemos um comentário em Pug, não devemos efetuar nenhuma ação. Em contrapartida, quando estamos perante o outro tipo de comentário, devemos guardar a sua informação, removendo os 2 primeiros caracteres, referentes às *slashes*. É importante que as funções sigam a ordem apresentada, de modo a que o comentário Pug não seja reconhecido como um comentário a ser mantido no texto exportado.

```
1 def t_pug_comment(t):
2     r"\/\/-[^\n]+"
3     pass
4
5 def t_COMMENT(t):
```

```
6       r"\/\/[^\n]+"
7       t.value = t.value[2:]
8       return t
```

Não adicionamos os comentários em bloco ao nosso projeto, apesar de acreditarmos que são de fácil implementação. A sua análise léxica é muito semelhante aos blocos de texto anteriormente apresentados, fazendo-se uso da indentação; apenas diferem na medida em que os blocos de texto são introduzidos após a presença de um ponto, enquanto que os blocos de comentário são intrudozidos após a presença dos *tokens* que introduzem os comentários, já referidos.

4.2 Analisador sintático

Finalizada a análise léxica, partimos para a análise sintática. Tal como na análise léxica, fomos implementando as partes principais, expandindo depois a gramática até abranger todas as funcionalidades que pretendíamos.

4.2.1 Definição da gramática

Primeiramente, antes de avançarmos para a implementação do analisador sintático, tivemos que definir a gramática relativa à linguagem Pug.

A nossa regra principal designa-se por "pug". Um documento em Pug é dado por uma lista de expressões, lista esta que pode conter uma ou mais expressões.

```
1       pug -> expression_list
2
3       expression_list -> expression
4                        | expression_list expression
```

Cada expressão corresponde a uma linha da linguagem Pug, podendo esta ter várias possibilidades. As linhas podem ser um comentário, uma definição de uma variável, uma condicional, um iterador ou podem representar uma *tag*. Na presença de uma *tag* podemos ter atributos, texto ou um bloco de expressões. Este último é representado pela sequência de *tokens* "INDENT" "expression_list" "DEDENT".

```
1  expression -> TAG
2  expression -> TAG INDENT expression_list DEDENT
3  expression -> TAG text
4  expression -> TAG text INDENT expression_list DEDENT
5  expression -> TAG attributes
6  expression -> TAG attributes INDENT expression_list DEDENT
7  expression -> TAG attributes text
8  expression -> TAG attributes text INDENT expression_list DEDENT
9  expression -> TAG DOT INDENT text DEDENT
10 expression -> TAG attributes DOT INDENT text DEDENT
11 expression -> TAG ASSIGN JSEXPRESSION
12 expression -> TAG ASSIGN JSEXPRESSION INDENT expression_list DEDENT
13 expression -> TAG attributes ASSIGN JSEXPRESSION
14 expression -> TAG attributes ASSIGN JSEXPRESSION INDENT expression_list DEDENT
```

```
15 expression -> IF JSEXPRESSION INDENT expression_list DEDENT
16 expression -> IF JSEXPRESSION INDENT expression_list DEDENT ELSE INDENT
       expression_list DEDENT
17 expression -> UNLESS JSEXPRESSION INDENT expression_list DEDENT
18 expression -> VARDEF VAR ASSIGN JSEXPRESSION
19 expression -> EACH VAR IN list INDENT expression_list DEDENT
20 expression -> WHILE JSEXPRESSION INDENT expression_list DEDENT
21 expression -> COMMENT
```

Os atributos foram desenvolvidos de modo a garantir que cada *tag* tem apenas 1 atributo "id". Recorremos novamente a uma lista, onde podemos ter 0 ou n atributos.

```
1      attributes -> ID
2      attributes -> ID attribute_list
3      attributes -> attribute_list ID
4      attributes -> attribute_list
5      attributes -> attribute_list ID attribute_list
6
7      attribute_list -> CLASS
8      attribute_list -> LBRACKET RBRACKET
9      attribute_list -> LBRACKET attributecont RBRACKET
10     attribute_list -> attribute_list CLASS
11     attribute_list -> attribute_list LBRACKET RBRACKET
12     attribute_list -> attribute_list LBRACKET attributecont RBRACKET
13
14     attributecont -> ATTRIBUTENAME ASSIGN QUOTE ATTRIBUTE QUOTE
15     attributecont -> attributecont ATTRIBUTESPACE ATTRIBUTENAME ASSIGN QUOTE
          ATTRIBUTE QUOTE
```

Como podemos ter interpolação de expressões "JavaScript" nas *strings*, desenvolvemos a seguinte regra para o texto:

```
1      text -> INTERPOLATION text
2      text -> INTERPOLATION
3      text -> TEXT text
4      text -> TEXT
```

Para as listas, sabemos que estas são limitadas por parênteses retos e podem ser vazias. Quando a lista contém mais do que um elemento, estes devem ser separados por vírgulas.

```
1      list -> LSQBRACKET RSQBRACKET
2      list -> LSQBRACKET listcontent RSQBRACKET
3
4      listcontent -> ITEM
5      listcontent -> ITEM COMMA listcontent
```

4.2.2 Implementação do *parser*

Com a gramática já definida, procedemos à construção das funções, com a respetiva sintaxe do "Yacc", que iriam validar a gramática.

Já com as funções todas definidas, levantou-se o problema sobre como viríamos a converter o Pug em HTML, pelo que realizamos duas abordagens.

A primeira abordagem passava por processar o HTML nas funções do "Yacc", como podemos verificar no seguinte exemplo.

```
1    def p_expression_tag_text(p):
2        """expression : TAG TEXT"""
3        p[0] = '<' + p[1] + '>' + p[2] + '</' + p[1] + '>'
```

No entanto, esta estratégia não se revelou ser a melhor, uma vez que não tinhamos a flexibilidade que prentendiamos. Desta forma, optamos por utilizar uma estrutura intermédia que corresponde a uma árvore. Esta árvore é representada pelo ficheiro "tree.py" que, por sua vez, contém várias classes que representam cada tipo de nodo que a árvore pode ter.

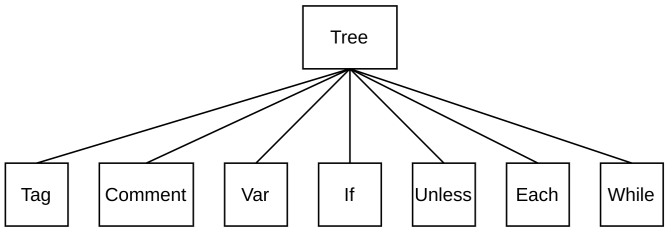

Figura 4.1: Nodos possíveis da estutura intermédia.

Vimos a necessidade de criar diferentes tipos de nodos, uma vez que tinhamos de guardar diferentes informações consoante as diferentes expressões da gramática. Por exemplo, no nodo relativo à implementação da "Tag" necessitamos de guardar a *tag*, atributos e texto caso existam. Já no "IfNode", precisamos de ter informação acerca da condição. De notar que todos os nodos possuem uma variável de instância relativa aos blocos que cada nodo pode possuir, o que torna esta estrutura numa árvore.

Posteriomente, toda esta árvore é processada, de modo a que cada nodo tome o comportamento necessário, ou seja, no caso do "EachNode", terá de iterar numa lista. Em simultâneo, a *string* final também é gerada. Desta forma, cada classe/nodo contém o método "pug_to_html", responsável por gerar a respetiva representação da informação em HTML e devolvendo-a numa *string*.

Expressões em *JavaScript*

Um dos primeiros impasses consistiu em como avaliar expressões extensas, contendo vários tipos de variáveis. De facto, não seria viável criar funçõs de reconhecimento para listas, objetos, expressões aritméticas, dicionários, bem como funções em "JavaScript". Para além disso, estas expressões poderiam conter variáveis cujo valor teria de ser armazenado previamente na sua definição. Com isto, a solução resumiu-se no uso da biblioteca "js2py", que nos permite correr código "JavaScript" em "Python".

Inicialmente, é criado um contexto que vai guardar todas as variáveis que foram declaradas até ao momento. Sempre que uma variável é declarada, o método "pug_to_html" da classe "VarNode" vai tratar de a adicionar ao contexto através da função "execute" do Js2Py.

```
    def pug_to_html(self, string="", context=js2py.EvalJs()):
        try:
            context.execute(f"var {self.name} = {self.value}")
        except:
            context.execute(f"var {self.name} = '{self.value}'")

        return string
```

Quando estamos perante expressões que necessitam de ser avaliadas em nodos como o "If", "While", "Unless" ou até mesmo na "Tag" quando lhe é atribuída uma expressão, invocamos a função "eval" que devolve o resultado do código *JavaScript* dado.

```
    try:
        string += str(context.eval(self.expression))
    except:
        string += ""
```

4.3 Apresentação dos resultados

Como tínhamos abordado anteriomente neste relatório, o conversor deverá ser capaz de efetuar a leitura de um ficheiro e exportar o texto gerado para um novo ficheiro.

Inicialmente, é possibilitado ao utilizador a introdução da diretoria do ficheiro de *input* em Pug, como é exemplificado na seguinte figura.

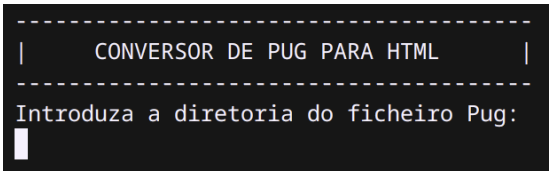

Figura 4.2: Interface do conversor.

Em seguida, é realizada a análise léxica e sintática do texto e produzido o resultado da conversão. Contudo, o grupo pretendeu que este resultado fosse apresentado de uma forma visualmente atrativa, devidamente indentado. Uma vez que o foco do trabalho não se direcionou propriamente para esta parte, acabamos por recorrer a uma biblioteca para tratar de indentar, de forma devida, o texto. A biblioteca utilizada foi a "Beautiful Soup"', que pode ser instalada através do seguinte comando: `pip install beautifulsoup4`. Deste modo, a utilização desta ferramenta de abstração poupou bastante tempo ao grupo, tempo este que foi útil para a implementação de novas funcionalidades e garantir o funcionamento correto do programa.

4.4 Testes realizados e resultados obtidos

Nesta secção, apresentamos alguns testes realizados pelo grupo de modo a garantir a correção do conversor, tentando cobrir toda a parte da linguagem Pug que consegue ser reconhecida.

Inicialmente, fazemos um teste simples onde estão definidos 2 identificadores para a mesma *tag*. Sabemos que isso não é possível em HTML e foi uma das restrições adicionais cuja implementação interessou ao grupo. Assim sendo, o resultado esperado é um erro, tal como obtido.

Teste

```
1 html#principal(id="ola")
```

Resultado Obtido

```
1 Erro sintatico no input!ID(ola) on line 1
```

Em seguida, fazemos um teste englobando as várias representações possíveis de atributos em Pug. A *tag*, que neste caso é esperado que seja uma *div*, deve ainda conter o texto "texto".

Teste

```
1 html
2     head
3     body
4         p().class1#id(class="class2").class3() texto
```

Resultado Obtido

```
1 <html>
2     <head>
3     </head>
4     <body>
5         <p class="class1 class2 class3" id="id">
6             texto
7         </p>
8     </body>
9 </html>
```

Para o terceiro teste, aplicamos um bloco de texto à *tag* "p". Esta foi uma das funcionalidades requeridas pelo enunciado.

Teste

```
p.
    text1
    text2
    text3
```

Resultado Obtido

```
<p>
    text1
text2
text3
</p>
```

De modo a testar a definição de variáveis e as expressões em *JavaScript*, realizamos o seguinte teste, onde onde a variável "v1" deverá guardar o número "3.5". Em relação à variável "text", o valor que lhe deverá ser atribuído deverá resultar da soma de 2 com 1, que será 3; com a concatenação deste número da *string* vazia, o resultado passará a ter um tipo diferente: uma *string* que, neste caso, contém desde já o carater "3"; em seguida, é concatenado o número 3 e, ainda, o número 3.5, que é o valor de "v1", sendo que ao concatenarmos cada número à *string* estamos a tranformá-lo numa *string* e a concatená-la, sem que sejam efetuadas operações aritméticas. Também é possível verificar que podemos aplicar funções, neste caso a strings.

Teste

```
html
    -var v1=2+3/2
    text= 1+2+"teste".toUpperCase()+3+v1
```

Resultado Obtido

```
<html>
    <text>
        3TESTE33.5
    </text>
</html>
```

No próximo teste, usufruimos das instruções condicionais "if" e "unless". Primeiramente, verificamos, também, o correto funcionamento da expressão em *JavaScript* ao receber o nome de uma variável, sendo que o valor da variável, sendo que é, efetivamente, verificado o valor dessa variável em questão. É, portanto, esperado desde já a *string* "Verdade" no resultado final. Em seguida, como a seguinte condição não se verifica, deve ser exportado o conteúdo do bloco do "else". Por último, é ainda testada a instrução condicional "unless" - dado que a condição não é verídica, é esperado que seja exportado o conteúdo desse bloco. Realçamos que as *strings* iniciadas por "if" foram propositadas, de modo a demonstrar que o texto, relativo à *tag* não pode ser interpretado como uma instrução condicional.

Teste

```
1  html
2      -var v1=1
3      if v1
4          p Verdade
5              if 1+1==1
6                  p if aninhado verdade
7              else
8                  p if aninhado falso
9              p Correu bem
10     else
11         p Falso
12     unless 1+1==1
13         p=2*5
```

Resultado Obtido

```
1  <html>
2      <p>
3          Verdade
4          <p>
5              if aninhado falso
6          </p>
7          <p>
8              Correu bem
9          </p>
10     </p>
11     <p>
12         10
13     </p>
14 </html>
```

Em seguida, testamos os métodos iterativos implementados. Primeiramente, no ciclo "each", é esperado que apenas o "false" não seja exportado como conteúdo de uma *tag*. Dentro deste ciclo, temos, ainda, um ciclo "for" aninhado, onde é esperado que apenas seja exportada uma *tag* "p" com o conteúdo "1". Por último, é testado o ciclo "while", sendo o resultado obtido coincidente com o esperado. Neste exemplo, também é possível ver a interpolação de variáveis no texto em funcionamento.

Teste

```
1 html
2     each i in [1,2,"ola",1.5,false]
3         if i
4             p texto #{i}
5         for i in[1]
6             p=i
7     while i<3
8         p=1
9         li=i++
```

Resultado Obtido

```
1 <html>
2     <p>
3         texto 1
4     </p>
5     <p>
6         1
7     </p>
8     <p>
9         texto 2
10    </p>
11    <p>
12        1
13    </p>
14    <p>
15        texto ola
16    </p>
17    <p>
18        1
19    </p>
20    <p>
21        texto 1.5
22    </p>
23    <p>
24        1
25    </p>
26    <p>
27        1
28    </p>
29    <p>
30        1
31    </p>
32    <li>
33        1
34    </li>
35    <p>
36        1
37    </p>
38    <li>
39        2
```

```
40      </li>
41  </html>
```

Neste teste, vamos apresentar a funcionalidade de comentários em Pug. Existem dois tipos de comentários de linha , aqueles que são apenas do pug, e comentários que fazem parte do código HTML. Temos, portanto, o primeiro comentário que é o comentário do pug e como podemos ver não aparece no resultado final. Por outro lado, o segundo comentário já aparece no resultado final.

Teste

```
1  html
2      //- will not output within markup
3      p Sem Comentario
4      // will output within markup
5      p Com Comentario
```

Resultado Obtido

```
1  <html>
2      <p>
3          Sem Comentario
4      </p>
5      <!-- will output within markup-->
6      <p>
7          Com Comentario
8      </p>
9  </html>
```

Por último, apresentamos o teste disponibilizado no enunciado do projeto, juntamente com os resultados obtidos, que vão de encontro ao *output* também fornecido no enunciado.

Teste disponibilizado

```
1  html(lang="en")
2      head
3          title= pageTitle
4          script(type='text/javascript').
5              if (foo) bar(1 + 5)
6      body
7          h1 Pug - node template engine
8          #container.col
9              if youAreUsingPug
10                 p You are amazing
11             else
12                 p Get on it!
13             p.
14                 Pug is a terse and simple templating language with a
15                 strong focus on performance and powerful features
```

Resultado obtido

```
<html lang="en">
    <head>
        <title>
        </title>
        <script type="text/javascript">
            if (foo) bar(1 + 5)
        </script>
    </head>
    <body>
        <h1>
            Pug - node template engine
        </h1>
        <div class="col" id="container">
            <p>
                Get on it!
            </p>
            <p>
                Pug is a terse and simple templating language with a
strong focus on performance and powerful features
            </p>
        </div>
    </body>
</html>
```

Ao longo dos testes, foi evidente a importância de considerar uma ampla gama de cenários de forma a validar as funcionalidades que desenvolvemos. Assim, olhando para todos os testes construídos de forma satisfatória, fazemos um balanço positivo visto que eles cumprem com aquilo que esperavamos.

Capítulo 5

Conclusão

Neste projeto fizemos uso do gerador de compiladores baseados em gramáticas tradutoras, Yacc, e do gerador de analisadores léxicos, Lex, para criar um conversor da linguagem de *template* Pug para a linguagem de marcação de hipertexto HTML.

Acreditamos que a realização deste conversor aumentou, efetivamente, a nossa capacidade de produção de gramáticas. Todas as funcionalidades básicas propostas no enunciado estão implementadas, juntamente com outras funcionalidades mais avançadas que valorizam o projeto do ponto de vista do grupo, fazendo-se por isso uma avaliação positiva do trabalho desenvolvido.

Futuramente, o conversor elaborado poderia vir a suportar as restantes funcionalidades que se encontram disponíveis na documentação do Pug, como é o caso dos *case statements* que não consideramos que tenham uma difícil implementação. Uma das funcionalidades em destaque do Pug são os *mixings*, que consistem em blocos de código reutilizáveis, constituíndo também um importante aspeto a ser adicionado a este conversor, assim como a herança entre vários ficheiros suportada, que promove a modularidade na criação de páginas *web*.

Apêndice A

Analisador Léxico

Listing A.1: Tokens recolhidos.

```
tokens = (
    'EACH',
    'IN',
    'WHILE',
    'UNLESS',
    'IF',
    'ELSE',
    'INDENT',
    'DEDENT',
    'TAG',
    'ATTRIBUTENAME',
    'ATTRIBUTESPACE',
    'ATTRIBUTE',
    'TEXT',
    'ID',
    'CLASS',
    'VAR',
    'ITEM',
    'VARDEF',
    'JSEXPRESSION',
    'INTERPOLATION',
    'COMMENT',
    'LBRACKET',
    'RBRACKET',
    'LSQBRACKET',
    'RSQBRACKET',
    'DOT',
    'COMMA',
    'ASSIGN',
    'QUOTE'
)
```

Listing A.2: Estados definidos.

```
states = (
    ('textblock', 'exclusive'),
    ('text', 'exclusive'),
```

```
4       ('attributes', 'exclusive'),
5       ('attributescontent', 'exclusive'),
6       ('conditional', 'exclusive'),
7       ('attribution', 'exclusive'),
8       ('interpolation', 'exclusive'),
9       ('vardefinition', 'exclusive'),
10      ('eachiteration', 'exclusive'),
11      ('whileiteration', 'exclusive'),
12      ('list', 'exclusive')
13  )
```

Apêndice B

Analisador Sintático

Listing B.1: Gramática.

```
1  pug -> expression_list
2
3  expression_list -> expression
4  expression_list -> expression_list expression
5
6  expression -> TAG
7  expression -> TAG INDENT expression_list DEDENT
8  expression -> TAG text
9  expression -> TAG text INDENT expression_list DEDENT
10 expression -> TAG attributes
11 expression -> TAG attributes INDENT expression_list DEDENT
12 expression -> TAG attributes text
13 expression -> TAG attributes text INDENT expression_list DEDENT
14 expression -> TAG DOT INDENT text DEDENT
15 expression -> TAG attributes DOT INDENT text DEDENT
16 expression -> TAG ASSIGN JSEXPRESSION
17 expression -> TAG ASSIGN JSEXPRESSION INDENT expression_list DEDENT
18 expression -> TAG attributes ASSIGN JSEXPRESSION
19 expression -> TAG attributes ASSIGN JSEXPRESSION INDENT expression_list DEDENT
20 expression -> IF JSEXPRESSION INDENT expression_list DEDENT
21 expression -> IF JSEXPRESSION INDENT expression_list DEDENT ELSE INDENT
       expression_list DEDENT
22 expression -> UNLESS JSEXPRESSION INDENT expression_list DEDENT
23 expression -> VARDEF VAR ASSIGN JSEXPRESSION
24 expression -> EACH VAR IN list INDENT expression_list DEDENT
25 expression -> WHILE JSEXPRESSION INDENT expression_list DEDENT
26 expression -> COMMENT
27
28 text -> INTERPOLATION text
29 text -> INTERPOLATION
30 text -> TEXT text
31 text -> TEXT
32
33 attributes -> attribute_list
34 attributes -> attribute_list ID
35 attributes -> attribute_list ID attribute_list
36 attributes -> ID
```

```
37 attributes -> ID attribute_list
38
39 attribute_list -> CLASS
40 attribute_list -> LBRACKET RBRACKET
41 attribute_list -> LBRACKET attributecont RBRACKET
42 attribute_list -> attribute_list CLASS
43 attribute_list -> attribute_list LBRACKET RBRACKET
44 attribute_list -> attribute_list LBRACKET attributecont RBRACKET
45
46 attributecont -> ATTRIBUTENAME ASSIGN QUOTE ATTRIBUTE QUOTE
47 attributecont -> attributecont ATTRIBUTESPACE ATTRIBUTENAME ASSIGN QUOTE ATTRIBUTE
       QUOTE
48
49 list -> LSQBRACKET RSQBRACKET
50 list -> LSQBRACKET listcontent RSQBRACKET
51
52 listcontent -> ITEM
53 listcontent -> ITEM COMMA listcontent
```

Bibliografia

[Bea] David Beazley. PLY (Python Lex-Yacc). https://www.dabeaz.com/ply/ply.html.

[Pug] PugJS. Getting Started - Pug. https://pugjs.org/api/getting-started.html.