

UNIVERSIDADE DO MINHO



PROCESSAMENTO DE LINGUAGENS

DEPARTAMENTO DE INFORMÁTICA

Conversor toml2json

Grupo 45:

João Abreu

Hugo Matias

João Coutinho

Número:

A84802

A85370

A86272

28 de Junho de 2020

Conteúdo

1	Introdução	1
2	Problema	2
3	Solução	3
3.1	Estrutura	3
3.2	Filtros de texto	3
3.2.1	Estado 0	3
3.2.2	Estado Value	3
3.2.3	Estado entreAspas	3
3.3	Gramática	4
3.3.1	Keys	4
3.3.2	Values	5
3.3.2.1	Tipos base	5
3.3.2.2	Array	9
3.4	Subconjunto escolhido/limitações do programa	10
3.5	Estruturas Auxiliares	11
4	Input/Output	12
5	Conclusão	14
6	Referências	15

1. Introdução

No âmbito da avaliação prática da cadeira de Processamento de Linguagens, foi-nos atribuído o problema **4**, chamado Conversor *toml2json*. Este consiste na conversão da linguagem **TOML** (*Tom's Obvious, Minimal Language*), que se trata de uma linguagem simples, fácil de ler e escrever que é usada para descrever estruturas complexas, para a sua linguagem equiparável e mais popular chamada **JSON**.

Neste projeto, pretende-se numa primeira fase desenvolver um reconhecedor de **TOML**, de seguida construir uma representação interna da respetiva estrutura e por fim, através desta, mostrar os dados no formato **JSON**.

Devido à imensidão de estruturas presentes na linguagem **TOML**, o projeto foi desenvolvido para um subconjunto da linguagem à nossa escolha, como foi referido no enunciado.

2. Problema

Escrever uma gramática independente de contexto para o reconhecimento de um subconjunto de estruturas no formato **TOML** e o respectivo analisador léxico. Criar um programa que processe uma especificação e que construa uma representação interna através de *Yacc*, com recurso a uma gramática tradutora. Por fim, com a representação criada apresentar os dados no formato **JSON**. Um exemplo da linguagem **TOML** encontra-se de seguida:

```
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"
]
```

3. Solução

3.1 Estrutura

A estrutura deste trabalho tem como processo de construção, a filtragem de um input em formato **TOML** recorrendo ao *Flex*, e por sua vez tal output irá ser usado e devidamente organizado em *Yacc* o que irá permitir a criação de uma representação interna do pretendido.

Será importante informar que, na nossa abordagem ao problema, optamos por não guardar informação em estruturas de dados e processar/reconhecer todos os dados linha-a-linha, ou seja, quando reconhecemos determinada estrutura da linguagem **TOML**, nós geramos **diretamente** os dados no formato **JSON** correspondente. Como seria de esperar, esta abordagem terá as suas limitações, que podem ser vistas no capítulo 3.4.

3.2 Filtros de texto

A linguagem **TOML** tem como base a relação key/value. Deste modo, dividimos o filtro de texto desenvolvido nesta relação. Houve a necessidade de distinguir keys de values pois certos símbolos são aceites num lado e não no outro (Ex: enters são aceites dentro de arrays nos values, no entanto são ignorados em keys).

3.2.1 Estado 0

Este estado representa as chaves da linguagem, juntamente com outras estruturas que não necessitem obrigatoriamente de values (Ex: tables).

3.2.2 Estado Value

Este estado representa os valores que vão estar associados às suas respectivas chaves. Podem ser de vários tipos (Ex: floats, inteiros, strings, etc.) que podem ser encontrados abaixo.

3.2.3 Estado entreAspas

Encontramo-nos neste estado quando estamos a processar uma string. Mal é reconhecida uma aspa de abertura de string passamos para este estado. É útil pois, devido à sua natureza, conseguimos analisar cada carácter da string e tomar decisões para cada um. Isto é necessário pois se, por exemplo, encontrarmos uma aspa já dentro de uma string, ao estarmos neste estado conseguimos adicionar o carácter extra chamado *backslash* (\) para protegermos tal aspa e permitir que seja um código **JSON** válido.

3.3 Gramática

3.3.1 Keys

As *keys* têm 3 maneiras de ser representadas: bare, quoted ou dotted. As bare keys apenas podem conter caracteres ASCII (A-Za-z0-9_-), as quoted seguem as mesmas regras das strings básicas. As dotted são uma sequência de bare ou quoted keys unidas por um ponto, permitindo agrupar propriedades semelhantes. Quando encontramos uma table assumimos como se fosse uma key e seguimos para a próxima linha.

As tabelas são coleções de pares de key / value. Elas aparecem entre parêntesis rectos numa linha. Dá para distinguir tabelas de arrays pois arrays são sempre values (temos de ter um par key = value).

Yacc	
Pair	: Key '=' Value Key '=' Array
DottedPair	: Key '.' SubKey '=' Value Key '.' SubKey '=' Array
Table	: str
Key	: str
SubKey	: str
Flex	
[=]	{ /* ----- Fim de Key ----- */ BEGIN value; return yytext[0]; }
["\'']	{ /* ----- Quotted Keys ----- */ BEGIN entreAspas; initArray(&keyvalue,20); insertArray(&keyvalue,yytext[0]); }
[A-Za-z0-9_-]+	{ /* ----- Bare Keys ----- */ // Tirar o espaço if(yytext[yytextlen-1] == ' '){ yytext[yytextlen-1] = '\0'; yylval.string = strdup(yytext); return str; }
\[[.A-Za-z0-9_-]+\]	{ /* ----- Tables ----- */ yytext[yytextlen-1] = '\0'; yylval.string = strdup(yytext+1); return str; }
[\n]	{ /* Para ignorar espaços entre pares */

3.3.2 Values

Os *values* podem ser representados por todos os tipos básicos (*Integer*, *Float*, *Boolean*, *String* e *DateTime*), assim como por outras estruturas de dados que usam estes mesmo tipo básicos para a construção de *values* mais complexos (*Arrays* e *Tables*).

3.3.2.1 Tipos base

Os tipos base são identificados no input **TOML** recorrendo a expressões regulares específicas para cada um deles. Recorrendo a identificadores definidos em *Yacc* presentes numa estrutura customizada (*struct Info*) é possível reconhecer o tipo processado de forma a ser usado de forma correta.

Yacc

```
%union{
    char* string;
    char* s;
    float f;
    int n;

    union Data {
        char* s;
        float f;
        int n;
    } data;

    struct Info{
        int uniontype; // 0 - string, 1 - inteiro, 2 - boolean, 3 - float
        union Data valor;
    } info;
}
```

De modo a que os valores identificados a partir do *Flex* estejam "conectados", foi necessário criar uma union principal onde as variáveis correspondem ao output produzido pelo *Flex*. Foi necessário criar também uma *union Data* de modo a que cada valor só pudesse ser de um tipo de cada vez. Usamos também um inteiro que serviu para identificar o tipo de dados usado no momento, uma vez um valor, ao ser processado, pode ser representado com um tipo de dados diferente.

- *String*

Processamento, identificação e representação do tipo *String* em **Flex**:

Flex

```

<valor>
...
([\\"' ]|\\\"\\\"|\\\"\\\"\\n|\\'\\'|\\'\\'|\\'\\'|\\'\\'|\\n)    {BEGIN entreAspas;
                                                                    initArray(&keyvalue,20);
                                                                    insertArray(&keyvalue,yytext[0]);
                                                                    if(strcmp(yytext, "\\\"") != 0 &&
                                                                    strcmp(yytext, "\\'") != 0)
                                                                    {inside_3_quotes = 1;};
                                                                    }
...

<entreAspas>
  (\\\"\\\"|\\'\\'|\\'\\')/[\\ \n#]    /* Representa o fim de multi line strings */
                                   BEGIN value;
                                   yynval.info.valor.s = strdup(getText(&keyvalue)+1);
                                   yynval.info.uniontype = 0;
                                   inside_3_quotes = 0;
                                   return val;
                                   }
...

```

- *Integer*

Processamento, identificação e representação do tipo *Integer* em **Flex**:

Flex

```

<valor>
...
[-+]?[0-9]+                {yynval.info.uniontype = 1;
                              return val;}
...

```


- *Boolean*

Processamento, identificação e representação do tipo *Boolean* em **Flex**:

Flex

```

<valor>
...
(true|false)                {yyval.info.uniontype = 2;
                             yyval.info.valor.s = strdup(yytext);
                             return val;}
...

```

- *Float*

Processamento, identificação e representação do tipo *Float* em **Flex**:

Flex

```

<valor>
...
[-+]?[0-9]*\.[0-9]+([eE][-+]?[0-9]+)?$ {yyval.info.uniontype = 3;
                                         return val;}
...

```

- *Offset Date-Time & Local Date-Time & Local Date*

Processamento, identificação e representação dos tipos *Offset Date-Time & Local Date-Time & Local Date* com formato RFC 3339 em **Flex**:

Flex

```

OffsetDateTime (([0-9]+)-([01-9]|1[012])-([01-9]|12)[0-9]|3[01])
                ([Tt]([01][0-9]|2[0-3]):([0-5][0-9]):
                ([0-5][0-9]|60)(\.[0-9]+)?
                (([Zz])|([\+|\-]([01][0-9]|2[0-3]):[0-5][0-9]))?)?)

<valor>
...
{OffsetDateTime}|{LocalTime} {yyval.info.uniontype = 0;
                              yyval.info.valor.s = strdup(yytext);
                              return val;}
...

```

- *Local-Time*

Processamento, identificação e representação do tipo *Local-Time* com formato RFC 3339 em **Flex**:

Flex

```
LocalTime (([01][0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]|60)(\.[0-9]+)?)
<valor>
...
{OffsetDateTime}|{LocalTime}    {yyval.info.uniontype = 0;
                                   yyval.info.valor.s = strdup(yytext);
                                   return val;}
...
```

3.3.2.2 Array

A estrutura de dados *Array* foi implementada tendo em conta a identificação dos símbolos terminais constituintes deste ("*/*", "*,*" e "*/*"). Para os elementos do array serão usados os tipo básicos definidos acima. É também de importante referência que estão também implementados *Nested Arrays*, estes foram possíveis de implementar recorrendo à recursividade à esquerda como se poderá verificar no código em *Yacc*.

Flex

```
<value>
...
\[          {inside_array = 1;return yytext[0];}
\[          {inside_array = 0;return yytext[0];}
,          {return yytext[0];}
...
```

Yacc

```
Array : '[' Elem ']'
Elem  : Value
      | Elem ',' Value
```

Implementação de *Nested Arrays* recorrendo à recursividade à esquerda.

```
| Array
| Elem ',' Array
;
```

3.4 Subconjunto escolhido/limitações do programa

A estratégia que usamos para escolher o subconjunto da linguagem foi primeiro implementar todas as funcionalidades básicas, como tipos de dados e estruturas indispensáveis. Optamos por não incluir as alternativas que a linguagem apresenta de estruturas básicas, por exemplo, apenas aceitamos *tables* e não *inline-tables*, pois estas últimas são uma alternativa das primeiras. Não incluímos também funcionalidades que, apesar de não se tratarem de alternativas de outras, não foram possíveis devido à nossa abordagem do problema de geração da informação no formato **JSON direta** a partir da gramática, como foi explicado anteriormente no capítulo 3.1. A lista de funcionalidades que não consideramos oportunas incluir são as seguintes:

- Inline tables
- Dottedkeys fora de ordem
- Uso de underscores para melhorar a legibilidade de um número grande (Ex: 5_349_221)
- Inteiros no formato binário e hexadecimal
- Valores infinitos ou NaN
- Espaços entre Dotted Tables (Ex: [g . h . i])

A lista de funcionalidades que não conseguimos incluir devido à nossa limitação do programa são as seguintes:

- Array of Tables
- Identação de Arrays
- Subtables de subtables (Ex: dog.tater.man)
- Dottedkeys de dottedkeys (Ex: key.value.type)

Gostaríamos de reforçar a ideia que todas as propriedades acima apresentadas que não conseguimos incluir seriam bastante plausíveis de se realizarem se tivéssemos lido todo o ficheiro input, guardássemos tudo em estrutura de dados, e de seguida gerado toda a informação estruturada em linguagem **JSON**.

3.5 Estruturas Auxiliares

A única estrutura de dados auxiliar que usada foi com o propósito de armazenamento dinâmico em memória do conteúdo de strings. Este é usado durante o filtro de texto *flex* quando estamos a processar uma string. Como não sabemos quão longa é uma string decidimos criar esta ferramenta. Foi desenhada da seguinte forma:

```
typedef struct{
    char *array;
    int used;
    int size;
} Array;

void initArray(Array *a, int initialSize);
void insertArray(Array *a, char element);
void closeArray(Array *a);
char *getText(Array *a);
void freeArray(Array *a);
int aspaOrPelica(Array *a); /* Serve apenas para propósitos de controlo de aspas/pelicas */
                             /* 0 se for primeiro char for aspa,
                             1 se for pelica, -1 se nenhum */
```

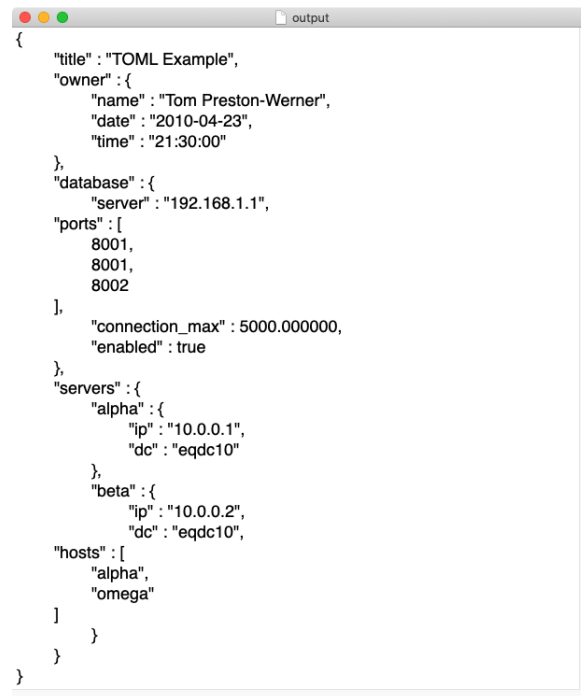
4. Input/Output

Juntamente com o nosso programa final, colocamos uma pasta de inputs possíveis ao nosso trabalho, de modo a facilitar **todas** as estruturas que decidimos incluir na nossa gramática. Para o output, criamos um pequeno script na makefile que o faz automaticamente, dado um path para um ficheiro input à escolha do utilizador, abrindo o respetivo output após ser gerado. O output será guardado num ficheiro chamado *output*. Os passos para recriar a criação do ficheiro output encontram-se abaixo:

```
> make
yacc toml2json.y
flex toml2json.l
cc y.tab.c -o toml2json Array.c

> make run I=inputs/input
./toml2json < inputs/input > output
open output
```

Em baixo encontra-se o ficheiro output se o input for "*inputs/input*". Este input particular trata-se do exemplo mostrado no enunciado.



```
{
  "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": 5000.000000,
    "enabled": true
  },
  "servers": {
    "alpha": {
      "ip": "10.0.0.1",
      "dc": "eqdc10"
    },
    "beta": {
      "ip": "10.0.0.2",
      "dc": "eqdc10",
    },
    "hosts": [
      "alpha",
      "omega"
    ]
  }
}
```

Figura 4.1: Output do exemplo do enunciado

No que conta à indentação do ficheiro produzido, esta foi sempre tida em atenção e tentamos gerar o ficheiro o mais fácil de ler possível. No entanto não conseguimos fazer com que os arrays seguissem as regras de indentação, fazendo com que o output se torne mais complicado de analisar junto a estas estruturas. Em todas as outras a indentação foi realizada com sucesso.

5. Conclusão

A solução de software resultante permite, de uma forma simples e eficaz, reconhecer uma especificação em **TOML**. Adicionalmente, através da gramática tradutora, é criada um ficheiro na linguagem **JSON** diretamente à medida que vamos reconhecendo cada linha de entrada. O subconjunto foi escolhido de maneira que incluíssemos as partes essenciais da linguagem, no entanto, mesmo assim ficamos limitados a tal devida à nossa abordagem do problema usando o reconhecimento-geração de **JSON** direto, sem nenhum armazenamento de informação em estrutura de dados.

6. Referências

Para a aprendizagem da linguagem **TOML**, sua respetiva validação juntamente com a validação de ficheiros **JSON** foram usadas as seguintes ferramentas:

- <https://pseitz.github.io/toml-to-json-online-converter> (Conversor TOML -> JSON)
- <https://www.toml-lint.com> (TOML validator)
- <https://jsonlint.com> (JSON validator)
- <https://toml.io/en/v1.0.0-rc.1> (Syntax TOML)