Processamento de Linguagens (3^{o} ano de LEI)

Trabalho Prático

Relatório de Desenvolvimento

José Moreira
(a95522)

Santiago Domingues
(a96886)

28 de maio de 2023

Resumo

O presente documento visa retratar o desenvolvimento de um conversor de linguagem Pug para HTML, desenvolvido em Python, no contexto do trabalho prático da Unidade Curricular de Processamento de Linguagens

Índice

Capítulo 1

Introdução

Pug é uma linguagem de marcação simplificada e de fácil leitura, cujo principal objetivo passa por aumentar a produtividade e a eficiência no desenvolvimento de páginas HTML. Devido ao seu formato e sintaxe relativamente mais simples em comparação com a linguagem HTML permite desenvolver estruturas mais complexas de uma página HTML de uma forma mais simples e rápida. Enquanto que HTML é uma linguagem de marcação muito mais complexa para a construção de páginas web.

Durante o processo de compilação do Pug, este é convertido para HTML, esta conversão ocorre através de um compilador específico para a linguagem Pug. O objetivo principal deste trabalho é desenvolver este conversor e, a partir de um excerto em código Pug gerar o respetivo código HTML. Para isso é necessário entender a estrutura e o funcionamento de um conversor deste género, podendo dividir a sua execução em 2 fases: Análise Léxica (Lexer) e Análise Sintática (Parser). A primeira fase é responsável pela conversão de uma sequência de caracteres numa sequência de tokens. Já a segunda fase é traduzida na análise de uma determinada sequência recebida como input e na realização da verificação da sua estrutura gramatical de acordo com a gramática formal previamente definida. A análise sintática garante, a partir de um texto de entrada, a capacidade de gerar uma árvore de derivação através de *parsers* , como LL(1), LR(0), LALR, entre outros.

Capítulo 2

Enunciado

Transcrição do enunciado do trabalho prático referido por este documento

Construa um conversor que aceite um subconjunto da linguagem Pug e gere o HTML correspondente (defina claramente a parte coberta).

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

Listing 2.1: Exemplo da notação Pug

```
<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>
```

Listing 2.2: HTML correspondente

Capítulo 3

Conversor PUG-HTML

3.1 Análise Léxica - Lexer

3.1.1 Tokens

Inicialmente definiu-se um conjunto de tokens fundamentais de modo a tornar possível a leitura de caracteres e a sua conversão nestes tokens, através da execução das respetivas funções.

```
tokens = ('TAG', 'INDENT', 'CLASS', 'ID', 'ATTS',
          'COMMENT', 'TEXT', 'LPAR', 'RPAR',
          'DOT', 'SPACE','S_VAR', 'VAR','D_DOT',
          'S_C_ATTS', 'INTERPOL', 'ASSIGN', 'IF',
          'ELSE', 'DEDENT', 'SAMEDENT','TEXT_INDENT',
          'TEXT_DEDENT', 'TEXT_SAMEDENT', 'UNLESS', 'DOCTYPE'
)
```

Listing 3.1: Tokens

Cada um destes tokens representa um caracter, ou uma sequência de caracteres da linguagem Pug, como por exemplo:

```
'TAG': representa a tag correspondente (html, head, body...)
'INDENT': representa um nível de identação superiores ao anterior =>    html
                                                                            head

'DEDENT': representa um nível de identação inferiores ao anterior =>        script
                                                                        body

'SAMEDENT': representa um nível de identação equivalente ao anterior=>    h1
                                                                          p


'ATTS': representa os atributos que um determinado elemento pode conter adicionalmente,
        estando sempre entre parêntisis (html(lang="en"))
```

```
    'TEXT': representa qualquer texto presente no documento
    'LPAR' e 'RPAR': representam os caracteres de abertura e fecho de parêntisis, respetivamente
    'DOT': representa o ponto '.'
    'SPACE': representa um ou mais espaços
    'S_VAR' e 'VAR': representam o elemento '-var' e o nome da respetiva
                     variavel e a sua atribuição (-var test = true)
```

De notar que o funcionamento dos tokens de identação de texto (TEXT_INDENT, TEXT_DEDENT, TEXT_SAMEDENT) é análogo à generalização dos tokens de identação.

3.1.2 Estados

Na definição de alguns tokens, devido à sua similaridade, ocorreram alguns conflitos em que não era possível captar um determinado caracter no token pretendido visto que este estava a ser captado por outro token, normalmente, mais geral que o anterior. Por exemplo, qualquer texto que viesse após o token `TAG` seria assimilado como `TAG` e não como `TEXT`, que seria o pretendido. Deste modo, para resolver este problema optou-se pela criação de estados. Estes estados permitem tratar de situações específicas durante a análise léxica

No lexer desenvolvido destacamos os seguintes estados:

```
    states = (
    ('content', 'exclusive'),
    ('atts', 'exclusive'),
    ('text', 'exclusive'),
    ('multiline', 'exclusive'),
    ('var', 'exclusive'),
    ('assignment', 'exclusive')
)
```

Listing 3.2: Tokens

Para ilustrar o funcionamento dos estados no nosso programa podemos utilizar como exemplo o estado *content*. Sempre que o analisador léxico detetar a entrada de uma `TAG`, `CLASS` ou `ID` entrará no estado *content*, sendo que para sair deste estado o lexer necessita de detetar uma nova linha (caracter `\n`). O comportamento é semelhante para os restantes estados e permite, em situações mais específicas, detetar atributos e variáveis de forma precisa e eficaz, recorrendo aos estados *atts* e *var*, respetivamente. O primeiro estado inicia-se pela deteção de um parêntisis esquerdo e termina quando detetar um parêntisis direito, ou seja, vai ser capaz de captar todos os atributos pertencentes a uma tag ou classe, por exemplo.

```
    LexToken(LPAR,'(',1,4)
    LexToken(ATTS,'lang="en"',1,5)
    LexToken(RPAR,')',1,14)
```

Neste exemplo, este conjunto de tokens iniciam o estado *atts*, capturam o atributo pretendido e por fim saem do estado *atts*

Sumariamente, os outros estados definidos podem ser analisados na seguinte tabela:

Estado	Descrição	Exemplo Pug	Lexer Output do Estado
atts	Atributos de um elemento. Inicia-se pela identificação de um parêntisis esquerdo	html(lang="en")	LexToken(LPAR,'(',1,4) LexToken(ATTS,'lang="en"',1,5) LexToken(RPAR,')',1,14)
var	Variáveis usadas para armazenar valores. Inicia-se pela identificação do símbolo prédefinido '-var' ou pela identificação de um 'if'	- var test = true	LexToken(S_VAR,'- var ',1,20) LexToken(VAR,'test = true',1,26)
text	Texto no documento, incluindo comentários. Inicia-se pela identificação de um ou mais espaços ou do símbolo de comentário (//)	p Get on it!	LexToken(SPACE,' ',1,311) LexToken(TEXT,'Get on it!',1,312)
multiline	Todo o texto que se encontre numa identação superior ao objeto referente. Inicia-se pela identificação de um ponto ('.')	p. Pug is a terse and simple templating language with a strong focus on performance and powerful features	LexToken(DOT,'.',1,336) LexToken(TEXT_INDENT,'\n ',1,337) LexToken(TEXT,'Pug is a terse and simple templating language with a',1,354) LexToken(TEXT_SAMEDENT,'\n ',1,406) LexToken(TEXT,'strong focus on performance and powerful features',1,423)
assignment	Inicia-se pela captura do sinal de atribuição ('=')	title= pageTitle	LexToken(ASSIGN,'=',1,60) LexToken(VAR,'pageTitle',1,62)

Tabela 3.1: Tabela de Análise de Estados

3.1.3 Definições

Para ser possível o funcionamento do Lexer, incluindo a correta captura dos caracteres e a conjugação de estados, foi fundamental criar um conjunto de funções que através de expressões regulares tivessem a capacidade de captar atomicamente os caracteres pretendidos e manipular os estados definidos de forma a obter uma análise léxica completa e eficiente. As definições são o cerne do analisador léxico, pois são responsáveis pela captura dos caracteres necessários para a respetiva conversão em tokens realizada pelo Lexer, ou seja, são capazes de identificar e extrair do texto pequenas partes que correspondam aos padrões por elas definidos.

As funções definidas têm também a capacidade de conjugar os estados e os manusear de forma a prevenir que um caracter que se pretenda captar sob uma determinada regra seja captado por outra regra. Para cada token mencionado na Listing 3.1, foram definidas funções que captassem o padrão correspondente, para exemplificar:

- Definição de TAG

```
def t_TAG(t):
r'[a-z0-9]+\/?'
t.lexer.begin('content')
return t
```

- Definição de COMMENT

```
def t_COMMENT(t):
r'\/\/\-?'
t.lexer.begin('text')
return t
```

- Definição de ATTS

```
def t_atts_ATTS(t):
r'[^\)]+'
if '\n' in t.value:
    t.value = t.value.replace(" ","").replace("\n"," ").replace("'",'"').strip()
else:
    if t.value[0] == '{':
        t.value = t.value.replace('{"','')
                         .replace('{"','')
                            .replace('"}','"')
                                .replace("'}",'"')

        t.value = t.value.replace('": \'','="')
                            .replace('": "','="')
                                .replace("': \"",'="')
                                    .replace("': \'",'="')
```

```
        t.value = t.value.replace("', '",'" ')
                         .replace("', \"",'" ')
                            .replace('", \'','" ')
                                .replace('", "','" ')
    return t
```

Na tabela 4.1 podem ser analisados exemplos de definições do nosso programa, bem como a respetiva expressão regular, um exemplo de um excerto de um texto em linguagem Pug e o padrão produzido pelo Lexer em concordância com a definição.

Definição	Expressão Regular	Exemplo Pug	Output
TAG	`r'[a-z0-9]+\/?'`	html(lang="en")	LexToken(TAG,'html',1,0)
IF	`r'if'`	if youAreUsing-Pug	LexToken(IF,'if',1,225)
INTERPOL	`r'\#\{[^\{]+\}'`	p You are amazing #{helloPug}	LexToken(INTERPOL,'#{helloPug}',1,277)
S_VAR	`r'\-[ \s]*var[ \s]+'`	-var title=true	LexToken($S_VAR,' -var', 1, 33$)
VAR	`r'[^\n]+'`	if youAreUsing-Pug	LexToken(VAR,'youAreUsingPug',1,254)
ID	`r'\#[a-z][a-zA-Z0-9\-]*'`	h1 Pug - node template engine container.col	LexToken(ID,'container',1,198)
CLASS	`r'\.[a-z][a-zA-Z0-9\-]*'`	h1 Pug - node template engine container.col	LexToken(CLASS,'.col',1,208)
COMMENT	`r'\/\/\-?'`	//pugUtilization	LexToken(COMMENT,'//',1,29)

Tabela 3.2: Tabela de Definições

3.1.4 Código Lexer

As restantes funções bem como a plenitude do código referente ao Lexer pode ser visualizado no Apêndice C.

Capítulo 4

Análise Sintática - Parser

Em computação, a análise sintática é uma fase de execução de um compilador cujo principal objetivo passa por averiguar se a sequência de tokens gerada pela análise léxica (Lexer) está de acordo com a gramática definida e construir árvore de derivação correspondente.

A análise sintática recebe uma sequência de tokens, que correspondem a caracteres convertidos pelo Lexer, e analisa sintaticamente esta sequência, isto é, através de um conjunto de regras gramaticais definidas verifica se os tokens estão em concordância com as regras e constrói a árvore de derivação (*Parse Tree*) que traduz a estrutura hierárquica do programa.

No mundo da computação existem diversos tipos de parsers, dependendo de como estes analisam as regras gramaticais e constroem as árvores de derivação, podendo ser divididos em dois grandes grupos:

1. Parser Descendente (*Top Down Parser*)
    1. LL(1);
    2. Recursive Descent Parser;
    3. LL(k)

2. Parser Ascendente (*Bottom Up Parser*)
    1. LR(0)
    2. SLR(1)
    3. LALR(1)

As principais diferença entre os *Top Down* e os *Bottom Up* parsers residem na orientação em que é construída a árvore de derivação e a ordem pela qual são aplicadas as regras gramaticais.

Top Down Parser	Bottom Up Parser
Estratégia de parsing que se inicia no nível mais alto da árvore, ou seja, na raíz e aplica as regras numa direção descendente	Estratégia de parsing que se inicia no nível mais baixo da árvore e aplica as regras numa direção ascendente
Tenta encontrar as derivações mais à esquerda para um determinado texto de entrada	Tenta reduzir os dados de entrada ao símbolo inicial de uma gramática
Utiliza técnicas de *Left Most Derivation*	Utiliza técnicas de *Right Most Derivation*
A principal decisão passa por selecionar que regra de produção (gramatical) usar para construir o texto de entrada	A principal decisão passa por selecionar que regra de produção utilizar para reduzir o texto de entrada até ao símbolo inicial da gramática

Tabela 4.1: Top Down e Bottom Up Parsers

Em específico, para o desenvolvimento do nosso projeto foi utilizado um parser ascendente, em concreto, LALR(1).

A sigla LALR significa "Look-Ahead Left-to-Right"e o número 1 indica que o parser usa apenas 1 token de *lookahead* para tomar decisões durante a análise sintática. O parser LALR(1) surge de uma variação do parser LR(1) que usa uma tabela de parsing LALR para realizar a sua análise. Esta tabela, em comum com a maioria das tabelas de parsers Bottom Up contém as transições dos estados e as ações a serem seguidas com base no estado atual e no valor do *lookahead*. Uma das principais diferenças para os outros parsers da mesma categoria reside no facto de o LALR(1) agrupar estados com conjuntos de itens semelhantes, mesmo que tenham lookaheads diferentes, o que permite uma redução na tabela de análise, logo apresenta uma grande vantagem em relação a outros tipos de parsers LR, visto que é capaz de lidar com gramáticas mais expressivas sem aumentar significativamente a complexidade da tabela de análise.

4.1 Regras Gramaticais

As regras gramaticais são a base fundamental para a definição da estrutura gramatical de uma linguagem, pois são responsáveis por conjugar os tokens de forma a criar as expressões pretendidas e a guiar o analisador de modo a ser possível a análise correta e a construição da respetiva árvore de derivação da estrutura de entrada.

De forma canónica, uma regra gramatical é construida a partir de tokens não-terminais, terminais e produções. No nosso projeto foram definidos os seguintes tokens terminais e não-terminais:

1. Terminais

   - ASSIGN
   - ATTS
   - CLASS
   - COMMENT
   - DEDENT
   - DOT
   - D_DOT
   - ELSE
   - ID
   - IF
   - INDENT
   - INTERPOL
   - LPAR
   - RPAR
   - SAMEDENT
   - SPACE
   - S_C_ATTS
   - S_VAR
   - TAG
   - TEXT
   - TEXT_DEDENT
   - TEXT_INDENT
   - TEXT_SAMEDENT
   - VAR
   - UNLESS

2. Não-Terminais

   - attLine
   - atts
   - classList
   - classListLine
   - components
   - content
   - elem
   - element
   - html
   - idLine
   - ifClause
   - keyLine
   - stuff
   - textline

A regra inicial do nosso projeto é definida por:

```
def p_html(p):
    '''
    html : content
    '''
    global tag_stack
    global space
    p[0] = ''
    if len(tag_stack) > 0:
        p[0] += p[1] + f'</{tag_stack.pop()[0]}>'
        while len(tag_stack) > 0:
            last_tag = tag_stack.pop()
            spaces = last_tag[1] * space
            p[0] += '\n' + spaces + f'</{last_tag[0]}>'
    else:
        p[0] += p[1]
```

Sendo que o não-terminal 'content' pode ser expandido em:

```
def p_content(p):
    '''
    content : element content
            | element
    '''
    p[0] = p[1]
    if len(p) == 3:
        p[0] += p[2]
```

Por sua vez 'element' representa todo o conjunto de elementos que podem ser identificados na linguagem Pug e convertidos em HTML. Visto ser um não-terminal, pode ser expandido num conjunto de objetos, não-terminais e terminais, que recursivamente completam o conjunto de regras gramaticais que definem o parser do projeto.

Doctype

Permite realizar a conversão de elementos *Doctype* caso tenham o valor de 'html' ou 'xml'

```
def p_element_doctype(p):
    '''
    element : DOCTYPE
    '''
    if 'html' in p[1]:
        p[0] = '<!DOCTYPE html>'
    else:
        p[0] = '<?xml version="1.0" encoding="utf-8" ?>'
```

Condicionais

Definição da estrutura condicional, com os elementos:

- *'if'*

```
def p_elem_if(p):
    '''
    elem : ifClause
    '''
    p[0] = p[1]



def p_ifClause_if(p):
    '''
    ifClause : IF VAR
             | ELSE
    '''
    global state
    global vars
    global tag_state
    p[0] = ''
    if len(p) == 3:
        if p[2] in vars and vars[p[2]] != 'false':
            state = 'if'
        else:
            state = 'no'
        tag_state = 'if'
    else:
        if state == 'if':
            state = 'no'
        else:
            state = 'else'
        tag_state = 'else'
```

- *'unless'*

```
def p_ifClause_unless(p):
    '''
    ifClause : UNLESS VAR
    '''
    global state
    global vars
    global tag_state
    p[0] = ''
    if p[2] not in vars or (p[2] in vars and vars[p[2]] == 'false'):
        state = 'if'
    else:
        state = 'no'
    tag_state = 'if'
```

Ambos seguidos de um objeto terminal 'VAR', sendo representado neste contexto por uma condição, a ser analisado pela estrutura condicional correspondente

Variáveis

Permite a criação e armazenamento de variáveis e do valor a elas atribuido para futura utilização

```
def p_elem_var(p):
    '''
    elem : S_VAR VAR
    '''
    global var_stack
    global comment
    p[0] = ''
    split_var = p[2].split('=')
    var_stack.append((split_var[0].strip(),split_var[1].strip()))
    comment = 1
```

Interpolação

Permite a introdução de uma determinada variável numa parte do texto

```
def p_textLine_interpol(p):
    '''
    textLine : INTERPOL
             | INTERPOL TEXT
             | TEXT INTERPOL TEXT
    '''
    p[0] = ''
    if len(p) < 4:
        var = p[1][2:-1]
        if var in vars:
            add = vars[var].replace('"','')
            p[0] += f'{add}'
        if len(p) == 3:
            p[0] += p[2]
    else:
        var = p[2][2:-1]
```

```
        if var in vars:
            add = vars[var].replace('"','')
            p[0] += f'{p[1]}{add}{p[3]}'
        else:
            p[0] += f'{p[1]}{p[3]}'
```

Comentários

Permite documentar a linha de código em que o símbolo de comentário está presente. Podendo ser identificados 2 tipos de comentários:

1. Comentários Pug
   - Comentários apenas para a linguagem Pug, não sendo apresentados no resultado da conversão em HTML, são iniciados pelo símbolo `'//-'`
2. Comentários
   - Comentários que serão apresentados no resultado final da conversão em HTML, são iniciados pelo símbolo `'//'`

```
def p_elem_comment(p):
    '''
    elem : COMMENT textLine
    '''
    global state
    global indent_level
    if len(p[1]) == 3:
        p[0] = ''
    else:
        p[0] = f'<!--{p[2]}-->'
```

Atributos

Permite a inserção de atributos, fornecendo informação extra sobre um determinado elemento. Atributos podem estar presentes numa tag, numa classe ou num id, sob a forma:

```
> tag|id|class(atributo)
```

São geralmente representados por um nome e um valor separados pelo caracter '='. No entanto, existem ainda variações como por exemplo `&attributes` (*and attributes*)e atributos multilinha.

```
def p_atts(p):
    '''
    atts : LPAR ATTS RPAR
         | S_C_ATTS LPAR ATTS RPAR
    '''
    if len(p) == 4:
```

```
        p[0] = p[2]
    else:
        p[0] = p[3]
```

Tags

Um dos principais elementos da linguagem Pug e da linguagem HTML, fundamentais para a criação de elementos HTML. Podem ser convertidos valores simples de tags como `head` ou `body`, bem como *nested tags* (`a: img`) ou então *self-closing tags*, que como o nome indica na conversão para HTML o seu fecho é realizado de forma autónoma.

De forma a ser capaz de guardar as tags identificadas, para, aquando da conversão para HTML, realizar o seu fecho, foi necessário definir uma variável `tag_stack` e uma `second_stack`, esta com o objetivo de guardar as tags e a sua indentação e posteriormente adicionar estes elementos à `tag_stack`.

```
def p_keyLine_tag(p):
    '''
    keyLine : TAG
            | TAG classListLine
            | TAG idLine
    '''
    global second_stack
    p[0] = ''
    if p[1][-1] != '/':
        p[0] = f'{p[1]}'
        if len(p) == 3:
            p[0] += f' {p[2]}'
    else:
        p[1] = p[1].replace('/','')
    if state != 'no':
        second_stack.append(p[1])
```

Classes

Elemento utilizado para identificar um determinado nome para ser mais facilmente manuseado no contexto CSS. São identificadas pelo símbolo `'.'`, podendo ser ou não precedido de um elemento `ID`.

```
def p_keyLine_class_id(p):
    '''
    keyLine : classListLine
            | idLine
    '''
    global second_stack
    p[0] = f'div {p[1]}'
    if state != 'no':
        second_stack.append('div')


def p_classListLine(p):
    '''
```

```
    classListLine : classList
                  | classList ID
    '''
    if len(p) == 2:
        p[0] = f'class="{p[1]}"'
    else:
        p[0] = f'class="{p[1]}" id="{p[2][1:]}"'


def p_classList(p):
    '''
    classList : CLASS classList
              | CLASS
    '''
    if len(p) == 3:
        p[0] = p[1][1:] + ' ' + p[2]
    else:
        p[0] = p[1][1:]
```

ID

Elemento utilizado para atribuir uma identificação específica a um elemento HTML. É sinalizado pelo símbolo # e pode ser procedido de uma classe.

```
def p_idLine(p):
    '''
    idLine : ID
           | ID classList
    '''
    if len(p) == 2:
        p[0] = f'id="{p[1][1:]}"'
    else:
        p[0] = f'class="{p[2]}" id="{p[1][1:]}"'
```

Texto

Excertos de texto presentes nos dados de entrada. Sendo identificados apenas numa linha ou então em várias linhas, se antecedidos de um ponto.

```
def p_elem_text(p):
    '''
    elem : textLine
    '''
    p[0] = p[1]

def p_textLine_text(p):
    '''
    textLine : TEXT
             | TEXT INTERPOL
    '''
    p[0] = p[1]
    if len(p) == 3:
        var = p[2][2:-1]
```

```
        if var in vars:
            p[0] += vars[var].replace('"','')
```

Indentação

Uma das principais características das linguagens Pug e HTML é a sua indentação, isto é, a maneira como estão distribuidos os espaços e as quebras de linha em concordância com os elementos.

A indentação pode ser classificada de acordo com nível em que se encontra em relação ao elemento anterior. Caso um elemento possua uma indentação superior ao anterior, então tem o valor de `INDENT`, caso seja inferior `DEDENT` e caso seja igual `SAMEDENT`, como específicado no Capítulo 3

De forma a tornar mais simples a análise de valores de identação, foram criados dois grupos:

1. Identação de Texto

(a)
```
def p_element_text_id(p):
    '''
    element : TEXT_INDENT elem
    '''
    p[0] = ''
    global state
    if state != 'no':
        if state in ['if','else']:
            p[1] = p[1].replace(' ','',4)
        p[0] = p[1] + p[2]



def p_element_text_dd(p):
    '''
    element : TEXT_DEDENT elem
    '''
    p[0] = ''
    global state
    if state != 'no':
        if state in ['if','else']:
            p[1] = p[1].replace(' ','',4)
        p[0] = p[1] + p[2]



def p_element_text_sd(p):
    '''
    element : TEXT_SAMEDENT elem
    '''
    p[0] = ''
    global state
    if state != 'no':
        if state in ['if','else']:
            p[1] = p[1].replace(' ','',4)
        p[0] = p[1] + p[2]
```

2. Identação para a restante estrutura

(a)

```
def p_element_normal_id(p):
    '''
    element : elem
            | INDENT elem
    '''
    global indent_level
    global min_indent
    global state
    global state_indent
    global tag_state
    global var_stack
    p[0] = ''

    var = ('','')
    if len(var_stack) > 0:
        var = var_stack.pop()

    if p[1] != '':
        if state != 'no':
            if var[0] != '':
                vars[var[0]] = var[1]
            if len(p) == 2:
                if p[1] == '<>':
                    p[1] = ''
                p[0] += p[1]
                if min_indent != 0:
                    elem = p[1].split('>')[0][1:]
                    p[0] += f'</{elem}>'
                    if len(second_stack) > 0:
                        second_stack.pop()
            else:
                if p[2] != '':
                    if state in ['if','else']:
                        p[1] = p[1].replace(' ','',4)
                    if p[2] == '<>':
                        p[2] = ''
                    p[0] += p[1] + p[2]
                    if min_indent == 0:
                        min_indent = len(p[1]) - 1

                    indent_level = len(p[1]) - 1


            if len(second_stack) > 0:
                tag_stack.append((second_stack.pop(),indent_level))

    if state in ['if','no'] and state_indent == -1:
        state_indent = indent_level + min_indent
    if tag_state == 'if':
        tag_state = ''



def p_element_normal_dd(p):
    '''
    element : DEDENT elem
    '''
```

```
    global indent_level
    global min_indent
    global state
    global state_indent
    global tag_state
    p[0] = ''

    var = ('','')
    if len(var_stack) > 0:
        var = var_stack.pop()

    indent = len(p[1]) - 1
    if state in ['if','else']:
        indent -= min_indent

    if state in ['no','if','else'] and indent <= state_indent:
        if tag_state == 'else' or tag_state == 'if':
            tag_state = ''
        else:
            state = 'normal'
            state_indent = -1
            indent += min_indent


    if state != 'no':
        if var[0] != '':
            vars[var[0]] = var[1]
        if p[2] != '' and len(tag_stack) > 0:
            if state in ['if','else']:
                p[1] = p[1].replace(' ','',4)
            if p[2] == '<>':
                p[2] = ''

            cl_tag = tag_stack.pop()
            p[0] += f'</{cl_tag[0]}>'
            if len(tag_stack) > 0:
                while tag_stack[-1][1] >= indent:
                    last_tag = tag_stack.pop()
                    spaces = last_tag[1] * ' '
                    p[0] += '\n' + spaces + f'</{last_tag[0]}>'
                    if len(tag_stack) == 0:
                        break

            p[0] += p[1] + p[2]


        indent_level = len(p[1]) - 1
        if len(second_stack) > 0:
            tag_stack.append((second_stack.pop(),indent_level))


    if state == 'else' and state_indent == -1:
        state_indent = indent_level



def p_element_normal_sd(p):
    '''
    element : SAMEDENT elem
```

```
    '''
    global indent_level
    global state
    global vars
    global var_stack
    global comment
    p[0] = ''

    var = ('','')
    if len(var_stack) > 0:
        var = var_stack.pop()

    if state != 'no':
        if p[2] != '':
            if var[0] != '':
                vars[var[0]] = var[1]
            if state in ['if','else']:
                p[1] = p[1].replace(' ','',4)
            if p[2] == '<>':
                p[2] = ''

            if len(tag_stack) > 0 and '<!' not in p[2]:
                if comment == 0:
                    p[0] += f'</{tag_stack.pop()[0]}>'
                elif comment == 1:
                    comment = 2
                elif comment == 2:
                    comment = 0
                    p[0] += f'</{tag_stack.pop()[0]}>'

            p[0] += p[1] + p[2]


        indent_level = len(p[1]) - 1
        if len(second_stack) > 0:
            tag_stack.append((second_stack.pop(),indent_level))
```

De modo a generalizar o controlo da conversão, por exemplo, a execução de estruturas condicionais, adição de variáveis, entre outras operações foi criada uma variável que representa o estado, podendo ter os valores de `'normal'` ou `'no'`, caso se queira proceder à execução do processo ou não, respetivamente.

4.1.1 Código Parser

O código completo do parser, incluindo todas as regras gramaticais podem ser visualizadas no Apêndice D

4.2 Gramática

Gramática final, com todas as regras de produção definidas:

```
Rule 0     S' -> html
Rule 1     html -> content
Rule 2     content -> element content
Rule 3     content -> element
Rule 4     element -> DOCTYPE
Rule 5     element -> elem
Rule 6     element -> INDENT elem
Rule 7     element -> DEDENT elem
Rule 8     element -> SAMEDENT elem
Rule 9     element -> TEXT_INDENT elem
Rule 10    element -> TEXT_DEDENT elem
Rule 11    element -> TEXT_SAMEDENT elem
Rule 12    elem -> ifClause
Rule 13    ifClause -> IF VAR
Rule 14    ifClause -> ELSE
Rule 15    ifClause -> UNLESS VAR
Rule 16    elem -> stuff components
Rule 17    stuff -> keyLine
Rule 18    stuff -> keyLine attLine
Rule 19    keyLine -> TAG
Rule 20    keyLine -> TAG classListLine
Rule 21    keyLine -> TAG idLine
Rule 22    keyLine -> classListLine
Rule 23    keyLine -> idLine
Rule 24    classListLine -> classList
Rule 25    classListLine -> classList ID
Rule 26    classList -> CLASS classList
Rule 27    classList -> CLASS
Rule 28    idLine -> ID
Rule 29    idLine -> ID classList
Rule 30    attLine -> atts attLine
Rule 31    attLine -> atts
Rule 32    atts -> LPAR ATTS RPAR
Rule 33    atts -> S_C_ATTS LPAR ATTS RPAR
Rule 34    components -> D_DOT
Rule 35    components -> SPACE textLine
Rule 36    components -> <empty>
Rule 37    components -> DOT
Rule 38    components -> ASSIGN VAR
Rule 39    elem -> textLine
Rule 40    textLine -> INTERPOL
```

```
Rule 41    textLine -> INTERPOL TEXT
Rule 42    textLine -> TEXT INTERPOL TEXT
Rule 43    textLine -> TEXT
Rule 44    textLine -> TEXT INTERPOL
Rule 45    elem -> COMMENT textLine
Rule 46    elem -> S_VAR VAR
```

Capítulo 5

Notação Coberta

De forma resumida, foram cobertos os seguintes elementos da linguagem Pug, de maneira a tornar o nosso programa o mais completo e capaz de traduzir excertos de linguagem Pug em HTML:

1. Atributos
    1.1. Inline
    2.2. Multiline
    3.3. De classe
    4.4. &atributos (*and attributes*)
2. Comentários
    2.1. Comentários
    2.2. Comentários Pug
3. Condicionais
    3.1. if
    3.2. unless
4. Doctype
    4.1. Extensão html
    4.2. Extensão xml
5. Interpolação
    5.1. Texto
6. Texto Simples
    6.1. Inline
    6.2. Multiline
7. Classes

8. ID
9. Tags
    9.1. Simple Tags
    9.2. *Self-Closing*
    9.3. *Nested tags*

Capítulo 6

Main

O ficheiro `main.py` é responsável pelo controlo dos dados de entrada e de saída do conversor do nosso projeto. É capaz de ler um ficheiro Pug e escrever o resultado num ficheiro HTML. Para ser executado apenas é necessário o comando:

```
> python3 main.py
```

Aparecerá um mensagem para escrever o nome do ficheiro e basta introduzir [nome].pug e estará completo o processo.

Capítulo 7

Conclusão

O desenvolvimento deste projeto mostrou-se um teste de exigência elevada às nossas capacidades de decisão e controlo da matéria leccionada na Unidade Curricular de Processamento de Linguagens, permitindo aprofundar de maneira exímia o conhecimento dos processos de um compilador, especificamente a análise léxica e sintática, bem como todo o processo envolvente, desde a definição de tokens até à produção de regras gramaticais. Pensamos ter atingido os objetivos requiridos na definição do projeto e portanto estamos satisfeitos com o resultado produzido.

De notar que se encontram no Apêndice A exemplos de conversão de excertos de documentos Pug em HTML, usando o conversor desenvolvido neste projeto.

Apêndice A

Exemplos de Conversão

A.1 Dataset 1

Código Pug:

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

Código HTML correspondente:

```
<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>
```

A.2 Dataset 2

Código Pug:

```
doctype html
html(lang="pt")
    -var var1 = 'complex'
    head
        title= pageTitle
        script(type='text/javascript')
    body
        h1 Pug - node template engine
        #container.col
            if youAreUsingPug
                p You are amazing
            else
                p Get on it!
            p.
                Pug is a terse and not #{var1} templating language with a
                strong focus on performance and powerful features
```

Código HTML correspondente:

```
<!DOCTYPE html>
<html lang="pt">
    <head>
        <title></title>
        <script type='text/javascript'></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 not complex templating language with a
                strong focus on performance and powerful features</p>
        </div>
    </body>
</html>
```

A.3 Dataset 3

Código Pug:

```
doctype xml
html(lang="en")
    head
        title= pageTitle
        script(
            type='text/javascript'
            name='pug'
        )
    body
        h1 Pug - node template engine
        p This is a test
```

Código HTML correspondente:

```
<?xml version="1.0" encoding="utf-8" ?>
<html lang="en">
    <head>
        <title></title>
        <script type="text/javascript" name="pug"></script>
    </head>
    <body>
        <h1>Pug - node template engine</h1>
        <p>This is a test</p>
    </body>
</html>
```

A.4 Dataset 4

Código Pug:

```
html(lang="en")
    // This is supposed to show up
    //- This is not supposed to show up
    head
        title
        script(type='text/javascript')
    body
        h1 Pug - node template engine
        p This is an experiment
```

Código HTML correspondente:

```
<html lang="en">
    <!-- This is supposed to show up-->
    <head>
        <title></title>
        <script type='text/javascript'></script>
    </head>
    <body>
        <h1>Pug - node template engine</h1>
        <p>This is an experiment</p>
    </body>
</html>
```

A.5 Dataset 5

Código Pug:

```
html(lang="en")
    head
        title= pageTitle
        script(type='text/javascript')
    body
        h1 Pug - node template engine
        #container.col
            unless youAreUsingPug
                p This is HTML
            p.
                Rather you are using HTML or Pug this will appear
```

Código HTML correspondente:

```
<html lang="en">
    <head>
        <title></title>
        <script type='text/javascript'></script>
    </head>
    <body>
        <h1>Pug - node template engine</h1>
        <div class="col" id="container">
            <p>This is HTML</p>
            <p>
                Rather you are using HTML or Pug this will appear</p>
        </div>
    </body>
</html>
```

A.6 Dataset 6

Código Pug:

```
html(lang="en")
    head
        title: script
        img/
    body
        #foo(data-bar="foo")&attributes({'data-foo': 'bar'})
```

Código HTML correspondente:

```
<html lang="en">
    <head>
        <title><script></script></title>
        </img>
    </head>
    <body>
        <div id="foo" data-bar="foo" data-foo="bar"></div>
    </body>
</html>
```

Apêndice B

Main

```
from _parser import parser

def main():
    file_name = input('Type the file name: ')
    with open('../pug_files/' + file_name) as file:
        content = file.read()

    html_file = file_name.replace('.pug','.html')
    with open('../converted_html_files/' + html_file,'w') as result:
        result.write(parser.parse(content))

if __name__ == '__main__':
    main()
```

Apêndice C

Lexer

```
import ply.lex as lex


tokens = ('TAG', 'INDENT', 'CLASS', 'ID', 'ATTS',
          'COMMENT', 'TEXT', 'LPAR', 'RPAR', 'DOT', 'SPACE','S_VAR', 'VAR',
          'D_DOT', 'S_C_ATTS', 'INTERPOL', 'ASSIGN', 'IF', 'ELSE', 'DEDENT', 'SAMEDENT',
          'TEXT_INDENT', 'TEXT_DEDENT', 'TEXT_SAMEDENT', 'UNLESS', 'DOCTYPE')


states = (
    ('content', 'exclusive'),
    ('atts', 'exclusive'),
    ('text', 'exclusive'),
    ('multiline', 'exclusive'),
    ('var', 'exclusive'),
    ('assignment', 'exclusive')
)

indent_level = 0
text_indent = 0


def t_DOCTYPE(t):
    r'doctype[ \s]+ (html|xml)'
    return t


def t_IF(t):
    r'if'
    t.lexer.begin('var')
    return t


def t_ELSE(t):
    r'else'
    return t


def t_UNLESS(t):
    r'unless'
    t.lexer.begin('var')
```

```
    return t


def t_TAG(t):
    r'[a-z0-9]+\/?'
    t.lexer.begin('content')
    return t


def t_content_ASSIGN(t):
    r'\='
    t.lexer.begin('assignment')
    return t


def t_var_assignment_VAR(t):
    r'[^\n]+'
    t.value = t.value.replace("'",'"')
    t.lexer.begin('INITIAL')
    return t


def t_text_multiline_INTERPOL(t):
    r'\#\{[^\{]+\}'
    return t


def t_content_S_C_ATTS(t):
    r'&attributes'
    return t


def t_content_LPAR(t):
    r'\('
    t.lexer.begin('atts')
    return t


def t_atts_RPAR(t):
    r'\)'
    t.lexer.begin('content')
    return t


def t_atts_ATTS(t):
    r'[^\)]+'
    if '\n' in t.value:
        t.value = t.value.replace(" ","").replace("\n"," ").replace("'",'"').strip()
    else:
        if t.value[0] == '{':
            t.value = t.value.replace('{"','').replace("{'",'').replace('"}','"').replace(
                "'}",'"')
            t.value = t.value.replace('": \'','="').replace('": "','="').replace("': \"",'
                ="').replace("': \'",'="')
            t.value = t.value.replace("', '",'" ').replace("', \"",'" ').replace('", \'','
                " ').replace('", "','" ')
    return t
```

```
def t_INITIAL_content_text_newline(t):
    r'\n[ \t]*'
    t.lexer.begin('INITIAL')
    global indent_level
    if len(t.value) - 1 > indent_level:
        t.type = 'INDENT'
    elif len(t.value) - 1 < indent_level:
        t.type = 'DEDENT'
    else:
        t.type = 'SAMEDENT'

    indent_level = len(t.value) - 1
    return t


def t_multiline_newline(t):
    r'\n[ \t]*'
    global indent_level
    global text_indent
    if len(t.value) - 1 < indent_level:
        if len(t.value) - 1 <= text_indent:
            t.lexer.begin('INITIAL')
            t.type = 'DEDENT'
        else:
            t.type = 'TEXT_DEDENT'
    elif len(t.value) - 1 > indent_level:
        t.type = 'TEXT_INDENT'
    else:
        t.type = 'TEXT_SAMEDENT'
    indent_level = len(t.value) - 1
    return t


def t_S_VAR(t):
    r'\-[ \s]*var[ \s]+'
    t.lexer.begin('var')
    return t


def t_INITIAL_content_ID(t):
    r'\#[a-z][a-zA-Z0-9\-]*'
    t.lexer.begin('content')
    return t


def t_INITIAL_content_CLASS(t):
    r'\.[a-z][a-zA-Z0-9\-]*'
    t.lexer.begin('content')
    return t


def t_content_D_DOT(t):
    r'\:'
    t.lexer.begin('INITIAL')
    return t


def t_content_DOT(t):
    r'\.'
```

```
    global indent_level
    global text_indent
    text_indent = indent_level
    t.lexer.begin('multiline')
    return t


def t_content_SPACE(t):
    r'[ \s]'
    t.lexer.begin('text')
    return t


def t_COMMENT(t):
    r'\/\/\-?'
    t.lexer.begin('text')
    return t


def t_text_multiline_TEXT(t):
    r'[^\n\#]+'
    return t


t_INITIAL_assignment_var_ignore = ' '

t_content_atts_text_multiline_ignore = ''

def t_ANY_error(t):
    print(f"Invalid token: {t.value[0]}")
    t.lexer.skip(1)

lexer = lex.lex()
```

Apêndice D

Parser

```
import ply.yacc as yacc
from _lexer import tokens

tag_stack = []
second_stack = []
indent_level = 0
min_indent = 0
vars = {}
state = 'normal'
state_indent = -1
tag_state = ''
var_stack = []
space = ' '
comment = 0


def p_html(p):
    '''
    html : content
    '''
    global tag_stack
    global space
    p[0] = ''
    if len(tag_stack) > 0:
        p[0] += p[1] + f'</{tag_stack.pop()[0]}>'
        while len(tag_stack) > 0:
            last_tag = tag_stack.pop()
            spaces = last_tag[1] * space
            p[0] += '\n' + spaces + f'</{last_tag[0]}>'
    else:
        p[0] += p[1]


def p_content(p):
    '''
    content : element content
            | element
    '''
    p[0] = p[1]
```

```
    if len(p) == 3:
        p[0] += p[2]


def p_element_doctype(p):
    '''
    element : DOCTYPE
    '''
    if 'html' in p[1]:
        p[0] = '<!DOCTYPE html>'
    else:
        p[0] = '<?xml version="1.0" encoding="utf-8" ?>'


def p_element_normal_id(p):
    '''
    element : elem
            | INDENT elem
    '''
    global indent_level
    global min_indent
    global state
    global state_indent
    global tag_state
    global var_stack
    global space
    p[0] = ''

    var = ('','')
    if len(var_stack) > 0:
        var = var_stack.pop()

    if p[1] != '':
        if state != 'no':
            if var[0] != '':
                vars[var[0]] = var[1]
            if len(p) == 2:
                if p[1] == '<>':
                    p[1] = ''
                p[0] += p[1]
                if min_indent != 0:
                    elem = p[1].split('>')[0][1:]
                    p[0] += f'</{elem}>'
                    if len(second_stack) > 0:
                        second_stack.pop()
            else:
                if p[2] != '':
                    if state in ['if','else']:
                        p[1] = p[1].replace(' ','',4)
                    if p[2] == '<>':
                        p[2] = ''
                    p[0] += p[1] + p[2]
                    if min_indent == 0:
                        min_indent = len(p[1]) - 1
                        #space = p[1][1:]

                    indent_level = len(p[1]) - 1
```

```
            if len(second_stack) > 0:
                tag_stack.append((second_stack.pop(),indent_level))

    if state in ['if','no'] and state_indent == -1:
        state_indent = indent_level + min_indent
    if tag_state == 'if':
        tag_state = ''



def p_element_normal_dd(p):
    '''
    element : DEDENT elem
    '''
    global indent_level
    global min_indent
    global state
    global state_indent
    global tag_state
    global comment
    p[0] = ''

    var = ('','')
    if len(var_stack) > 0:
        var = var_stack.pop()

    indent = len(p[1]) - 1
    if state in ['if','else']:
        indent -= min_indent

    if state in ['no','if','else'] and indent <= state_indent:
        if tag_state == 'else' or tag_state == 'if':
            tag_state = ''
        else:
            state = 'normal'
            state_indent = -1
            indent += min_indent


    if state != 'no':
        if var[0] != '':
            vars[var[0]] = var[1]
        if p[2] != '' and len(tag_stack) > 0:
            if state in ['if','else']:
                p[1] = p[1].replace(' ','',4)
            if p[2] == '<>':
                p[2] = ''

            if comment == 0:
                cl_tag = tag_stack.pop()
                p[0] += f'</{cl_tag[0]}>'
                if len(tag_stack) > 0:
                    while tag_stack[-1][1] >= indent:
                        last_tag = tag_stack.pop()
                        spaces = last_tag[1] * space
                        p[0] += '\n' + spaces + f'</{last_tag[0]}>'
                        if len(tag_stack) == 0:
```

```
                        break
            elif comment == 1:
                comment = 2
            else:
                comment = 0

            p[0] += p[1] + p[2]


        indent_level = len(p[1]) - 1
        if len(second_stack) > 0:
            tag_stack.append((second_stack.pop(),indent_level))


    if state == 'else' and state_indent == -1:
        state_indent = indent_level



def p_element_normal_sd(p):
    '''
    element : SAMEDENT elem
    '''
    global indent_level
    global state
    global vars
    global var_stack
    global comment
    p[0] = ''

    var = ('','')
    if len(var_stack) > 0:
        var = var_stack.pop()

    if state != 'no':
        if p[2] != '':
            if var[0] != '':
                vars[var[0]] = var[1]
            if state in ['if','else']:
                p[1] = p[1].replace(' ','',4)
            if p[2] == '<>':
                p[2] = ''

            if len(tag_stack) > 0 and '<!' not in p[2]:
                if comment == 0:
                    p[0] += f'</{tag_stack.pop()[0]}>'
                elif comment == 1:
                    comment = 2
                elif comment == 2:
                    comment = 0
                    p[0] += f'</{tag_stack.pop()[0]}>'

            p[0] += p[1] + p[2]


        indent_level = len(p[1]) - 1
        if len(second_stack) > 0:
            tag_stack.append((second_stack.pop(),indent_level))
```

```
def p_element_text_id(p):
    '''
    element : TEXT_INDENT elem
    '''
    p[0] = ''
    global state
    if state != 'no':
        if state in ['if','else']:
            p[1] = p[1].replace(' ','',4)
        p[0] = p[1] + p[2]


def p_element_text_dd(p):
    '''
    element : TEXT_DEDENT elem
    '''
    p[0] = ''
    global state
    if state != 'no':
        if state in ['if','else']:
            p[1] = p[1].replace(' ','',4)
        p[0] = p[1] + p[2]


def p_element_text_sd(p):
    '''
    element : TEXT_SAMEDENT elem
    '''
    p[0] = ''
    global state
    if state != 'no':
        if state in ['if','else']:
            p[1] = p[1].replace(' ','',4)
        p[0] = p[1] + p[2]


def p_elem_if(p):
    '''
    elem : ifClause
    '''
    p[0] = p[1]


def p_ifClause_if(p):
    '''
    ifClause : IF VAR
             | ELSE
    '''
    global state
    global vars
    global tag_state
    p[0] = ''
    if len(p) == 3:
```

```
        if p[2] in vars and vars[p[2]] != 'false':
            state = 'if'
        else:
            state = 'no'
        tag_state = 'if'
    else:
        if state == 'if':
            state = 'no'
        else:
            state = 'else'
        tag_state = 'else'



def p_ifClause_unless(p):
    '''
    ifClause : UNLESS VAR
    '''
    global state
    global vars
    global tag_state
    p[0] = ''
    if p[2] not in vars or (p[2] in vars and vars[p[2]] == 'false'):
        state = 'if'
    else:
        state = 'no'
    tag_state = 'if'



def p_elem_stuff(p):
    '''
    elem : stuff components
    '''
    p[0] = p[1] + p[2]



def p_stuff(p):
    '''
    stuff : keyLine
          | keyLine attLine
    '''
    p[0] = f'<{p[1]}'
    if len(p) == 3:
        p[0] += f' {p[2]}'
    p[0] += f'>'



def p_keyLine_tag(p):
    '''
    keyLine : TAG
            | TAG classListLine
            | TAG idLine
    '''
    global second_stack
    p[0] = ''
    if p[1][-1] != '/':
```

```
        p[0] = f'{p[1]}'
        if len(p) == 3:
            p[0] += f' {p[2]}'
    else:
        p[1] = p[1].replace('/','')
    if state != 'no':
        second_stack.append(p[1])



def p_keyLine_class_id(p):
    '''
    keyLine : classListLine
            | idLine
    '''
    global second_stack
    p[0] = f'div {p[1]}'
    if state != 'no':
        second_stack.append('div')



def p_classListLine(p):
    '''
    classListLine : classList
                  | classList ID
    '''
    if len(p) == 2:
        p[0] = f'class="{p[1]}"'
    else:
        p[0] = f'class="{p[1]}" id="{p[2][1:]}"'



def p_classList(p):
    '''
    classList : CLASS classList
              | CLASS
    '''
    if len(p) == 3:
        p[0] = p[1][1:] + ' ' + p[2]
    else:
        p[0] = p[1][1:]



def p_idLine(p):
    '''
    idLine : ID
           | ID classList
    '''
    if len(p) == 2:
        p[0] = f'id="{p[1][1:]}"'
    else:
        p[0] = f'class="{p[2]}" id="{p[1][1:]}"'



def p_attLine(p):
```

```
    '''
    attLine : atts attLine
            | atts
    '''
    p[0] = p[1]
    if len(p) == 3:
        p[0] += f' {p[2]}'


def p_atts(p):
    '''
    atts : LPAR ATTS RPAR
         | S_C_ATTS LPAR ATTS RPAR
    '''
    if len(p) == 4:
        p[0] = p[2]
    else:
        p[0] = p[3]


def p_components_dds(p):
    '''
    components : D_DOT
               | SPACE textLine
               |
    '''
    p[0] = ''

    if len(p) == 3:
        p[0] += p[2]


def p_components_da(p):
    '''
    components : DOT
               | ASSIGN VAR
    '''
    global vars

    p[0] = ''
    if len(p) == (3):
        if p[2] in vars:
            p[0] += vars[p[2]]


def p_elem_text(p):
    '''
    elem : textLine
    '''
    p[0] = p[1]


def p_textLine_interpol(p):
    '''
```

```
    textLine : INTERPOL
             | INTERPOL TEXT
             | TEXT INTERPOL TEXT
    '''
    p[0] = ''
    if len(p) < 4:
        var = p[1][2:-1]
        if var in vars:
            add = vars[var].replace('"','')
            p[0] += f'{add}'
        if len(p) == 3:
            p[0] += p[2]
    else:
        var = p[2][2:-1]
        if var in vars:
            add = vars[var].replace('"','')
            p[0] += f'{p[1]}{add}{p[3]}'
        else:
            p[0] += f'{p[1]}{p[3]}'


def p_textLine_text(p):
    '''
    textLine : TEXT
             | TEXT INTERPOL
    '''
    p[0] = p[1]
    if len(p) == 3:
        var = p[2][2:-1]
        if var in vars:
            p[0] += vars[var].replace('"','')


def p_elem_comment(p):
    '''
    elem : COMMENT textLine
    '''
    global state
    global indent_level
    global comment
    if len(p[1]) == 3:
        p[0] = ''
    else:
        p[0] = f'<!--{p[2]}-->'
    comment = 1


def p_elem_var(p):
    '''
    elem : S_VAR VAR
    '''
    global var_stack
    global comment
    p[0] = ''
    split_var = p[2].split('=')
    var_stack.append((split_var[0].strip(),split_var[1].strip()))
```

```
    comment = 1



def p_error(p):
    print("Syntax error in input: ", p)

parser = yacc.yacc(debug=True)
```