Processamento de Linguagens (3º ano de Curso)

Trabalho Prático

Relatório de Desenvolvimento

Bianca Araújo do Vale (A95835)

João Pedro Machado Ribeiro (A95719)

Telmo José Pereira Maciel (A96569)

28 de maio de 2023

Resumo

Neste trabalho iremos desenvolver um compilador de código *TOML* gerando o respetivo código num formato *JSON*, através do uso de um gerador de compiladores baseado em gramáticas e analisadores léxicos, atavés do *Lex/Yacc* do *Ply/Python*.

Conteúdo

Capítulo 1

Introdução

1.1 Conversor toml-json

Área: Processamento de Linguagens

O presente relatório tem como objetivo expor o trabalho prático desenvolvido na Unidade Curricular de Processamento de Linguagens do terceiro ano da Licenciatura em Engenharia Informática.
O trabalho escolhido pelo grupo foi o **Conversor toml-json (2.6)**, sendo que iremos abordar, ao longo do relatório, o processo de resolução do respetivo trabalho.
Para a realização do respetivo projeto, colocamos em prática diversos conhecimentos adquiridos nas aulas teóricas e práticas da respetiva UC: o uso de geradores de compiladores baseados em gramáticas tradutoras, tais como o *Yacc* do *PLY* do *Python* e o gerador de analisadores léxicos *Lex*, também do *PLY* do *Python*.

1.2 Estrutura do Relatório

O presente relatório encontra-se dividido em quatro capítulos:

- O primeiro capítulo refere-se à introdução, que descreve, de forma breve, o problema e retrata o objetivo do trabalho.
- O capítulo dois refere-se à concepção/desenho da resolução, onde o grupo apresenta a estratégia que utilizou para resolver o trabalho, desde os tokens criados até à gramática desenvolvida. Ainda neste capítulo está presente o controlo de erros, onde o grupo explica o que realizou para proceder à ultrapassagem dos mesmos.
- O terceiro capítulo caracteriza-se pelas funcionalidades que o grupo implementou no trabalho prático.
- No quarto e último capítulo está contida uma reflexão crítica, por parte do grupo, acerca do trabalho realizado.

Capítulo 2

Concepção/desenho da Resolução

2.1 Concepção da Linguagem

Primeiramente, para a realização do presente trabalho prático, começamos por desenvolver um analisador léxico com a ferramenta *Lex*. Este tem como objetivo identificar os tokens (definidos pelo grupo).
Posteriormente, procedemos à realização da gramática, que irá ser apresentada com mais detalhe mais à frente.

2.2 Analisador Léxico

O analisador léxico é responsável por analisar o código-fonte caracter por caracter e transformá-lo em sequências de *tokens*, preparando o nosso código para as próximas etapas do processo de compilação.

2.2.1 Tokens

Com o objetivo de conseguirmos representar as diversas funcionalidades do nosso programa, definimos alguns tokens, que se encontram no nosso ficheiro **lex**.
Para além da vírgula (**COMMA**), dos parêntes retos (**LBRACKECT**, **RBRACKECT**) e da igualdade (**EQUALS**), definimos um conjunto de *tokens* que achamos necessários para a resolução do nosso problema, um compilador de linguagem *toml*.
Fora os *tokens* anteriormente referidos, temos também os seguintes:

BASIC_STRING e LITERAL_STRING: Representa todas as strings multi-line;

STRING: Representa uma string;

COMMENT: Representa um comentário;

NUMBER: Representa um número, int ou float;

DATE, TIME e DATE_TIME: Representa todos os tipos de datas e horas suportadas;

BOOLEAN: Representa qualquer palavra do tipo *bool*;

KEY: Representa as chaves dos valores, isto é, tudo que vem atrás do símbolo '=';

OBJECT e SUBOBJECT: Representa todos os objetos da nossa linguagem.

```
tokens = (
    'BASIC_STRING',      'LITERAL_STRING',    'STRING',
    'COMMENT',           'NUMBER',            'IP',
    'DATE',              'TIME',              'BOOLEAN',
    'KEY',               'OBJECT',            'SUBOBJECT',
    'LBRACKET',          'RBRACKET',          'EQUALS',
    'COMMA',             'DATE_TIME',
    )

literals = '-:.",=[]'

t_ignore = ' \t\n'

def t_BASIC_STRING(t):
    r'\'\'\'((.|\n)*?)\'\'\''
    t.value = str(t.value[3:-3])
    return t


def t_LITERAL_STRING(t):
    r'\"\"\"((.|\n)*?)\"\"\"'
    t.value = str(t.value[3:-3])
    return t

def t_BOOLEAN(t):
    r'TRUE|FALSE|true|false'
    t.value = True if t.value == 'true' else False
    return t

def t_IP(t):
    r'\"\d+\.\d+\.\d+\.\d+\"'
    t.value = t.value[1:-1]  # Remove aspas
    return t

def t_DATE_TIME(t):
    r'\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d+)?Z?'
    t.value = str(t.value)
    return t

def t_DATE(t):
    r'\d+\-\d+\-\d+'
    return t

def t_TIME(t):
    r'\d+\:\d+\:\d+'
    return t

def t_NUMBER(t):
    r'-?\d+(?:\.\d+)?(?:[eE][-+]?\d+)?'
    try:
        t.value = int(t.value)
    except ValueError:
        t.value = float(t.value)
```

```
54      return t
55
56  def t_COMMENT(t):
57      r'(\#\s.*)'
58      pass
59      #return t
60
61  def t_STRING(t):
62      r'("[^"]*" | \'[^\']*\' | """[^"]*""" | \'\'\'[^\']*\'\'\')'
63      t.value = t.value[1:-1]  # Remove aspas
64      return t
65
66  def t_KEY(t):
67      r'([a-z_\d]+)\s'
68      return t
69
70  def t_SUBOBJECT(t):
71      r'[a-zA-Z_\d]*\.[a-zA-Z_\d]+'
72      match = re.match(r'([a-zA-Z_\d]*)\.([a-zA-Z_\d]+)', t.value)
73      if match:
74          t.value = match.group(2)
75      return t
76
77  def t_OBJECT(t):
78      r'[a-zA-Z_\d]+'
79      return t
80
81  t_LBRACKET = r'\['
82  t_RBRACKET = r'\]'
83  t_EQUALS = r'='
84  t_COMMA = r','
```

2.3 Analisador Sintático

Após o desenvolvimento do analisador léxico, pasamos para a elaboração de um analisador sintático.

Um analisador sintático é responsável por analisar a estrutura gramatical do código-fonte de um programa, aplicando as regras sintáticas definidas. Este constrói uma árvore sintática que captura a estrutura hierárquica do programa.

2.4 Gramática

Tendo os tokens todos definidos e apresentados, passaremos de seguida a demonstrar a nossa gramática e como o código da linguagem *toml* é traduzido para a linguagem que nós queremos *json*. De maneira a melhor entender a forma como se encontra estruturada, a gramática será apresentada em diferentes secções.

É importante referir que se trata de uma gramática LR, ou seja, *Bottom-up*, de forma a podermos usufruir ao máximo das funcionalidades do PLY.

Apesar de normalmente apenas serem apresentadas produções de gramática, é nesta que também estão estabelecidas as declarações necessárias para a nossa gramática criar um dicionário em *python*, facilitando, assim, mais tarde, a nossa conversão para a linguagem desejada.

De referir, também, que todos os valores que tiverem em letra maiúscula são simbolos terminais (apresentados anteriormente no lex) e os restantes são não terminais.

```
1
2  # P1  : FT                : Section_list
3  # P2  : Section_list      : Section section_list
4  # P3  :                   | Section
5  # P4  : Section           : '[' Section_name ']' Section_content
6  # P5  :                   | '[' Section_name ']' Subsection
7  # P6  :                   | Content
8  # P7  : Table             : LBRACKET LBRACKET Section_name RBRACKET RBRACKET
       Section_content Table
9  # P8  :                   | LBRACKET LBRACKET Section_name RBRACKET RBRACKET
       Section_content
10 # P9  : Subsection        : '[' Subsection_name ']' Section_content Subsection
11 # P10 :                   | '[' Subsection_name ']' Section_content
12 # P11 : Section_name      : OBJECT
13 # P12 : Subsection_name   : SUBOBJECT
14 # P13 : Section_content   : Content Section_content
15 # P14 :                   | Content
16 # P15 :                   | COMMENT
17 # P16 : Content           : Key '=' Value
18 # P17 : Key               : KEY
19 # P18 : Value             : BASIC_STRING
20 # P19 :                   | LITERAL_STRING
21 # P20 :                   | STRING
22 # P21 :                   | NUMBER
23 # P22 :                   | IP
24 # P23 :                   | DATE
25 # P24 :                   | TIME
26 # P25 :                   | DATE_TIME
27 # P26 :                   | BOOLEAN
28 # P27 :                   | Array
29 # P28 : Array             : '[' Value_list ']'
30 # P29 : Value_list        : Value ',' Value_list
31 # P30 :                   | Value
```

2.4.1 Programa principal

Primeiramente, decidimos começar o nosso programa como uma lista de secções, que por sua vez, podem-se dividir em apenas uma secção ou uma secção seguida de mais secções, de forma recursiva.
O nosso programa tem, obrigatoriamente, uma secção, pois o código toml apresentado nunca vai ser vazio.

```
1
2  def p_FT(p):
3      "FT : Section_list"
4      p[0] = p[1]
5
6  def p_Section_list1(p):
7      "Section_list : Section Section_list"
8      p[0] = {**p[1], **p[2]}
9
10 def p_Section_list2(p):
```

```
    "Section_list : Section"
    p[0] = p[1]
```

2.4.2 Secções

Uma secção pode ser dividida de quatro formas: nome da secção e respetivo conteúdo , nome da secção seguida de uma subsecção, apenas conteúdo (quando não temos referência a um objeto json, isto é, apenas um valor json) ou uma tabela. Aqui quando nos referimos a uma secção estamos a falar de um objeto json.

Como podemos ver pelas produções, todos os valores já estão a ser guardados num formato de dicionário para, mais tarde, facilitar a conversão.

```

def p_Section1(p):
    "Section  : LBRACKET Section_name RBRACKET Section_content"
    p[0] = { p[2]: p[4] }

def p_Section2(p):
    "Section  : LBRACKET Section_name RBRACKET Subsection"
    p[0] = { p[2]: p[4] }

def p_Section3(p):
    "Section  : Content"
    p[0] = p[1]

def p_Section4(p):
    "Section  : Table"
    p[0] = p[1]
```

2.4.3 Subsecções

Aqui quando nos referimos a subsecções estamos a falar de objetos dentro de objetos (json). Neste caso, uma subsecção é composta por um nome e o seu conteúdo. Temos também a primeira produção que vai fazer isto de forma recursiva de forma a apanhar todas as subsecções disponíveis no código.

```

def p_Subsection1(p):
    "Subsection : LBRACKET Subsection_name RBRACKET Section_content Subsection"
    p[0] = {p[2]: p[4]}
    p[0].update(p[5])

def p_Subsection2(p):
    "Subsection : LBRACKET Subsection_name RBRACKET Section_content"
    p[0] = { p[2]: p[4] }
```

2.4.4 Tabela

Com tabela estamo-nos a referir a tabelas aninhadas, e aqui temos duas produções, a segunda (mais simples) vai apenas retirar o nome da tabela e o seu conteúdo, já a primeira é responsável por fazer o mesmo da segunda, retirar nome e conteúdo da tabela, e ainda verificar se a tabela que vem a seguir (caso exista) tem o mesmo nome da anterior e, em caso positivo, vai apenas unir os seus conteúdos num array e guardar no respetivo dicionário.

```
1
2  def p_Table1(p):
3      "Table : LBRACKET LBRACKET Section_name RBRACKET RBRACKET Section_content Table"
4      keys_list = list(p[7].keys())
5      if p[3] == keys_list[0]:
6          p[0] = {p[3]: [p[6]]+p[7][p[3]]}
7      else:
8          p[0] = {p[3]: [p[6]]}
9          p[0].update(p[7])
10
11 def p_Table2(p):
12     "Table : LBRACKET LBRACKET Section_name RBRACKET RBRACKET Section_content"
13     p[0] = { p[3]: [p[6]] }
```

2.4.5 Nome da secção ou subsecção

Nestas produções apenas vamos retirar o nome das secções e subsecções, através do uso do respetivo *token*.

```
1
2  def p_Section_name(p):
3      "Section_name : OBJECT"
4      p[0] = p[1]
5
6  def p_Subsection_name(p):
7      "Subsection_name : SUBOBJECT"
8      p[0] = p[1]
```

2.4.6 Conteúdo da secção

Na parte do conteúdo da secção temos duas alternativas, ou este é um comentário que mais tarde vai ser ignorado ou é uma linha de conteúdo.

```
1
2  def p_Section_content1(p):
3      "Section_content : Content Section_content"
4      p[0] = {**p[1], **p[2]}
5
6  def p_Section_content2(p):
7      "Section_content : Content"
8      p[0] = p[1]
9
10 def p_Section_content3(p):
11     "Section_content : COMMENT"
12     p[0] = p[1]
```

2.4.7 Conteúdo

Uma linha de conteúdo é, obrigatoriamente, representada por uma chave e um valor.

```
1
```

```
def p_Content(p):
    "Content : Key EQUALS Value"
    p[0] = { p[1] : p[3] }
```

2.4.8 Chave

Esta produção, tal como a do nome da secção ou subsecção, apenas vai verificar o respetivo *token*.

```

def p_Key(p):
    "Key : KEY"
    p[0] = p[1]
```

Aqui temos a produção responsável por captar todos os valores de uma linha *toml*. A função *p_Value* vai verificar todos os *tokens* que tenham a ver com possíveis valores de uma linha de código *toml*. Temos, ainda, uma linha responsável por verificar a existência de um *Array*

2.4.9 Valores

```

def p_Value(p):
    '''Value : BASIC_STRING
             | LITERAL_STRING
             | STRING
             | NUMBER
             | IP
             | DATE
             | TIME
             | BOOLEAN
             | Array
             | DATE_TIME
            '''
    p[0] = p[1]
```

2.4.10 Arrays

Estas produções são responsáveis por verificar a existência de um *array*. A primeira vai retirar os valores que se encontram dentro de um *array* e a segunda é responsável por percorrer o *array* valor a valor.

```

def p_Array(p):
    "Array : LBRACKET Value_list RBRACKET "
    p[0] = p[2]

def p_Value_list(p):
    '''Value_list : Value COMMA Value_list
                  | Value'''
    if len(p) > 2:
        p[0] = [p[1]] + p[3]
    else:
        p[0] = [p[1]]
```

2.5 Controlo de Erros

De modo a controlarmos os erros que apareciam ao longo da realização do projeto, testámos escrever diversos exemplos de várias formas, de modo a conseguirmos que a nossa gramática ficasse mais ”universal”. Alguns exemplos foram extraídos da internet e também utilizamos o que nos foi dado no enunciado, efetuando o grupo várias alterações ao mesmo.

Por outro lado, de forma a controlar os erros do nosso compilador fomos criando funções que detetassem os mesmos e nos devolvessem o erro.

Ao contruir o lex, criamos a função *t_error* de forma a devolver um erro caso não consiga fazer o lexer.

```
def t_error(t):
    print("Erro: " + t.value[0], file=sys.stderr)
    t.lexer.skip(1)
```

De seguida, ainda com o mesmo pensamento do caso anterior, no yacc criamos a função *p_error* que nos devolve erro caso não consiga interpretar alguma linha do nosso código.

```
def p_error(p):
    parser.success = False
    print(f"Sintax error: {p.value}")
```

Por fim, para não estarmos a trabalhar com códigos *toml* mal formatados criamos estas linhas de código de forma a ler o ficheiro introduzido para converter e detetar logo se há erros de formatação ou não, devolvendo erro em caso afirmativo.

```
try:
    parsed_data = toml.loads(data)
    print("O arquivo TOML esta formatado corretamente.")

except toml.TomlDecodeError as e:
    print("Erro ao decodificar o arquivo TOML:")
    print(e)
```

Capítulo 3

Funcionalidades implementadas

Como pedido no trabalho prático, a nossa primeira funcionalidade a ser implementada foi o conversor de *toml* para *json*. No entanto, uma vez que ao longo das produções da gramática fomos guardando todos os valores num dicionário *python*, reparamos que, com isso, eram facilmente construídos outros conversores para outras linguagens, como por exemplo, *yaml* e *xml*

Em todos os exemplos que se vão apresentar de seguida, o ficheiro input é o exemplo apresentado no enunciado e *parsed_dict* é o dicionário resultante do analisador sintático.

3.1 Conversor *toml* para *json*

Esta foi a funcionalidade pedida no trabalho prático, logo foi a primeira a ser feita. Como referido anteriormente, os nossos valores estão todos armazenados num dicionário python, logo para converter o nosso dicionário para um ficheiro *json* apenas temos de adicionar a seguinte linha de código que vai dar *dump* ao nosso dicionário para o ficheiro *output*.

```
1 json.dump(parsed_dict, f, ensure_ascii=False)
```

3.1.1 Ficheiro output

```
1  {
2      "title": "TOML Example",
3      "owner": {
4          "name": "Tom Preston-Werner",
5          "date": "2010-04-23",
6          "time": "21:30:00"
7      },
8      "database": {
9          "server": "192.168.1.1",
10         "ports": [
11             8001,
12             8001,
13             8002
14         ],
15         "connection_max": 5000,
16         "enabled": true
17     },
```

```
    "servers": {
        "alpha": {
            "ip ": "10.0.0.1",
            "dc ": "eqdc10"
        },
        "beta": {
            "ip ": "10.0.0.2",
            "dc ": "eqdc10",
            "hosts ": [
                "alpha",
                "omega"
            ]
        }
    }
}
```

3.2 Conversor *toml* para *yaml*

Esta foi uma funcionalidade extra que o grupo decidiu implementar. Aqui é percorrido o dicionário com a tradução do *toml*, que vai ser convertido para *yaml* com a seguinte linha de código:

```
yaml_data = yaml.dump(parsed_dict)
```

3.2.1 Ficheiro output

```

database:
  'connection_max ': 5000
  'enabled ': true
  'ports ':
  - 8001
  - 8001
  - 8002
  'server ': 192.168.1.1
owner:
  'date ': '2010-04-23'
  'name ': Tom Preston-Werner
  'time ': '21:30:00'
servers:
  alpha:
    'dc ': eqdc10
    'ip ': 10.0.0.1
  beta:
    'dc ': eqdc10
    'hosts ':
    - alpha
    - omega
    'ip ': 10.0.0.2
'title ': TOML Example
```

3.3 Conversor *toml* para *xml*

Igual à funcionalidade anterior, esta foi uma funcionalidade extra que o grupo decidiu implementar. Aqui é percorrido o dicionário com a tradução do *toml*, que vai ser convertido para *xml* com as seguintes linhas de código:

```
def create_xml_element(parsed_dict, parent):
    for key, value in parsed_dict.items():
        if isinstance(value, dict):
            element = ET.SubElement(parent, key)
            create_xml_element(value, element)
        else:
            element = ET.SubElement(parent, key)
            element.text = str(value)

root = ET.Element('root')

create_xml_element(parsed_dict, root)

# Criar a  rvore  XML
tree = ET.ElementTree(root)
```

3.3.1 Ficheiro output

```

<?xml version='1.0' encoding='utf-8'?>
<root>
    <title >TOML Example</title >
    <owner>
        <name >Tom Preston-Werner</name >
        <date >2010-04-23</date >
        <time >21:30:00</time >
    </owner>
    <database>
        <server >192.168.1.1</server >
        <ports >[8001, 8001, 8002]</ports >
        <connection_max >5000</connection_max >
        <enabled >True</enabled >
    </database>
    <servers>
        <alpha>
            <ip >10.0.0.1</ip >
            <dc >eqdc10</dc >
        </alpha>
        <beta>
            <ip >10.0.0.2</ip >
            <dc >eqdc10</dc >
            <hosts >['alpha', 'omega']</hosts >
        </beta>
    </servers>
</root>
```

3.4 Funcionamento do programa

Ao correr o programa com ***python3 converter.py***, o utilizador vai ser pedido para introduzir o path do ficheiro que quer converter e, de seguida, qual é a linguagem para a qual deseja converter o seu ficheiro. Neste processo o programa vai verificar se o ficheiro existe, se está num formato *toml* correto e se a linguagem para a qual o utilizador deseja converter é suportada.

Capítulo 4

Conclusão

Finalizada a realização deste projeto desenvolvido ao longo do semestre, sentimos que pudemos aprofundar os conhecimentos e aspetos adquiridos ao longo das aulas da respetiva UC, nomeadamente o uso das ferramentas *Yacc* e *Lex* do *PLY*.

Ao longo da realização do presente trabalho prático, deparamo-nos com algumas dificuldades, mas rapidamente superamos, analisando os erros e tentanto perceber o porquê de estarem a acontecer.

Uma das partes mais desafiadoras deste trabalho foi a construção do analisador sintático. Durante o processo, o grupo deparou-se com vários erros e também encontrou melhores soluções para o desenvolvimento da gramática. Isto exigiu várias tentativas de escrita até chegarmos a uma forma mais correta.

Em jeito de conclusão, finalizado este trabalho, o grupo está satisfeito com o produto final, considerando ir de encontro ao que foi solicitado.