Processamento de Linguagens (3º ano de LEI)

Trabalho Prático

Relatório de Desenvolvimento

Free Top G

Gonçalo Santos
(A95354)

Daniel Furtado
(A97327)

Nuno Costa
(A96897)

28 de maio de 2023

Universidade do Minho

Resumo

No âmbito da Unidade Curricular de Processamento de Linguagens, foi proposto ao grupo, como forma de desenvolver os seus conhecimentos à cerca dos conteúdos lecionados nas aulas, a realização de um trabalho prático. Tivemos a oportunidade de escolher entre vários enunciados de diferentes trabalhos, sendo que o grupo ficou com o enunciado nº2.6 que corresponde a um conversor *TOML* para *JSON*. Deste modo, o objetivo deste trabalho prático passa por usar as ferramentas *Flex* e *Yacc* do módulo *ply* do python para implementar o tal conversor.

Conteúdo

Capítulo 1

Introdução

Estando o grupo a frequentar o 3º ano da Licenciatura em Engenharia Informática, foi-nos proposto, no contexto da Unidade Curricular de Processamento de Linguagens, que aprofundássemos os nossos conhecimentos na área. Posto isto, este trabalho consiste no desenvolvimento de um reconhecedor léxico e sintático para *TOML* com recurso ao par de ferramentas Flex e Yacc.

TOML é uma linguagem de configuração minimalista que tem como objetivo ser mapeada numa tabela de hash. Devido à sua sintaxe básica, é simples mapear esta linguagem em estruturas de dados em outras linguagens (como é o exemplo do JSON).

Posto isto, vamos passar para a análise léxica da linguagem.

Capítulo 2

Analisador Léxico

A primeira etapa do desenvolvimento deste trabalho prático baseou-se na elaboração do analisador Léxico com a utilização da ferramenta Lex. O analisador Léxico é responsável por identificar os tokens individuais dentro do código TOML, dividindo o texto em unidades léxicas significativas. O lexer analisa o arquivo de entrada caráter por caráter e reconhece os padrões definidos para cada token.

Os tokens são as componentes elementares, ou seja, não divisíveis, da linguagem.

A definição de um analisador léxico passa pela definição dos padrões dos seus tokens.

2.1 Tokens

Nesta secção, apresentamos todos os tokens que compõem o nosso analisador léxico.

```
tokens = ['OBJETO', 'AOT', 'COMENTARIO', 'ATRIBUTO', 'STR', 'LITSTR',
    'INT', 'FLOAT', 'NOTACIEN', 'INFNAN', 'SC', 'DATA', 'TEMPO',
    'BOOLEANO']

literals = '{=[],}'

def t_COMENTARIO(t):
    r'\#.*'
    return t

def t_ATRIBUTO(t):
    r'[\w\-]+(\s*\.\s*[\w\-]+)*(?=\s*=)'
    return t

def t_STR(t):
    r'".*?"'
    return t

def t_LITSTR(t):
    r"'.*?'"
    return t

def t_DATA(t):
```

```
22      r'\d+\-\d+\-\d+((T|␣)\d+:\d+:\d+(\.\d+)?(Z|[+\-]\d+:\d+)?)?'
23      return t
24
25  def t_TEMPO(t):
26      r'\d+:\d+:\d+(\.\d+)?'
27      return t
28
29  # Bin rio , octal e hexadecimal
30  def t_SC(t):
31      r'(0b[01]+|0o[0-7]+|0x([0-9]|[A-F]|[a-f])+)'
32      return t
33
34  def t_INFNAN(t):
35      r'[+\-]?(inf|nan)'
36      return t
37
38  def t_NOTACIEN(t):
39      r'[+\-]?\d+(\.\d+)?(e|E)[+\-]?\d+'
40      return t
41
42  def t_FLOAT(t):
43      r'[+\-]?\d+\.\d+'
44      return t
45
46  def t_INT(t):
47      r'[+\-]?\d+'
48      return t
49
50  def t_BOOLEANO(t):
51      r'(true|false)'
52      return t
53
54  def t_AOT(t):
55      r'\[\[\S*\s*\]\]'
56      t.value = t.value[2:-2]
57      return t
58
59  def t_OBJETO(t):
60      r'(?<=\[)\S*\s*(?=\])'
61      return t
```

2.1.1 Detalhes dos Tokens

Em alguns tokens, tentamos agrupar o maior número de casos que achamos possível. Desses tokens podemos destacar o **DATA** e o **SC**, por exemplo.

```
[teste1]
# Todos os casos que o nosso token DATA captura
odt1 = 1979-05-27T07:32:00Z
odt2 = 1979-05-27T00:32:00-07:00
odt3 = 1979-05-27T00:32:00.999999-07:00
odt4 = 1979-05-27 07:32:00Z
ldt1 = 1979-05-27T07:32:00
ldt2 = 1979-05-27T00:32:00.999999
ld1 = 1979-05-27

[teste2]
# Todos os casos que o nosso token SC captura
# hexadecimal with prefix `0x`
hex1 = 0xDEADBEEF
hex2 = 0xdeadbeef
hex3 = 0xdead_beef

# octal with prefix `0o`
oct1 = 0o01234567
oct2 = 0o755 # useful for Unix file permissions

# binary with prefix `0b`
bin1 = 0b11010110
```

Capítulo 3

Analisador Sintático

Após a conclusão do desenvolvimento do analisador léxico, avançamos para a elaboração do analisador sintático. Nesta etapa, utilizamos o Yacc para construir a gramática que sustenta o nosso trabalho prático e possibilita a conversão do TOML. Esta fase foi especialmente desafiadora e complexa, já que exigiu várias tentativas de escrita da gramática até chegarmos à sua versão final.

3.1 Estratégia seguida

Na elaboração da gramática chegamos à conclusão que o melhor a fazer era converter a linguagem *TOML* numa estrutura de dados intermédia, ou numa RI (Representação Intermédia), que depois serviria como suporte para a conversão noutras linguagens.

A estrutura de dados na qual convertemos o *TOML* é num dicionário já que o próprio TOML define uma tabela de hashing que pode ser vista como um dicionário.

3.2 Gramática

Nesta secção, apresentamos as produções que compõem a nossa gramática. Através dessas produções, é possível compreender a estrutura da linguagem TOML.

```
p1  : toml -> linha toml
p2  : toml  | &
p3  : linha -> COMENTARIO
p4  : linha  | chaveValor
p5  : linha  | obj
p6  : linha  | aot
p7  : obj -> '[' OBJETO ']' contObj
p8  : contObj -> contOb
p9  : contOb -> COMENTARIO contOb
p10 : contOb  | chaveValor contOb
p11 : contOb  | &
p12 : aot -> AOT arrayCont
p13 : arrayCont -> elem arrayCont
p14 : arrayCont  | &
p15 : elem -> chaveValor
```

```
p16 : elem -> COMENTARIO
p17 : chaveValor -> ATRIBUTO '=' val
p18 : val -> STR
p19 : val  | LITSTR
p20 : val  | INT
p21 : val  | FLOAT
p22 : val  | NOTACIEN
p23 : val  | SC
p24 : val  | INFNAN
p25 : val  | DATA
p26 : val  | TEMPO
p27 : val  | BOOLEANO
p28 : val  | '{' contDict '}'
p29 : val  |'{' '}'
p30 : val  | lista
p31 : contDict -> chaveValor ',' contDict
p32 : contDict  | chaveValor
p33 : lista -> '[' ']'
p34 : lista  | '[' cont ']'
p35 : cont -> val
p36 : cont  | val ',' cont
```

Capítulo 4

Funcionalidades Implementadas

Como foi explicado anteriormente, o analisador sintático usa a gramática para criar um dicionário que representa uma conversão direta da linguagem *TOML*.

Após termos este dicionário construído, usamos o mesmo para a conversão numa das 3 linguagens implementadas: *JSON*, *YAML* ou *XML*.

4.1 Conversor *TOML* para *JSON*

Esta funcionalidade trata-se da funcionalidade exigida no nosso enunciado, e com isto foi a primeira a estar concluída. No caso do formato *JSON* percorremos o dicionário com o conteúdo já convertido e para os diferentes tipos de valores associados às respetivas chaves do dicionário, temos diferentes tipos de tratamento. No caso do valor ser um dicionário a função é chamada recursivamente, se o valor corresponde a uma string apenas são acrescentadas as aspas em volta do valor, se o valor for um booleano é efetuada uma pequena conversão para os booleanos do JSON, caso contrário é apenas escrito para o ficheiro JSON o output normalmente.

4.2 Conversor *TOML* para *XML*

Para tornar o trabalho mais complexo e dinâmico com mais opções e novas linguagens de conversão, decidimos que era interessante também incluir a conversão de *TOML* para *XML*. Com a gramática já implementada e os dados guardados no dicionário foi simples a conversão em *XML*. Através do método *dictToXML* convertemos um dicionário em uma representação *XML*. A função recebe três parâmetros: a *string* que armazena o resultado da conversão para XML, outra string que armazena a identação atual para cada linha do XML e o dicionário a ser convertido. Dependendo do tipo do valor associado à respetiva chave no dicionário, é tratado de forma diferente, podendo conter tipos mais complexos como dicionários e listas ou tipos mais simples que são os restantes casos, como *strings* e inteiros. Depois de converter o dicionário numa string formato *XML* apenas é necessário escrevê-la no ficheiro *output*.

4.3 Conversor *TOML* para *YAML*

Outra funcionalidade extra realizada foi a conversão *TOML* para *YAML*. Tal como na conversão para *XML* também é percorrido o dicionário com a tradução do *TOML* no entanto o formato para o qual vai converter é *YAML*. A linguagem *YAML* possui sintaxe diferente das restantes e, por isso, é tratada de acordo com as

suas características. As chaves são seguidas de dois pontos ":"e os valores são representados diretamente, sem a necessidade de tags de abertura e de fecho. Portanto, a função *dicToYAML* escreve as chaves e valores diretamente no ficheiro *output*, separados por dois pontos, possui também um tratamento especial no caso de listas e dicionários tal como em *XML*.

4.4 Servidor Conversor de *TOML*

Com o intuito de tornar o trabalho mais desafiante, decidimos desenvolver, utilizando node.js, um servidor que fosse capaz de implementar numa só plataforma os três tipos de conversores mencionados, de forma a que o resultado fosse visualmente mais atrativo. Este servidor permite ao utilizador inserir o código *TOML* (na caixa do lado esquerdo) e obter (na caixa do lado direito) o código correspondente à linguagem selecionada (*JSON*, *YAML* ou *XML*).

Ao aceder ao servidor, o utilizador é direcionado para uma página onde pode escolher a linguagem para a qual deseja converter o código *TOML*.

Figura 4.1: Página inicial do conversor

Após selecionar a linguagem desejada, o utilizador é redirecionado para uma nova página, que é a página principal onde a conversão efetivamente ocorre. Nessa página, é possível encontrar a opção de alterar a linguagem na parte superior da página. A visualização é simples e intuitiva, com o objetivo de facilitar a compreensão e utilização do servidor.

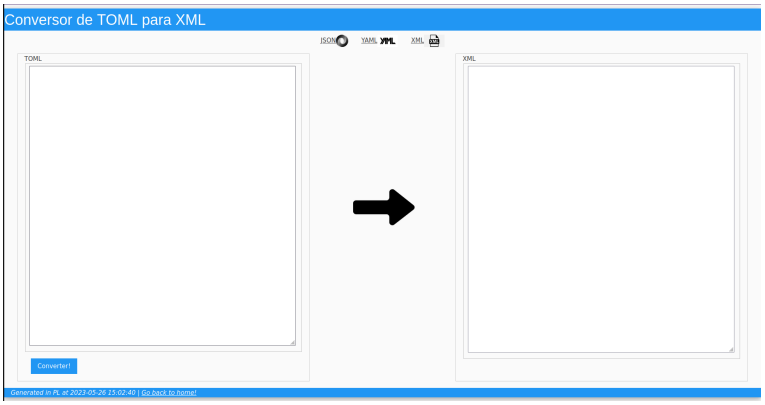

Figura 4.2: Página principal do conversor

Capítulo 5

Exemplos de utilização

5.1 Conversão de *TOML* para *JSON*

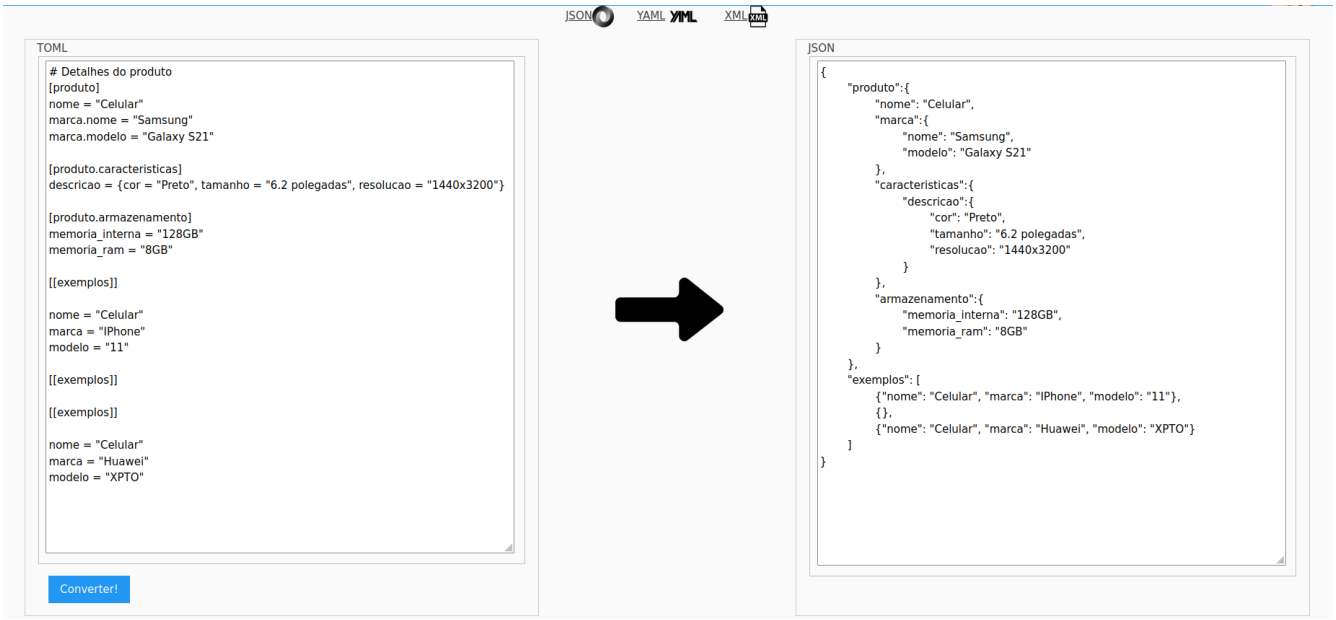

Figura 5.1: Conversão de TOML para JSON

5.2 Conversão de *TOML* para *YAML*

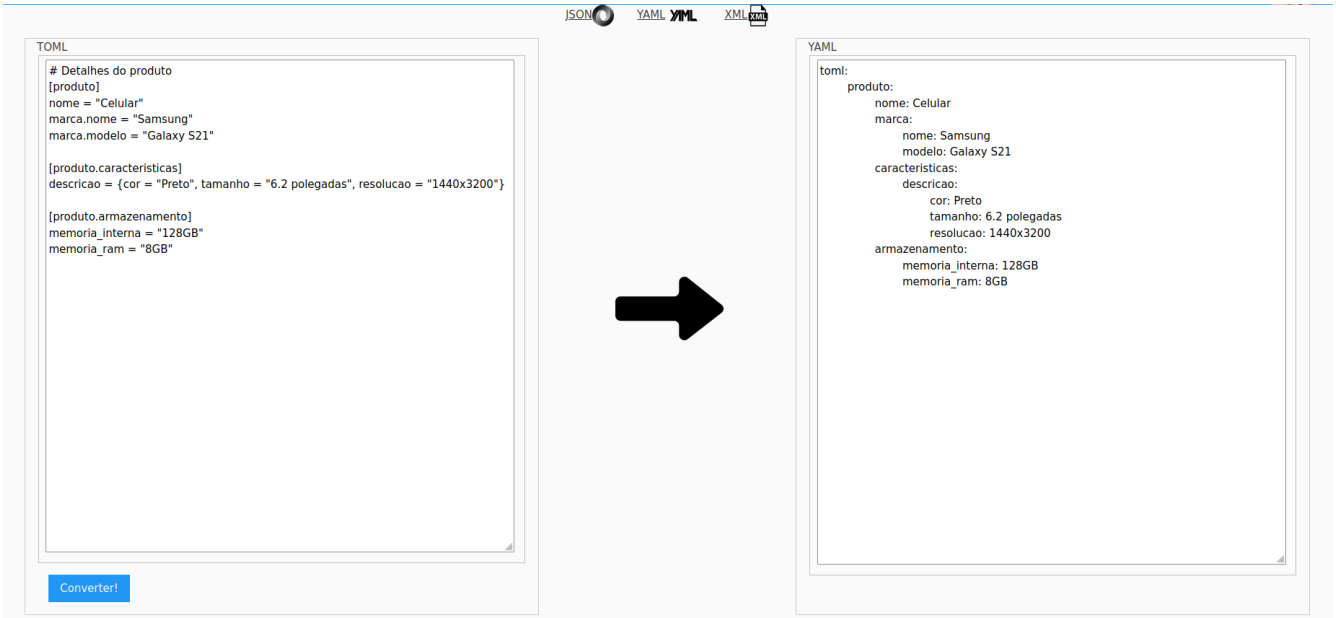

Figura 5.2: Conversão de TOML para YAML

5.3 Conversão de *TOML* para *XML*

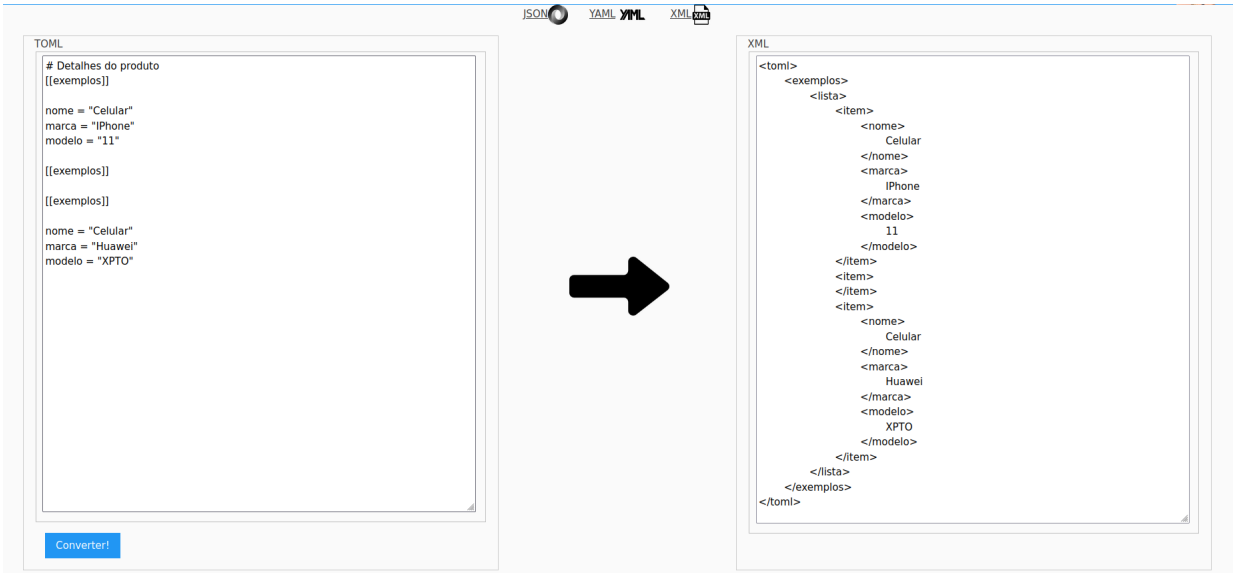

Figura 5.3: Conversão de TOML para XML

Capítulo 6

Conclusão

Finalizado este trabalho prático da Unidade Curricular de Processamento de Linguagens, podemos fazer um balanço do mesmo.

Como era de esperar, a realização deste trabalho revelou-se bastante enriquecedora e útil, na medida em que pudemos consolidar conhecimentos adquiridos nas aulas desta UC. Para além do aprofundamento dos nossos conhecimentos acerca de Expressões Regulares, Analisadores Léxicos e Sintáticos, pudemos ainda explorar as nossas capacidades no que toca ao uso de node.js, tornando este projeto ainda mais gratificante e desafiador.

Em suma, estamos satisfeitos com o nosso desempenho na elaboração deste trabalho, considerando que todas as nossas soluções foram bem estruturadas, justificadas e consistentes.