Processamento de Linguagens (3º - Ano)

Conversor Pug para HTML

Relatório de Desenvolvimento

Grupo nº51

João Gonçalo de Faria Melo (A95085)

Hugo Filipe Silva Abelheira (A95151)

Eduardo Miguel Pacheco Silva (A95345)

Universidade Do Minho

28 de maio de 2023

Resumo

Este relatório visa ilustrar o desenvolvimento do projeto de Processamento de Linuagens, um Conversor de Pug para HTML.

São abordados vários temas, desde a própria contextualização do trabalho à sua implementação, com particular ênface no desenvolvimento de cada uma das etapas necessárias, elaborando sobre os desafios enfrentados e as nossas decisões. Explicamos alguns conceitos e analisamos os resultados obtidos.

Conteúdo

Capítulo 1

Introdução

1.1 Estrutura do Relatório

O relatório está dividido em 6 capítulos: Introdução, Análise e Especificação, Concepção da Resolução, Codificação e Testes, Conclusão e ainda Fontes e Referências.

Este capítulo de Introdução serve para dar uma sucinta explicação deste relatório, bem como contextualizar o projeto.

No capítulo da Análise e Especificação fazemos uma descrição informal do problema em si.

Em seguida, na Concepção da Resolução, falamos do processo de desenvolvimento do projeto, elaborando cada uma das etapas com uma descrição sucinta.

É apenas no capítulo da Codificação e Testes que explicamos como aplicamos a metodologia de trabalho, as ferramentas que utilizamos as decisões que o grupo tomou, bem como os problemas e soluções de cada uma das fases. Nesta fase do relatório explicaremos também alguns conceitos necessários para a compreensão do projeto.

Finalizamos com o capítulo da Conclusão onde discutimos os resultados obtidos e elaboramos sobre o decorrer do projeto, bem como o seu futuro. Segue-se um capítulo com as fontes usadas que nos auxiliaram no desenvolver do trabalho.

1.2 Contextualização

No âmbito da disciplina Processamento de Linguagens, vamos desenvolver um conversor de Pug para HTML, aplicando conhecimentos sobre *regex*, construção de gramáticas e *parsings*. Este relatório ilustra o problema, o nosso processo de desenvolvimento e as resultados obtidos, bem como algumas conclusões e exemplos.

Capítulo 2

Análise e Especificação

2.1 Descrição do problema

O enunciado escolhido para desenvolvimento corresponde ao enunciado **2.5. Conversor de *Pug* para HTML**.

O projeto em si trata-se do desenvolvimento de um conversor que aceite um subconjunto da linguagem *Pug* e gere o HTML correspondente. *Pug* é uma linguagem de modelagem para HTML que simplifica a escrita de código por meio de uma sintaxe mais concisa e intuitiva, no entanto, às vezes pode ser necessário converter esse código para HTML, por exemplo, problemas de compatibilidade.

Este projeto apresenta alguns desafios sólidos. Por exemplo, a sintaxe do Pug é significativamente diferente da sintaxe do HTML, o que requer a implementação de regras de tradução precisas. Para além disso, uma das maiores dificuldades está em definir a hierarquia entre as diferentes *tags*. Em HTML, a identificação da abertura e fecho de uma *tag* estão bem claras através da notação $< tag >$ para abertura e $< /tag >$ para fecho. No entanto, em Pug, esta gestão hierárquica é feita através de níveis de identação.

Capítulo 3

Concepção da Resolução

3.1 Etapas de Implementação

Para a resolução do problema em questão, ficou claro para o grupo que o nosso programa deveria seguir as seguintes fases:

Análise Léxica → Análise Sintática → Análise Semântica → *Tree-Walk Interpretation*

3.2 Análise Léxica

A análise léxica é a primeira etapa do processo de compilação de um programa. O nosso analisador léxico deverá ser capaz de analisar o código Pug dado como entrada e dividi-lo em unidades parciais, designadas por *tokens*. Assim, devemos definir um léxico adaptado à sintaxe de Pug, que seja capaz de diferenciar indistiguivelmente as características de cada elemento. A definição deste léxico exige que enumeremos *tokens*, estados e ainda listas de palavras reservadas, caso necessário. Cada *token* representará uma sequência de um ou mais caracteres com significado específico. São ainda definidas as regras para a dispensabilidade de determinados caracteres.

O analisador léxico vai, portanto, percorrer o nosso texto de entrada e identificar os *tokens* conforme os padrões que definirmos com a ajuda de expressões regulares.

O resultado da análise léxica é importante para a próxima etapa do processo, a análise sintática, na qual usaremos o reconhecimento dos *tokens* para a definição de uma gramática e de regras de produção.

3.3 Análise Sintática

A análise sintática corresponde à segunda fase de resolução. A sua função é analisar a sequência de *tokens* obtida a partir do léxico e verificar se este segue a estrutura gramatical definida. É, portanto, nesta fase que elaboramos uma gramática que nos ajude, através de um algoritmo de *parsing*, construir a nossa estrutura de dados que será analisada posteriormente por um *Tree-Walk Interpreter*.

A nossa gramática formal será uma gramática independente de contexto (GIC), que descreve as regras sintáticas da linguagem. Será aplicado um algoritmo LALR para a construção da *stack* e árvore de derivação.

3.4 Análise Semântica

A etapa relativa à análise semântica visa verificar se as regras semânticas da linguagem estão a ser respeitadas. No nosso projeto, esta análise é feita conjuntamente e em simultâneo da análise sintática.

3.5 *Tree-Walk Interpretation*

A fase *Tree-Walk Interpretation* ou fase de Interpretação e Percurso da Árvore é a última etapa para a resolução do problema e finalizar a conversão de *Pug* para *HTML*. É aqui que será percorrida e interpretada a árvore de derivação construída na análise sintática, por forma a realizar a conversão. A estrutura de dados utilizada corresponde a uma AST (*Abstract Syntax Tree*). Os detalhes relativamente à forma como decidimos definir a nossa estrutura de dados estão descritos no sub-capítulo 4.2.4.

Capítulo 4

Codificação e Testes

4.1 Ferramentas utilizadas

O projeto foi implementado utilizando a linguagem de programação *python* e o PLY (*Python Lex-Yacc)* que inclui ferramentas de construção de compiladores como o *lex*, para a análise léxica, e o *yacc*, destinado à análise sintática. Estas foram as ferramentas abordadas na UC ao longo do semestre e esperadas para a realização do projeto.

4.2 Alternativas, Decisões e Problemas de Implementação

O código elaborado está divido em três ficheiros *python* que abordam por completo as ferramentas especificadas em 4.1:

- *lex.py* - **Analisador Léxico**
- *yacc.py* - **Analisador Sintático**
- *html_nodes.py* - **Estrutura de dados**

4.2.1 Analisador Léxico

O nosso analisador léxico (*lex.py*) define as *tokens* e os estados necessários. É utilizado o módulo *re* de *python* para a utilização de expressões regulares. Alguns exemplos de *tokens* que criámos são: TAG_NAME, TEXT, ATTRIBUTE_NAME, ATTRIBUTE_VALUE, etc, que visam corresponder a sintaxe característica de Pug como *tags*, conteúdo textual e atributos. Fizemos ainda uso de estados exclusivos como: 'tag', 'attribute', etc, que ajudam a identificar e alterar as nossas regras de reconhecimento depois de entrarmos dentro de uma *tag* ou dentro de um atributo.

4.2.2 Analisador Sintático

O primeiro passo na definição do nosso analisador sintático é a definição da gramática formal.

Em seguida, na figura 4.2.2 é descrita, em notação BNF (*Backus-Naur Form*), a gramática desenvolvida para a resolução do problema, que visa ter em conta aspetos como a hierarquia de *tags*.

Para cada elemento gramatical foram definidas as respetivas regras de produção que criam a nossa AST, cujas características estão especificadas no ficheiro *html_nodes.py*

```
<line> := <dent> <tag> <line> | <tag> <line> | <dent_tag_aux>

<dent_tag_aux> := <dent> <tag>

<dent> := <indent> | <outdent> | <nodent>

<indent> := INDENT

<outdent> := OUTDENT

<notdent> := NODENT

<tag> := TAG_NAME <inside_tag> | <div_tag>

<div_tag> := <inside_tag>

<inside_tag> := <attributes_aux> <text>

<attributes_aux> := <attributes> | <attribute_empty>

<attributes> := LPAREN <attributes_list> RPAREN <special_attribute
                | <special_attribute> <attributes>
                | <special_attribute> <attribute_empty>
                | <attribute_empty>

<attributes_list> := <attribute> COMMA <attributes_list> | attribute

<attribute> := ATTRIBUTE_NAME EQUALS ATTRIBUTE_VALUE | <attribute_empty>

<special_attribute> := <class_attribute>
                     | <id_attribute>

<class_attribute> := DOT SPECIAL_ATTRIBUTE <id_attribute>
                   | <class_attribute_empty

<id_attribute> := HASTAG SPECIAL_ATTRIBUTE <class_attribute> | <id_attribute_empty>

<text> := SPACE TEXT | <text_empty>
```

Figura 4.1: Gramática

O analisador sintático desenvolvido utiliza o algoritmo LALR *(Look-Ahead LR)* para analisar eficientemente a estrutura gramatical do programa. O LALR combina o algoritmo LR com o conceito de **lookahead**, permitindo ao analisador tomar decisões com base num número limitado de símbolos seguintes. Isso resulta numa análise rápida e precisa, graças à construção de conjuntos de itens LR(0) e LR(1) e à utilização de tabelas de análise.

Utilizamos bastante a ferramenta de *debug* disponível na biblioteca *yacc* para a identificação e correção de erros na nossa gramática.

4.2.3 Stack

De maneira a construir a nossa árvore DOM (Modelo de Objeto de Documento), a estrutura hierárquica de um documento HTML, composta por nodos, onde cada nodo representa um elemento HTML, utilizamos uma *stack* para inserir os nodos e o seu nível de indentação. Isto porque, como o analisador sintático LALR é um *parser Bottom-Up* a nossa árvore é construida a partir do final do ficheiro *Pug*.

Por exemplo, se usarmos o código Pug que se segue para converter para HTML:

```
h1 Processamento de Linguagens
    h2 Analisadores Lexicos
        p lexer
    h2 Analisadores Sintaticos
        h3 LALR
            p bottom-up
    h2 Conclusao
```

O funcionamento da *stack* seria o seguinte:

Estado Inicial

Este é o momento em que a nossa *stack* se encontra vazia, pois ainda não houve nenhum *reduce* de uma *line*, o que na nossa gramática representa representa as indentações mais a *tag*, ou seja, "`h2 Conclusao`"ainda não é colocada na *stack*.
Stack – []

1ª Iteração

Nesta iteração o nodo da *tag* "`h2 Conclusao`"é adicionada à *stack* juntamente com o seu nível de indentação, neste caso, seria 1, através de um tuplo.
Stack - [("<h2> Conclusao<h2>", 1)]

2ª Iteração

Como o próximo nodo tem maior nível de indentação vamos apenas dar *push* desse nó na *stack* de maneira a ficar no topo da *stack*. É de notar que o lado direito da *stack* é o nosso topo da *stack*.
Stack - [("<h2> Conclusao<h2>", 1), (bottom-up , 3)]

3ª Iteração

Neste passo, o nosso nodo "`h3 LALR`"tem menor indentação que o nodo que se encontra no topo da *stack*. Quando isso acontece, significa que o nodo com menor indentação é pai do nodo do topo da *stack*, ou seja, damos *pop* ao nodo, e adicionamos esse nodo aos filhos do pai. É de mencionar que este processo deve ser repetido até o nodo do topo da *stack* se encontrar com um nível de indentação igual ou inferior ao atual.
Stack - [("<h2>Conclusao<h2>", 1), (<h3>LALR bottom-up </h3>, 2)]

4ª Iteração

Agora é só repetir o processo de inserção na *stack* como foi referido nos passos anteriores.
Stack - [("<h2> Conclusao<h2>", 1),
(<h2>Analisadores Sintaticos<h3>LALR bottom-up </h3></h2>, 1)]

5ª Iteração

Stack - [("<h2> Conclusao<h2>", 1),
(<h2>Analisadores Sintaticos<h3>LALR bottom-up </h3></h2>, 1),
(lexer ,2)]

6ª Iteração

Stack - [("<h2> Conclusao<h2>", 1),
(<h2>Analisadores Sintaticos<h3>LALR bottom-up </h3></h2>, 1),
(<h2>Analisadores Lexicos lexer </h2>,1)]

7ª Iteração

Stack - [(<h1>Processamento de Linguagens
<h2>Analisadores Lexicos lexer </h2>
<h2>Analisadores Sintaticos<h3>LALR bottom-up </h3></h2>
<h2>Conclusao<h2></h1>, 0)]

Como tem indentação 0, significa que é o nodo do topo da árvore.

4.2.4 Estrutura de dados

A estrutura de dados que decidimos utilizar e que está definida em *html_nodes.py* é a seguinte:

TagNode(name, attributes, text, children, parent)

Este objeto *TagNode* não é mais que uma árvore com as seguintes variáveis de instância:

- *name*: Nome da *tag*
- *attributes*: Lista de atributos, em que cada atributo é um *AttributeNode*
- *text*: Conteúdo de uma *tag*
- *children*: Lista de objetos *TagNode*, que são os filhos do *TagNode* em questão
- *parent*: Objeto *TagNode*, que é o pai do *TagNode* em questão

AttributeNode(name, value)

Este objeto apenas define as variáveis de instância de um atributo de uma *tag* HTML:

- *name*: Nome do atributo
- *value*: Valor do atributo

Um aspeto importante a notar é que, para ambos estes objetos o método *__str__* imprime os valores de cada variável conforme extraída do segmento *Pug*, enquanto que o método *__repr__* transforma os segmentos para seu formato HTML. Desta forma, conseguimos aceder muito facilmente à representação HTML do código *Pug*. Mais ainda, é assim que resumimos o nosso *Tree-Walk Interpreter* e, portanto, última fase do projeto, no método *__repr__*.

4.2.5 *Dataset* - Ficheiros de entrada e saída

Os ficheiros de entrada e saída correspondem, respetivamente, aos ficheiros *test.pug* e *test.html*. O programa recebe o ficheiro de entrada, realiza todos os processos necessários à conversão para HTML e escreve o resultado dessa transformação no ficheiro *test.html*.

4.3 Guia de Utilização

Infelizmente, não desenvolvemos nenhuma *interface* que permitisse uma utilização ao utilizador comum. Desta forma, o funcionamento do programa por parte do utilizador consiste nos seguintes passos:

1. Alteração do ficheiro *test.pug* para o código *Pug* de entrada desejado a conversão
2. Execução do ficheiro *yacc.py*
3. Verificação da conversão escrita em *test.html*

4.4 Testes realizados

Nesta secção são apresentados alguns exemplos de testes que realizamos para verificar o funcionamento correto do nosso programa.

4.4.1 Teste 1

Input Pug:

```
h1 Processamento de Linguagens
    h2 Analisadores Lexicos
        p lexer
    h2 Analisadores Sintaticos
        h3 LALR
            p bottom-up
    h2 Conclusao
```

Output HTML:

```
<h1>Processamento de Linguagens
  <h2>Analisadores Lexicos
    <p>lexer</p>
  </h2>
  <h2>Analisadores Sintaticos
    <h3>LALR
      <p>bottom-up</p>
    </h3>
  </h2>
  <h2>Conclusao</h2>
</h1>
```

4.4.2 Teste 2

Input Pug:

```
h1(name="Mario", id="pai") Mario Dias
    #mae.falar(name="joao") Maria Gloria
        .classificador Não tem Filhos
        h3 IRS
    b Avô
```

Output HTML:

```
<h1 name="Mario" id="pai">Mario Dias
    <div id="mae" class="impostos" name="joao">Maria Gloria
        <b>Não tem Filhos
            <h3>IRS</h3>
        </b>
    </div>
</h1>
```

Capítulo 5

Conclusão

Concluindo, o grupo considera que atingiu os objetivos pretendidos, visto que ficou muito mais confortável com a utilização das ferramentas mencionadas ao longo do relatório e também aprofundou diversos conceitos relacionados com Processamento de Linguagens. No entanto, existem ainda muitas funcionalidades que ficaram por implementar, incluindo algumas ilustradas no enunciado, como: blocos condicionais e conteúdo textual multi-linha.

Em geral, aquele que pensamos que foi o maior desafio foi a definição de uma gramática que permitisse definir corretamente as relações pai-filho entre os nodos e, sendo que conseguimos ultrapassar esse obstáculo, ficamos contentes com o resultado.

Capítulo 6

Fontes e Referências

- Ply Documentation : `https://www.dabeaz.com/ply/ply.html`
- Pug Documentation : `https://pugjs.org/api/getting-started.html`