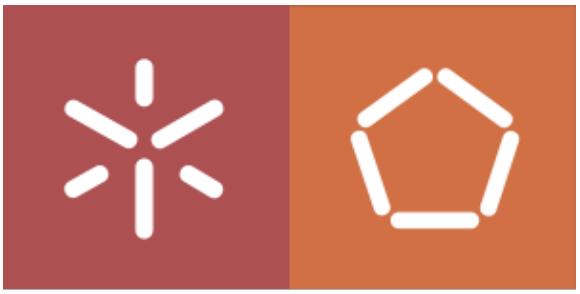

Universidade do Minho
Escola de Engenharia

2.6 Conversor TOML-JSON

Processamento de Linguagens

Gonçalo Semelhe Sousa Braga (a97541)
João Miguel Ferreira Loureiro (a97257)
Simão Oliveira Alvim Barroso (a96834)

28 de maio de 2023

Grupo 31 - Os empresários

2022/2023

Índice

Lista de Figuras

1 Introdução

O presente relatório diz respeito ao trabalho prático da Unidade Curricular de Processamento de Linguagens. Ao longo deste vamos discutir os vários aspetos do trabalho como a escolha do tema, a arquitetura da nossa resolução, e a nossa resolução do problema. Por fim vamos apresentas as nossa reflexões críticas do trabalho feito, bem como possíveis melhorias que podemos fazer caso seja necessário desenvolver ainda mais o projeto.

1.1 Tema Escolhido

O projeto é composto por vários enunciados dos quais apenas um deve ser selecionado, sendo, portanto, uma escolha livre. Optámos por escolher o tema **2.6**, mais propriamente, o **Conversor TOML-JSON**.

A nossa escolha recaiu sobre o facto de termos achado o tema interessante, uma vez que é bastante usado pela sociedade hoje em dia e pelo facto de ser uma forma de consolidar-mos a matéria dada na UC.

1.2 Objetivos do Trabalho

Os dois principais objetivos do nosso trabalho prático são:

- Ambientação com o **lexer** e criação de um analisador léxico para uma determinada linguagem;
- Ambientação com o **yacc** e criação de uma gramática que suporte o funcionamento do nosso coversor de linguagens;

Além destes objetivos principais o nosso grupo definiu alguns objetivos adicionais, tais como:

- Criação de um conversor para mais do que uma linguagem(além do obrigatório no trabalho prático), tais como TOML-XML, TOML-YAML, JSON-TOML.
- Criação de um ambiente gráfico, de forma a tornar semelhante a experiência de utilização entre diferentes tipos de utilizadores.

2 Arquitetura do Trabalho

2.1 Estrutura de pastas

O trabalho encontra-se divido em vários ficheiros e pastas. Temos uma pasta, denominada de **input**, com vários ficheiros de texto (*.txt*) que utilizamos para irmos testando o programa ao longo da sua resolução. Optámos por este sistema de testes em vez da criação de testes automatizados pois não nos pareceu viável a sua implementação tendo em comparação com os outros extras que fizemos.

Possuímos outra pasta, denominada de **output**, com os ficheiros dados como resultado pelos nossos conversores. Decidimos adotar esta estratégia, para ser relativamente mais fácil vermos os resultados dos conversores.

Para utilizar os **inputs** e dar como o resultado os **outputs**, possuímos os conversores de várias linguagens numa pasta denominada **src**, como forma de agrupar todo o código dos conversores.

2.2 Componentes de Software

As várias componentes de software[1] utilizadas ao longo do projeto foram o **Ply**, mais concretamente o **Lex** para fazer a analise léxica e o **Yacc** para fazer análise sintática.

Além disto, foi também usado o módulo ***PyQt5*** para construir a interface gráfica, que nos permite aproximar o trabalho de uma aplicação verdadeira (invés de ser feito pela linha de comandos).

2.3 Conversor TOML-JSON

Neste relatório do trabalho prático, vamos abordar de forma detalhada o conversor TOML-JSON, pois é aquele que é estritamente necessário.

Este conversor possui todas as funcionalidades da documentação do *TOML* implementadas, tais como:

[1] Software é um termo técnico que foi traduzido para a língua portuguesa como suporte lógico e trata-se de uma sequência de instruções a serem seguidas e/ou executadas, na manipulação, redirecionamento ou modificação de um dado ou acontecimento

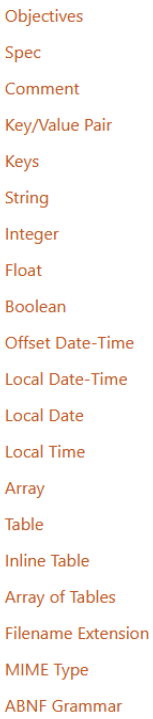

Figura 2.1: Documentação TOML

É possível ver todos os parâmetros da documentação no seguinte link: ***https://toml.io/en/v1.0.0***

2.3.1 Explicação da subdivisão do conversor

O conversor por nós concebido tenta tirar partido das ferramentas fornecidas pelas classes em *python*, desta forma nos decidimos criar uma classe denominada de **Conversor**.

Nesta classe, possuímos as seguintes variáveis:

```
self.inDicInAOT = 0 # Flag que indica que estamos num dicionrio e
    dentro de um array of tables

self.numberAOT = 0 # Flag que indica o nmero de array of tables
    existentes

self.dicionario = "" # String que possui o dicionrio que estamos a
    tratar dentro do array of tables

self.aot = [] # Armazenamento temporrio do array of table formado

self.fileStates = [] # Estado do ficheiro input que estamos a
    tratar no momento

self.documentTitle = "" # Ttulo do docuemnto de output

self.documentData = dict() # Dicionrio que possui a  informao
    convertida de um ficheiro para o outro

self.keyEmpty = 0 #  Nmero  de chaves que so deste tipo: '' ou ""

self.auxListas = [] # Lista que auxilia a construo das listas em
```

```
        JSON

    self.auxinlineTables = dict() # Dicionrio que auxila a construo das
        Inline Tables em JSON

    self.listaKeysUsadas = [] # Lista das keys j usadas nas Inlines
        Tables

    self.listaKeysNRetirar = [] # Lista das keys a no retirar do
        documentData

    self.listaKeysRetirar = [] # Lista das keys a retirar do
        documentData

    self.keyOndeRet = [] # Chaves do diconrio principal onde retirar

    self.inlineTables = 0 # Flag que indica que estamos numa Inline
        Table

    self.inAOT = 0 # Flag que indica que estamos num array of tables

    self.lastAOT="" # String que indica o ltimo array of tables tratado
```

Desta forma conseguimos perceber o porquê de termos criado o conversor assim.
Além disto possuímos um **analisador léxico** e um **parser** que juntos nos dão a integridade do programa.

2.3.2 Explicação da abordagem do problema por nós utilizada

O nosso grupo de trabalho decidiu abordar a leitura do ficheiro de input da seguinte maneira:

- 1º Travessia do ficheiro: Nesta primeira travessia do ficheiro, nós agrupamos as estruturas de dados que se encontram dispostas em diferentes linhas do ficheiro, de forma a agrupa-las numa só linha do ficheiro, como acontece por exemplo com as listas.

  Assim esta primeira travessia do ficheiro serve para preparar o ficheiro passado como input para a nossa ferramenta, uma vez que esta tem aspetos que depende desta primeira travessia do ficheiro.

- 2º Travessia do ficheiro: Nesta segunda travessia do ficheiro, nós fazemos o que é suposto fazer, que é a passagem pelo analisador léxico, a "passagem"pelo parser e as ações semânticas necessárias para construirmos o ficheiro de *output*.

  Esta fase já não necessita de tratar de alguns problemas que teria de tratar caso não houvesse a primeira travessia, como por exemplo a existência de ***newline*** no meio de conteúdo do ficheiro *input* .

Esta estratégia por nós utilizada é semelhante aquilo que o ***Latex*** faz, uma vez que ambos os softwares utilizam 2 travessias dos ficheiros de *input* para gerar um ficheiro *output*.
Esta estratégia foi devidamente explicada ao corpo docente, que nos indicou, que a melhor opção era explicar devidamente a nossa estratégia no relatório final, algo feito por nós acima.

2.3.3 Explicação das funções auxiliares utilizadas

As funções auxiliares por nós utilizadas no conversor são as seguintes:

- **giveListas** - Esta função recebe uma *string* correspondente a uma lista de listas e devolve uma lista de listas com o conteúdo existente na *string*, mas do tipo ***list*** invés de ser com o tipo **str**.
- **contaAPR** - Esta função recebe uma *string* e conta o número de abertura de parêntesis retos existentes na *string* passada como *input*.
- **contaFPR** - Esta função recebe uma *string* e conta o número de fecho de parêntesis retos existentes na *string* passada como *input*.
- **splitData** - Esta função recebe os dados do ficheiro e divide-os da forma mais correta possível, isto é, juntando as estruturas de dados, que estão separadas em diversas linhas do ficheiro, da forma mais compacta possível.
- **levelsData** - Esta função recebe uma *string* existente dentro de um dicionário ou sub-dicionário e dividia pelas pontos existentes nessa mesma *string*.

2.3.4 Explicação do Analisador Léxico

O nosso analisador léxico reconhece os seguintes ***tokens***, que são expostos em baixo:

```
tokens = (
        "WORD", "INT", "FLOAT", "PLICA", "FPR", "APR",
        "VIRG", "ASPA", "IGUAL", "CONTENT", "DATE", "TIME",
            "NEWDICTIONARY", "NEWSUBDICTIONARY",
        "SIGNAL", "INTWITHUNDERSCORE", "HEXADECIMAL", "OCTAL",
            "BINARIO", "EXPONENCIACAO", "FLOATWITHUNDERSCORE",
        "OFFSETDATETIME", "LOCALDATETIME", "LOCALDATE",
            "LOCALTIME","BOOL","APC","FPC", "AOT"
    )
    literals = (':', '-')

    t_AOT = r'\[\[.+\]\]'
    t_BOOL = r'True|False|true|false|Verdadeiro|Falso|verdadeiro|falso'
    t_WORD =
        r'([0-9]+)?[A-Za-z_\-]+([0-9]+|\.)?([A-Za-z_\-\.]+)?([0-9]+)?'
    t_FLOAT = r'\d+\.\d+'
    t_INT = r'\d+'
    t_PLICA = r'\''
    t_FPR = r'\]'
    t_APR = r'\['
    t_VIRG = r'\,'
    t_ASPA = r'"'
    t_IGUAL = r'\='
    t_CONTENT = r'("|\').[^=]*("|\')'
    t_DATE = r'\d+\-\d+\-\d+'
    t_TIME = r'\d+\:\d+:\d+'
    t_SIGNAL = r'(\+|-){1}'
    t_INTWITHUNDERSCORE = r'\d+(?=\_)|\_|(?<=\_)\d+'
    t_FLOATWITHUNDERSCORE =
        r'((\d+\.\d+)|\d+)(?=\_)|\_|(?<=\_)(\d+\.\d+|\d+)'
    t_HEXADECIMAL = r'0[xX][0-9a-fA-F\_]+'
    t_OCTAL = r'0[oO][0-7]+'
    t_BINARIO = r'0[bB][0-1]+'
```

```
t_EXPONENCIACAO = r'(\+|\-)?(\d+|\d+\.\d+){1}[eE](\+|\-)?\d+'
# linebreak no seguinte regex apenas  utilizado neste relatrio por
    uma questo de legibilidade
t_OFFSETDATETIME =
    r'\d{4}\-\d{2}\-\d{2}(T|\s)\d{2}\:\d{2}\:((\d{2}Z|\d+\.\d+\-\d{2}:\d{2})|
\d{2}\-\d{2}\:\d{2})'
t_LOCALDATETIME =
    r'\d{4}\-\d{2}\-\d{2}(T|\s)\d{2}\:\d{2}\:(\d+\.\d+|\d{2})'
t_LOCALDATE = r'\d{4}-\d{2}-\d{2}'
t_LOCALTIME = r'\d{2}:\d{2}:(\d+\.\d+|\d{2})'
t_APC = "{"
t_FPC = "}"
t_ANY_ignore = ' \t'
```

Além dos *tokens* que são expostos em cima, possuímos os seguintes estados:

```
def t_COMMENTARY(t):
        r'\#.*'
        pass

    def t_NEWLINE(t):
        r"""\n+"""
        t.lexer.lineno += t.value.count('\n')
        return t

    def t_NEWDICTIONARY(t):
        r'\[[a-zA-Z\d\-]+(\s+)?\]'
        print("Entrei no estado NEWDICTIONARY")
        t.lexer.begin('NEWDICTIONARY')
        return t

    def t_NEWSUBDICTIONARY(t):
        r'(\[)([a-zA-Z\d\-]+(\s+)?\.(\s+)?[a-zA-Z\d\-]+(\s+)?\])|
        (\[([a-zA-Z\-\d]+(\s+)?\.(\s+)?)+[a-zA-Z\d\-]+(\s+)?\])'
        print("Entrei no estado NEWSUBDICTIONARY")
        t.lexer.begin('NEWSUBDICTIONARY')
        return t

    def t_NEWDICTIONARY_NEWSUBDICTIONARY_END(t):
        r'\n\['
        print('Sai do meu estado atual, e vou voltar ao meu estado
            inicial')
        t.lexer.begin('INITIAL')
        return t

    def t_ANY_error(t):
        print('Lexical error: "' + str(t.value[0]) + '" in line ' +
            str(t.lineno))
        t.lexer.skip(1)
```

Desta forma conseguimos perceber, quais são os ***tokens*** que o analisador léxico nos irá fornecer para que o ***parser*** consiga juntamente com ações semânticas originar o ficheiro output com os ***tokens*** fornecidos pelo analisador léxico.

Um exemplo de output do analisador léxico é o seguinte:

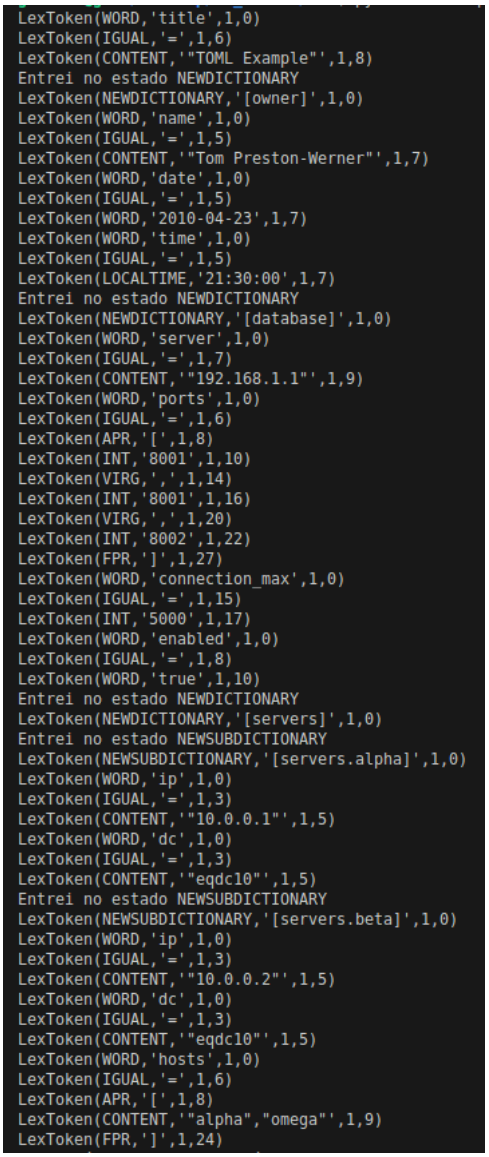

Figura 2.2: Output obtido do analisador léxico

Se repararmos no output do analisador léxico, os ***newlines*** não são necessários uma vez que os dados já se encontram divididos da forma correta, por isso não necessitamos de ter cuidado com os ***newlines***. Este foi um dos principais problemas solucionados com duas travessias do mesmo ficheiro.

2.3.5 Explicação do Parser

O **Parser** pode ser encarado no nosso trabalho prático como o ponto principal do trabalho, uma vez que a gramática juntamente com as ações semânticas necessárias dão-nos o resultado final do trabalho.

Olhemos agora para a gramáticas por nós pensada para suportar o trabalho prático:

```
Rule 0     S' -> Dados
Rule 1     Dados -> WORD IGUAL Content
Rule 2     Dados -> ASPA ASPA IGUAL Content
Rule 3     Dados -> PLICA PLICA IGUAL Content
Rule 4     Dados -> INT IGUAL Content
```

```
Rule 5    Dados -> CONTENT IGUAL Content
Rule 6    Dados -> WORD IGUAL APC Conteudo FPC
Rule 7    Dados -> WORD IGUAL APC FPC
Rule 8    Dados -> NEWDICTIONARY
Rule 9    Dados -> NEWSUBDICTIONARY
Rule 10   Dados -> AOT
Rule 11   Dados -> APR CONTENT FPR
Rule 12   Dados -> APR WORD FPR
Rule 13   Dados -> WORD WORD IGUAL Content
Rule 14   Dados -> WORD WORD WORD IGUAL Content
Rule 15   Dados -> INT WORD INT IGUAL Content
Rule 16   Dados -> FLOAT IGUAL Content
Rule 17   Dados -> WORD IGUAL AOT
Rule 18   Content -> CONTENT
Rule 19   Content -> DATE
Rule 20   Content -> TIME
Rule 21   Content -> Lista
Rule 22   Content -> Palavras
Rule 23   Content -> LittleEndian
Rule 24   Content -> LittleEndianFloat
Rule 25   Content -> HEXADECIMAL
Rule 26   Content -> OCTAL
Rule 27   Content -> BINARIO
Rule 28   Content -> EXPONENCIACAO
Rule 29   Content -> signalInf
Rule 30   Content -> OFFSETDATETIME
Rule 31   Content -> LOCALDATETIME
Rule 32   Content -> LOCALDATE
Rule 33   Content -> LOCALTIME
Rule 34   Content -> BOOL
Rule 35   signalInf -> SIGNAL WORD
Rule 36   Conteudo -> Conteudo VIRG Dados
Rule 37   Conteudo -> Dados
Rule 38   LittleEndianFloat -> FLOATWITHUNDERSCORE OtherEndiansFloat
Rule 39   LittleEndian -> INTWITHUNDERSCORE OtherEndians
Rule 40   OtherEndians -> INTWITHUNDERSCORE OtherEndians
Rule 41   OtherEndiansFloat -> FLOATWITHUNDERSCORE OtherEndiansFloat
Rule 42   OtherEndians -> <empty>
Rule 43   OtherEndiansFloat -> <empty>
Rule 44   Content -> INT
Rule 45   Content -> FLOAT
Rule 46   Content -> SIGNAL INT
Rule 47   Content -> SIGNAL FLOAT
Rule 48   Content -> WORD FLOAT
Rule 49   Lista -> APR Elementos FPR
Rule 50   Lista -> APR FPR
Rule 51   Elementos -> Elementos VIRG Elemento
Rule 52   Elementos -> Elementos VIRG
Rule 53   Elementos -> Elemento
Rule 54   Elemento -> WORD
Rule 55   Elemento -> CONTENT
Rule 56   Elemento -> INT
Rule 57   Elemento -> FLOAT
Rule 58   Palavras -> WORD Palavras
Rule 59   Palavras -> <empty>
```

Como podemos ver pela gramática por nós definida, o axioma da mesma é: **Dados**.

Olhando agora atentamente para a regra **Dados**, esta corresponde ás atribuições existentes no ficheiro input bem como aos dicionários e sub-dicionários , tal como:

```
name = "Tom Preston-Werner"
date = 2010-04-23
time = 21:30:00
[owner]
[database]
[servers]
[servers.alpha]
[servers.beta]
```

Olhando atentamente para a regra **Content**, esta engloba todo o conteúdo da atribuição, isto é, tudo o que está à direita do igual existente na atribuição:

```
"Tom Preston-Werner"
2010-04-23
21:30:00
```

Esta regra abrange todo o tipo de variáveis possíveis de serem encontradas num ficheiro **.json**, tal como*strings*, datas, tempo, listas, frases, *littleEndian*, *littleEndian* em *floats*, hexadecimais, octal, binario, exponenciação, infinito, *offsetdatetime*, *localdatetime*, *localdate*, *localtime* e *bool*.

Assim, como podemos ver pela gramática apresentada neste relatório, juntamente com ações semânticas conseguimos ter como ficheiro final um output tal como:

Figura 2.3: *Output* obtido do conversor

3 Extras

Nesta secção vamos falar de uns extras que fizemos para o trabalho ao longo do trabalho de modo a explorarmos diversas funcionalidades da ferramenta desenvolvida, de aprofundar o nosso conhecimento na matéria da UC e de testar a ferramenta criada em diversos problemas.

3.1 Conversor TOML-YAML e TOML-XML

Estando o conversor TOML-JSON a funcionar o nosso primeiro pensamento foi em pensar de como usar o dicionário produzido pelo *yacc* para outras especificações. Sendo o dicionário uma representação bastante versátil, este conjunto de desafios teve uma resolução muito similar: são ambas funções recursivas que a cada nível de indentação se chama com o valor de espaços aumentado por 1.

```
def dictToYAML_P(dic, espacos):
result = ""
for e in dic:
    if type(dic[e]) == dict:
        result += (espacos * " ") + e + ":\n"
        result += dictToYAML_P(dic[e], espacos + 1)
    elif type(dic[e]) == list:
        result += (espacos * " ") + e + ":\n"
        for l in dic[e]:
            result += ((espacos + 1) * " ") + "- " + str(l) + "\n"
    else:
        result += (espacos * " ") + e + ": " + str(dic[e]) + "\n"
return result


def dict2xml(dic, n):
    result = ""
    for i in dic:
        result += 4 * (n - 1) * " " + "<" + i + ">\n"
        if type(dic[i]) == dict:
            result += dict2xml(dic[i], n + 1)
        else:
            result += 4 * n * " " + str(dic[i]) + "\n"
        result += 4 * (n - 1) * " " + "</" + i + ">\n"
    return result
```

Ambas estas funções encontram-se no ficheiro *toYAML_XML.py*.

Estas duas funções recebem um dicionário (chamando o conversor TOML-JSON que produz esta representação intermédia) e um número de espaços, geralmente 0 para a função do *YAML* e 1 para a do *XML*, mas que podem ser aumentados consoante a identação. É de realçar que este número está *hardcoded* e permitir ao utilizador a sua alteração através do uso da *interface* (explicado mais à frente) será certamente uma tarefa futura.

Temos de ressalvar sendo estas principalmente *features* adicionais pode levar à existência de algum erros, principalmente no que toca ao *XML*. Baseamos-nos nos vários conversores online já existentes para a execução desta tarefa: em particular neste para *XML* e neste para o *YAML*.

Vamos agora ver em baixo um exemplo de cada uma destas conversões. Do lado esquerdo está o input em *TOML* e do lado direito o output.

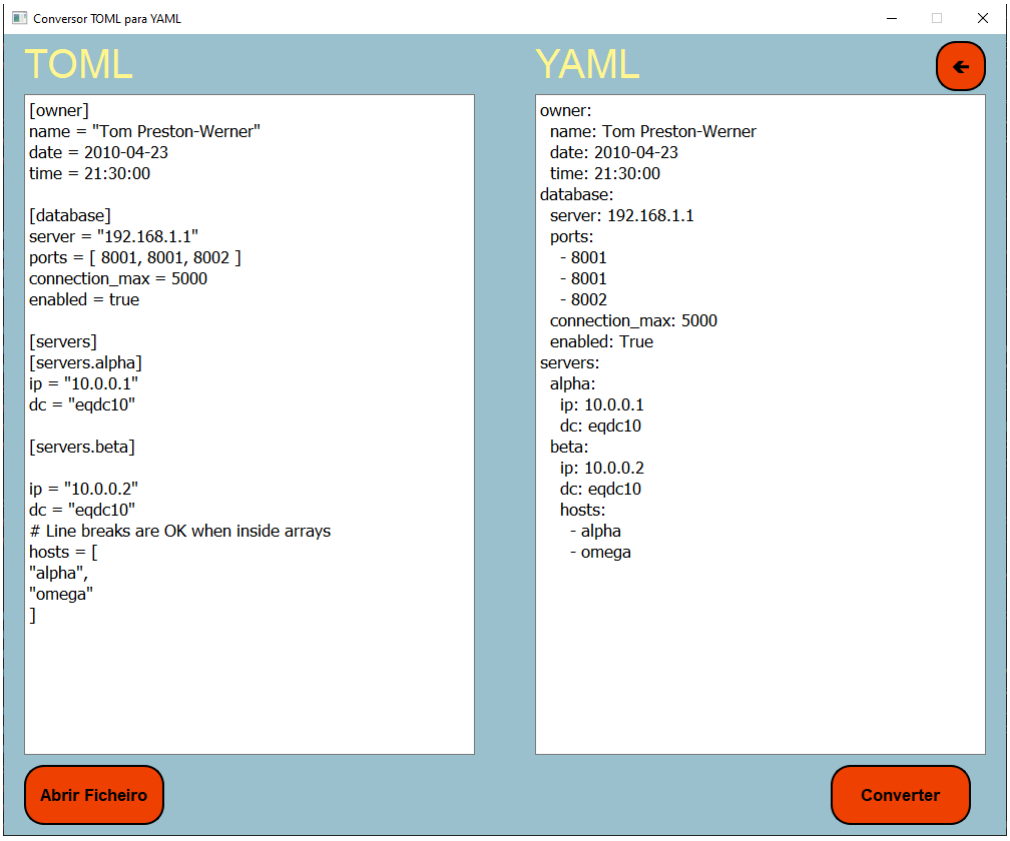

Figura 3.1: Exemplo de conversão de *TOML* para *YAML*

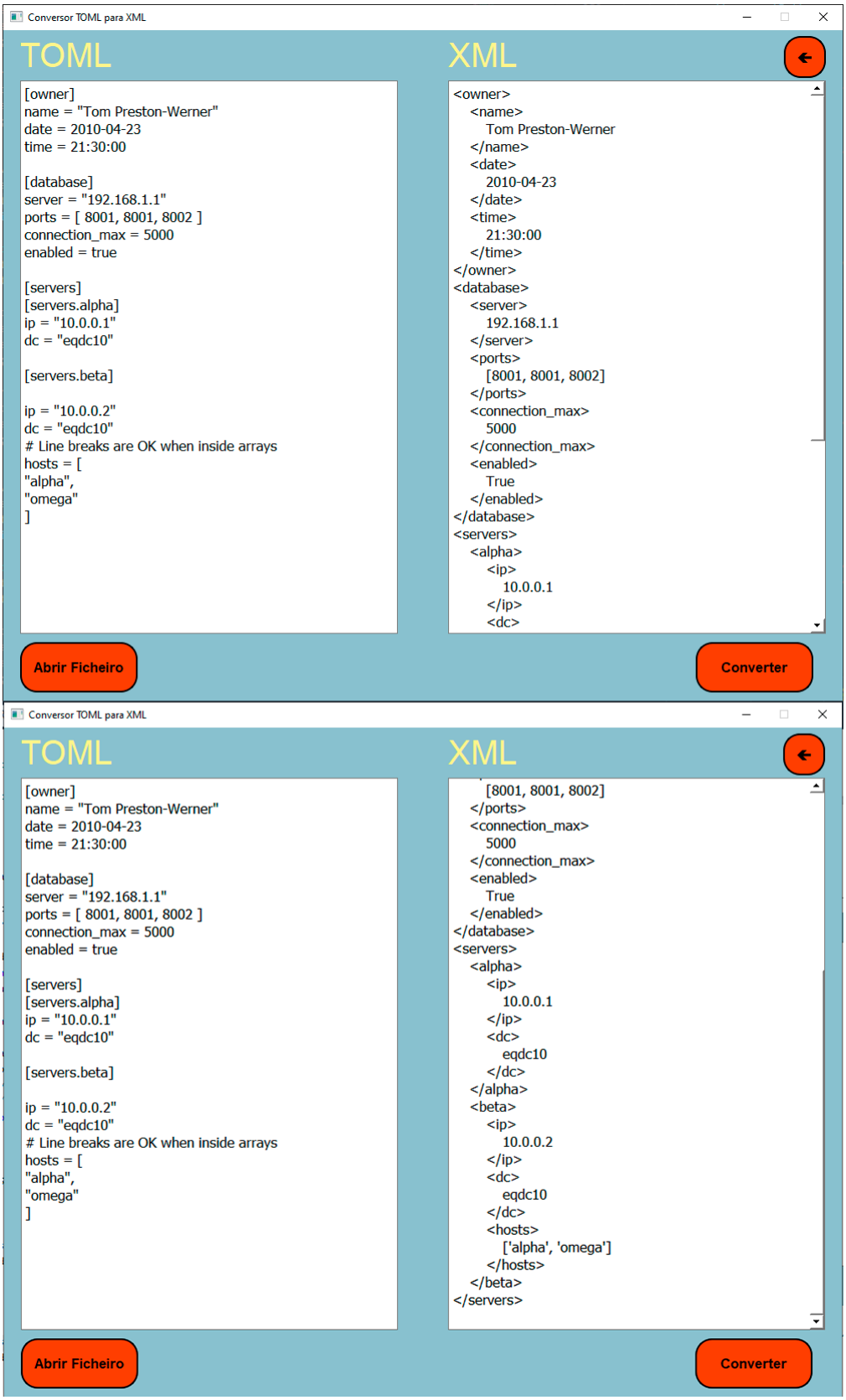

Figura 3.2: Exemplo de conversão de *TOML* para *XML*

3.2 Conversor JSON-TOML

Após retiramos uma dúvida com o professor Ramalho sobre o trabalho foi nos sugerido que como desafio e complementar ao trabalho fizéssemos um conversor de *JSON* para *TOML*, ou seja o completo contrário deste trabalho, o que iria implicar uma nova análise léxica e sintática. Optámos por fazer-lo.

É de mencionar que esta resolução não está perfeita, estando apenas uma versão que funciona para a o básico do *JSON*, servindo na nossa estimativa para uma boa quantidade dos casos. Um dos exemplos em que não funciona é com *Numeric Literal* (por exemplo: 350_000_203). Não encontramos mais nenhum caso de erro flagrante, no entanto como não testamos efusivamente esta parte, não garantimos a sua não existência.

A nossa resolução desta parte encontra-se no ficheiro *jsonToToml.py*.

A estrutura deste programa é bastante simples, muito parecida à feita durante as aulas práticas: temos uma parte do programa (lex) que faz a análise léxica e uma parte do programa que faz a análise sintática (yacc).

Relativamente ao lex, os *tokens* que utilizamos e a descrição do que capturam segue em baixo:

```
tokens = (
    'STRING', # Da match com as string
    'NUM', # Da match com qualquer numero, inclusive decimais # (\+|-)?
    'LPR', # Abrir parentese reto
    'RPR', # Fechar parentese reto
    'LCHAVETA', # Abrir chaveta
    'RCHAVETA', # Abrir chaveta
    'VIRG', # Captura a virgula
    'DOISPONTOS', # Captura os dois pontos (:)
    'BOOL', # r'true|false'
    'NULL' # captura null
)
```

Devido à simplicidade do *JSON*, a quantidade de *tokens* também não precisava de ser substancial, pelo que estes chegam e sobram para os vários exemplos que testamos, o mais complicado dos quais se encontra na figura em baixo da demonstração final.

Relativamente à gramática, esta é muito básica tendo apenas 13 produções. Em baixo segue a especificação da gramática dada pelo ficheiro *parser.out* (para se obter este ficheiro basta pôr a *flag debug=True* na função *yacc.yacc*).

```
    Grammar

Rule 0     S' -> object
Rule 1     object -> LCHAVETA members RCHAVETA
Rule 2     members -> par
Rule 3     members -> par VIRG members
Rule 4     par -> STRING DOISPONTOS value
Rule 5     array -> LPR elementos RPR
Rule 6     elementos -> value
Rule 7     elementos -> value VIRG elementos
Rule 8     value -> object
Rule 9     value -> NULL
Rule 10    value -> array
Rule 11    value -> BOOL
```

```
Rule 12   value -> NUM
Rule 13   value -> STRING
```

Através da análise do ficheiro *parser.out* também não há conflitos na gramática.

A estrutura de um ficheiro *JSON* é, na sua versão mais simples, uma lista de *key value pairs*. Os delimitadores das listas são as chavetas. O valor da *"key"* é uma *string* e os *values* podem ser de vários tipos diferentes.

Um dos aspetos negativos e que poderia ajudar no problema dos *numeric literals* era dividir em mais os *tokens* para depois estender mais a gramática para incluir esses novos *tokens*.

Associado a esta produções são acrescentadas ações semânticas de modo a gerar um dicionário. cada *members* é uma *key* do dicionário e a seu valor é um o valor dessa *key*.

A passagem do dicionário para *TOML* é tratada pela função *toml.dumps* fornecida pelo módulo toml (*import toml*).

Figura 3.3: Exemplo de conversão de *JSON* para *TOML*

3.3 Interface

Por fim fizemos uma *interface* de maneira a tornar o processo mais intuitivo e apelativo e a defesa deste trabalho mais eficiente. Para isto recorremos ao módulo ***PyQt5***. Ao longo deste trabalho foram apresentadas várias imagens da *interface* que passemos agora a explicar a sua resolução.

Trata se de uma *interface* básica, mas que cumpre todas as necessidades requeridas como as várias traduções entre os vários formatos e a possibilidade de abrir e guardar ficheiros.

> **Nota:** Para poder utilizar a interface, é necessário instalar o módulo ***PyQt5***, no caso este ainda não estar instalado na máquina pretendida.

Todo o código relacionado com a *interface* encontra-se localizado no ficheiro *menu.py*, estando este dividido em duas classes: a ***MainMenu***, que constrói a página principal e contém todos os métodos relacionados a esta, e a ***ConvertMenu***, que para além de construir uma determinada página de conversão, contém e utiliza os métodos a ela relacionados.

Assim, a *interface* está dividida em dois tipos de página: a página principal e a página de conversão. Ao iniciar o programa é apresentado a página principal onde o utilizador escolhe qual conversor deseja utilizar:

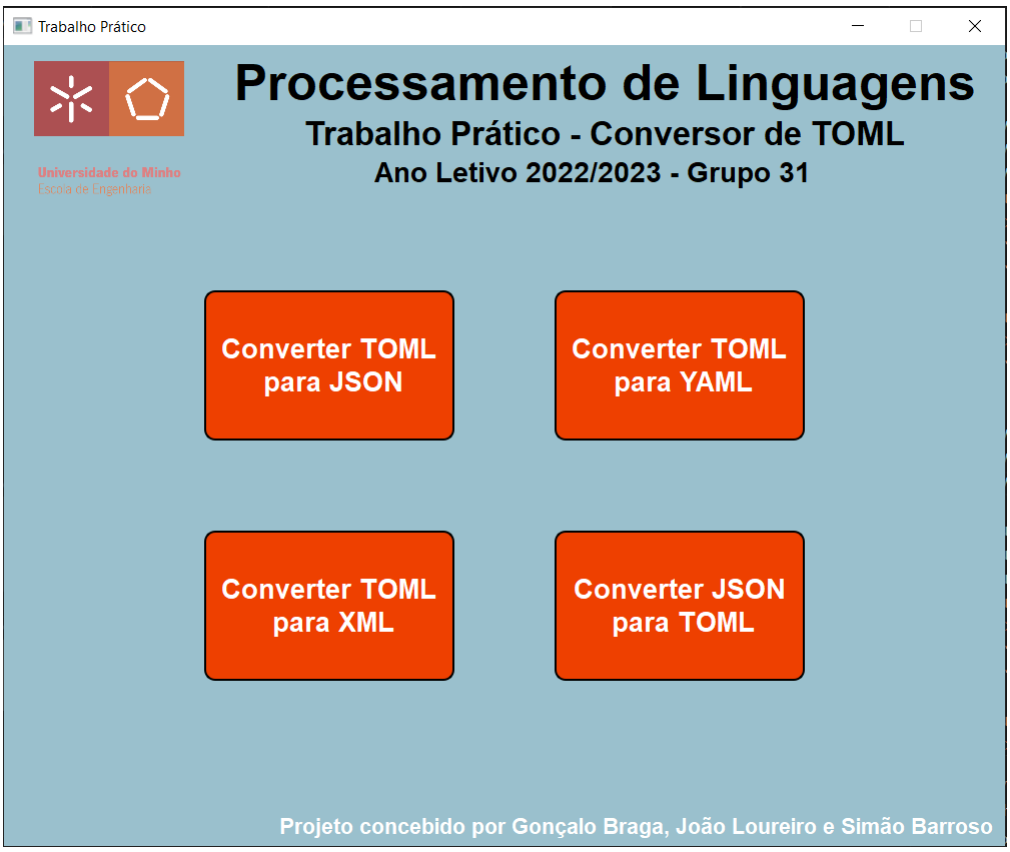

Figura 3.4: Página principal

Escolhido o conversor, é apresentada a página de conversão correspondente. Esta página é construida através de um *template* genérica que é preenchida com os dados correspondentes (tipo de *input* e de *output*, quais as funções aquando a conversão e se pode ou não guardar o resultado em ficheiro):

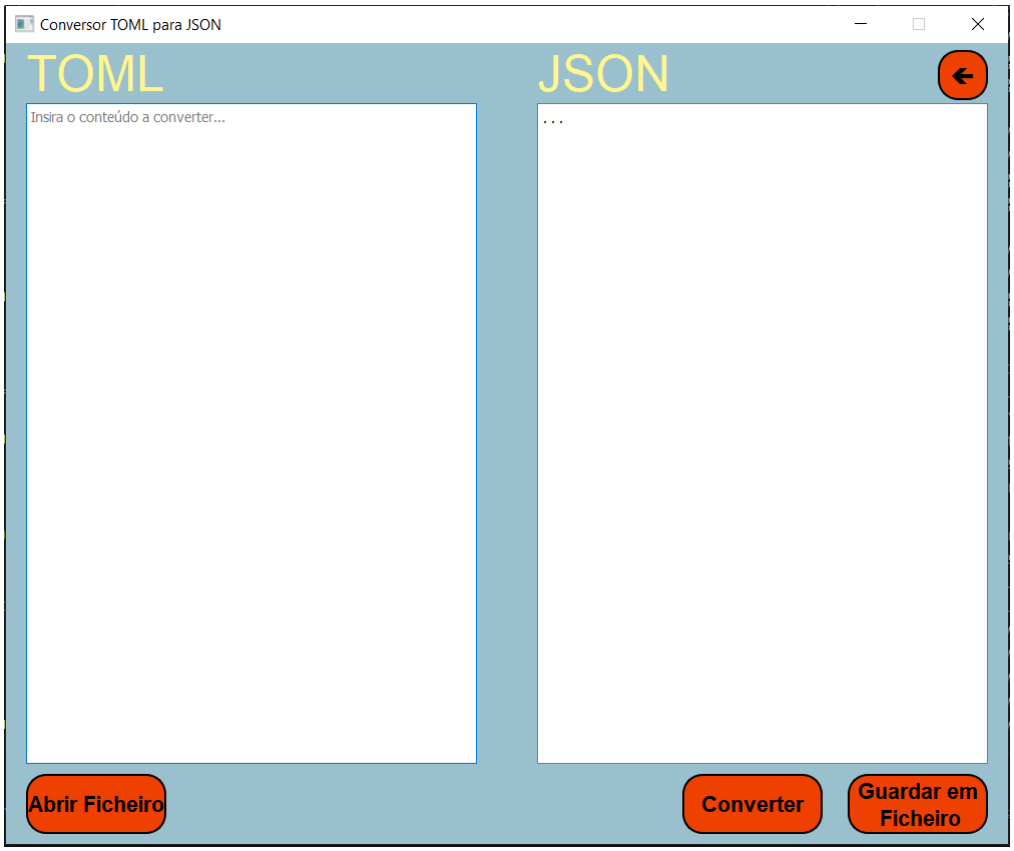

Figura 3.5: Uma página de conversão (*TOML* para *JSON*, neste caso)

O utilizador pode então escrever diretamente na *textbox* da esquerda o conteúdo que pretende converter ou carregar no botão **"Abrir Ficheiro"** para abrir um ficheiro à sua escolha, cujo conteúdo é transferido para a mesma.

Feito isto, o utilizador apenas necessita de carregar no botão **"Converter"** para que o seu *input* seja convertido para o tipo do *output* desejado.

Se o tipo de *input* for TOML, é criada uma instância da classe **Conversor** (explorada anteriormente), cujo *output* é utilizado como base para criar o *output* desejado pelo o utilizador, através de funções auxiliares.

Caso contrário, simplesmente é utilizada a função correspondente à conversão desejada, sem a utilização da **Conversor**.

De qualquer das maneiras, o resultado é impresso na *textbox* da esquerda e apresentado ao utilizador. No caso da conversão ser de *TOML* para *JSON*, o utilizador pode ainda optar por guardar a conversão em um ficheiro *.json*, cuja localização e nome, que tem que ser terminado por *.json*, é determinado por este mesmo através de uma janela.

Apenas implementamos esta funcionalidade para a conversão de *TOML* para *JSON* devido a esta ser o foco principal do projeto, bem como algumas dificuldades técnicas que surgiram com o menu na fase de desenvolvimento devido a esta funcionalidade.

Se o utilizador desejar voltar ao menu principal, pode o fazer a qualquer altura através do botão **Voltar** localizado no canto superior direito.

De forma a controlar a apresentação das páginas, utilizamos uma *stack* de *widgets* (elementos da *interface*), *QtWidgets.QStackedWidget()*, que já se encontra pré-definida no módulo do *PyQt5*. Ao inicializar uma nova instância de uma página, esta é inserida na *stack* e apresentada ao utilizador, e caso o utilizador deseje voltar

atrás, é removida a página atual da *stack*, e apresentada a seguinte nela inserida.

Outro aspeto a notar sobre a *interface* é que podem existir certas combinações consecutivas de ações tomadas pelo o utilizador que podem resultar na terminação abrupta do programa, pelo o qual não conseguimos determinar a razão destes acontecerem. O mesmo acontece caso o utilizador tente converter algo de um tipo diferente do tipo exigido pelo o *input*, pelo o que o utilizador deverá ter cuidado com o que quer converter.

Tirando estas situações, a *interface* encontra-se funcional sendo, na nossa opinião, uma forma mais acessível e apelativa de utilizar e testar o nosso projeto.

4 Aspetos Positivos e Negativos

Nesta secção vamos falar de alguns aspetos positivos e negativos do trabalho.

Relativamente a aspetos negativos, o primeiro a abordar tem a ver com a nossa resolução do problema. Achamos que apesar de resultar e estar relativamente eficiente, havia aspetos que podíamos melhorar, principalmente devido ao facto de percorremos o ficheiro duas vezes. Isto deve-se ao facto de termos começado o trabalho antes de termos noção completa do que é o *ply* e cremos que isso notou-se.

Relativamente aos aspetos positivos, ficamos satisfeitos uma vez que conseguimos percorrer a documentação toda do *TOML* e colocar-la a funcionar no nosso trabalho. Também o facto de termos feito de termos feito extras como interface e a tradução para outras linguagens é um aspeto bastante positivo.

5 Trabalho Futuro / Melhorias

Alguns dos aspetos que podemos melhorar e/ou fazer num futuro próximo:

- Criação de testes automatizados : a criação de testes automáticos iria ajudar mais num trabalho futuro e até ajudava a entender melhor a passagem de *TOML* para *JSON*.
- Adição de aspetos extra *TOML*: novas funcionalidades, novas especificações , entre outros.
- Compactar o programa de maneira que possa se tornar um executável.
- Acrescentar outras linguagens e conversões: entendido com o fim deste trabalho o processo de conversão é agora um assunto do qual nos interessamos.
- Melhoramento da *interface* e adição de novas opções a estas (customizar indentação do *output*, opção de guardar fichreiro para todos os conversores, etc...).

6 Conclusão

Em suma, apesar dos altos e baixos que tivemos ao longo do desenvolvimento desta aplicação, estamos bastantes satisfeitos com o resultado final. Cremos que aprendemos e compreendemos o que era para ser feito e a matéria lecionada ao longo da cadeira.

No decorrer deste projeto, adquirimos habilidades valiosas em processamento de linguagens, programação em *Python* e resolução de problemas. Aprendemos a projetar e implementar gramáticas, a construir análises léxicas e sintáticas robustas e a transformar dados de maneira eficiente. Essas habilidades serão benéficas em futuros projetos e pesquisas relacionadas a processamento de linguagens e desenvolvimento de ferramentas.

Além do conversor *TOML* para *JSON*, expandimos nosso projeto para incluir conversores adicionais, como *TOML* para *YAML* e *JSON* para *TOML*. Essa ampliação permitiu-nos aprofundar o nosso conhecimento em processamento de linguagens e explorar diferentes abordagens para a conversão entre formatos.