Trabalho Prático Processamento de Linguagens

André Gonçalves A95442 | Henrique Fernandes A95323 | João Brito A95641

(17 de julho de 2024)

1 Introdução

No âmbito da cadeira de processamento de linguagens, foi nos proposto vários enunciados de problemas/desafios, do qual teríamos de selecionar um para resolver. Assim sendo, pensámos que o mais interessante seria o conversor de Pug para HTML(2.5), que apesar de serem duas linguagens relativamente parecidas, têm alguma diferenças que foram complicadas de implementar. Para isto foi usada a linguagem de programação python, e a libraria ply, que nos fornece as ferramentas ply.lex e ply.yacc, usando isso criamos um lexer e um parser.

2 Conversão de Pug para HTML

Tal como já foi referido, Pug e HTML são duas linguagens relativamente parecidas, tendo algumas diferenças que podem ser complicadas de implementar. Assim sendo para implementar este programa foi necessário criar um lexer de tokens, para conseguirmos identificar quais os tokens presentes na linguagem Pug, e atribuir um nome a cada um para depois podermos decidir o que fazer com cada um deles no parser, e também criar uma estrutura a partir desse tokens que faça sentido na linguagem HTML sem destruir o significado, que esta tinha em Pug. Para isso criamos então, um lexer para dividir os varios tokens e um parser para construir a nossa lingugem HTML, sendo usadas as ferramentas, ply.lex e ply.yacc, respetivamente.

2.1 Lexer(Analisador léxico)

Para contruir um Lexer em python é necessário, primeiramente, definir quais os tokens que irão ser necessários capturar e quais os vários estados que os tokens se poderiam inserir, assim sendo os vários tokens e estados que foram necessários definir.

Começamos então por definir os tokens que seria necessários. No entanto foi necessário criar estados para alguns desses tokens, pois alguns tokens poderiam

ter formatos diferentes em vários casos, sendo ambos os estados exclusivos para as regras do estado inicial não interferirem com as regras do estado em si.

Figura 1: Tokens e estados do lexer

(a) Tokens do lexer

```
tokens = (
    'IDENTACAO',
    'SPACE',
    'NEWLINE',
    'TAG',
    'LPAREN',
    'RPAREN',
    'EQUAL',
    'ATRIBUT',
    'ID',
    'CLASS',
    'TEXTO',
    'TEXTPLAIN',
    'COMA',
    'IF',
    'ELSE',
    'VAR'
)
```

(b) Estados do lexer

```
states = (
    ('atributos', 'exclusive'),
    ('tag', 'exclusive'),
    ('frase', 'exclusive'),
    ('textplain', 'exclusive'),
    ('variaveis', 'exclusive')
)
```

2.1.1 Estado INITIAL

Este é o estado inicial do lexer, nele podemos receber um token VAR o que vai inicializar o estado 'variaveis', IF e o ELSE que iram para o estado 'frase' onde no caso do IF ira receber a sua condição, também poderá ir para o estado 'tag' se apanhar um os tokens ID,CLASS e TAG mas o mais importante é o token IENTACAO onde traduz os espaços e os tab's para o seu nível de indentação(um tab equivale a 4 e um espaço a 1) guardando numa variavel nível no lexer que será útil futuramente.

2.1.2 Estado tag

Primeiramente existem três tokens que poderão recorrer ou que irão iniciar ao estado 'tag', o proprio TAG que irá corresponder a qualquer palavra, o CLASS que irá corresponder a todas as palavras que venham a seguir a um ponto e o ID que irá corresponder a todas as palavras após um cardinal. A partir disto, sempre que o analisador encontrar qualquer um destes tokens irá iniciar o estado 'tag' , já dentro do estado, será retornado o token que ele leu(TAG,CLASS ou ID), se dentro desse estado ele encontrar um ponto irá iniciar o estado 'textplain', caso encontre o token LPAREN irá iniciar o estado 'atributos', caso encontre uma sequência de espaço ou um espaço irá iniciar o estado 'frase', caso encontre um igual(EQUAL) apenas retorna o token, e este deve ser seguido de um espaço e uma frase que será o conteúdo da variável, encontrado e por fim quando encontrar um newline acrescenta um ao valor que indentifica a linha do token encontraado, para sinalizar a mudança de linha e retorna ao estado INITIAL.

2.1.3 Estado frase

No estado INITIAL vão existir dois tokens que irão iniciar o estado frase, o IF que quando encontra um if e pelo menos um espaço, muda o valor do token para if apenas e inicia o estado frase, e também o ELSE que quando encontra um else começa também o estado frase. O estado frase em si é bastante simples, pois apenas vai retornar qualquer carater até encontrar um newline e quando encontrar soma 1 ao valor da linha do lexer para simular a modança de linha e retorna ao estado inicial.

2.1.4 Estado variaveis

No estado inicial, quando o programa encontra o token VAR, ou seja '- var', começa o estado variaveis. Já dentro do estado em si, e caso o lexer encontre o token EQUAL retorna o token, caso encontre um token TEXTO que será o conteudo ou o nome da variavel em si, que será quanquer carater ate ser encontrado um espaço, e por fim caso encontre um newline irá retornar ao estado incial e somar 1 ao valor da linha do lexer.

2.1.5 Estado atributos

Tal com ja foi referido este estado não vai ser iniciado no estado inicial, apenas é inciado quando o lexer encontra o token LPAREN no estado tag. Então ao entrar no estado tag, se encontrar um token ATRIBUT, que irá corresponder um uma palavra antes de um igual que irá ser o nome do atributos e á frente do igual irá estar o valor desse atributo entre ou ", caso encontre apenas um token RPAREN retorna ao estado tag, significando que terminou de ler o atributo.

2.1.6 Estado textplain

Quando uma tag acaba num ponto significa que as próximas linhas indentadas serão texto desta tag, para isso verificamos se a indentação é maior o menor que o nível de indentação guardado na variável nível do lexer, se for maior entao vai apanhar o texto todo, se for menor vai para o estado inicial.

2.1.7 Exemplo de output do lexer

Após ter definido todos os tokens e estados necessários, para analisar um texto em Pug, o output para este exemplo de um código em Pug é o seguinte:

```
html(lang="en")
    head
        title= pageTitle
        script(type='text/javascript').
            if (foo) bar(1 + 5)
        - var pageTitle = TP_PL
        subtitle= pageTitle
    body
        h1 Pug - node template engine
        #container.col
            - var youAreUsingPug = true
            if youAreUsingPug
                p.
                    You are amazing
            else
                .oi Get on it!
            p.
                Pug is a terse and simple templating language with a
                strong focus
                performance and powerful features
```

Figura 2: Código Pug

Figura 3: Outputs

```
LexToken(TAG,'html',1,0)
LexToken(LPAREN,'(',1,4)
LexToken(ATRIBUT,'lang="en"',1,5)
LexToken(RPAREN,')',1,14)
LexToken(NEWLINE,'\n',1,15)
LexToken(IDENTACAO,4,2,16)
LexToken(TAG,'head',2,17)
LexToken(NEWLINE,'\n',2,21)
LexToken(IDENTACAO,8,3,22)
LexToken(TAG,'title',3,30)
LexToken(EQUAL,'=',3,35)
LexToken(SPACE,' ',3,36)
LexToken(TEXTO,'pageTitle',3,37)
LexToken(NEWLINE,'\n',3,46)
LexToken(IDENTACAO,8,4,47)
LexToken(TAG,'script',4,55)
LexToken(LPAREN,'(',4,61)
LexToken(ATRIBUT,"type='text/javascript'",4,62)
LexToken(RPAREN,')',4,84)
LexToken(COMA,'.',4,85)
LexToken(NEWLINE,'\n',4,86)
LexToken(IDENTACAO,12,5,87)
LexToken(TEXTPLAIN,'if (foo) bar(1 + 5)',5,99)
LexToken(NEWLINE,'\n',5,118)
LexToken(IDENTACAO,8,6,119)
LexToken(VAR,'- var',6,127)
LexToken(TEXTO,'pageTitle',6,133)
LexToken(EQUAL,'=',6,143)
LexToken(TEXTO,'TP_PL',6,145)
LexToken(NEWLINE,'\n',6,150)
LexToken(IDENTACAO,8,7,151)
LexToken(TAG,'subtitle',7,159)
LexToken(EQUAL,'=',7,167)
LexToken(SPACE,' ',7,168)
LexToken(TEXTO,'pageTitle',7,169)
```

(a) Output 1

```
LexToken(NEWLINE,'\n',7,178)
LexToken(IDENTACAO,4,8,179)
LexToken(TAG,'body',8,180)
LexToken(NEWLINE,'\n',8,184)
LexToken(IDENTACAO,8,9,185)
LexToken(SPACE,' ',16,408)
LexToken(TEXTO,'Get on it!',16,409)
LexToken(NEWLINE,'\n',16,419)
LexToken(IDENTACAO,12,17,420)
LexToken(TAG,'p',17,432)
LexToken(COMA,'.',17,433)
LexToken(NEWLINE,'\n',17,434)
LexToken(IDENTACAO,16,18,435)
LexToken(TEXTPLAIN,'Pug is a terse and simple templating language with a',18,451)
LexToken(NEWLINE,'\n',18,503)
LexToken(IDENTACAO,16,19,504)
LexToken(TEXTPLAIN,'strong focus ',19,520)
LexToken(NEWLINE,'\n',19,533)
LexToken(IDENTACAO,16,20,534)
LexToken(TEXTPLAIN,'performance and powerful features',20,550)
```

(b) Output 2

2.2 Parser(yacc)

Após termos todos os tokens presentes no código que queremos converter, é necessário organizar esses tokens de forma a construir uma linguagem que faça sentido em HTML, para isso utilizamos uma ferramenta da bibleoteca ply em python com o nome de yacc. Este parser irá analisar o texto recebido e recursivamente (Bottom-up), irá usar os tokens definidos para construir frases que façam sentido, para tal foi necessários definir varias funçoes que se que fossem organizando os tokens consoante necessário.

2.2.1 P html

Este tem a seguinte representação: "html: linhas", que basicamente será o conjunto de todas a linhas que as restantes funções retornaram, tendo essas linhas simplesmente converte as mesmas para html usando a função html(), definida no ficheiro blocks.py.

2.2.2 P linhas

Este tem a seguinte estrutura:
linhas : linhas NEWLINE linha_normal
— linha_normal
— linhas NEWLINE linha_codigo
— linha_codigo
— linhas NEWLINE linha_var
— linha_var
— linhas NEWLINE linha_textplain

Estes são todos os tipos de linhas que podem aparecer em código pug. Assim sendo, se for a primeira linha do input sabemos que temos de criar a estrutura geral onde iremos guardar todos os sub blocos de código HTML, portanto nos casos em que temos apenas uma linha começamos por criar a estrutura blocks que será o p[0] nesse caso, caso seja uma linha normal adicionámos esse bloco á lista de sub blocos da estrutura blocks, caso seja uma linha de código ira na mesma guardar esse bloco na lista de sub blocos mas muda o valor da condição para falso e por fim caso seja uma variável cria na mesma a estrutura blocks mas ao invés de guardar na lista de sub blocos guarda essa variável na lista de variáveis da estrutura geral.
No caso em que já lemos mais linhas antes, significa que a estrutura geral já está criada, assim sendo caso seja um linha de código, vamos verificar se a condição existe nas variáveis da estrutura geral e caso existe colocámos o seu valor na variável bol da estrutura code caso não existe será automaticamente falsa.
No caso em que é uma linha normal vai guardar a estrutura block no p[3]. Após isto, vai percorrer as listas de sub blocos ate encontrar a lista de sub blocos á qual terá de adicionar o bloco que crio seja code ou block. No caso em que recebe uma tinha de texto, vai procurar qual o bloco a que pertence consoante

o nível de indentação atual e irá adicionar esse texto ao campo texto do bloco. Por fim se for uma variável vamos simplesmente adicionar o valor da variável recebida ao nodo do dicionário das váriavéis da estrutura bloco correspondente.

2.2.3 P linha_var

Esta função vai devolver um tuple, em que o primeiro campo será o nome da variável lida, e o segundo será o valor da variável em si.

2.2.4 P linha_codigo

Aqui criamos uma estrutura Code com as informações do codigo, a variável e se é um if ou else.

2.2.5 P linha_normal

Nesta função criamos um tuplo com o Block da linha e com um boolean que indica se o texto do Block é uma variável ou não.

2.2.6 P linha_textplain

O único tratamento que fazemos aqui é acrescer a devida indentação ao texto apesar de não ser obrigatório pois o html não é preciso indentação.

2.2.7 P corpo

Aqui retornamos um tuplo com as seguintes informações (informações da tag,texto da linha, boolean que indica se o texto da linha é uma variável).

2.2.8 P tag

O objetivo desta função é retornar um tuplo com os seguintes informações da tag (nome da tag,id,classes,atributos).

2.2.9 P atributos e P atributos_linha

Esta função simplesmente retorna uma lista com os atributos que estavam dentro de parênteses.

2.3 Estruturas

Cada uma das seguintes estruturas tem uma função html que passa as suas informações para código html guardado em uma string.

2.3.1 Blocks

Nesta estrutura é uma estrutura geral onde temos um array com todos os Code's e Block's, para alem disso é onde guardamos as variáveis.

```
class Blocks:
    def __init__(self):
        self.sub_blocks = []
        self.vars = {}

    def html(self):
        html=''
        for b in self.sub_blocks:
            html+=b.html()
        return html
```

Figura 4: Estrutura Blocks

2.3.2 Block

No Block guardamos todos os outros Code's e Block's que estejam na sua indentação, para alem disso todas as informações referentes ao texto, a tag e aos atributos deste block.

```
class Block:
    def __init__(self,nivel_atual,info):···

    def new_sub_block(self,block):···

    def html(self):
        node = ' '*self.nivel_atual
        if(self.atributos!= ''):
            node += f"<{self.tag} {self.atributos}>"
        else:
            node += f"<{self.tag}>"

        if(self.texto!=''):
            node+=self.texto
        if len(self.sub_blocks)>0:
            code_ty_ant = None
            code_bol_ant = None
            for b in self.sub_blocks:
                if type(b) is Block:
                    node+="\n"+ b.html()
                elif type(b) is Code:
                    if b.bol==None:
                        if code_ty_ant == 'if':
                            b.bol= not code_bol_ant
                        else:
                            print("error else without if\n")
                    code_ty_ant = b.type
                    code_bol_ant = b.bol
                    node+=b.html()
            node+= '\n'+' '*self.nivel_atual + f"</{self.tag}>"
        else:
            node+= f"</{self.tag}>"

        return node
```

Figura 5: Estrutura Block

2.3.3 Code

No Code guardamos todos os outros Code's e Block's que estejam na sua indentação, para alem disso a veracidade desta estrutura pois esta estrutura contem as informações que estão dentro dos if´s e dos else's.

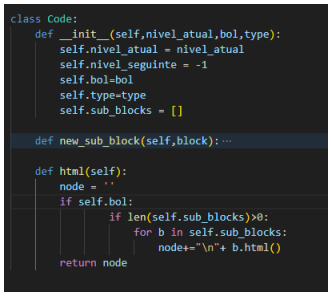

Figura 6: Estrutura Code

2.3.4 Output de exemplo do yacc

Utilizando o exemplo da figura 2, que foi o mesmo utilizado para o exemplo do output do lexer, este foi o resultado que obtemos ao executá-lo no yacc, criando a seguinte estrutura:

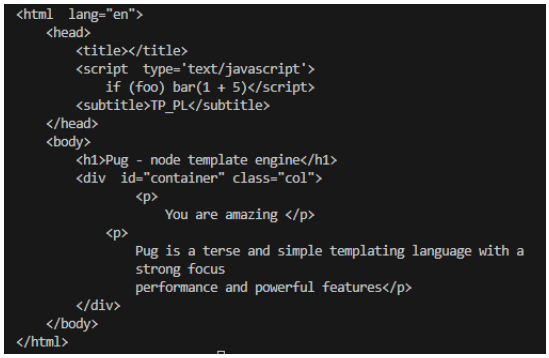

Figura 7: Output yacc

3 Conclusão

Este trabalho mostrou ser desafiador, mas no entanto achámos que conseguimos atender ao que o enunciado nos pedia de forma eficiente. A parte que nos trouxe mais dificuldades foi de facto encontrar uma maneira de resolver o problema de indentação, ou seja, com iríamos saber consoante a indentação a qual tag teria

de corresponder determinado código ou como saberíamos se uma tag faz parte de outra em HTML, apesar disso conseguimos encontrar uma maneira de resolver essa questão. Assim sendo, apesar de não implementarmos todas as funcionalidades possíveis em Pug como por exemplo o for, conseguimos implementar maior parte das funcionalidades presentes em Pug.