

Processamento de Linguagens (3.º ano de Curso)
Trabalho Prático

Relatório de Desenvolvimento
Tema 2.6 - Conversor toml-json

Grupo 10 - MOP

Miguel Silva Pinto
A96106

Orlando José da Cunha Palmeira
A97755

Pedro Miguel Castilho Martins
A97613

5 de junho de 2023

Resumo

Este relatório foi elaborado no âmbito da cadeira de Processamento de Linguagens, em que descrevemos os vários aspetos do projeto realizado para a disciplina, que consiste num conversor de ficheiros de texto em TOML para o formato JSON.

Conteúdo

1	Introdução	3
1.0.1	Enquadramento e Contexto	3
1.0.2	Problema e Objetivos	3
1.0.3	Estrutura do Relatório	4
2	Análise e Especificação	5
2.1	Descrição informal do problema	5
2.2	Especificação do Requisitos	5
3	Concepção/desenho da Resolução	6
3.1	Análise Léxica	6
3.1.1	Estados	7
3.1.2	Tratamento dos <i>Tokens</i>	7
3.2	Gramática	10
4	Testes realizados e Resultados	17
4.1	Testes	17
4.2	Mensagens de erro	18
5	Conclusão	19
A	Testes	21
A.1	<i>Array</i>	21
A.2	<i>Bool</i>	22
A.3	<i>Comment</i>	22
A.4	<i>Datetime</i>	24
A.5	<i>Float</i>	24
A.6	<i>Integer</i>	24

A.7	<i>Inline tables</i>	25
A.8	<i>Key</i>	28
A.9	<i>Spec</i>	30
A.10	<i>String</i>	31
A.11	<i>Table</i>	33
A.12	<i>Others</i>	34
B	Mensagens de erro	36
B.1	Chave duplicada na <i>Inline Table</i>	36
B.2	Chave duplicada	36
B.3	Token EOF inesperado	37
B.4	Redefinição de valor	37
B.5	Redefinição de valor em tabela	38

Capítulo 1

Introdução

1.0.1 Enquadramento e Contexto

Área: Processamento de Linguagens

Um programa capaz de realizar a conversão de ficheiros TOML para JSON pode ser útil em diversas situações, principalmente no desenvolvimento de software e integração de sistemas.

TOML é uma linguagem de serialização de dados que se concentra na facilidade de leitura e de escrita por humanos, enquanto JSON é um formato amplamente utilizado para troca de dados entre aplicações *web*. Portanto, a conversão de ficheiros TOML para JSON permite que esses dados sejam mais facilmente manipulados e integrados em sistemas que utilizam o formato JSON.

Por exemplo, se uma empresa possui um sistema que utiliza JSON para troca de dados entre diferentes aplicações, mas recebe dados de um fornecedor em formato TOML, essa conversão pode ser necessária para integrar esses dados no sistema existente. Em resumo, um projeto de conversão de ficheiros TOML para JSON pode ser útil em diversas situações em que a interoperabilidade entre sistemas que utilizam esses formatos diferentes é necessária.

1.0.2 Problema e Objetivos

Neste projeto, o problema consistiu em desenvolver um programa capaz de converter ficheiros TOML para o formato JSON.

Com a realização deste projeto, os seus principais objetivos foram:

- aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), reforçando a capacidade de escrever gramáticas, quer independentes de contexto (GIC), quer tradutoras (GT);
- desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora;
- utilizar geradores de compiladores baseados em gramáticas tradutoras, concretamente o Yacc, versão PLY do Python, completado pelo gerador de analisadores léxicos Lex, também versão PLY do Python.

1.0.3 Estrutura do Relatório

Este relatório está organizado em cinco capítulos diferentes.

No capítulo 1, **Introdução**, faz-se uma introdução ao trabalho, relatando o seu enquadramento, contexto e objetivos.

No capítulo 2, **Análise e Especificação**, é descrito informalmente o problema, sendo feita uma breve especificação dos requisitos do projeto.

No capítulo 3, **Desenho da Resolução**, detalhamos as estratégias adotadas para o desenvolvimento do trabalho, com a descrição de grande parte da lógica do nosso programa, apresentando a implementação da análise léxica e da análise sintática, como também se especificam certos detalhes técnicos relativos ao programa.

No capítulo 4, **Testes realizados e Resultados**, são apresentados os diversos testes a que submetemos o nosso programa, bem como os seus resultados.

No capítulo 5, **Conclusão**, é feita uma análise final do nosso projeto, concluindo o nosso relatório.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

Criar um programa que traduza ficheiros no formato TOML para o formato JSON, através da elaboração de uma gramática tradutora capaz de gerar um ficheiro JSON com a correspondente informação do ficheiro TOML.

2.2 Especificação do Requisitos

De maneira a abordar o problema em questão, foram definidos os requisitos a cumprir ao longo da sua elaboração:

- Definir os *tokens* da linguagem TOML
- Desenvolver as regras da gramática
- Estabelecer as estruturas de dados a utilizar para gerar o resultado.
- Redigir um relatório de desenvolvimento

Capítulo 3

Concepção/desenho da Resolução

Após a fase de análise e especificação, passamos para a parte da implementação. A implementação do programa baseia-se em duas partes fundamentais, a **análise léxica** e a **análise sintática**, elaboradas através da utilização de dois módulos, *lex* e *yacc* da ferramenta **PLY** do Python.

Com o objetivo de aprimorar a organização das nossas estruturas e alcançar um resultado final mais estruturado, optamos por implementar **classes** para o Lexer e o Parser do nosso programa. Esta abordagem segue princípios similares aos ensinados nas aulas de Processamento de Linguagens, mas com o benefício adicional de oferecer **encapsulamento** e **organização** que são essenciais em projetos de maior complexidade.

Nesta secção do relatório, serão explicadas com algum detalhe as implementações relativas à parte **léxica** e à parte **gramatical**, sendo também demonstrados certos aspetos do comportamento do programa final.

3.1 Análise Léxica

Primeiramente foram definidos os *tokens* da linguagem:

- KEY → *Strings* que representam a chave dos pares chave-valor.
- STRING → Tipo de valor delimitado entre aspas ou plicas.
- INT → Tipo de valor com um valor numérico inteiro.
- FLOAT → Tipo de valor com um valor numérico de vírgula flutuante.
- BOOL → Tipo de valor com um booleano.
- DATETIME → Tipo de valor com uma representação de uma data.
- TABLE → Nome de uma tabela.
- NEWLINE → Uma ou mais quebras de linha, usado para fazer o controlo das restrições de linhas.
- EOF → *Token* indicador de fim de ficheiro, usado para fazer o controlo das restrições de linhas.

Para além dos *tokens*, foi também necessário especificar os literais que fazem parte da linguagem TOML.

- '[' , ']' → Definem *arrays* e tabelas
- '{' , '}' → Definem *inline tables*
- '.' → Aninha tabelas.
- '=' → Atribuição de valores.
- ',' → Separação de elementos de *arrays* e dicionários.

3.1.1 Estados

O analisador léxico do nosso trabalho tem cinco estados possíveis, a saber:

1. **INITIAL**
2. **RTABLE**
3. **RVALUE**
4. **RDICT**
5. **RARRAY**

Para gerir as transições entre os estados, foi necessário utilizar uma lógica de uma *stack* de estados, de forma a manter em memória o estado prévio para o qual se pretende regressar. Para isso, foram utilizados os métodos *push_state* e *pop_state*, disponibilizados pelo módulo Lex.

O estado **INITIAL** é o estado padrão do *lexer* (sendo que é o estado com que este é iniciado). Neste estado ocorre a leitura de pares chave-valor em que, após a leitura de uma chave, se for encontrado um carácter '=', é feito um *push* do estado **RVALUE** que encarregar-se-á de ler o valor associado à chave. Se for detetado um carácter '[', inicia a leitura do nome de uma tabela (*push* do estado **RTABLE**).

O estado **RTABLE** serve para ler nomes de tabelas e só é acionado pelo estado **INITIAL** quando este encontra o carácter '['. Este estado foi criado para limitar os *tokens* que podem aparecer nos nomes de tabelas, permitindo apenas capturar os caracteres '[', ']' e *strings*, eventualmente divididas por '.', que determinam o nome da tabela.

O estado **RVALUE** serve exclusivamente para ler valores concretos (números, *strings*, booleanos, etc...). Este estado é apenas ativado após a leitura de um *token* '='. É importante referir que, se a seguir a uma chave estiver uma lista ou um dicionário, o estado **RVALUE** é retirado da *stack* e substituído pelo estado **RARRAY** ou **RDICT**, respetivamente, uma vez que o *array* ou dicionário serão o valor em questão.

O estado **RARRAY** serve para a captura de *tokens* de listas. Este estado é apenas ativado após a leitura do carácter '[' (nos estados **RVALUE**, **RARRAY** e **RDICT**) e está preparado para capturar valores atómicos (*strings*, números, etc...) bem como outras listas e dicionários. Este estado tem uma característica diferente dos restantes, o facto de ignorar *newlines*, uma vez que um *array* pode abranger várias linhas, podendo assim simplificar a gramática de construção de *arrays*.

O estado **RDICT** serve para a captura de *tokens* de dicionários (tabelas *inline*). Este estado é apenas ativado após a leitura do carácter '{' (nos estados **RVALUE**, **RARRAY** e **RDICT**) e está preparado para *tokens* de pares chave-valor. Neste estado, quando é detetado um *token* 'KEY', é feito um *push* do estado **RVALUE** que se encarrega de processar o valor associado à chave, reaproveitando as regras utilizadas para ler valores, definidas pelo estado **RVALUE**.

3.1.2 Tratamento dos *Tokens*

Há casos em que *tokens* definidos no *Lexer* apresentam formatos diferentes, porém, são considerados o mesmo tipo de *token*. Exemplos disto são as *strings* com quatro tipos diferentes possíveis ou as **datas** com os seus diferentes formatos. De maneira a melhor organizar o tipo de tratamento que cada "sub-tipo" necessita, dividimo-los em diferentes regras, sendo no final transformados no seu respetivo tipo de *token* mais geral.

KEY

O *token* **KEY** representa todas as chaves dos pares chave-valor que aparecerem num TOML. O tratamento necessário, apenas consiste na eventual remoção de aspas delimitadoras que possam ser usadas para definir uma chave.

STRING

O *token* **STRING** apresenta características mais complexas que os outros *tokens*, uma vez que existem quatro maneiras diferentes de representar *strings*: *basic*, *multi-line basic*, *literal*, *multi-line literal*.

Basic strings são delimitadas por aspas (”), podendo apenas estar contidas numa só linha. Nestas *strings* podem estar contidos quaisquer caracteres Unicode, à exceção dos que têm de ser *escaped* como o *literal*, e as próprias aspas.

Multi-line basic strings são delimitadas por três aspas em cada lado, podendo ter uma ou duas aspas seguidas em qualquer lugar da *string*, e permitem que a *string* abranja várias linhas. Um *newline* imediatamente após o delimitador de abertura será removido. Todos os outros espaços em branco e caracteres *newline* permanecem intactos. Para além destas características, este tipo de *string* permite escrever *strings* longas sem introduzir espaços em branco supérfluos ao usar um *backslash* (\) no fim de uma linha. Com isto, todos os espaços em branco, incluindo *newlines*, são removidos até ser encontrado um carácter que não seja espaço branco ou *newline*.

Literal strings são delimitadas por plicas (’), e tal como as *basic strings* só podem estar numa linha. O que distingue estas *strings* das básicas, é o facto que não há *escaping* nestas *strings*, ou seja, o que se escreve é o que se obtém.

Multi-line literal strings são delimitadas por três plicas permitindo que a *string* abranja várias linhas. Tal como *literal strings* não é feito qualquer *escaping*, e também como nas *multi-line basic strings*, um *newline* imediatamente após o delimitador de abertura será removido e pode-se escrever uma ou duas plicas consecutivas em qualquer lugar da *string*.

INT

O *token* **INT** captura todos os inteiros, tendo em conta todas as suas possíveis variantes:

- Os números podem ter um sinal (+ ou -) como prefixo.
`int1 = +99`
`int2 = -17`
- Os números podem ter *underscores* entre eles.
`int3 = 5_349_221`
- Os inteiros podem ser representados em hexadecimal, octal e binário.
`hex = 0xdeadbeef`
`oct = 0o755`
`bin = 0b11010110`

Todos os *tokens* que obedeçam a estes diferentes formatos, são convertidos para o tipo de dados *inteiro*, incluindo os números com representação não decimal que são convertidos para a representação decimal.

Na documentação do TOML, são também especificadas algumas restrições relativas aos inteiros que são respeitadas no nosso projeto. Não devem ser aceites números com zeros à esquerda e números inteiros de 64 bits (de -2^{63} a $2^{63} - 1$) devem ser aceites e tratados sem perdas, mas se um inteiro não puder ser representado sem perdas, um erro é gerado.

FLOAT

O *token* **FLOAT** captura todos os valores de vírgula flutuante. Estes valores podem ser representados com os seguintes formatos:

- Tal como nos inteiros, os *floats* podem ter sinais como prefixos.
`flt1 = +1.0`
`flt2 = -0.01`
- Os *floats* podem ter *underscores* entre eles.
`flt3 = 224_617.445_991`
- Os *floats* podem ter uma parte exponencial com um "e" ou "E" seguido de uma parte inteira.
`flt4 = 5e+22`
`flt5 = 1e06`
`flt6 = -2E-2`
- Os *floats* podem também ser representados com valores especiais *inf* e *nan*.
`sf1 = inf`
`sf2 = -inf`
`sf3 = nan`

Todos estes valores serão convertidos para o tipo de dados *float*, de maneira a que estejam corretamente apresentados no ficheiro JSON resultante.

BOOL

O *token* **BOOL** representa valores de verdadeiro e falso que possam aparecer num ficheiro TOML. Este *token* apenas toma o valor "true" ou "false" pelo que o único tratamento que se faz numa *string* deste *token* é a conversão no valor booleano correspondente.

DATETIME

O *token* **DATETIME** representa datas (que podem incluir horas e fusos horários) ou apenas horas. Estes são alguns valores exemplificativos que este *token* pode tomar:

- 1979-05-27T07:32:00Z - Data e hora no formato ISO8601 (a letra Z indica que a data e hora estão no fuso horário UTC).
- 1979-05-27T00:32:00-07:00 - Data e hora no formato ISO8601 em que é indicado um atraso de 7 horas em relação ao UTC.
- 1979-05-27 - Datas no formato AAAA-MM-DD
- 07:32:00 - Horas no formato HH:MM:SS

Todos os valores que este *token* possa ter sofrerão o mesmo tratamento, independentemente do formato. O tratamento é feito através da função `parse` do módulo `dateutil.parser` que retorna um objeto `datetime`.

No caso do *token* só conter uma data ou uma hora, guardamos apenas a data ou a hora desse objeto `datetime`. No final, este objeto será transformado em *string*, de modo a que seja compatível com o formato JSON.

TABLE

O *token* **TABLE** representa nomes de tabelas TOML. Por exemplo, na *string* "[tabela1]", o *token* TABLE terá o valor "tabela1". O tratamento destes *tokens*, tal como nas chaves dos pares chave-valor, apenas consiste na eventual remoção de aspas delimitadoras que possam ser usadas para definir um nome de uma tabela.

3.2 Gramática

Após definirmos os *tokens* que vão ser passados ao *parser*, passamos para a definição das regras da nossa **gramática tradutora** capaz de interpretar TOML e transformar em JSON, e que satisfaça a condição LR. De maneira a facilitar a deteção de alguns erros, demos preferência à implementação de **produções gramaticais** com **recursividade à esquerda**.

Durante a análise sintática do ficheiro TOML, iremos criar um **dicionário** em Python que se irá assemelhar ao objeto JSON desejado, tirando partido da compatibilidade entre as estruturas de dados em **Python** e um objeto **JSON**.

Para isso criamos algumas funções que nos permitiram agrupar os valores obtidos na análise léxica, em estruturas de dados de Python, que terão a estrutura do objeto JSON requerido e que serão melhor explicados no contexto das regras gramaticais envolvidas.

Axioma

O axioma da nossa gramática é o símbolo não terminal "**toml**", podendo ser derivado nestas 5 regras gramaticais.

```
1 def p_toml_eof(self, p):
2     'toml : toml EOF'
3     p[0] = p[1]
4
5 def p_newlines_toml(self, p):
6     'toml : newlines toml'
7     p[0] = p[2]
8
9 def p_toml_kvps_tables(self, p):
10    'toml : kvaluepairs tables'
11    p[0] = merge_dictionaries([p[1], p[2]])
12
13 def p_toml_kvps_or_tables(self, p):
14    '''
15    toml : kvaluepairs
16          | tables
17    '''
18    p[0] = p[1]
```

```
19
20 def p_empty_toml(self, p):
21     'toml : '
22     p[0] = {}
```

No cenário mais abrangente de um ficheiro TOML, o axioma fundamental será composto por "**kvaluepairs**" e "**tables**", definido pela regra *p_toml_kvps_tables*. Os pares chave/valor representam as entradas da **tabela de nível superior**, conhecida como a tabela de topo, conforme definida na documentação do TOML. As tabelas serão compostas por uma série de pares chave-valor que permitem a definição de valores específicos.

A estrutura final desta regra será resultado da aplicação da função **merge_dictionaries**, que combina os dados dos símbolos não terminais "kvaluepairs" e "tables", na estrutura de dados adequada.

No entanto, um ficheiro TOML pode também ser só constituído por **pares chave/valor** ou só **tabelas**, ou até não ter **nada** definido nele. Nestes casos o símbolo "toml" terá apenas o valor do símbolo que lhe for constituinte, sendo um dicionário vazio no caso do ficheiro não ter nada.

Uma vez que *tokens* como *newlines* e EOF são retornados pelo lexer, com o intuito de controlar restrições de linhas relacionadas à sintaxe de alguns elementos, foram criadas as regras *p_toml_eof* e *p_newlines_toml* que simplesmente tratam de casos específicos que é necessário ter em atenção ao ter estes *tokens*.

Em todas estas regras, o valor final de p[0], o axioma, irá ser a estrutura de dados Python final que é equivalente ao objeto JSON desejado.

Pares chave/valor

Os **pares chave/valor** são um bloco de construção fundamental de um documento TOML, estando presentes em grande parte das estruturas definidas no TOML.

Primeiramente, como uma chave pode ser encadeada em sequência com outras chaves através da utilização do ponto ("."), definimos este conjunto de produções.

```
1 def p_key(self, p):
2     'key : KEY "." key'
3     p[0] = [p[1]] + p[3]
4     p.set_lineno(0, p.lineno(1))
5     p.set_lexpos(0, p.lexpos(1))
6
7 def p_single_key(self, p):
8     'key : KEY'
9     p[0] = [p[1]]
10    p.set_lineno(0, p.lineno(1))
11    p.set_lexpos(0, p.lexpos(1))
```

Este conjunto de regras gramaticais, agrupam todas as chaves pontilhadas numa lista, sendo esse o valor armazenado no símbolo não terminal "**key**", sendo esta lista utilizada para gerar o objeto resultante da junção de uma chave com o seu valor, estando a ação implementada na produção geradora do símbolo não gramatical "**kvaluepair**" que representa um **par chave/valor**.

```
1 def p_kvaluepair(self, p):
2     'kvaluepair : key "=" value'
```

```
3     p[0] = calcObject(p[1],p[3])
4     p.set_lineno(0, p.lineno(1))
5     p.set_lexpos(0, p.lexpos(1))
```

A função *calcObject* recebendo uma lista de chaves e o respetivo valor, é capaz de calcular a estrutura de dados equivalente ao formato JSON, permitindo a construção de um objeto JSON com base nos dados fornecidos.

De maneira a agrupar um conjunto de pares chave/valor, foi definido um símbolo não terminal "**kvaluepairs**", que através da função *merge_dictionaries*, é obtido o formato da estrutura de dados pretendida.

```
1 def p_kvaluepairs(self, p):
2     '''
3     kvaluepairs : kvaluepair newlines
4                  | kvaluepairs kvaluepair EOF
5     '''
6     try:
7         p[0] = merge_dictionaries([p[1], p[2]])
8     except myException as e:
9         e.set_lineno(p.lineno(2))
10        e.set_lexpos(p.lexpos(2))
11        raise e
12
13 def p_single_kvaluepairs(self, p):
14     '''
15     kvaluepairs : kvaluepair newlines
16                  | kvaluepair EOF
17     '''
18     p[0] = p[1]
```

A fim de cumprir com a restrição de que um par chave/valor deve estar numa única linha, é necessário que um "kvaluepair" seja seguido por uma nova linha ou pelo *token* EOF, no caso de ser o último par chave/valor do documento, garantindo que a estrutura do documento TOML de entrada esteja correta.

Se algum erro for detetado na *merge_dictionaries*, é gerada uma exceção, indicando a linha e a posição do elemento que causou o erro, sendo posteriormente gerada uma mensagem de erro para o utilizador, que serão melhor exemplificadas em secções posteriores do relatório.

Tabelas

Na linguagem TOML, estão definidos dois tipos de tabelas, tabelas normais e tabelas de *array*. Essas duas tabelas têm sintaxes, sendo definidas nas suas regras gramaticais.

```
1 def p_normaltable(self, p):
2     'normaltable : "[" tablename "]" newlines kvaluepairs'
3     p[0] = calcObject(p[2],p[5])
4     p.set_lineno(0, p.lineno(2))
5
6 def p_empty_normaltable(self, p):
7     '''
8     normaltable : "[" tablename "]" newlines
```

```

9         | "[" tablename "]" EOF
10     '''
11     p[0] = calcObject(p[2],{})
12     p.set_lineno(0, p.lineno(2))
13
14 def p_arraytable(self, p):
15     'arraytable : "[" "[" tablename "]" "]" newlines kvaluepairs'
16     p[0] = calcObjectArrayTable(p[3],p[7])
17     p.set_lineno(0, p.lineno(3))
18
19 def p_empty_arraytable(self, p):
20     '''
21     arraytable : "[" "[" tablename "]" "]" newlines
22                 | "[" "[" tablename "]" "]" EOF
23     '''
24     p[0] = calcObjectArrayTable(p[3],{})
25     p.set_lineno(0, p.lineno(3))

```

Nestas quatro regras, são estabelecidos os símbolos que irão gerar estas tabelas, contemplando os casos em que as tabelas estejam vazias para ambos os tipos de tabela. Para as tabelas normais, a geração da estrutura resultante é feita de maneira similar aos pares chave/valor, através da função *calcObject*.

Já para as tabelas de *array*, recorre-se a uma função especial parecida com a *calcObject* mas que tem em conta as particularidades deste tipo de tabela, sendo usada a *calcObjectArrayTable* colocando os pares chave/valor num *array*.

De notar que as tabelas são apenas válidas se após o seu cabeçalho, estiverem *newlines* ou um *token* EOF, tal como era exigido aos "kvaluepairs" (pares chave/valor), estando isso contemplado nas produções que geram as tabelas.

De modo a não tornar o relatório repetitivo, não iremos mostrar todas as regras gramaticais relativas a geração do símbolo "tablename" uma vez que tem um processo de criação parecido com o símbolo "key", uma vez que o TOML indica que as mesmas regras das chaves dos pares chave/valor, são aplicadas nos nomes dos cabeçalhos de tabelas.

De maneira a juntar estas tabelas no símbolo "**tables**" usado na geração do axioma, foram elaboradas as seguintes regras gramaticais.

```

1 def p_tables_normalt(self, p):
2     'tables : tables normaltable'
3     try:
4         p[0] = merge_tables([p[1],p[2]])
5     except myException as exc:
6         exc.set_linetable(p.lineno(2))
7         raise exc
8
9 def p_tables_arrayt(self, p):
10    'tables : tables arraytable'
11    try:
12        p[0] = merge_tables([p[1],p[2]])
13    except myException as exc:
14        exc.set_linetable(p.lineno(2))
15        raise exc

```

```
16
17 def p_single_tables(self, p):
18     '''
19     tables : arraytable
20             | normaltable
21     '''
22     p[0] = p[1]
```

Como podemos ver, nestas produções foi usada recursividade à esquerda, tal como foi previamente enunciado como a nossa preferência, uma vez que nos permite ter uma maneira de manter uma noção de ordem durante a análise sintática onde os símbolos vão sendo reduzidos. Assim, se algum elemento causar um erro, esse elemento é facilmente identificável, podendo gerar uma mensagem de erro mais sugestiva que indique onde o utilizador deve corrigir o problema.

Valores

Um valor é um símbolo extremamente abrangente, podendo assumir o valor de qualquer estes *tokens*:

- STRING
- INT
- FLOAT
- BOOL
- DATETIME

Para que sejam considerados como valores, foram criadas estas regras gramaticais simples que passam o valor do respetivo *token* para o símbolo "**value**" que é usado na produção de um "**kvaluepair**".

```
1 def p_value_int(self, p):
2     'value : INT'
3     p[0] = p[1]
4
5 def p_value_float(self, p):
6     'value : FLOAT'
7     p[0] = p[1]
8
9 def p_value_string(self, p):
10    'value : STRING'
11    p[0] = p[1]
12
13 def p_value_bool(self, p):
14    'value : BOOL'
15    p[0] = p[1]
16
17 def p_value_datetime(self, p):
18    'value : DATETIME'
19    p[0] = p[1]
```

Mas para além destes *tokens*, um valor pode ser um *array* ou uma tabela *inline* (dicionário). Para definir esse tipo de valor, foi preciso criar regras gramaticais para a deteção de *arrays* e tabelas *inline*.

Aqui estão apresentadas as produções que envolvem a geração de um símbolo não terminal que representa um **array**.

```
1 def p_empty_array(self, p):
2     'array : "[" "]"'
3     p[0] = []
4
5 def p_array(self, p):
6     'array : "[" arraycontent "]"'
7     p[0] = p[2]
8
9 def p_arraycontent(self, p):
10    'arraycontent : arraycontent arrelem'
11    p[0] = p[1] + [p[2]]
12
13 def p_arraycontent_single(self, p):
14    'arraycontent : arrelem'
15    p[0] = [p[1]]
16
17 def p_arrelem(self, p):
18    '''
19    arrelem : value ","
20             | value
21    '''
22    p[0] = p[1]
```

Um **array** será um conjunto de valores (podendo ser de diferentes tipos) delimitado por dois parênteses retos. Estes *arrays* podem abranger **mais que uma linha**, mas como o nosso *tokenizer* consegue detetar que está a ler um *array*, ele ignora *newlines*. Isto significa que não há necessidade de abordar essa questão na análise gramatical.

É também importante destacar que a sintaxe dos *arrays* em TOML permite uma **vírgula de terminação** (também chamada de vírgula à direita) após o último valor do *array*. Para acomodar esta particularidade foi definida a regra gramatical **p_arrelem** em que um símbolo constituinte de um *array* já possa incluir a vírgula como parte dele, englobando a necessidade de haver uma vírgula entre os valores do *array* e a possibilidade do último valor ter uma vírgula de terminação.

Para definir *inline tables*, as regras são semelhantes às da geração de um símbolo "array", mas com algumas diferenças.

```
1 def p_empty_dictionary(self, p):
2     'dictionary : "{" "}"'
3     p[0] = dict()
4
5 def p_dictionary(self, p):
6     'dictionary : "{" dictcontent "}"'
7     p[0] = p[2]
8
9 def p_dictcontent(self, p):
10    'dictcontent : dictcontent "," kvaluepair'
```

```
11     try:
12         p[0] = merge_dictionaries([p[1], p[3]])
13     except myException as e:
14         e.set_lineno(p.lineno(3))
15         e.set_lexpos(p.lexpos(3))
16         raise e
17
18 def p_single_dictcontent(self, p):
19     'dictcontent : kvaluepair'
20     p[0] = p[1]
21     p.set_lineno(0, p.lineno(1))
22     p.set_lexpos(0, p.lexpos(1))
```

Primeiramente, os *newlines* são tidos em conta na leitura de um dicionário, uma vez que uma tabela *inline* está restrita a **uma só linha**. Isto não implica nenhum cuidado específico na gramática, apenas é previsto que *newlines* sejam retornados, e se algum *newline* for detetado, a gramática automaticamente irá gerar um **erro**.

Outra diferença, é o facto que uma tabela *inline* não pode ter uma vírgula de terminação como os *arrays*.

Como a estrutura sintática de um par chave/valor do TOML é a mesma de um par chave/valor de uma tabela *inline*, reaproveitamos o símbolo não terminal "**kvaluepair**" para defini-lo como um **elemento** do dicionário (símbolo "**dictcontent**"). À medida que se progride na sequência de símbolos "**kvaluepair**", estes vão sendo agregados num dicionário, sendo esse dicionário o valor do símbolo não terminal "**dictionary**" que representa uma **tabela inline**.

Com estes símbolos não terminais definidos, *array* e *dictionary*, estes podem passar a ser tratados como valores através da definição das seguintes produções.

```
1 def p_value_array(self, p):
2     'value : array'
3     p[0] = p[1]
4
5 def p_value_dictionary(self, p):
6     'value : dictionary'
7     p[0] = p[1]
```

Capítulo 4

Testes realizados e Resultados

4.1 Testes

Para testar o funcionamento do nosso conversor, recorreremos à utilização de testes unitários. Esses testes foram implementados com recurso ao módulo `unittest` e foram obtidos através deste repositório do GitHub: <https://github.com/BurntSushi/toml-test/tree/master/tests>.

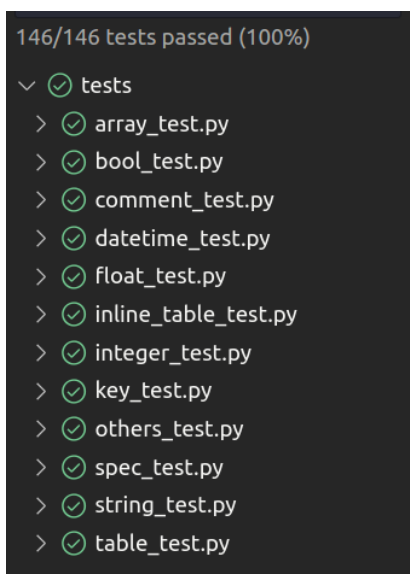


Figura 4.1: Testes unitários

Nestes testes encontram-se diversos tipos de ficheiros TOML, juntamente com o seu respetivo resultado esperado em JSON, abrangendo todos os aspetos da linguagem TOML.

No apêndice A é possível consultar uma lista de testes efetuados com alguns dos ficheiros TOML que consideramos mais relevantes, mostrando o ficheiro TOML e o resultado obtido pelo nosso programa em JSON.

4.2 Mensagens de erro

Levando em consideração a perspectiva do utilizador, decidimos implementar um sistema gerador de **mensagens de erro**.

Para tal, utilizamos funcionalidades de rastreio de linha e posição disponibilizadas pelo PLY. Apesar do PLY fornecer uma funcionalidade opcional de rastreio de todos os símbolos gramaticais, decidimos usar apenas os métodos *set_lineno* e *set_lexpos* para apenas propagar a informação de posição entre símbolos gramaticais específicos, em vez de ter essa informação propagada por todos os símbolos gramaticais, exigindo processamento extra e diminuindo o desempenho do programa.

No **apêndice B** é possível ver alguns **exemplos** das diferentes mensagens de erro que o nosso programa é capaz de gerar.

Com a informação da linha e da posição do símbolo que levantou um problema, somos capazes de determinar a coluna em que o erro acontece, permitindo ao utilizador localizar o problema no seu ficheiro TOML. Pelos exemplos, é possível ver que em alguns casos de erro a mensagem de erro indica exatamente onde o erro ocorre na linha, apontando a área com o símbolo "^^".

Capítulo 5

Conclusão

Dada por concluída a realização do trabalho prático, podemos dizer que o projeto foi essencial para o desenvolvimento e aprimoramento dos nossos conhecimentos relativos à área de processamento de linguagens, desde a aplicação de expressões regulares, à escrita de gramáticas.

Ao longo do projeto deparamo-nos com algumas dificuldades, destacando a expressão regular necessária para a captura de *strings* com aspas escapadas, e a lógica necessária para a correta integração das *array tables* do TOML nas nossas estruturas de dados.

Apesar das dificuldades, elas foram superadas, estando bastante satisfeitos com o nosso projeto final, que se demonstrou capaz de converter corretamente 146 diferentes ficheiros que na totalidade abordam todos os aspetos da linguagem TOML. Como trabalho futuro, existe sempre a possibilidade de melhorar a qualidade da apresentação das mensagens de erro que já temos, como também seria possível estender a possibilidade de conversão para outros tipos de formatos para além de JSON, como por exemplo para YAML.

Bibliografia

TOML: English v1.0.0, a. URL <https://toml.io/en/v1.0.0>.

Toml to json, b. URL <https://transform.tools/toml-to-json>.

Apêndice A

Testes

A.1 *Array*

Teste - TOML

```
contributors = [  
  "Foo Bar <foo@example.com>",  
  { name = "Baz Qux", email = "bazqux@example.com", url = "https://example.com/bazqux" }  
]
```

```
# Start with a table as the first element. This tests a case that some libraries  
# might have where they will check if the first entry is a table/map/hash/assoc  
# array and then encode it as a table array. This was a reasonable thing to do  
# before TOML 1.0 since arrays could only contain one type, but now it's no  
# longer.
```

```
mixed = [{k="a"}, "b", 1]
```

Resultado - JSON

```
{  
  "contributors": [  
    "Foo Bar <foo@example.com>",  
    {  
      "name": "Baz Qux",  
      "email": "bazqux@example.com",  
      "url": "https://example.com/bazqux"  
    }  
  ],  
  "mixed": [  
    {  
      "k": "a"  
    },  
    "b",  
    1  
  ]  
}
```

```
]
}
```

A.2 *Bool*

Teste - TOML

```
t = true
f = false
```

Resultado - JSON

```
{
  "t": true,
  "f": false
}
```

A.3 *Comment*

Teste - TOML

```
[section]#attached comment
#[notsection]
one = "11"#cmt
two = "22#"
three = '#'

four = ""# no comment
# nor this
#also not comment""#is_comment

five = 5.5#66
six = 6#7
8 = "eight"
#nine = 99
ten = 10e2#1
eleven = 1.11e1#23

["hash#tag"]
"#!" = "hash bang"
arr3 = [ "#", '#', ""###"" ]
arr4 = [ 1,# 9, 9,
2#,9
,#9
3#]
```



```
,4]
arr5 = [[[["#"],
["#"]]]]#]
]
tbl1 = { "#" = '}'#'}#}}}
```

Resultado - JSON

```
{
  "section": {
    "one": "11",
    "two": "22#",
    "three": "#",
    "four": "# no comment\n# nor this\n#also not comment",
    "five": 5.5,
    "six": 6,
    "8": "eight",
    "ten": 1000.0,
    "eleven": 11.1
  },
  "hash#tag": {
    "#!": "hash bang",
    "arr3": [
      "#",
      "#",
      "###"
    ],
    "arr4": [
      1,
      2,
      3,
      4
    ],
    "arr5": [
      [
        [
          [
            ["#"]
          ]
        ]
      ]
    ],
    "tbl1": {
      "#": "}"#"
```

```

    }
  }
}

```

A.4 *Datetime*

Teste - TOML

```

utc1  = 1987-07-05T17:45:56.1234Z
utc2  = 1987-07-05T17:45:56.6Z
wita1 = 1987-07-05T17:45:56.1234+08:00
wita2 = 1987-07-05T17:45:56.6+08:00

```

Resultado - JSON

```

{
  "utc1": "1987-07-05 17:45:56.123400+00:00",
  "utc2": "1987-07-05 17:45:56.600000+00:00",
  "wita1": "1987-07-05 17:45:56.123400+08:00",
  "wita2": "1987-07-05 17:45:56.600000+08:00"
}

```

A.5 *Float*

Teste - TOML

```

before = 3_141.5927
after  = 3141.592_7
exponent = 3e1_4

```

Resultado - JSON

```

{
  "before": 3141.5927,
  "after": 3141.5927,
  "exponent": 3000000000000000.0
}

```

A.6 *Integer*

Teste - TOML

```

bin1 = 0b11010110
bin2 = 0b1_0_1

```

```

oct1 = 0o01234567
oct2 = 0o755
oct3 = 0o7_6_5

hex1 = 0xDEADBEEF
hex2 = 0xdeadbeef
hex3 = 0xdead_beef
hex4 = 0x00987

```

Resultado - JSON

```

{
  "bin1": 214,
  "bin2": 5,
  "oct1": 342391,
  "oct2": 493,
  "oct3": 501,
  "hex1": 3735928559,
  "hex2": 3735928559,
  "hex3": 3735928559,
  "hex4": 2439
}

```

A.7 *Inline tables*

Teste - TOML

```

inline = {a.b = 42}

many.dots.here.dot.dot.dot = {a.b.c = 1, a.b.d = 2}

a = { a.b = 1 }
b = { "a"."b" = 1 }
c = { a . b = 1 }
d = { 'a' . "b" = 1 }
e = {a.b=1}

[tbl]
a.b.c = {d.e=1}

[tbl.x]
a.b.c = {d.e=1}

[[arr]]
t = {a.b=1}
T = {a.b=1}

```

```
[[arr]]
t = {a.b=2}
T = {a.b=2}
```

Resultado - JSON

```
{
  "inline": {
    "a": {
      "b": 42
    }
  },
  "many": {
    "dots": {
      "here": {
        "dot": {
          "dot": {
            "dot": {
              "a": {
                "b": {
                  "c": 1,
                  "d": 2
                }
              }
            }
          }
        }
      }
    }
  },
  "a": {
    "a": {
      "b": 1
    }
  },
  "b": {
    "a": {
      "b": 1
    }
  },
  "c": {
    "a": {
      "b": 1
    }
  },
  "d": {
    "a": {
```

```

        "b": 1
    }
},
    "e": {
        "a": {
            "b": 1
        }
    },
    "tbl": {
        "a": {
            "b": {
                "c": {
                    "d": {
                        "e": 1
                    }
                }
            }
        }
    },
    "x": {
        "a": {
            "b": {
                "c": {
                    "d": {
                        "e": 1
                    }
                }
            }
        }
    },
    "arr": [
        {
            "t": {
                "a": {
                    "b": 1
                }
            },
            "T": {
                "a": {
                    "b": 1
                }
            }
        },
        {
            "t": {
                "a": {
                    "b": 2
                }
            }
        }
    ]
}

```

```

    },
    "T": {
        "a": {
            "b": 2
        }
    }
}
]
}

```

A.8 *Key*

Teste - TOML

Note: this file contains literal tab characters.

```

name.first = "Arthur"
"name".last = "Dent"

```

```

many.dots.here.dot.dot.dot = 42

```

Space are ignored, and key parts can be quoted.

```

count.a      = 1
count . b    = 2
"count"."c"  = 3
"count" . "d" = 4
'count'.'e'  = 5
'count' . 'f' = 6
"count". 'g' = 7
"count" . 'h' = 8
count.'i'    = 9
count      . 'j'      = 10
"count".k    = 11
"count" . l    = 12

```

```

[tbl]

```

```

a.b.c = 42.666

```

```

[a.few.dots]

```

```

polka.dot = "again?"

```

```

polka.dance-with = "Dot"

```

```

[[arr]]

```

```

a.b.c=1

```

```

a.b.d=2

```

```

[[arr]]

```

```
a.b.c=3
a.b.d=4
```

Resultado - JSON

```
{
  "name": {
    "first": "Arthur",
    "last": "Dent"
  },
  "many": {
    "dots": {
      "here": {
        "dot": {
          "dot": {
            "dot": 42
          }
        }
      }
    }
  },
  "count": {
    "a": 1,
    "b": 2,
    "c": 3,
    "d": 4,
    "e": 5,
    "f": 6,
    "g": 7,
    "h": 8,
    "i": 9,
    "j": 10,
    "k": 11,
    "l": 12
  },
  "tbl": {
    "a": {
      "b": {
        "c": 42.666
      }
    }
  },
  "a": {
    "few": {
      "dots": {
        "polka": {
          "dot": "again?",
          "dance-with": "Dot"
        }
      }
    }
  }
}
```

```

    }
  }
},
"arr": [
  {
    "a": {
      "b": {
        "c": 1,
        "d": 2
      }
    }
  },
  {
    "a": {
      "b": {
        "c": 3,
        "d": 4
      }
    }
  }
]
}

```

A.9 *Spec*

Teste - TOML

```

[[fruits]]
name = "apple"

[fruits.physical] # subtable
color = "red"
shape = "round"

[[fruits.varieties]] # nested array of tables
name = "red delicious"

[[fruits.varieties]]
name = "granny smith"

[[fruits]]
name = "banana"

[[fruits.varieties]]
name = "plantain"

```


Resultado - JSON

```
{
  "fruits": [
    {
      "name": "apple",
      "physical": {
        "color": "red",
        "shape": "round"
      },
      "varieties": [
        {
          "name": "red delicious"
        },
        {
          "name": "granny smith"
        }
      ]
    },
    {
      "name": "banana",
      "varieties": [
        {
          "name": "plantain"
        }
      ]
    }
  ]
}
```

A.10 *String*

Teste - TOML

NOTE: this file includes some literal tab characters.

```
multiline_empty_one = ""
```

A newline immediately following the opening delimiter will be trimmed.

```
multiline_empty_two = ""
""
```

*# \ at the end of line trims newlines as well; note that last \ is followed by
two spaces, which are ignored.*

```
multiline_empty_three = ""\
  ""
```

```
multiline_empty_four = ""\
```

```

\
\
"""

equivalent_one = "The quick brown fox jumps over the lazy dog."
equivalent_two = """
The quick brown \

fox jumps over \
the lazy dog."""

equivalent_three = """\
The quick brown \
fox jumps over \
the lazy dog.\
"""

whitespace-after-bs = """\
The quick brown \
fox jumps over \
the lazy dog.\
"""

no-space = """a\
b"""

# Has tab character.
keep-ws-before = """a          \
b"""

escape-bs-1 = """a \\
b"""

escape-bs-2 = """a \\\
b"""

escape-bs-3 = """a \\\\
b"""

```

Resultado - JSON

```

{
  "multiline_empty_one": "",
  "multiline_empty_two": "",
  "multiline_empty_three": "",
  "multiline_empty_four": "",
  "equivalent_one": "The quick brown fox jumps over the lazy dog.",

```

```

"equivalent_two": "The quick brown fox jumps over the lazy dog.",
"equivalent_three": "The quick brown fox jumps over the lazy dog.",
"whitespace-after-bs": "The quick brown fox jumps over the lazy dog.",
"no-space": "ab",
"keep-ws-before": "a  \tb",
"escape-bs-1": "a \\nb",
"escape-bs-2": "a \\b",
"escape-bs-3": "a \\b\\n b"
}

```

A.11 *Table*

Teste - TOML

```

[[albums]]
name = "Born to Run"

[[albums.songs]]
name = "Jungleland"

[[albums.songs]]
name = "Meeting Across the River"

[[albums]]
name = "Born in the USA"

[[albums.songs]]
name = "Glory Days"

[[albums.songs]]
name = "Dancing in the Dark"

```

Resultado - JSON

```

{
  "albums": [
    {
      "name": "Born to Run",
      "songs": [
        {
          "name": "Jungleland"
        },
        {
          "name": "Meeting Across the River"
        }
      ]
    }
  ],
}

```

```

{
  "name": "Born in the USA",
  "songs": [
    {
      "name": "Glory Days"
    },
    {
      "name": "Dancing in the Dark"
    }
  ]
}
]
}

```

A.12 *Others*

Teste - TOML

```

#Useless spaces eliminated.
title="TOML Example"
[owner]
name="Lance Uppercut"
dob=1979-05-27T07:32:00-08:00#First class dates
[database]
server="192.168.1.1"
ports=[8001,8001,8002]
connection_max=5000
enabled=true
[servers]
[servers.alpha]
ip="10.0.0.1"
dc="eqdc10"
[servers.beta]
ip="10.0.0.2"
dc="eqdc10"
[clients]
data=[["gamma","delta"],[1,2]]
hosts=[
  "alpha",
  "omega"
]

```

Resultado - JSON

```

{
  "title": "TOML Example",
  "owner": {

```

```

    "name": "Lance Uppercut",
    "dob": "1979-05-27 07:32:00-08:00"
  },
  "database": {
    "server": "192.168.1.1",
    "ports": [
      8001,
      8001,
      8002
    ],
    "connection_max": 5000,
    "enabled": true
  },
  "servers": {
    "alpha": {
      "ip": "10.0.0.1",
      "dc": "eqdc10"
    },
    "beta": {
      "ip": "10.0.0.2",
      "dc": "eqdc10"
    }
  },
  "clients": {
    "data": [
      [
        "gamma",
        "delta"
      ],
      [
        1,
        2
      ]
    ],
    "hosts": [
      "alpha",
      "omega"
    ]
  }
}

```

Apêndice B

Mensagens de erro

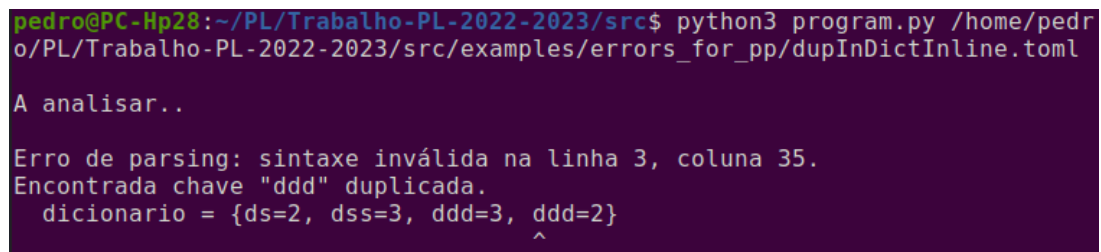
B.1 Chave duplicada na *Inline Table*

Ficheiro TOML inválido

```
test = 1

dicionario = {ds=2, dss=3, ddd=3, ddd=2}
```

Mensagem de erro

A terminal window with a dark background. The prompt is 'pedro@PC-Hp28:~/PL/Trabalho-PL-2022-2023/src\$'. The command executed is 'python3 program.py /home/pedro/PL/Trabalho-PL-2022-2023/src/examples/errors_for_pp/dupInDictInline.toml'. The output shows the program analyzing the file and then reporting a parsing error: 'Erro de parsing: sintaxe inválida na linha 3, coluna 35. Encontrada chave "ddd" duplicada.' followed by the TOML snippet '{ds=2, dss=3, ddd=3, ddd=2}' with a caret pointing to the second 'ddd'.

B.2 Chave duplicada

Ficheiro TOML inválido

```
nome = "Jose"
nome = "Andre"
apelido = "Teixeira"
apelido = "Fontes"
```

Mensagem de erro

```

pedro@PC-Hp28:~/PL/Trabalho-PL-2022-2023/src$ python3 program.py /home/pedro/PL/Trabalho-PL-2022-2023/src/examples/errors_for_pp/duplicateKey.toml

A analisar..

Erro de parsing: sintaxe inválida na linha 2, coluna 1.
Encontrada chave "nome" duplicada.
  nome = "Andre"
    ^

```

B.3 Token EOF inesperado

Ficheiro TOML inválido

```

# unexpected ending
[error

```

Mensagem de erro

```

pedro@PC-Hp28:~/PL/Trabalho-PL-2022-2023/src$ python3 program.py /home/pedro/PL/Trabalho-PL-2022-2023/src/examples/errors_for_pp/eof.toml

A analisar..

Erro de parsing: sintaxe inválida na linha 2, coluna 7.
Fim de ficheiro inesperado.
[error
  ^

```

B.4 Redefinição de valor

Ficheiro TOML inválido

```

# THE FOLLOWING IS INVALID

# This defines the value of fruit.apple to be an integer.
fruit.apple = 1

# But then this treats fruit.apple like it's a table.
# You can't turn an integer into a table.
fruit.apple.smooth = true

```

Mensagem de erro

```

pedro@PC-Hp28:~/PL/Trabalho-PL-2022-2023/src$ python3 program.py /home/pedro/PL/Trabalho-PL-2022-2023/src/examples/errors_for_pp/intToTable.toml

A analisar..

Erro de redefinição da chave "apple", na linha 8, na coluna 1.
  fruit.apple.smooth = true
    ^

```

B.5 Redefinição de valor em tabela

Ficheiro TOML inválido

```

[fruit]
apple = "red"

[fruit.apple]
texture = "smooth"

```

Mensagem de erro

```

pedro@PC-Hp28:~/PL/Trabalho-PL-2022-2023/src$ python3 program.py /home/pedro/PL/Trabalho-PL-2022-2023/src/examples/errors_for_pp/strnoAttr.toml

A analisar..

Erro de redefinição da chave "apple", na tabela da linha 4.

```