Processamento de Linguagens (3º ano de Curso)

Trabalho Prático

Relatório de Desenvolvimento

Grupo 35

João Faria (a97652)

Tiago Ferreira (a97141)

Miguel Neiva (a92945)

28 de maio de 2023

Resumo

Este relatório serve de suporte à entrega do projeto final, que consiste no desenvolvimento completo de um processador para uma *Domain Specific Language*. No nosso caso, trata-se da construção em Python de uma ferramenta que converte um subconjunto de Toml para JSON, recorrendo ao Flex e Yacc.

Conteúdo

Capítulo 1

Introdução

Supervisor: José Carlos Ramalho
Área: Processamento de Linguagens

1.1 Conversor toml-json

O presente relatório descreve todo o processo de desenvolvimento de um conversor de linguagens TOML-JSON, como projeto final da cadeira de processamento de linguagens. O projeto visa aumentar a nossa experiência em engenharia de linguagens e em programação generativa, consolidando os conhecimentos adquiridos ao longo do semestre nas aulas teóricas e práticas, bem como desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.

A linguagem TOML é uma linguagem simples e intuitiva que permite a definição de estruturas complexas, como dicionários generalizados, frequentemente utilizados em ficheiros de configuração mas podem ser aplicados em vários outros domínios. No entanto, pode ser necessário converter essas estruturas para outras linguagens, como, por exemplo, o JSON, para a sua aplicação em diferentes contextos.

O objetivo deste projeto é construir uma ferramenta que converta um subconjunto de dados escritos no formato TOML para o formato JSON. Para isso, utilizamos em Python ferramentas de geração de compiladores baseadas em gramáticas tradutoras: o Flex e o Yacc; este relatório serve de suporte à entrega final do projeto e apresenta todas as informações complementares ao código desenvolvido.

O problema que se pretende resolver é efetuar esta conversão de forma eficiente, transformando assim todo o tipo de estruturas de um documento TOML num documento correto e bem formatado JSON, tornando a conversão mais fácil e acessível, por exemplo, a desenvolvedores que precisem de trabalhar com diferentes tipos de linguagens de configuração de forma rápida. O principal objetivo do projeto é fornecer uma solução eficiente e confiável para a conversão descrita anteriormente.

Neste relatório, também se discutem os resultados obtidos e os contributos do projeto para a área de processamento de linguagens.

Estrutura do Relatório

No primeiro capítulo, a introdução, já apresentamos de forma sucinta e objetiva uma visão geral do projeto destacando os principais pontos que serão abordados no relatório.

Já no capítulo 2, será feita uma descrição informal do problema, uma secção mais detalhada sobre o nosso

problema, seguida da especificação dos requisitos, incluindo os dados, no capítulo 2.2.

O capítulo 3 abordará a conceção e o desenho da resolução, incluindo a definição das estruturas de dados e algoritmos utilizados.

No capítulo 4, serão apresentadas as alternativas, decisões e problemas de implementação encontrados, bem como os testes realizados e os resultados obtidos.

Finalmente, no capítulo 5, será apresentada a conclusão do projeto, destacando-se os principais contributos, reflexões e as possíveis extensões para trabalhos futuros.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

TOML é uma linguagem de marcação minimalista usada para a configuração de ficheiros. O nome "*Tom's Obvious, Minimal Language*", é uma referência ao seu criador, Tom Preston-Werner.

Este tipo de ficheiro foi projetado para a sua linguagem ser facilmente legível tanto para humanos quanto para máquinas, com ênfase na simplicidade e clareza. Usa uma sintaxe que lembra a do INI, mas adiciona recursos adicionais, como suporte para tipos de dados mais complexos, desde arrays a tabelas aninhadas, e ainda suporta comentários. É frequentemente usado em projetos de software para armazenar configurações, mas também pode ser usado para outros fins, como a configuração em servidores web, em sistemas operacionais e outro tipo de aplicações. Ela é compatível com várias linguagens de programação, incluindo Python, Ruby, Java e C++.

Já a linguagem JSON (*JavaScript Object Notation*) é um formato leve de troca de dados amplamente utilizado na comunicação entre sistemas. A sua sintaxe é baseada em pares chave-valor, o que facilita a representação de informações estruturadas de forma legível tanto para humanos quanto para máquinas. JSON é independente da plataforma e é suportado por diversas linguagens de programação, tornando-o altamente interoperável. É frequentemente utilizado em aplicações web e APIs para transmitir e armazenar dados de forma eficiente, devido à sua natureza simples e flexível.

No contexto de projetos de engenharia informática, por exemplo, o uso da linguagem JSON é fundamental para a troca de informações entre os diferentes componentes de sistemas. Ao adotar o JSON como formato de dados, um desenvolvedor pode estruturar e transmitir informações relevantes de maneira concisa e eficiente. Além disso, a legibilidade e a flexibilidade da sintaxe JSON permitem uma fácil integração com outras tecnologias e sistemas externos, contribuindo para a escalabilidade e a interoperabilidade do nosso projeto.

Como já referido, o objetivo central deste projeto é abordar a questão da conversão eficiente de estruturas de documentos TOML para documentos JSON corretos e bem formatados. Esta conversão visa facilitar o trabalho de desenvolvedores que lidam com diversas linguagens de configuração, permitindo que eles realizem a tarefa de forma ágil e descomplicada. O principal propósito é disponibilizar uma solução confiável e eficiente para esse processo de conversão. Além disso, procuramos tornar a conversão mais acessível, ampliando o alcance de utilização para diferentes contextos e aplicações.

2.2 Especificação do Requisitos

2.2.1 Dados

Para a elaboração deste trabalho prático, estabelecemos de imediato dois grandes requisitos a cumprir: Elaboração do nosso analisador léxico, e elaboração do nosso analisador sintático. Decidimos começar, naturalmente, pela elaboração do nosso analisador léxico. Fizemos uma pesquisa a fundo acerca de todos os tipos de representações possíveis em documentos TOML, consultando a documentação oficial no site da linguagem, e começamos a definir as nossas regras e expressões regulares. Esta linguagem ainda possui uma boa quantidade de tipos de blocos de texto, então foi uma tarefa que nos ocupou algum tempo. Ainda assim, a certo ponto, decidimos partir para a implementação do nosso analisador sintático, ainda com alguns casos por definir, no nosso *lexer*.

No desenvolvimento do nosso analisador sintático procuramos, inicialmente, informar-nos sobre as regras semânticas às quais um ficheiro TOML obedecia, procurando garantir as mesmas. Definimos qual a estrutura da nossa gramática e, chegando a um consenso, entendemos que o mais correto seria identificar todos os elementos possíveis de um ficheiro toml, um a um. Existiam muitas particularidades com a linguagem toml que teríamos de garantir na nossa gramática entre elas: a existência de listas, objetos, tabelas e *arrays* de tabelas, listas aninhadas, objetos aninhados, entre outros.

Por fim, após estes dois tópicos cumpridos, teríamos então todas as ferramentas para poder, efetivamente, concluir o nosso objetivo final: efetuar a conversão para um ficheiro de output JSON.

Capítulo 3

Concepção/desenho da Resolução

3.1 Estruturas de Dados

Ao longo de todo o nosso programa utilizamos vários dicionários auxiliares fundamentais para a conversão do nosso ficheiro toml em partes, conforme os elementos presentes no ficheiro fossem reconhecidos. No final restou-nos recorrer a um dicionário global para realizar a junção de todos os dicionários auxiliares num único só, fazendo as devidas verificações típicas de um dicionário em *python*. Este dicionário global servirá para compor a nossa estrutura de elementos num formato semelhante ao pretendido no final.

3.2 Algoritmos

Por fim, o nosso projeto apresenta dentro da pasta */src* 5 programas *python* distintos elaborados por nós:

tokenizer.py

O programa tokenizer.py é responsável por realizar a análise léxica de um arquivo TOML (com extensão .toml). Inicialmente, ele importa o módulo lexer do ficheiro analisador_lexico.py, que contém, como já referido, a definição do analisador léxico.
De seguida, é aberto o ficheiro TOML especificado por nós em modo de leitura e as linhas do mesmo são armazenadas numa lista *lines*. Depois, as linhas são concatenadas numa única string chamada *result*.Por questões de otimização decidimos que o nosso programa deveria ler o ficheiro do *stdin*, sendo necessário, no momento em que corremos o programa, passarmos o caminho para o ficheiro toml que iremos reconhecer.

```
>> python3 tokenizer.py < ../TOML/toml.toml
```

O analisador léxico é inicializado com a entrada definida como a string *result*. Depois, utilizando um *loop* contínuo, o analisador léxico obtém o próximo *token* chamando *lexer.token()*. Se não houver mais *tokens*, o *loop* é interrompido. Cada *token* é depois impresso na saída padrão.
Em resumo, este programa lê um arquivo TOML, realiza a análise léxica usando o analisador léxico fornecido e imprime os *tokens* resultantes.

gparser.py

O programa denominado gparser.py é o responsável por correr o nosso analisador sintático sobre um documento TOML. Começa por importar o nosso *parser* proveniente do nosso programa analisador_sintático.py, que contém, como anteriormente referido, a definição da nossa gramática.
Em seguida são definidas algumas variáveis globais que foram essenciais ao nosso programa, temos então:

- `parser.toml`: Importa a classe TOML do ficheiro toml.py, de forma a ser possível manipular uma estrutura final que irá ser importada para o nosso ficheiro result.json.
- `parser.tabel_token`: Um booleano que nos permite saber se nos encontramos diante de um caso em que temos duas tabelas seguidas, sendo uma delas vazia.
- `parser.stack`: Esta variável funciona como uma stack, permitindo que sejam guardados conjuntos de atribuições pela forma como os mesmos aparecem no ficheiro toml principal e não pela forma como são reconhecidos pela gramática.
- `parser.final` Esta variável, como o próprio nome indica é responsável por compor o dicionário final para cada tabela e guardar o mesmo numa lista. Mais tarde todos os dicionários desta lista serão passados à estrutura final.
- `parser.size`: Variável que permite manter um contador sobre o número de elementos já reconhecidos, de forma a saber quando atingimos o final do ficheiro.

O próximo passo foi abrir o ficheiro TOML especificado por nós em modo de leitura e armazenar as linhas do mesmo numa lista *lines*. Depois, as linhas são concatenadas numa única string chamada *result*. Por questões de otimização decidimos que o nosso programa deveria ler o ficheiro do *stdin*, sendo necessário, no momento em que corremos o programa, passarmos o caminho para o ficheiro toml que iremos converter.

```
>> python3 gparser.py < ../TOML/toml.toml
```

Por fim passamos ao *parser* a string *result* e utilizando o método *parser.toml.toJSON()*, presente na classe TOML do ficheiro toml.py, obtemos a nossa estrutura JSON com uma indentação igual a 4 e com todos os dados fornecidos pelo ficheiro TOML. Resta-nos agora abrir um ficheiro *result.json* e copiar para o mesmo o resultado json obtido.

toml.py

Este programa consiste numa implementação de classes e métodos relacionados à manipulação de ficheiros no formato TOML. O objetivo é fornecer funcionalidades para converter estruturas de dados em TOML para o formato JSON.
O programa inclui três classes principais: "Assignment"para representar atribuições de valores numa estrutura TOML, "Table"para representar tabelas (secções) no TOML e "TOML"como uma classe auxiliar que engloba várias funcionalidades auxiliares ao nosso analisador sintático. A classe "TOML"possui métodos para adicionar elementos ao dicionário de dados, tanto no âmbito global como em tabelas específicas, métodos para a criação de "Assignments"partindo de uma lista de nomes e do valor que o *assignment* tem associado e, para além disso, oferece métodos para converter o dicionário em formato JSON. É neste programa que temos ainda a inicialização do nosso dicionário final, este dicionário começa vazio e à medida que o analisador sintático avança são acrescentadas novas estruturas ao mesmo, no final obtemos um dicionário já devidamente estruturado e com toda a informação presente no ficheiro TOML utilizado inicialmente.

A classe "TOML"utiliza uma abordagem recursiva para lidar com dicionários aninhados, permitindo a adição de elementos de forma hierárquica. A conversão para JSON é realizada utilizando a biblioteca padrão json, garantindo a formatação adequada e a representação correta dos dados.

analisador_lexico.py

Como o nome indica, trata-se do nosso analisador léxico, define a função do analisador léxico utilizando a biblioteca Ply (Python Lex-Yacc) para realizar a análise léxica dos nossos ficheiro TOML de *input*.

As regras de tokenização são definidas utilizando expressões regulares e são associadas a nomes de *tokens* específicos. Quase todos são bastante sugestivos, alguns dos *tokens* definidos incluem:

- `COMMENT`: Comentário iniciado com `#`.
- `COMMA`: Vírgula (,).
- `DOT`: Ponto (.).
- `VAR`: Identificadores, que podem ser sequências alfanuméricas ou incluir hífens (-).
- `LEFTBRACKET` e `RIGHTBRACKET`: Parênteses ({ e }) utilizados para delimitar tabelas no TOML.
- `LEFTSQUAREBRACKET` e `RIGHTSQUAREBRACKET`: Parênteses ([e]) utilizados para delimitar arrays no TOML.
- `EQUAL`: Sinal de igual (=) utilizado para atribuições de valores.
- `DATE`, `INT`, `FLOAT`, `INF`, `NAN`, `STRING`, `LITERALSTRING`, `MSTRING`, `MLSTRING`, `BOOLEAN`, `TIME`, `DATETIME`, `OFFSET`, `OFFSETDATETIME`, `HEXADECIMAL`, `BINARY`, `OCTAL`: Tokens que representam diferentes tipos de valores válidos no TOML, como datas, números inteiros e de vírgula flutuante, strings, booleanos, etc.
- `NEWLINE`: Quebra de linha.

Existem também funções definidas para tratar de forma prioritária alguns tipos de tokens mais complexos, como TIME, OFFSETDATETIME, DATETIME, OFFSET e DATE.

Além disso, há uma função ***t_error*** para tratar caracteres inválidos, exibindo uma mensagem de erro indicando o caractere ilegal e a linha correspondente.

Por fim, o objeto lexer é criado utilizando a função lex.lex() da biblioteca Ply para inicializar o analisador léxico.

Em resumo, aqui definimos as regras de tokenização, utilizando a biblioteca Ply para criar um analisador léxico capaz de identificar e classificar os *tokens* presentes no ficheiro de entrada.

analisador_sintatico.py

O analisador_sintático.py trata-se do ficheiro onde desenvolvemos o nosso analisador sintático. Como parte essencial do nosso sistema de compilação, o analisador sintático desempenha um papel fundamental na verificação da nossa estrutura gramatical e na deteção de erros de sintaxe presentes ao longo do ficheiro toml de leitura.

Começamos por fazer a devida importação dos *tokens* já reconhecidos pelo nosso analisador léxico e que serão fundamentais para a elaboração da nossa gramática. Na construção da nossa gramática, procuramos ser o mais simples possível, o nosso estado inicial é o "program"que será constituído por "statements",

sendo os "statements"constituídos por vários "statement", um "statement"é uma produção que identifica os vários elementos possíveis no no toml, tables, assignements, newlines e comments. Recuando para a função *p_program* e para a função *p_statements*, correspondentes às funções que incluem as produções para o "program"e para os "statements", respetivamente, é nestas funções que ocorrem as principais tarefas do nosso analisador sintático. Começando pela função *p_statements* é nesta função que os "Assignments"são guardados numa stack, variável global *p.parser.stack* anteriormente referida, pela ordem com que são reconhecidos. Caso se identifica uma "Table", é feita a devida análise para verificar o seu tipo e, após a junção do nome associado à tabela com as atribuições que a compõe ou sem atribuições, caso em que a tabela se encontra vazia, obtemos uma estrutura final de um dicionário, que será inserida na variável global *p.parser.final* anteriormente referida. Da função *p_statements* para a função *p_program* é passado o dicionário atual, é só na função contendo a produção raiz do nosso programa que iremos acrescentar todos as atribuições que não estão associadas a nenhuma tabela, uma vez mais pela ordem inversa com que são identificas, resultando no dicionário final correspondente à conversão de todo o ficheiro toml. De referir ainda a utilidade da função *p_statement* que, para além de identificar, um a um, todos os elementos presentes no nosso ficheiro toml, ainda realiza a devida contagem de elementos, mais à frente iremos referir o porquê de termos seguido esta estratégia de identificação de elementos um a um e o porquê de realizarmos uma contagem de elementos em vez de linhas.

Tanto o nome das tabelas como o nome associado às atribuições em toml podem ser bastante complexos, uma vez que, podem apresentar *quoted names*, caso em que temos estruturas de nomes aninhados. Devido a complexidade destas estruturas foi necessário a criação de duas produções, "name"e "name2", que identificassem estes casos, resultando, das respetivas funções, uma lista de nomes ordenada pela ordem com que os mesmos aparecem nos *quoted names* presentes nas tabelas ou nas *keys* de uma atribuição. Com esta lista de nomes é agora possível criar dicionários aninhados.

Para além de todos os tipos elementares, já referidos quando abordamos o analisador léxico, tivemos ainda que criar produções que lidassem com os tipos de elemento lista e tabelas alinhadas (*inline table*), uma vez que os mesmos são tipos mais complexos que não podiam ser identificados pelo *lexer*.

Por fim, decidimos ainda fazer a devida conversão de tipos no analisador sintático. Sabíamos que esta tarefa podia ser feita no analisador léxico, e que ao acrescenta-la no analisador sintático estávamos a incutir ainda mais complexidade ao mesmo, no entanto, queríamos ter uma verificação de tipos mais precisa e rigorosa, e ainda remover complexidade do nosso analisador léxico de forma a que o mesmo apenas fosse responsável por identificar o *token* especifico, precavendo possíveis problemas devido à grande quantidade de tipos possíveis na linguagem TOML, o que resultou num analisador léxico simples e eficaz.

O nosso analisador sintático conta com 68 produções que juntas são capazes de transformar qualquer ficheiro TOML num ficheiro no formato JSON.

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

Durante a implementação do analisador léxico, enfrentamos alguns desafios e tomamos decisões importantes. Um dos primeiros desafios foi definir corretamente os *tokens* a serem reconhecidos, considerando a gramática do TOML e os requisitos específicos do projeto. Foi necessário lidar com uma variedade de tipos de dados, como strings, datas, números e valores booleanos, e garantir que o analisador fosse capaz de identificar e classificar cada token adequadamente. A correta priorização dos reconhecimentos foi um ponto que nos trouxe algum trabalho, até conseguirmos acertar com os *matches* pretendidos.

Outro desafio significativo foi lidar com strings multi linhas, um recurso do TOML. Para isso, foi necessário ajustar a implementação para reconhecer corretamente as strings multi linhas delimitadas por três aspas triplas, permitindo a presença de quebras de linha e mantendo a integridade dos caracteres especiais dentro dessas strings; inicialmente este ponto era um problema pois estávamos a enviar como *input* ao nosso *lexer* apenas uma linha de cada vez, e, como seria de esperar, isto impossibilitava-o de reconhecer a nossa expressão regular definida para uma string multi linha. Pensamos ainda implementar a utilização de estados para solucionar este problema, mas acabamos por achar mais correto e eficaz fazer a leitura do ficheiro completo ao invés de enviar linha a linha, e com esta medida, conseguimos alcançar o nosso objetivo.

Em resumo, a implementação do *lexer* utilizando a biblioteca *Ply* envolveu a definição cuidadosa de expressões regulares e a ordenação adequada das regras para lidar com os desafios de reconhecimento de *tokens* específicos. As decisões tomadas ao longo do processo permitiram obter um *lexer* capaz de identificar corretamente diferentes tipos de *tokens*, incluindo números inteiros, *floats* especiais, expressões de data e hora com *offset*, entre outros.

Quando passamos ao analisador sintático, enfrentamos, de igual modo, vários desafios e, uma vez mais, tivemos que chegar a um consenso para tomar várias decisões importantes. Inicialmente verificamos que um dos maiores problemas que enfrentávamos era de identificar quando é que uma atribuição pertencia a uma tabela, e qual era essa tabela, ou não pertencia, o que implicava estruturas diferentes para os dois casos. Decidimos então, simplificar, e realizar uma travessia de forma a que identificássemos elemento a elemento (com elemento estamos a referir-nos a tabelas, atribuições, comentários e linhas vazias) e desta forma, e utilizando algumas variáveis globais, já conseguíamos identificar se o objeto pertencia ou não a uma tabela, bastava, para isso, o analisador sintático encontrar a identificação de uma tabela e associar todos as atribuições que recolheu ao longo da travessia a esta tabela.

Outro problema com que nos deparamos teve a ver com sintaxe de um documento TOML. Num ficheiro TOML as linhas são únicas a um elemento, ou seja, apenas podiam conter uma indicação de tabela, ou de uma atribuição, ou um comentário ou a própria linha ser vazia, sendo que, se a linha contivesse uma indicação

de tabela, nessa mesma linha não podia haver mais nenhuma tabela nem uma atribuição, e o mesmo caso a linha tivesse uma atribuição, não podia haver nenhuma indicação de tabela nem mais nenhuma atribuição na mesma linha, obrigatoriamente teríamos de mudar de linha, apenas comentários são possíveis de incluir em linhas que já contêm um elemento. Para resolver este problema tivemos que, no fim de cada elemento, obrigar a existência de uma mudança de linha, estratégia que se aplica a todas as tabelas e atribuições do nosso programa, exceto ao último elemento do ficheiro, pois, por questões de estética, não queríamos que o utilizador fosse obrigado a inserir uma última mudança de linha no último elemento do ficheiro, não nos pareceu correto o mesmo. Assim surgiu-nos um novo problema com o nosso analisador sintático, tínhamos de encontrar uma forma de saber se nos encontrávamos na última linha do ficheiro. Foi então que decidimos, inicialmente, guardar o número de linhas obtidas no momento de leitura do ficheiro numa nova variável global a todo o programa do analisador sintático, assim podíamos incluir uma nova produção nas nossas funções que identificavam tabelas e atribuições, com esta produção o elemento não era obrigado a terminar com a mudança de linha, no entanto tivemos que garantir que tal só acontecia quando nos encontrávamos com um número de *t.lexer.lineno* (número de linhas identificado pelo analisador léxico através da leitura de um carácter de mudança de linha no decorrer da analise léxica) igual ao número total de linhas, ou seja, na última linha do nosso ficheiro, e caso contrário encontrávamos um erro.

Mais tarde com a introdução de alguns tipos mais complexos do toml, como por exemplo as strings multi linha, criou-se uma dificuldade em contar o número de mudanças de linha do ficheiro, pois estas mudanças de linha seriam incluídas no *token* correspondente à multi linha, não contando assim para o número de linhas do *lexer*(*t.lexer.lineno*). Alteramos completamente a nossa estratégia de contagem de linhas, pois continuávamos a precisar de saber quando encontraríamos a última linha do nosso ficheiro. Beneficiamos da estratégia referida anteriormente de identificar elemento a elemento e, sempre que um elemento fosse identificado, incrementamos uma variável global *p.parser.size* a fim de contar o número de elementos do ficheiro já percorridos em vez do número de linhas, realizando, igualmente, uma comparação com o número do *t.lexer.lineno* para verificar que nos encontrávamos na última linha do ficheiro.

4.2 Testes realizados e Resultados

Mostram-se a seguir alguns testes feitos (valores introduzidos) e os respetivos resultados obtidos, com alguns dos ficheiros presentes na pasta TOML:

[Toml]

```toml
tittle = "TOML example"

[[fruits]]
name = "apple"

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

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

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

[[fruits]]
name = "banana"

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

[Json]

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

Figura 4.1: Exemplo da conversão Toml para Json

[Toml]

```toml
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
date = 2010-04-23
time = 21:30:00

[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"

# Line breaks are OK when inside arrays
hosts = ["alpha","omega"]
```

[Json]

```json
{
    "title": "TOML Example",
    "owner": {
        "name": "Tom Preston-Werner",
        "date": "010-04-2",
        "time": "1:30:0"
    },
    "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",
            "hosts": [
                "alpha",
                "omega"
            ]
        }
    }
}
```

Figura 4.2: Exemplo da conversão Toml para Json

[Toml]

[Json]

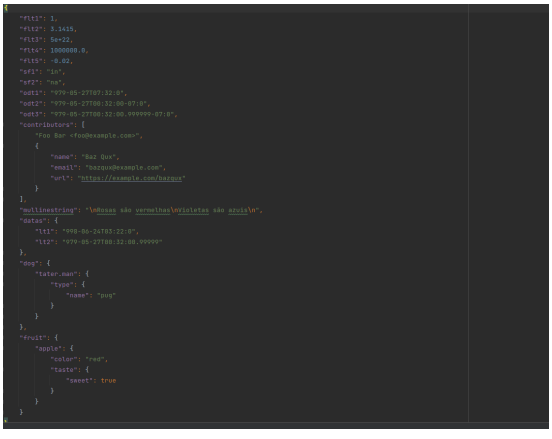


Figura 4.3: Exemplo da conversão Toml para Json

[Toml]

```toml
[legumes]

["batatas"."vermelha".quantidade]

["batatas"."branca".quantidade]

[[bebidas]]

[variavel1.variavel2.variavel3]
nome = "variavel"
idade = 1

[nome."primeiro".1]
```

[Json]

```json
{
    "legumes": {},
    "batatas": {
        "vermelha": {
            "quantidade": {}
        },
        "branca": {
            "quantidade": {}
        }
    },
    "bebidas": [],
    "variavel1": {
        "variavel2": {
            "variavel3": {
                "nome": "variavel",
                "idade": 1
            }
        }
    },
    "nome": {
        "primeiro": {
            "1": {}
        }
    }
}
```

Figura 4.4: Exemplo da conversão Toml para Json

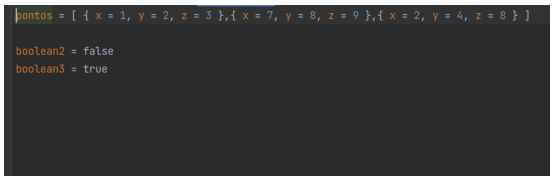
[Toml]

```toml
tittle = "TOML EXAMPLE"

[person]
name.proprio = "John"
name.apelido = "Doe"
age.birth = 30

[animal]
type = "raposa"
pet_name = "coccy"
pet_age = 8

[animal.pet]
type = "dog"
pet_name = "Puffy"
pet_age = 3

[address.road.California]
street = "123 Main St"
city = "Anytown"
state = "CA"
zip = 12345
habitantes = [{ nome = "Alex", idade = 24 }]
```

[Json]

Figura 4.5: Exemplo da conversão Toml para Json

[Toml]



[Json]

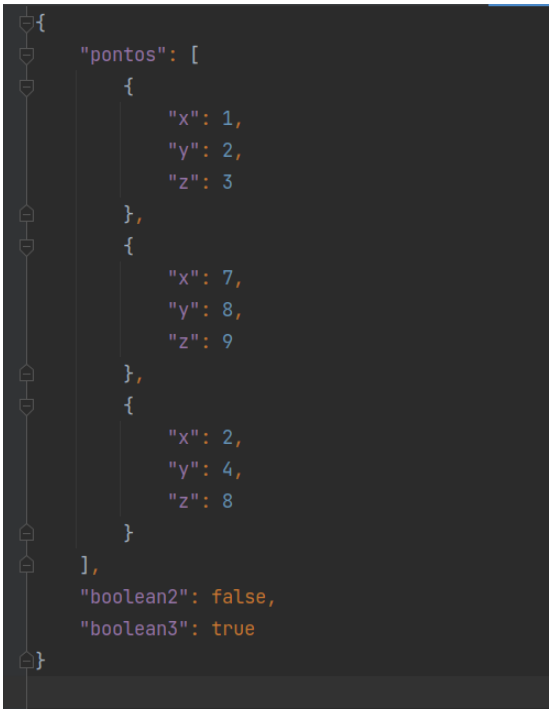


Figura 4.6: Exemplo da conversão Toml para Json

Capítulo 5

Conclusão

Ao longo deste relatório foi apresentado o projeto de desenvolvimento de um conversor de toml para json. Foi realizada uma análise do problema, especificação dos requisitos e conceção da solução, incluindo a escolha de estruturas de dados e algoritmos adequados.

A implementação foi realizada com sucesso, tendo sido apresentadas alternativas, decisões e problemas de implementação. Foram realizados testes e apresentamos resultados que consideramos que satisfazem os requisitos.

Conclui-se que este projeto permitiu 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), bem como desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora.

Por fim, este trabalho permitiu também a utilização de 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.

Em suma, este projeto, além de nos ajudar a consolidar conhecimentos aprendidos ao longo das aulas, permitiu ainda adquirir novas competências e conhecimentos em processamento de linguagens, gramáticas formais e compiladores. Pudemos compreender e ganhar experiência do que são todas as etapas de um processo destes, e por fim conseguimos desenvolver uma ferramenta útil para conversão de toml para json.

Apêndice A

Código do Programa

Lista-se a seguir o códigos dos diferentes programas desenvolvidos:

Listing A.1: tokenizer.py

```
import sys
from analisador_lexico import lexer

lines=[]
for line in sys.stdin:
    lines.append(line)

result=""
for line in lines:
    result+=line

lexer.input(result)
while True:
    tok = lexer.token()
    if not tok:
        break  # No more input
    print(tok)
```

Listing A.2: gparser.py

```
import sys
import toml as toml
from analisador_sintatico import parser

parser.toml = toml.TOML()
parser.table_token = False
parser.stack = []
parser.final = []
parser.size = 1

lines=[]
for line in sys.stdin:
    lines.append(line)

result=""
for line in lines:
    result+=line

parser.parse(result)

print(parser.toml.toJSON())

with open('../out/result.json','w') as file:
    file.write(parser.toml.toJSON())
```

Listing A.3: toml.py

```
import json

class Assignment:
    def __init__(self,content):
        self.content = content

class Table:
    def __init__(self,name,type):
        self.type = type
        self.name = name
        self.data = {}

class TOML:

    def __init__(self):
        self.data = {}

    def add_element(self,new_dict):
        for key, value in new_dict.items():
            if key not in self.data:
                self.data[key] = value
            else:
                if isinstance(value, dict) and isinstance(self.data[key], dict):
                    self.data[key] = self.add_element_aux(self.data[key], value)
                elif isinstance(self.data[key], list) and isinstance(value,list):
                    self.data[key].append(value[0])
                elif isinstance(self.data[key], list):
                    self.add_element_aux(self.data[key][-1], value)
                else:
                    self.data[key] = value
        return self.data

    def add_element_table(self, atual_dict,new_dict):
        for key, value in new_dict.items():
            if key not in atual_dict:
                atual_dict[key] = value
            else:
                if isinstance(value, dict) and isinstance(atual_dict[key], dict):
                    atual_dict[key] = self.add_element_aux(atual_dict[key], value)
                else:
                    atual_dict[key] = value
        return atual_dict

    def add_element_aux(self, dict1, dict2):
        for key, value in dict2.items():
            if key not in dict1:
                dict1[key] = value
            else:
                if isinstance(value, dict) and isinstance(dict1[key], dict):
                    dict1[key] = self.add_element_aux(dict1[key], value)
                elif isinstance(value, list):
                    if len(value) > 0:
                        dict1[key].append(value[0])
```

```
54                      else:
55                          dict1[key].append([])
56                  else:
57                      dict1[key] = value
58          return dict1
59
60      def new_assignment(self,names,value):
61          obj = {}
62          for nome in reversed(names):
63              if obj:
64                  obj = {nome: obj}
65              else:
66                  obj = {nome: value}
67          return obj
68
69      def toJSON(self):
70          return json.dumps(self.data, indent=4, ensure_ascii=False)
```

Listing A.4: analisador_lexico.py

```
import ply.lex as lex

tokens = ('COMMENT', 'COMMA', 'DOT', 'VAR',
          'LEFTBRACKET', 'RIGHTBRACKET', 'LEFTSQUAREBRACKET', 'RIGHTSQUAREBRACKET',
          'EQUAL', 'DATE', 'INT', 'FLOAT', 'INF', 'NAN', 'STRING', 'LITERALSTRING', 'MSTRING', 'MLSTRING', 'BOOLEAN',
          'TIME', 'DATETIME', 'OFFSET', 'OFFSETDATETIME', 'HEXADECIMAL', 'BINARY', 'OCTAL',
          'NEWLINE')


t_COMMA = r'\,'
t_DOT = r'\.'
t_LEFTSQUAREBRACKET = r'\['
t_RIGHTSQUAREBRACKET = r'\]'
t_LEFTBRACKET = r'\{'
t_RIGHTBRACKET = r'\}'
t_EQUAL = r'\='
t_BOOLEAN = r'(false|true)'
t_STRING = r'(\"[^"]+\")'
t_LITERALSTRING = r'(\'[^\']+\')'
t_MSTRING = r'\"\"\"[^\"]*\"\"\"'
t_MLSTRING = r'\'\'\'[^\']*\'\'\''
t_INT = r'[\+\-]?(0|[1-9](\_?[0-9])*)'
t_FLOAT = r'[\+\-]?(0|[1-9](\_?[0-9])*)(\.[0-9](\_?[0-9])*([eE][\+\-]?[0-9](\_?[0-9])*)?|[eE][\+\-]?[0-9](\_?[0-9])*)'
t_INF = r'[\+\-]?inf'
t_NAN = r'[\+\-]?nan'
t_HEXADECIMAL = r'0x[0-9A-Fa-f]([0-9A-Fa-f]|_[0-9A-Fa-f])*'
t_BINARY = r'0b[01]([01]|_[01])*'
t_OCTAL = '0o[0-7]([0-7]|_[0-9])*'
t_VAR = r'(\w+|\-)+'
t_COMMENT = r'\#\s*.*'

t_ignore = ' \t'

def t_TIME(t):
    r'\d{2}:\d{2}:\d{2}(\.\d{6})?'
    return t

def t_OFFSETDATETIME (t):
    r'\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d{6})? [+-]\d{2}:\d{2}'
    return t

def t_DATETIME (t):
    r'\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}(\.\d{6})?'
    return t

def t_OFFSET (t):
    r'[+-]\d{2}:\d{2}'
    return t

def t_DATE(t):
```

```
51      r'\d{4}-\d{2}-\d{2}'
52      return t
53
54  def t_NEWLINE(t):
55      r'\n'
56      t.lexer.lineno += 1
57      return t
58
59  def t_error(t):
60      print(f"Illegal character {t.value[0]} on line {t.lineno}")
61      t.lexer.skip(1)
62
63  lexer = lex.lex()
```

Listing A.5: analisador_sintatico.py

```
import ply.yacc as yacc
import analisador_lexico as lexer
from toml import Assignment
from toml import Table

tokens = lexer.tokens

start = 'program'

def p_program(p):
    """
    program : statements
    """
    for assg in p.parser.stack:
        p.parser.final.insert(0,assg)
    for obj in p.parser.final:
        p.parser.toml.add_element(obj)

    p[0] = p[1]

def p_statements(p):
    """
        statements : statement
                   | statement statements
    """

    if isinstance(p[1],Assignment):
        p.parser.stack.append(p[1].content)
        p.parser.table_token = False
    elif isinstance(p[1],Table):
        if p[1].type == 'table_dict':
            name = p[1].name
            if p.parser.table_token:
                obj = p.parser.toml.new_assignment(name,{})
                p.parser.final.insert(0,obj)
            else:
                for assg in reversed(p.parser.stack):
                    p.parser.toml.add_element_table(p[1].data,assg)
                data = p[1].data
                obj = p.parser.toml.new_assignment(name,data)
                p.parser.final.insert(0,obj)
        elif p[1].type == 'table_list':
            name = p[1].name
            if p.parser.table_token:
                obj = p.parser.toml.new_assignment(name,[])
                p.parser.final.insert(0,obj)
            else:
                for assg in reversed(p.parser.stack):
                    obj = p.parser.toml.new_assignment(name,[assg])
                    p.parser.final.insert(0,obj)
        p.parser.stack = []
        p.parser.table_token = True
    else:
```

```
54          pass
55      p[0] = parser.toml.data
56
57
58  def p_statement(p):
59      """
60          statement : table
61                    | assignment
62                    | comment
63                    | newline
64      """
65      p.parser.size +=1
66      p[0] = p[1]
67
68  def p_newline(p):
69      """
70          newline : NEWLINE
71      """
72      p[0] = {'type': 'newline', 'value':p[1]}
73
74  def p_comment(p):
75      """
76          comment : COMMENT NEWLINE
77      """
78      p[0] = {'type': 'comment', 'value':p[1]}
79
80  def p_table(p):
81      """
82          table : header1
83                | header2
84      """
85      p[0] = p[1]
86
87  def p_header1(p):
88      """
89          header1 : LEFTSQUAREBRACKET name RIGHTSQUAREBRACKET
90                  | LEFTSQUAREBRACKET name RIGHTSQUAREBRACKET NEWLINE
91                  | LEFTSQUAREBRACKET name RIGHTSQUAREBRACKET COMMENT
92      """
93      if len(p) == 4:
94          if p.lexer.lineno == p.parser.size:
95              p[0] = Table(p[2],'table_dict')
96          else:
97              raise Exception(f'Unexpected character, table expected only newlines or
                    comments till end of line at row {p.lexer.lineno}')
98
99      else:
100         p[0] = Table(p[2],'table_dict')
101
102 def p_header2(p):
103     """
104         header2 : LEFTSQUAREBRACKET LEFTSQUAREBRACKET name RIGHTSQUAREBRACKET
                RIGHTSQUAREBRACKET
105             | LEFTSQUAREBRACKET LEFTSQUAREBRACKET name RIGHTSQUAREBRACKET
```

```
                RIGHTSQUAREBRACKET NEWLINE
106             | LEFTSQUAREBRACKET LEFTSQUAREBRACKET name RIGHTSQUAREBRACKET
                RIGHTSQUAREBRACKET COMMENT
107
108     """
109     if len(p) == 6:
110         if p.lexer.lineno == p.parser.size:
111             p[0] = Table(p[3],'table_list')
112         else:
113             raise Exception(f'Unexpected character, table expected only newlines or
                    comments till end of line at row {p.lexer.lineno}')
114
115     else:
116         p[0] = Table(p[3],'table_list')
117
118 def p_assignment(p):
119     """
120         assignment : name EQUAL elemento
121                    | name EQUAL elemento NEWLINE
122     """
123     if len(p) == 4:
124         if p.parser.size == p.lexer.lineno:
125             content = p.parser.toml.new_assignment(p[1], p[3])
126             p[0] = Assignment(content)
127         else:
128             raise Exception(f'Unexpected character, assignment expected only newlines
                    or comments till end of line')
129     else:
130         content = p.parser.toml.new_assignment(p[1], p[3])
131         p[0] = Assignment(content)
132
133 def p_assignment_object(p):
134     """
135         assignment_object : name EQUAL elemento
136     """
137     p[0] = p.parser.toml.new_assignment(p[1],p[3])
138
139 def p_name(p):
140     """
141         name : elementoVar
142              | elementoVar name2
143     """
144     if len(p) == 2:
145         p[0] = [p[1]]
146     else:
147         p[0] = [p[1]] + p[2]
148
149 def p_name2(p):
150     """
151         name2 : DOT elementoVar
152               | DOT elementoVar name2
153     """
154     if len(p)==3:
155         p[0] = [p[2]]
```

```
156     else:
157         p[0] = [p[2]] + p[3]
158 
159 def p_elemento_var(p):
160     """
161         elementoVar : VAR
162                     | string
163                     | int
164     """
165     p[0] = p[1]
166 
167 def p_lista(p):
168     """
169         lista : LEFTSQUAREBRACKET RIGHTSQUAREBRACKET
170             | LEFTSQUAREBRACKET NEWLINE lista2 RIGHTSQUAREBRACKET
171             | LEFTSQUAREBRACKET lista2 RIGHTSQUAREBRACKET
172     """
173     if len(p) == 2:
174         p[0] = []
175     elif len(p) == 5:
176         p.parser.size += 1
177         p[0] = p[3]
178     else:
179         p[0] = p[2]
180 
181 def p_lista2(p):
182     """
183         lista2 : conteudo_lista
184                 | conteudo_lista COMMA
185     """
186     p[0] = p[1]
187 
188 def p_conteudo_lista(p):
189     """
190         conteudo_lista : elemento
191                 | elemento NEWLINE
192                 | elemento COMMA conteudo_lista
193                 | elemento COMMA NEWLINE conteudo_lista
194     """
195     if len(p) == 3:
196         p.parser.size+=1
197         p[0] = [p[1]]
198     elif len(p) == 5:
199         p.parser.size+=1
200         p[0] = [p[1]] + p[4]
201     elif len(p) == 4:
202         p[0] = [p[1]] + p[3]
203     else:
204         p[0] = [p[1]]
205 
206 def p_object(p):
207     """
208         object : LEFTBRACKET RIGHTBRACKET
209                 | LEFTBRACKET conteudo_object RIGHTBRACKET
```

```
210     """
211     if len(p) == 2:
212         p[0] = {}
213     else:
214         p[0] = p[2]
215 
216 def p_conteudo_object(p):
217     """
218         conteudo_object : assignment_object
219                 | assignment_object COMMA conteudo_object
220     """
221     if len(p) == 2:
222         p[0] = p[1]
223     else:
224         p[0] = p.parser.toml.add_element_aux(p[1],p[3])
225 
226 def p_elemento(p):
227     """
228         elemento : int
229                 | float
230                 | inf
231                 | nan
232                 | hexadecimal
233                 | binary
234                 | octal
235                 | string
236                 | mstring
237                 | lstring
238                 | mlstring
239                 | boolean
240                 | date
241                 | time
242                 | datetime
243                 | offset_datetime
244                 | lista
245                 | object
246     """
247     p[0] = p[1]
248 
249 def p_mstring(p):
250     """
251         mstring : MSTRING
252     """
253     p[0] =  p[1][3:-3]
254 
255 def p_mlstring(p):
256     """
257         mlstring : MLSTRING
258     """
259     p[0] =  p[1][3:-3]
260 
261 def p_string(p):
262     """
263         string : STRING
```

```
264     """
265     p[0] =  p[1][1:-1]
266
267 def p_lstring(p):
268     """
269         lstring : LITERALSTRING
270     """
271     p[0] =  p[1][1:-1]
272
273 def p_number(p):
274     """
275         int : INT
276     """
277     p[0] = int(p[1])
278
279 def p_float(p):
280     """
281         float : FLOAT
282     """
283     p[0] = float(p[1])
284
285 def p_inf(p):
286     """
287         inf : INF
288     """
289     p[0] = p[1][1:-1]
290
291 def p_nan(p):
292     """
293         nan : NAN
294     """
295     p[0] = p[1][1:-1]
296
297 def p_hexadecimal(p):
298     """
299         hexadecimal : HEXADECIMAL
300     """
301     p[0] = p[1][1:-1]
302
303 def p_binary(p):
304     """
305         binary : BINARY
306     """
307     p[0] = p[1][1:-1]
308
309 def p_octal(p):
310     """
311         octal : OCTAL
312     """
313     p[0] = p[1][1:-1]
314
315 def p_boolean(p):
316     """
317         boolean : BOOLEAN
```

```
318     """
319     if p[1] == "true":
320         p[0] = True
321     else:
322         p[0] = False
323 
324 
325 def p_date(p):
326     """
327         date : DATE
328     """
329     p[0] = p[1][1:-1]
330 
331 def p_time(p):
332     """
333         time : TIME
334     """
335     p[0] = p[1][1:-1]
336 
337 def p_datetime(p):
338     """
339         datetime : DATETIME
340     """
341     p[0] = p[1][1:-1]
342 
343 def p_offset_datetime(p):
344     """
345         offset_datetime : OFFSETDATETIME
346     """
347     p[0] = p[1][1:-1]
348 
349 def p_error(p):
350     if p:
351         print(f"Sintax error on line {p.lineno}, column {p.lexpos + 1}: "
352               f"unexpected token '{p.value}'")
353     else:
354         print("Syntax error: unexpected end of file")
355 
356 parser = yacc.yacc()
```