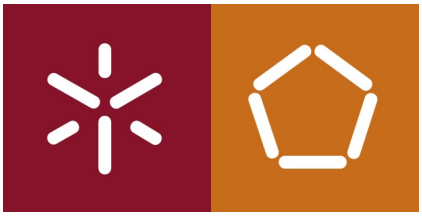

Universidade do Minho
Mestrado Integrado em Engenharia Informática
3ºano - 2º Semestre

Processamento de Linguagens

Trabalho Prático Nº.2 (YACC)

Grupo 33



a83732 – Gonçalo Rodrigues Pinto
a84197 – João Pedro Araújo Parente
a84829 – José Nuno Martins da Costa

28 de Junho de 2020

Conteúdo

Lista de Figuras

1 Introdução

No 2º semestre do 3º ano do Curso de Engenharia Informática da Universidade do Minho, existe uma unidade curricular denominada por Processamento de Linguagens, que tem como objectivo ajudar os estudantes a especificar linguagens de domínio específico através de gramáticas e/ou expressões regulares, a desenvolver processadores para essas linguagens, transformar qualquer formato textual num outro formato e especificar e implementar "front-ends" e "back-ends" para qualquer tipo de aplicação.

O presente trabalho teve como principais objectivos aumentar a experiência de uso do ambiente Linux, da linguagem imperativa C (para codificação das estruturas de dados e respectivos algoritmos de manipulação), e de algumas ferramentas de apoio à programação, também de rever e aumentar a capacidade de escrever gramáticas independentes de contexto (GIC), que satisfaçam a condição LR(), para criar Linguagens de Domínio Específico (DSL) além de desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora (GT) e por fim utilizar geradores de compiladores como o par flex/yacc.

O enunciado deste trabalho era composto por diferentes enunciados, dos quais foi nos solicitado resolver um escolhido em função do número do grupo (NGr), usando a fórmula $exe = (NGr\%6) + 1$. Dado que o nosso grupo é o número 33 corresponde assim sendo ao enunciado número **4**, *Conversor toml2json*.

2 Contextualização

TOML é uma linguagem simples, fácil de ler e escrever, para descrever estruturas complexas (usada frequentemente em ficheiros de configuração), desenhada para ser mapeada directamente e sem ambiguidades para dicionários generalizados. TOML, cujo nome significa *Tom's Obvious, Minimal Language* devido ao seu criador Tom Preston-Werner, é usada em muitos projetos de software sendo equiparável a JSON ou YAML.

Para ilustrar esta linguagem apresenta-se abaixo um exemplo da mesma.

```
# This is a TOML document.
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
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]
  # Indentation (tabs and/or spaces) is allowed but not required
  [servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

  [servers.beta]
  ip = "10.0.0.2"
  dc = "eqdc10"

[clients]
data = [ ["gamma", "delta"], [1, 2] ]

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

No contexto deste trabalho prático, pretendeu-se escrever uma gramática que cubra um subconjunto da linguagem TOML à nossa escolha como também construir um processador (flex, yacc) que reconheça e valide estruturas/dicionários escritos na DSL TOML definida acima, gerando o JSON correspondente. Para atingir então esse objectivo o nosso trabalho seguiu e respeita as regras apresentadas na documentação oficial desta linguagem, presente no `https://toml.io/en/` (consultado no dia 27 de Junho de 2020) como foi também guiou-se pelo repositório que contém a versão em desenvolvimento da especificação TOML (`https://github.com/toml-lang/toml`) (consultado no dia 27 de Junho de 2020).

3 Linguagem TOML

O TOML pretende ser um formato de um documento de configuração mínimo que seja fácil de ler devido à semântica óbvia. O TOML foi projectado para mapear sem ambiguidade numa tabela de hash. O TOML é fácil de analisar as estruturas de dados numa ampla variedade de linguagens.

Partindo dos sites acima referidos podemos constatar que TOML possui tipos nativos úteis tais como pares de chave / valor, listas, tabelas, tabelas "em linha", listas de tabelas, inteiros como decimais, valores lógicos e datas como também horas com a opção de *offsets*. No URL `https://toml.io/en/v1.0.0-rc.1` (consultado no dia 27 de Junho de 2020) podemos observar algumas transformações destes tipos para JSON, assim o programa desenvolvido que gera JSON foi baseado nestes exemplos encontrados.

No seguimento do mesmo link podemos observar que representar tabelas "em linha"é idêntico a escrever uma tabela em JSON logo por esta razão o nosso subconjunto da linguagem TOML não abrange este conceito de tabela "em linha"em vez escrevemos uma tabela.

3.1 Tipos considerados

3.1.1 Pares de chave / valor

O principal componente básico de um documento TOML é o par de chave/valor. As chaves estão à esquerda do sinal de igual e os valores à direita. O espaço em branco é ignorado em torno dos nomes e valores das chaves. A chave, sinal de igual e valor devem estar na mesma linha. O valor deve ser um destes tipos : uma palavra, inteiro, decimal, valor lógico, data e/ou hora e uma lista.

Uma chave pode ser simples, entre aspas ou constituída por pontos.

- **Simples:** pode conter apenas letras ASCII, dígitos ASCII, _ e -. Pode ser composta apenas por dígitos ASCII, por exemplo 1234, mas sempre interpretado como uma chave única;
- **Entre aspas:** exactamente as mesmas regras que a a chave simples mas permite usar um conjunto muito mais amplo de nomes de chave;
- **Constituída por pontos:** são uma sequência de chaves simples ou entre aspas unidas por um ponto, o que permite agrupar propriedades semelhantes.

3.1.2 Palavras

Existem quatro maneiras de expressar sequências de caracteres: básica, básica de várias linhas, literal e literal de várias linhas. Todas as cadeias devem conter apenas caracteres UTF-8 válidos.

Sequência de caracteres básica são limitadas por aspas. Qualquer caractere Unicode pode ser usado, excepto aqueles que precisam ser delimitados: aspas, barra invertida e caracteres de controlo.

Sequência de caracteres básica de várias linha são limitadas por três aspas de cada lado e permitem novas linhas. Uma nova linha imediatamente após o delimitador de abertura será descartada. Todos os outros caracteres de espaço em branco e nova linha permanecem intactos.

Sequência de caracteres literal são limitadas por aspas simples e como cadeias básicas, elas devem aparecer em uma única linha.

Sequência de caracteres literal de várias linhas são limitadas por três aspas simples de cada lado e permitem novas linhas. Uma nova linha imediatamente após o delimitador de abertura será cortada. Todo o outro conteúdo entre os delimitadores é interpretado como está, sem modificação.

3.1.3 Inteiros e Decimais

Inteiros são números inteiros. Números positivos podem ser prefixados com um sinal de mais. Os números negativos são prefixados com um sinal de menos. Para números grandes, podemos utilizar _ entre dígitos para melhorar a legibilidade, cada _ deve estar entre pelo menos um dígito de cada lado. Valores inteiros não negativos também podem ser expressos em hexadecimal, octal ou binário. Os valores hexadecimais não diferenciam maiúsculas de minúsculas. Os _ são permitidos entre dígitos (mas não entre o prefixo e o valor).

Um decimal consiste em uma parte inteira (que segue as mesmas regras que valores inteiros decimais) seguida por uma parte fracionária e /ou uma parte do expoente. Se uma parte fracionária e a parte expoente estiverem presentes, a parte fracionária deverá preceder a parte expoente. Uma parte fracionária é um ponto decimal seguido por um ou mais dígitos. Uma parte do expoente é um E (maiúscula ou minúscula) seguido por uma parte inteira (que segue as mesmas regras que valores inteiros decimais, mas pode incluir zeros à esquerda). Semelhante aos números inteiros, podemos usar _ para melhorar a legibilidade, cada _ deve estar entre pelo menos um dígito. Pode existir ainda valores decimais especiais que podem ser expressos, eles são sempre minúsculos que são o infinito(inf) e a representação de um não número (nan).

3.1.4 Valores Lógicos

Valores Lógicos são apenas os valores aos quais você estamos acostumados, *true* ou *false*, sempre em minúsculas.

3.1.5 Datas e horas com a opção de offsets

Para representar inequivocamente um instante específico no tempo, podemos usar uma data e hora no formato RFC3339 com *offset*, por exemplo, 1979-05-27T00:32:00.999999-07:00. Para facilitar a leitura, podemos substituir o delimitador T entre data e hora por um espaço.

É possível também omitir o *offset* de uma data e hora formatada pela RFC3339, ela representa a data e hora especificadas sem nenhuma relação com um fuso horário.

Como também é possível incluir apenas a parte da data de uma data e hora formatada, representando o dia inteiro sem nenhuma relação com fuso horário e ainda pode-se incluir apenas a parte da hora de uma data e hora formatada, representando assim a hora do dia sem nenhuma relação com um dia específico ou fuso horário

3.1.6 Listas

Listas encontram-se limitadas por [ ] com valores internos. Os elementos são separados por vírgulas. As listas podem conter valores dos mesmos tipos de dados permitidos nos pares chave / valor. Valores de tipos diferentes podem ser misturados.

Podem ainda abranger várias linhas. Pode haver um número arbitrário de novas linhas e comentários antes de um valor e antes do] de fecho.

3.1.7 Tabelas

As tabelas (também conhecidas como tabelas de hash ou dicionários) são colecções de pares de chave / valor. Elas aparecem entre [] em uma linha sozinhas, podendo diferenciar das listas porque elas contém apenas valores.

Abaixo de uma tabela, e até a próxima tabela ou fim do ficheiro são pares de chave/valor dessa tabela contudo não é garantido que os pares de chave/-valor nas tabelas estejam numa ordem específica. As regras de nomenclatura para tabelas são as mesmas que para as chaves (descrita acima) e tal como elas não pode definir nenhuma tabela mais de que uma vez. Tabelas vazias são permitidas e simplesmente não possuem pares de chave/valor dentro delas.

As chaves constituída por pontos definem tudo à esquerda de cada ponto como uma tabela. Como as tabelas não podem ser definidas mais de uma vez, a redefinição dessas tabelas usando um cabeçalho [tabela] não é permitida. Da mesma forma, o uso de chaves constituída por pontos para redefinir as tabelas já definidas no formato [tabela] não é permitido.

Contudo a forma [tabela] pode, no entanto, ser usado para definir sub-tabelas dentro de tabelas definidas através de chaves constituída por pontos.

3.1.8 Listas de Tabelas

O último tipo considerado expresso é uma lista de tabelas, isto pode ser expresso usando um nome de tabela entre [] duplos. Abaixo disso, e até a próxima tabela ou o fim do ficheiro são os chave/valor dessa lista. Cada tabela com o mesmo nome entre [] duplos e é um elemento na lista de tabelas. As tabelas são inseridas na ordem encontrada. Uma tabela com [] duplos sem pares de chave / valor será considerada uma tabela vazia.

3.2 Gramática Independente de Contexto

Nesta secção iremos definir a linguagem criada de forma a solucionar o problema, analisando todos os tipos acima referidos. De acordo com o estudado ao longo do semestre, define-se uma gramática para a representação de uma linguagem imperativa como a junção dos quatros conjuntos $< T, N, S, P >$, respectivamente Símbolos Terminais, Não-Terminais, Axioma da gramática e Produções.

3.2.1 Terminais

Os símbolos terminais são os que podem aparecer como entrada ou saída de uma produção, dos quais não se pode derivar mais nenhuma unidade. Por convenção, foram escritos a letra minúscula. As suas definições explicitam adequadamente as suas funções na execução do programa.

```
T = { 'string', 'datahora', 'dataa',
        'hora', 'hora', 'expo',
        'numero',  'numero_decimal, 'trueOrfalse'
    }
```

3.2.2 Não-Terminais

Os símbolos terminais são os que podem aparecer como saída de uma produção, dos quais obrigatoriamente deriva uma ou mais unidades. Por convenção foram escritos a letra maiúscula.

```
NT = { 'ListaAtribuicoes', 'Tabs', 'Tipo',
        'Tabela', 'Nome', 'Atribuicao', 'Valor', 'Array'
    }
```

3.2.3 Axioma

O axioma é o raiz da árvore de derivação, do qual deriva a primeira produção.

```
S = { 'TOML' }
```

3.2.4 Produções

Uma gramática é definida pelas regras de produção que especificam que símbolos podem substituir outros. Todas as derivações do conjunto de testes fornecido seguem as seguintes regras.

```
P = {
    p1: TOML -> ListaAtribuicoes Tabs
    p2: Tabs -> Tabs Tipo
    p3: Tabs ->
    p4: Tipo -> 't' Tabela
    p5: Tipo -> 'a' Tabela
    p6: Tabela -> Nome ':' ListaAtribuicoes
    p7: Nome -> Nome string
    p8: Nome ->
    p9: ListaAtribuicoes -> ListaAtribuicoes Atribuicao
    p10: ListaAtribuicoes ->
    p11: Atribuicao -> Nome '=' Valor
    p12: Valor -> string
    p13: Valor -> numero
    p14: Valor -> numero_decimal
    p15: Valor -> trueOrfalse
    p16: Valor -> expo
    p17: Valor -> dataa
    p18: Valor -> datahora
    p19: Valor -> hora
    p20: Valor -> '[' Array ']'
    p21: Array -> Array ',' Valor
    p22: Array -> Valor
}
```

A abordagem tomada para a criação da gramática independente de contexto que melhor representa linguagem a reconhecer foi uma top-down. A especificação da linguagem TOML encontra-se essencialmente divida em duas partes, uma em que são descritas atribuições gerais (ListaAtribuicoes) e a outra em que se encontram descritos as tabelas (Tabs).

A parte onde se encontram descritas as atribuições gerais é representada por uma lista de atribuições (p9 e p10) onde uma atribuição (p11) pode ser composta por um par chave/valor separado pelo símbolo igual. Esta chave tal como foi referido anteriormente pode ser simples ou uma sequência de chaves simples ou entre aspas unidas por um ponto (p7 e p8). O Valor ora descreve uma palavra, número, decimal, valor lógico, os diferentes tipos de datas com as características referidas também descreve uma sequência de valores (de p12 a p22).

A outra parte, que compreende as tabelas (p2 e p3) é subdivida no tipo de tabela em questão pois pode existir uma tabela simples ou uma lista de tabelas, assim é necessário identificar o tipo em causa (p4 e p5). Uma tabela tal como foi referido é simplesmente uma chave dentro de [] que possui várias as atribuições associadas (p6).

3.3 Implementação

3.3.1 Estruturas de Dados Utilizadas

De forma a criar um conversor toml2json recorreu-se às colecções GLib para gerir os dados de forma eficiente e elegante neste programa desenvolvido em linguagem C pois fornecem estruturas de dados mais complexos (como as funções e variáveis necessárias para manipular os dados) que são escassas nesta linguagem.

Tal como foi referido previamente o TOML foi projectado para mapear sem ambiguidade numa tabela de hash logo utilizou-se duas tabelas de hash disponibilizadas pela GLib, uma (*GHashTable**) designada de **atribuicoes** criada com o intuito de armazenar as atribuições que se encontram antes da primeira tabela e a outra tabela de hash designada de **tabelas** que por sua vez vai permitir guardar as tabelas como também atribuições e sub-tabelas associadas a uma determinada tabela.

Desta forma, para representar o conteúdo da tabela hash **atribuicoes** criou-se uma estrutura designada Atrib que vai permitir guardar a atribuição em si como também a chave associada a esta, tal como foi referido, a chave pode ser constituída por pontos levando à existência de uma hierarquia, neste sentido representou-se as chaves com uma colecção GLib disponibilizada que representa uma lista (*GSList**), com isto é possível criar os diferentes níveis entre chaves.

Por fim, para representar o conteúdo da tabela hash **tabelas** criou-se uma estrutura designada Values que vai permitir que guardar a informação das atribuições das tabelas e sub-tabelas associadas, para o primeiro caso, foi necessário utilizar outra tabela hash *atr* que vai representar as atribuições associada à tabela e para o segundo caso *filhos* criou-se uma nova estrutura de dados designada Tabela que possui duas listas através *GSList**, uma *nome* que representa a hierarquiedade do nome das tabelas e outra *at* que representa as várias de atribuições associadas à tabela.

```
typedef struct atrib {
    GSList* nome;
    char* valor_json;
} * Atrib;
typedef struct tabela {
    GSList* nome;
    GSList* at;
} * Tabela;
typedef struct values {
    GHashTable* atr;
    GHashTable* filhos;
    int flag_tabela_ou_array;
} * Values;
GHashTable* tabelas;
GHashTable* atribuicoes;
```

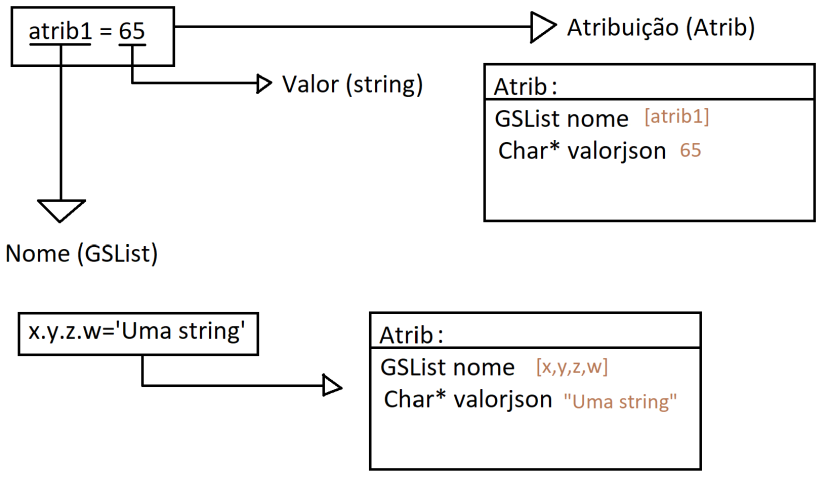

Figura 1: Exemplo da estrutura Atrib

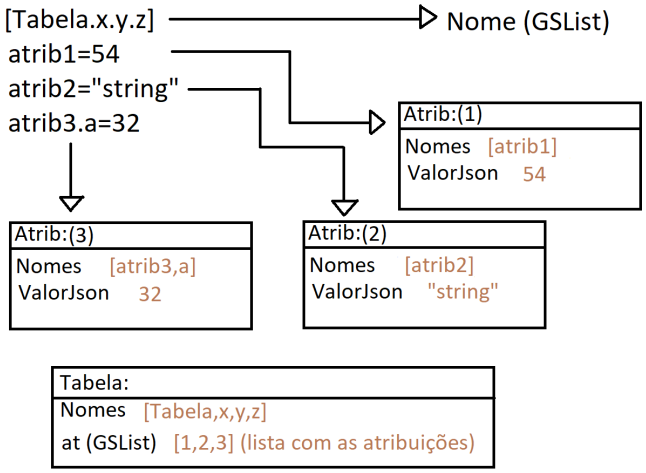

Figura 2: Estrutura Tabela, que posteriormente é passada como argumento à função 'insereTabela'

3.3.2 Analisador léxico

O analisador léxico desempenhou a função de traduzir a sequência de caracteres de entrada num conjunto de símbolos léxicos pré determinados que constituem as componentes da linguagem a reconhecer. Associado a cada um desses símbolo existe uma expressão regular que encapsula todas as instanciações da respectiva componente.

No analisador léxico decidimos criar alguns estados que conseguem representar facilmente a atribuição da linguagem TOML que é o elemento fundamental da mesma linguagem. Uma atribuição pode ser dividida em 2 partes fundamentais: chaves e valor e portanto criamos um estado para cada uma delas responsável de as reconhecer, nomeadamente os estados Chave e Valor. Ao longo do reconhecimento do ficheiro o programa gerado pelo flex andará em loop entre estes estados reconhecento todas as atribuições do ficheiro e passondo-as ao yacc depois do devido tratamento, que se pode observar ao pormenor nos anexos.

3.3.3 Gramática tradutora

Através do *yacc* foi criada uma gramática tradutora que teve como objectivo instanciar as estruturas de dados criados partindo da leitura de informação.

Referenciando a secção "Gramática Independente de Contexto", quando a gramática reconhece um conjunto de atribuições (combinação chave/valor) antes de aparecer a primeira tabela, preenche-se a tabela de hash **atribuicoes** com todas as encontradas, para que no fim do reconhecimento seja possível percorrer essa tabela de hash e imprimir no ecrã.

Quando a gramática reconhece uma tabela, é construída uma tabela do tipo da estrutura de dados Tabela, com o nome da tabela e as atribuições feitas dentro desta, inserimos assim a tabela criada na tabela de hash **tabelas**, sendo que esta tabela de hash faz a verificação se a tabela em questão é uma sub-tabela de outra ou não, se for, é colocada as atribuições no nível correcto, ou seja, para que seja possível referenciar novamente no ficheiro .toml. No final do reconhecimento esta tabela de hash **tabelas** é percorrida e as tabelas e as suas atribuições são impressas para o ecrã.

3.4 Resultados

```
{ "title": "TOML Example",
"owner": {
    "dob": "1979-05-27T07:32:00-08:00",
    "name": "Tom Preston-Werner"
},
"servers": {
    "beta": {
        "dc": "eqdc10",
        "ip": "10.0.0.2"
    },
    "alpha": {
        "dc": "eqdc10",
        "ip": "10.0.0.1"
    }
},
"clients": {
    "hosts": [
        "alpha",
        "omega"
    ],
```

```
        "data": [
            [
                "gamma",
                "delta"
            ],
            [
                1,
                2
            ]
        ]
    },
    "database": {
        "server": "192.168.1.1",
        "ports": [
            8001,
            8001,
            8002
        ],
        "connection_max": 5000,
        "enabled": true
    }
}
```

4 Conclusão

O presente relatório descreveu, de forma sucinta, o desenvolvimento do programa "toml2json"capaz de reconhecer e validar estruturas escritos na Linguagem de Domínio Específico TOML gerando o JSON correspondente.

Após a realização deste trabalho, ficamos conscientes das potencialidades que processadores de linguagens desenvolvidos segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora, utilizando para isso geradores de compiladores como o par flex/yacc, no sentido de permitir a fácil criação de uma Linguagem de Domínio Especifico o que permite representar bastante informação como por exemplo uma linguagem de programação.

Consideramos que os principais objectivos foram cumpridos.

Sentimos que a realização deste trabalho prático consolidou os nossos conhecimentos no uso do ambiente Linux, da linguagem imperativa C, de escrever gramáticas independentes de contexto para criar Linguagens de Domínio Específico.

5 Anexos

5.1 Analisador Léxico

```
[A-Za-z0-9_-]+  { yylval.str=strdup(yytext); return string; }
\"([^"\\]|\\.)+\" { yytext[yyleng-1]='\0';
                    yylval.str=strdup(yytext+1);
                    return string;
                  }
=               { BEGIN Valor; return yytext[0]; }
\[              { BEGIN Tabela; return 't'; }
\[\[            { BEGIN Tabela; return 'a'; }
<Tabela>[A-Za-z0-9_-]+      { yylval.str=strdup(yytext); return string; }
<Tabela>\"([^"\\]|\\.)+\"    { yytext[yyleng-1]='\0';
                               yylval.str=strdup(yytext+1);
                               return string;
                             }
<Tabela>\]  { BEGIN Chave; return ':'; }
<Tabela>\]\]    { BEGIN Chave; return ':'; }

<Chave>[A-Za-z0-9_-]+   { yylval.str=strdup(yytext); return string; }
<Chave>\"([^"\\]|\\.)+\"    { yytext[yyleng-1]='\0';
                    yylval.str=strdup(yytext+1);return string; }
<Chave>\[   { BEGIN Tabela; return 't';}

<Chave>\[\[ { BEGIN Tabela;return 'a';}

<Chave>=    { BEGIN Valor; return yytext[0];}


<Valor>(?i:true)    {   yylval.num = 1;
                        if (flag_array==0 ) { BEGIN Chave; }
                        return (trueOrfalse);
                    }

<Valor>(?i:false)   {   yylval.num = 0;
                        if (flag_array==0 ) { BEGIN Chave; }
                        return (trueOrfalse);
                    }

<Valor>([+\-])?(i?nan)  {   yylval.str = strdup("NaN");
                            if (flag_array==0 ) { BEGIN Chave; }
                            return string;
                        }
<Valor>([+\-])?(i?inf)  {  if (strlen(yytext) == 4) asprintf(&yylval.str, "%cInfinity", yytext[0]);
                            else yylval.str = strdup("Infinity");
                            if (flag_array==0 ) { BEGIN Chave; }
                            return string;
                        }
<Valor>([+\-])?[0-9]+   {yylval.num=atoi(yytext);
                           if (flag_array==0) { BEGIN Chave; }
                                                return numero;
                                        }

<Valor>([+\-])?[0-9]+(_[0-9]+)+             {   removeChar(yytext,'_');
                                                yylval.num=atoi(yytext);
                                                if (flag_array==0) { BEGIN Chave; }
                                                return numero;
        }

<Valor>([+\-])?[0-9]+(\.[0-9]+)?            {   yylval.fnum=atof(yytext);
                                                if (flag_array==0 ) { BEGIN Chave; }
                                                return numero_decimal;
                                            }

<Valor>([+\-])?[0-9]+(_[0-9]+)+(\.[0-9]+(_[0-9]+)+)?            {   removeChar(yytext,'_');
                                                                    yylval.fnum=atof(yytext);
                                                                    if (flag_array==0) { BEGIN Chave; }
                                                                    return numero_decimal;
                                                                }
```

```
<Valor>([+\-])?[0-9](\.[0-9]+)?[eE]([+\-])?[0-9]+                   {   if (yytext[0] == '+') {
                                yylval.str = strdup(yytext + 1);
                              }
                              else {
                                yylval.str = strdup(yytext);
                            }
                                                                        if (flag_array==0) { BEGIN Chave; }
                              return expo;
                            }

<Valor>[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[1-2][0-9]|3[01])[T ](0[0-9]|1[0-9]|2[0-4]):([0-5][0-9]|60):([0-5][0-9](\.[0-9]+)?|60)(Z|[\-+](0|1)[0-9]:([0-5][0-9]|60))?      {
                                                    yylval.str = strdup(yytext); if (flag_array==0) { BEGIN Chave; } return datahora;
                                                    }

<Valor>[0-9]{4}-(0[1-9]|1[0-2])-(0[1-9]|[1-2][0-9]|3[01])  { yylval.str = strdup(yytext); if (flag_array==0) { BEGIN Chave; } return dataa; }

<Valor>(0[0-9]|1[0-9]|2[0-4]):([0-5][0-9]|60):([0-5][0-9](\.[0-9]+)?|60)  { yylval.str = strdup(yytext); if (flag_array==0) { BEGIN Chave; } return hora; }

<Valor>0(x[0-9a-fA-F]{8}|x[0-9a-fA-F]{4}_[0-9a-fA-F]{4}|o[0-8]{8}|o[0-8]{3}|b[0-1]{8})            { yylval.str=strdup(yytext); if (flag_array==0){BEGIN Chave;} return string; }

<Valor>[A-Za-zO-9_-]+                                               {   yylval.str=strdup(yytext);
                                                                        if (flag_array==0)BEGIN Chave;
                                                                        return string;
                                                                    }

<Valor>\"([^"\\]|\\.)+\"                                            { yytext[yyleng-1]='\0'; yylval.str=strdup(yytext+1); if(flag_array==0) { BEGIN Chave; } return string; }

<Valor>\"\"\"\n([^"\\]|\\.)+\"\"\"                                  { yytext[yyleng-3]='\0'; yylval.str=strdup(yytext+4); if(flag_array==0) { BEGIN Chave; } return string; }

<Valor>\"\"\"([^"\\]|\\.)+\"\"\"                                    { yytext[yyleng-3]='\0'; yylval.str=strdup(yytext+3); if(flag_array==0) { BEGIN Chave; } return string; }

<Valor>'[^']+'                                                      { yytext[yyleng-1]='\0'; yylval.str=strdup(yytext+1); if(flag_array==0) { BEGIN Chave; } return string; }

<Valor>'''\n[^']+'''                                               { yytext[yyleng-3]='\0'; yylval.str=strdup(yytext+4); if(flag_array==0) { BEGIN Chave; } return string; }

<Valor>'''[^']+'''                                                 { yytext[yyleng-3]='\0'; yylval.str=strdup(yytext+3); if(flag_array==0) { BEGIN Chave; } return string; }

<Valor>\[                                                           { flag_array++; return yytext[0]; }

<Valor>,                                                            { return yytext[0]; }

<Valor>\]                                                           { flag_array--; if(flag_array==0) { BEGIN Chave; } return yytext[0];}


<*>#.+      { ; }

<*>.|\n     { ; }
```

5.2 Gramática Tradutora

```
%union{ char*str; int num; float fnum; GSList * list; Atrib at; Tabela t; }

%token <str>  string expo datahora hora dataa
%token <num>  numero trueOrfalse
%token <fnum> numero_decimal

%type <str> Valor Array
%type <at> Atribuicao
%type <t>  Tabela
%type <list> Nome
%type <list> ListaAtribuicoes


%%
TOML :  ListaAtribuicoes Tabs                  { insereAtribuicoes($1); }
     ;


Tabs : Tabs Tipo
     |
     ;


Tipo : 't' Tabela                              {   insereTabela($2,1); }
     | 'a' Tabela                              {   insereTabela($2,0); }
     ;




Tabela : Nome ':' ListaAtribuicoes             {
                        Tabela t = malloc(sizeof(struct tabela));
                        t->nome = $1;
                        t->at=$3;
                        $$ = t;}
       ;
```

```
Nome : Nome string                                  { $$ = g_slist_append ($1,$2); }
     |                                              { $$ = NULL; }
     ;


ListaAtribuicoes : ListaAtribuicoes Atribuicao      { $$=g_slist_append ($1,$2); }
                 |                                  { $$=NULL; }
                 ;


Atribuicao : Nome '=' Valor                         {
                                Atrib a = malloc(sizeof(struct atrib));
                                a->nome = $1;
                                a->valor_json = strdup($3);
                                $$ = a;}
           ;

Valor :   string                                    { asprintf(&$$,"\"%s\"",$1); }
      |   numero                                    { asprintf(&$$,"%d",$1); }
      |   numero_decimal                            { asprintf(&$$,"%f",$1); }
      |   trueOrfalse                               { if($1) $$="true";
                                                      else $$="false"; }
      |   expo                                      { asprintf(&$$,"%s",$1); }
      |   dataa                                     { asprintf(&$$,"\"%s\"",$1); }
      |   datahora                                  { asprintf(&$$,"\"%s\"",$1); }
      |   hora                                      { asprintf(&$$,"\"%s\"",$1); }
      |   '[' Array ']'                             { asprintf(&$$,"[ %s ]",$2);}
      ;

Array :  Array ',' Valor                            { asprintf(&$$,"%s,%s",$1,$3); }
      |  Valor                                      { asprintf(&$$,"%s",$1); }
      ;
%%
```

5.2.1 Makefile

```
CC=cc
GLIB_LIBS=‘pkg-config --cflags --libs glib-2.0‘

build:
flex toml2json.l
yacc -d -v toml2json.y
$(CC) -o toml2json y.tab.c $(GLIB_LIBS)

run:
./toml2json

example:
./toml2json < "./Files_toml/example.toml" >example.json

array:
./toml2json < "./Files_toml/array.toml" >array.json

keys:
./toml2json < "./Files_toml/keys.toml" >keys.json

tables:
./toml2json < "./Files_toml/tables.toml" >tables.json

values:
./toml2json < "./Files_toml/values.toml" >values.json

arraytables:
./toml2json < "./Files_toml/arraytables.toml" >arraytables.json

ficheiro:
./toml2json < "./Files_toml/ficheiro.toml" >ficheiro.json

clean:
rm -f lex.yy.c y.tab.c y.tab.h y.output toml2json *.dot *.json
```