Universidade do Minho
Escola de Engenharia

Processamento de Linguagens

Trabalho Prático nº 2

Conversor toml2json

Grupo 15

28 de Junho de 2020

MiEI - 3º Ano - 2º Semestre

Beatriz Rocha A84003

Filipe Guimarães A85308

Gonçalo Ferreira A84073

Conteúdo

Capítulo 1

Introdução

Este trabalho prático tem como principais objetivos:

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

Para este trabalho prático foi-nos dito que deveríamos resolver o "Conversor toml2json", visto que $(15 \% 6) + 1 = 4$. Este passa por escrever uma gramática que cubra a linguagem TOML e construir um processador (flex, yacc) que reconheça e valide estruturas/dicionários escritos na DSL TOML definida, gerando o JSON correspondente.

Capítulo 2

Análise e especificação

2.1 Descrição informal do problema

O projeto visa solucionar a transcrição de ficheiros com formato *TOML* para o seu equivalente em *JSON*. As notações podem ser encontradas nos sites `https://toml.io/en/` e `https://www.json.org/json-en.html`, assim possibilita verificar os diversos casos que serão processados pela solução proposta com uma gramática independente de contexto.

Requisitos a cumprir:

- Analisador Sintático
- Gramática Independente de Contexto
- Estrutura de Dados
- Gramática Tradutora

Capítulo 3

Proposta de resolução

3.1 Analisador sintático

A análise léxica desempenha a função de traduzir os diversos padrões possíveis para um conjunto de símbolos léxicos que constituem a linguagem a reconhecer. Na linguagem os símbolos encontrados foram:

OPEN_LIST, CLOSE_LIST OPEN_IN_LINE_TABLE CLOSE_IN_LINE_TABLE OPEN_TABLE CLOSE_TABLE OPEN_ARRAY_OF_TABLES, *CLOSE_ARRAY_OF_TABLES*, *KEY_EQ_VALUE*, *KEY_TOKEN*, *SEPARATE_VALUES*, *APOSTROPHE_TRI_OPEN*, *APOSTROPHE_TRI_CLOSE*, *QUOTE_TRI_OPEN*, *QUOTE_TRI_CLOSE*, *APOSTROPHE_OPEN*, *APOSTROPHE_CLOSE*, *QUOTE_OPEN*, *QUOTE_CLOSE* e *END*.

Esta notação pode ser um pouco excessiva, pois exige um maior controlo de variáveis de contexto. No entanto, oferece uma melhor modularidade.

Mais ainda, são estabelecidos os símbolos terminais que têm um valor atribuído, que são: undifined_numeric, apostrophe_char, hex_numeric, oct_numeric, bin_numeric, string_key, quote_char, boolean, integer, yyfloat e date, com o mesmo propósito anteriormente referido deve-se a modularidade. Pode ser observado nos anexos o código correspondente a este segmento do trabalho.

A questão da modularidade é levantada, pois trata-se de duas notações de objetos que podem ser atualizadas a qualquer momento, assim pretende-se conceber um produto que possa ter reutilização futura.

3.2 Gramática independente de contexto

A gramática independente de contexto começa pelo axioma da gramática representado pela regra "S"que tem terminação dado o símbolo END. No axioma é representado uma sequência com o símbolo não terminal *Sequence*, que tem como elementos o símbolo terminal *Sequenciable*, posteriormente este símbolo não terminal pode derivar em 3 símbolos não terminais nomeados *Pair*, *Table* e *ArrayOfTables*.

```
S
    : Sequence END
;

Sequence
    : Sequence Sequenciable
    | Sequenciable
;

Sequenciable
    : Pair
    | Table
    | ArrayOfTables
;
```

Seguidamente temos as regras respetivas aos símbolos não terminais que podem aparecer nas sequências anteriores. O símbolo *Table* é determinado pelo seu símbolo terminal de abertura seguido do símbolo não terminal *Key* acabando pelo seu símbolo terminal de fecho, analogamente *ArrayOfTables*. Já o símbolo *Pair* é definido pelo símbolo não terminal *Key* seguido do seu símbolo de igualdade para o símbolo não terminal *Value*, esta regra garante a atribuição de um valor a uma dada chave.

```
Table
    : OPEN_TABLE Key CLOSE_TABLE
;

ArrayOfTables
    : OPEN_ARRAY_OF_TABLES Key CLOSE_ARRAY_OF_TABLES
;

Pair
    : Key KEY_EQ_VALUE Value
;
```

Para satisfazer a complexidade proposta pelas chaves da notação, são propostas as regras abaixo, sendo que os símbolos *Key* e *DottedKey* se referem ao padrão das chaves complexas dado um determinado símbolo terminal *KEY_TOKEN*. O símbolo não terminal *KeyString* possui as regras com os diversos símbolos que constituem o nome de uma chave.

```
Key
    : DottedKey KeyString
;

DottedKey
    : DottedKey KeyString KEY_TOKEN
    | &
;

KeyString
    : string_key
    | APOSTROPHE_OPEN ApostropheString APOSTROPHE_CLOSE
    | QUOTE_OPEN QuoteString QUOTE_CLOSE
;
```

A representação dos valores atribuídos às chaves é imposta pelas regras do símbolo não terminal *Value*, que corresponde a 2 símbolos terminais *date* e *boolean* e 4 símbolos não terminais *String*, *Numeric*, *List* e *InLineTable*.

```
Value
    : String
    | Numeric
    | boolean
    | date
    | List
    | InLineTable
;
```

Os símbolos não terminais *InLineTable* e *List* são compostos pelos símbolos terminais de abertura e fecho, respetivamente. Também os seus símbolos não terminais que derivam de listagem dos seus elementos, *Pair* e *Value* para os diferentes tipos de listagem.

```
InLineTable
    : OPEN_IN_LINE_TABLE InLinable CLOSE_IN_LINE_TABLE
;

InLinable
    : Pair
    | InLinable SEPARATE_VALUES Pair
;

List
    : OPEN_LIST Listable CLOSE_LIST
;

Listable
    : Value
    | Listable SEPARATE_VALUES Value
    | Listable SEPARATE_VALUES
;
```

As regras abaixo representam o padrão para o reconhecimento de strings. Para tal existem quatro diferentes padrões de abertura e fecho de strings, como indicado na página da notação de objetos *TOML*. Abaixo estão os dois padrões que vão concatenar os caracteres existentes para cada tipo de string .

```
String
    : APOSTROPHE_TRI_OPEN ApostropheString APOSTROPHE_TRI_CLOSE
    | QUOTE_TRI_OPEN QuoteString QUOTE_TRI_CLOSE
    | APOSTROPHE_OPEN ApostropheString APOSTROPHE_CLOSE
    | QUOTE_OPEN QuoteString QUOTE_CLOSE
;

ApostropheString
    : apostrophe_char
    | ApostropheString apostrophe_char
;

QuoteString
    : quote_char
    | QuoteString quote_char
;
```

Por fim, as últimas regras da GIC representam os símbolos terminais do símbolo não terminal *Numeric*.

```
Numeric
    : yyfloat
    | integer
    | hex_numeric
    | oct_numeric
    | bin_numeric
    | undifined_numeric
;
```

3.3 Estrutura de dados

De modo a apoiar a gramática independente de contexto e análise sintática, utilizaram-se estruturas de dados. Esta estrutura de dados face aos requisitos do projeto deve possuir um tipo (por exemplo, objeto, *string*, lista, etc), uma chave única e um valor que pode ser qualquer tipo de informação. Como tal, é utilizada a *struct* abaixo com os campos *char type* que representa o tipo de dados, *char * key* respetivo à chave e *gpointer data* para o valor associado à chave.

```
struct storedata_st {
    char type;
    char * key;
    gpointer data;
};

typedef struct storedata_st * STOREDATA;
```

Como é possível observar usa-se ainda o apoio da biblioteca *glib* para manipulação de estruturas dinâmicas face a atingir o objetivo desejado. Juntamente com esta estrutura é desenvolvida uma API que visa a sua fácil utilização.

3.4 Gramática tradutora

Com recurso à ferramenta **YACC** é criada a gramática tradutora baseada na GIC com a adição comportamental da API fornecida pela estrutura de dados. O foco principal é a manipulação dos símbolos terminais e não terminais de modo a poder corresponder ao comportamento pretendido para as rotinas de padronização. Deste modo é necessário identificar os tipos dos símbolos, que se propõe:

```
%token OPEN_LIST CLOSE_LIST OPEN_IN_LINE_TABLE CLOSE_IN_LINE_TABLE
OPEN_TABLE CLOSE_TABLE OPEN_ARRAY_OF_TABLES CLOSE_ARRAY_OF_TABLES
KEY_EQ_VALUE KEY_TOKEN SEPARATE_VALUES APOSTROPHE_TRI_OPEN
APOSTROPHE_TRI_CLOSE QUOTE_TRI_OPEN QUOTE_TRI_CLOSE
APOSTROPHE_OPEN APOSTROPHE_CLOSE QUOTE_OPEN QUOTE_CLOSE END

%token <string_value> undifined_numeric apostrophe_char
hex_numeric oct_numeric bin_numeric string_key quote_char
boolean integer yyfloat date

%type <pointer> InLineTable InLinable Listable Value List Pair

%type <store_data> DottedKey Key

%type <string_value> ApostropheString QuoteString KeyString
Numeric String
```

Com esta proposta de tipos é possível manipular os dados de modo genérico assim garantindo a modularidade proposta, bem como a utilização das estruturas de dados de forma simples.

O comportamento associado às regras pode ser consultado no anexo 5.3. Contudo, é importante explicar o processo de reconhecimento das chaves e atribuição dos valores. A atribuição de um *Value* a uma *Key* é suportada pela verificação do tipo da chave e posteriormente a adição do valor à estrutura pretendido $1, finaliza retornando para $$ a estrutura em $1.

```
Pair
    : Key KEY_EQ_VALUE Value {
        if (store_data_get_type($1) == 'v') {
            store_data_set_key($3,store_data_get_key($1));
            store_data_add_value($1,$3);
        }

        $$ = $1;
    }
;

Key
    : DottedKey KeyString {
        $$ = store_data_next_key_value($1,$2);
        if (!$$) return erroSem("Key NULL");
    }
;

DottedKey
    : DottedKey KeyString KEY_TOKEN {
        $$ = store_data_next_key($1,$2);
    }
    | {
        if (parsing_InLineTable > 0 && !parsing_Table)
            $$ = in_line_table;
        else
            $$ = table_in_use;
    }
;
```

Ao observar as regras correspondentes a *DottedKey* nota-se que o vazio é o caso de paragem, que consoante o estado do *parser* retorna a tabela atualmente em uso. Complementando o caso de paragem a primeira regra é responsável por atualizar qual a próxima tabela com a função *store_data_next_key($1,$2)*, assim retornando o avanço nas estruturas. Por fim, o símbolo não terminal *Key* é constituído só por uma regra que valida o último campo da chave em que se este for *NULL* retorna um erro semântico de sobreposição de valores.

3.5 Testes

3.5.1 Requisitos

Para efetuar testes foram utilizados os exemplos disponíveis nos repositórios/sites públicos pertencentes à notação de objetos. Em particular, `https://toml.io/en/` e também diversos exemplos encontrados no *GitHub* oficial do *TOML* (`https://github.com/toml-lang`), estes exemplos podem ser consultados nos anexos do relatório.

Para efetuar os exemplos foram utilizados os comandos:

```
(./toml2json.exe < ../toml_examples/example-v0.3.toml)
> examplev3.json

(./toml2json.exe < ../toml_examples/hard_example.toml)
> hard.json

(./toml2json.exe < ../toml_examples/fruit.toml) > fruit.json
```

Estes ficheiros de *input* e *output* podem ser consultados no anexo 5.4 para verificação da sua integridade. Foram feitos mais testes que estão adicionados ao *zip* enviado com o código fonte.

3.5.2 Extra

Como elemento extra do trabalho ainda adicionamos a tradução para *XML*, este é apenas para demonstrar a versatilidade da proposta de resolução. Isto deve-se à utilização de estruturas para suportar os dados lidos do ficheiro *TOML*. Também poderíamos implementar outras formas de escrever esta informação.

```
(./toml2json.exe -xml < ../toml_examples/hard_example.toml)
> hard.xml
```

Capítulo 4

Conclusão

Com a resolução desde trabalho prático, pudemos expandir os conhecimentos adquiridos nas aulas teóricas e práticas deste unidade curricular relativamente à escrita de gramáticas independentes de contexto que satisfaçam a condição LR().

Acreditamos que os requisitos propostos foram implementados com sucesso. Para além daquilo que foi proposto ainda aproveitamos para melhorar o trabalho, na medida em que acabámos por implementar todos os subconjuntos da linguagem TOML e não apenas um como era pedido.

Capítulo 5

Anexos

5.1 FLEX

```
%option 8bit noyywrap yylineno stack

%{
#include "storedata.h"
#include "y.tab.h"
%}

dig [0-9]
signal [\+\-]



comment_regex #.*



boolean_regex (true|false)



string_key [A-Za-z0-9_\-]+



integer {dig}((_)?{dig})*
hex 0x[0-9A-Fa-f]+
oct 0o[0-7]+
bin 0b[01]+

integer_regex {signal}?{integer}
```

```
fractional \.{integer}
exponent [Ee]{signal}?{integer}
inf {signal}?inf
nan {signal}?nan

float_regex ({signal}?{integer}{fractional}?{exponent}?)



offset {dig}{2}:{dig}{2}
local_date {dig}{4}\-{dig}{2}\-{dig}{2}
local_time {dig}{2}:{dig}{2}:{dig}{2}(\.{dig}{1,6})?
local_date_time {local_date}T{local_time}
offset_date_time {local_date_time}(Z|\-{offset})

date_regex ({local_date}|{local_time}|{local_date_time}|{offset_date_time})


%x VALUE IN_LINE_TABLE LIST QUOTE_STR_L APOSTROPHE_STR_L QUOTE_STR APOSTROPHE_STR

%%

<QUOTE_STR>\n {
    yylval.string_value = strdup("\\n");
    return quote_char;
}
<QUOTE_STR>\\ {
    yylval.string_value = strdup("\\");
    return quote_char;
}
<QUOTE_STR>\" {
    yylval.string_value = strdup("\\\"");
    return quote_char;
}
<QUOTE_STR>\\[\n\ ]+ {
    yylval.string_value = "";
    return quote_char;
}

<QUOTE_STR>\"\"\" {
    yy_pop_state();
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return QUOTE_TRI_CLOSE;
}
<VALUE>\"\"\"\n? {
    yy_push_state(QUOTE_STR);
    return QUOTE_TRI_OPEN;
}
```

```
<QUOTE_STR_L,QUOTE_STR>\\\" {
    yylval.string_value = strdup(yytext);
    return quote_char;
}
<QUOTE_STR_L>\" {
    yy_pop_state();
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return QUOTE_CLOSE;
}
<INITIAL,VALUE,LIST,IN_LINE_TABLE>\" {
    yy_push_state(QUOTE_STR_L);
    return QUOTE_OPEN;
}

<QUOTE_STR,QUOTE_STR_L>[^\n] {
    yylval.string_value = strdup(yytext);
    return quote_char;
}




<APOSTROPHE_STR,APOSTROPHE_STR_L>\\ {
    yylval.string_value = strdup("\\\\");
    return apostrophe_char;
}
<APOSTROPHE_STR,APOSTROPHE_STR_L>\" {
    yylval.string_value = strdup("\\\"");
    return apostrophe_char;
}

<APOSTROPHE_STR>\n {
    yylval.string_value = strdup("\\n");
    return apostrophe_char;
}
<APOSTROPHE_STR>\\\' {
    yylval.string_value = strdup(yytext);
    return apostrophe_char;
}
<APOSTROPHE_STR>[^\'\n] {
    yylval.string_value = strdup(yytext);
    return apostrophe_char;
}
<APOSTROPHE_STR>\'\'\' {
    yy_pop_state();
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return APOSTROPHE_TRI_CLOSE;
}
<VALUE>\'\'\'\n? {
    yy_push_state(APOSTROPHE_STR);
```

```
    return APOSTROPHE_TRI_OPEN;
}

<APOSTROPHE_STR_L>\' {
    yy_pop_state();
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return APOSTROPHE_CLOSE;
}
<INITIAL,VALUE,LIST,IN_LINE_TABLE>\' {
    yy_push_state(APOSTROPHE_STR_L);
    return APOSTROPHE_OPEN;
}

<APOSTROPHE_STR,APOSTROPHE_STR_L>[^\n] {
    yylval.string_value = strdup(yytext);
    return apostrophe_char;
}




<VALUE>\n { yy_pop_state(); }


<*>[ \t\n\r] ;


<*>\. {
    return KEY_TOKEN;
}


<LIST,IN_LINE_TABLE>\, {
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return SEPARATE_VALUES;
}


<IN_LINE_TABLE>\} {
    yy_pop_state();
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return CLOSE_IN_LINE_TABLE;
}


<VALUE,LIST>\{ {
    yy_push_state(IN_LINE_TABLE);
    return OPEN_IN_LINE_TABLE;
}
```

```
<LIST>\] {
    yy_pop_state();
    if(YYSTATE == VALUE) { yy_pop_state(); }
    return CLOSE_LIST;
}


<VALUE,LIST>\[ {
    yy_push_state(LIST);
    return OPEN_LIST;
}


<*>\= {
    yy_push_state(VALUE);
    return KEY_EQ_VALUE;
}


<VALUE,LIST>{boolean_regex} {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return boolean;
}


<VALUE,LIST>{integer_regex} {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return integer;
}


<VALUE,LIST>{hex} {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return hex_numeric;
}


<VALUE,LIST>{oct} {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return oct_numeric;
}


<VALUE,LIST>{bin} {
    if(YYSTATE == VALUE) yy_pop_state();
```

```
    yylval.string_value = strdup(yytext);
    return bin_numeric;
}


<VALUE,LIST>{float_regex} {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return yyfloat;
}


<VALUE,LIST>({inf}|{nan}) {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return undifined_numeric;
}


<VALUE,LIST>{date_regex} {
    if(YYSTATE == VALUE) yy_pop_state();
    yylval.string_value = strdup(yytext);
    return date;
}



<INITIAL>\] {
    return CLOSE_TABLE;
}


<INITIAL>\[ {
    return OPEN_TABLE;
}


<INITIAL>\]\] {
    return CLOSE_ARRAY_OF_TABLES;
}


<INITIAL>\[\[ {
    return OPEN_ARRAY_OF_TABLES;
}


<INITIAL,IN_LINE_TABLE>{string_key} {
    yylval.string_value = strdup(yytext);
    return string_key;
```

```
}



<INITIAL,LIST,VALUE>{comment_regex} ;


<*><<EOF>> {
   return END;
}


<*>. {
   puts("ERROR TOKEN");
   return 0;
}

%%
```

5.2 GIC

```
S
    : Sequence END
;

Sequence
    : Sequence Sequenciable
    | Sequenciable
;

Sequenciable
    : Pair
    | Table
    | ArrayOfTables
;

Table
    : OPEN_TABLE Key CLOSE_TABLE
;

ArrayOfTables
    : OPEN_ARRAY_OF_TABLES Key CLOSE_ARRAY_OF_TABLES
;

InLineTable
    : OPEN_IN_LINE_TABLE InLinable CLOSE_IN_LINE_TABLE
;
```

```
InLinable
    : Pair
    | InLinable SEPARATE_VALUES Pair
;

List
    : OPEN_LIST Listable CLOSE_LIST
;

Listable
    : Value
    | Listable SEPARATE_VALUES Value
    | Listable SEPARATE_VALUES
;

Pair
    : Key KEY_EQ_VALUE Value
;

Key
    : DottedKey KeyString
;

DottedKey
    : DottedKey KeyString KEY_TOKEN
    | &
;

KeyString
    : string_key
    | APOSTROPHE_OPEN ApostropheString APOSTROPHE_CLOSE
    | QUOTE_OPEN QuoteString QUOTE_CLOSE
;

Value
    : String
    | Numeric
    | boolean
    | date
    | List
    | InLineTable
;

String
    : APOSTROPHE_TRI_OPEN ApostropheString APOSTROPHE_TRI_CLOSE
    | QUOTE_TRI_OPEN QuoteString QUOTE_TRI_CLOSE
    | APOSTROPHE_OPEN ApostropheString APOSTROPHE_CLOSE
    | QUOTE_OPEN QuoteString QUOTE_CLOSE
;
```

```
ApostropheString
    : apostrophe_char
    | ApostropheString apostrophe_char
;

QuoteString
    : quote_char
    | QuoteString quote_char
;

Numeric
    : yyfloat
    | integer
    | hex_numeric
    | oct_numeric
    | bin_numeric
    | undifined_numeric
;
```

5.3 YACC

```
%{
#include <stdio.h>
#include <string.h>

#include "storedata.h"

STOREDATA global_table    = NULL;
STOREDATA table_in_use    = NULL;
STOREDATA in_line_table   = NULL;

GPtrArray * inline_stack  = NULL;

int parsing_InLineTable   = 0;
int parsing_Table         = 0;

extern void asprintf();
extern int yylex();
extern int yylineno;
extern char *yytext;


char * take_of_under_score (char * s);
int yyerror();
int erroSem(char*);
%}

%union{
    char * string_value;
```

```
    gpointer pointer;
    STOREDATA store_data;
}


%token OPEN_LIST    // '['
%token CLOSE_LIST   // ']'


%token OPEN_IN_LINE_TABLE   // '{'
%token CLOSE_IN_LINE_TABLE  // '}'


%token OPEN_TABLE   // '['
%token CLOSE_TABLE  // ']'


%token OPEN_ARRAY_OF_TABLES     // '[['
%token CLOSE_ARRAY_OF_TABLES    // ']]'


%token KEY_EQ_VALUE     // '='
%token KEY_TOKEN        // '.'
%token SEPARATE_VALUES  // ','


%token APOSTROPHE_TRI_OPEN  // '''
%token APOSTROPHE_TRI_CLOSE // '''


%token QUOTE_TRI_OPEN   // """
%token QUOTE_TRI_CLOSE  // """


%token APOSTROPHE_OPEN  // '
%token APOSTROPHE_CLOSE // '


%token QUOTE_OPEN   // "
%token QUOTE_CLOSE  // "


%token END // <<EOF>>


%token <string_value>
    undifined_numeric
    apostrophe_char
    hex_numeric
    oct_numeric
```

```
    bin_numeric
    string_key
    quote_char
    boolean
    integer
    yyfloat
    date


%type <pointer>
    InLineTable
    InLinable
    Listable
    Value
    List
    Pair

%type <store_data>
    DottedKey
    Key

%type <string_value>
    ApostropheString
    QuoteString
    KeyString
    Numeric
    String

%%

S :
    {
        global_table = store_data_new_table("global");
        table_in_use = global_table;
        inline_stack = g_ptr_array_new();
    }
      Sequence END
    {
        print_2_JSON(global_table);
        return 0;
    }
;


Sequence
    : Sequence Sequenciable
    | Sequenciable
;
```

```
Sequenciable
    : Pair
    | Table
    | ArrayOfTables
;


Table
    : {
        table_in_use = global_table;
        parsing_Table = 1;
    }
    OPEN_TABLE Key CLOSE_TABLE
    {
        table_in_use = $3;
        if (store_data_get_type($3) == 'v') {
            store_data_set_data($3,g_hash_table_new(g_str_hash,g_str_equal));
            store_data_set_type($3,'h');
        }

        parsing_Table = 0;
    }
;


ArrayOfTables
    : {
        table_in_use = global_table;
        parsing_Table = 1;
    }
    OPEN_ARRAY_OF_TABLES Key CLOSE_ARRAY_OF_TABLES
    {
        if (store_data_get_type($3) != 'a') {
            store_data_set_data($3,g_ptr_array_new());
            store_data_set_type($3,'a');
        }

        STOREDATA s = store_data_new_table("");
        store_data_add_value($3,s);

        table_in_use = s;
        parsing_Table = 0;
    }
;


InLineTable
    : OPEN_IN_LINE_TABLE InLinable CLOSE_IN_LINE_TABLE {
        $$ = $2;
        parsing_InLineTable--;
```

```
        if (parsing_InLineTable > 0) in_line_table = g_ptr_array_index(inline_stack,parsin
    }
;


InLinable
    : {
        in_line_table = store_data_new_table("");
        g_ptr_array_insert(inline_stack, parsing_InLineTable, in_line_table);
        parsing_InLineTable++;
    }
    Pair
    {
        $$ = g_ptr_array_index(inline_stack,parsing_InLineTable-1);
    }
    | InLinable SEPARATE_VALUES Pair { $$ = $1; }
;


List
    : OPEN_LIST Listable CLOSE_LIST { $$ = $2; }
;


Listable
    : Value {
        STOREDATA s = store_data_new_array("");
        store_data_add_value(s,$1);
        $$ = s;
    }
    | Listable SEPARATE_VALUES Value { store_data_add_value($1,$3); $$ = $1; }
    | Listable SEPARATE_VALUES       { $$ = $1; }
;


Pair
    : Key KEY_EQ_VALUE Value {
        if (store_data_get_type($1) == 'v') {
            store_data_set_key($3,store_data_get_key($1));
            store_data_add_value($1,$3);
        }

        $$ = $1;
    }
;


Key
    : DottedKey KeyString { $$ = store_data_next_key_value($1,$2); if (!$$) return erroSem
;
```

```
DottedKey
    : DottedKey KeyString KEY_TOKEN { $$ = store_data_next_key($1,$2); }
    | {
        if (parsing_InLineTable > 0 && !parsing_Table) $$ = in_line_table;
        else $$ = table_in_use;
    }
;


KeyString
    : string_key                                        { asprintf(&$$,"%s",$1); }
    | APOSTROPHE_OPEN ApostropheString APOSTROPHE_CLOSE { asprintf(&$$,"%s",$2); }
    | QUOTE_OPEN QuoteString QUOTE_CLOSE                { asprintf(&$$,"%s",$2); }
;


Value
    : String        { $$ = store_data_new('s', "", $1); }
    | Numeric       { $$ = store_data_new('s', "", $1); }
    | boolean       { $$ = store_data_new('s', "", $1); }
    | date          { char * s; asprintf(&s,"\"%s\"",$1); $$ = store_data_new('s', "", s);
    | List          { $$ = $1; }
    | InLineTable   { $$ = $1; }
;


String
    : APOSTROPHE_TRI_OPEN ApostropheString APOSTROPHE_TRI_CLOSE { asprintf(&$$,"\"%s\"",$2
    | QUOTE_TRI_OPEN QuoteString QUOTE_TRI_CLOSE                { asprintf(&$$,"\"%s\"",$2
    | APOSTROPHE_OPEN ApostropheString APOSTROPHE_CLOSE         { asprintf(&$$,"\"%s\"",$2
    | QUOTE_OPEN QuoteString QUOTE_CLOSE                        { asprintf(&$$,"\"%s\"",$2
;


ApostropheString
    : apostrophe_char                   { asprintf(&$$,"%s",$1); }
    | ApostropheString apostrophe_char { asprintf(&$$,"%s%s",$1,$2); }
;


QuoteString
    : quote_char             { asprintf(&$$,"%s",$1); }
    | QuoteString quote_char { asprintf(&$$,"%s%s",$1,$2); }
;


Numeric
    : yyfloat           { $$ = take_of_under_score( *$1 == '+' ? $1 + 1 : $1 ); }
```

```
    | integer           { $$ = take_of_under_score( *$1 == '+' ? $1 + 1 : $1 ); }
    | hex_numeric       { asprintf(&$$,"%ld",strtol($1+2,NULL,16)); }
    | oct_numeric       { asprintf(&$$,"%ld",strtol($1+2,NULL,8)); }
    | bin_numeric       { asprintf(&$$,"%ld",strtol($1+2,NULL,2)); }
    | undifined_numeric { asprintf(&$$,"\"%s\"",$1); }
;

%%

int main(){
    yyparse();
    return 0;
}

int erroSem(char *s){
    printf("Erro Semântico na linha: %d, %s...\n", yylineno, s);
    return 0;
}

int yyerror(){
    printf("Erro Sintático ou Léxico na linha: %d, com o texto: %s\n", yylineno, yytext);
    return 0;
}

char * take_of_under_score (char * s) {
    char * r = malloc(strlen(s));
    int i = 0, j = 0;

    while (s[i]) {
        if (s[i] != '_') r[j++] = s[i];
        i++;
    }
    r[j] = '\0';

    return r;
}
```

5.4 Testes

5.4.1 example-v0.3.toml

```
# Comment
# I am a comment. Hear me roar. Roar.

# Table
# Tables (also known as hash tables or dictionaries) are collections of key/value pairs.
# They appear in square brackets on a line by themselves.

[Table]
```

```
key = "value" # Yeah, you can do this.

# Nested tables are denoted by table names with dots in them. Name your tables whatever cr

[dog.tater]
type = "pug"

# You don't need to specify all the super-tables if you don't want to. TOML knows how to d

# [x] you
# [x.y] don't
# [x.y.z] need these
[x.y.z.w] # for this to work

# String
# There are four ways to express strings: basic, multi-line basic, literal, and multi-line
# All strings must contain only valid UTF-8 characters.

[String]
basic = "I'm a string. \"You can quote me\". Name\tJos\u00E9\nLocation\tSF."

[String.Multiline]

# The following strings are byte-for-byte equivalent:
key1 = "One\nTwo"
key2 = """One\nTwo"""
key3 = """
One
Two"""

[String.Multilined.Singleline]

# The following strings are byte-for-byte equivalent:
key1 = "The quick brown fox jumps over the lazy dog."

key2 = """
The quick brown \


  fox jumps over \
    the lazy dog."""

key3 = """\
       The quick brown \
       fox jumps over \
       the lazy dog.\
       """

[String.Literal]
```

```
# What you see is what you get.
winpath  = 'C:\Users\nodejs\templates'
winpath2 = '\\ServerX\admin$\system32\'
quoted   = 'Tom "Dubs" Preston-Werner'
regex    = '<\i\c*\s*>'


[String.Literal.Multiline]

regex2 = '''I [dw]on't need \d{2} apples'''
lines  = '''
The first newline is
trimmed in raw strings.
   All other whitespace
   is preserved.
'''

# Integer
# Integers are whole numbers. Positive numbers may be prefixed with a plus sign.
# Negative numbers are prefixed with a minus sign.

[Integer]
key1 = +99
key2 = 42
key3 = 0
key4 = -17

# Float
# A float consists of an integer part (which may be prefixed with a plus or minus sign)
# followed by a fractional part and/or an exponent part.

[Float.fractional]

# fractional
key1 = +1.0
key2 = 3.1415
key3 = -0.01

[Float.exponent]

# exponent
key1 = 5e+22
key2 = 1e6
key3 = -2E-2

[Float.both]

# both
key = 6.626e-34
```

```
# Boolean
# Booleans are just the tokens you're used to. Always lowercase.

[Booleans]
True = true
False = false

# Datetime
# Datetimes are RFC 3339 dates.

[Datetime]
key1 = 1979-05-27T07:32:00Z
key2 = 1979-05-27T00:32:00-07:00
key3 = 1979-05-27T00:32:00.999999-07:00

# Array
# Arrays are square brackets with other primitives inside. Whitespace is ignored. Elements

[Array]
key1 = [ 1, 2, 3 ]
key2 = [ "red", "yellow", "green" ]
key3 = [ [ 1, 2 ], [3, 4, 5] ]
key4 = [ [ 1, 2 ], ["a", "b", "c"] ] # this is ok

#Arrays can also be multiline. So in addition to ignoring whitespace, arrays also ignore n
# Terminating commas are ok before the closing bracket.

key5 = [
  1, 2, 3
]
key6 = [
  1,
  2, # this is ok
]

# Array of Tables
# These can be expressed by using a table name in double brackets.
# Each table with the same double bracketed name will be an element in the array.
# The tables are inserted in the order encountered.

[[products]]
name = "Hammer"
sku = 738594937

[[products]]

[[products]]
name = "Nail"
sku = 284758393
```

```
color = "gray"


# You can create nested arrays of tables as well.

[[fruit]]
  name = "apple"

  [fruit.physical]
    color = "red"
    shape = "round"

  [[fruit.variety]]
    name = "red delicious"

  [[fruit.variety]]
    name = "granny smith"

[[fruit]]
  name = "banana"

  [[fruit.variety]]
    name = "plantain"
```

5.4.2 examplev3.json

```
{
    "Table": {
        "key": "value"
    },
    "Datetime": {
        "key1": "1979-05-27T07:32:00Z",
        "key3": "1979-05-27T00:32:00.999999-07:00",
        "key2": "1979-05-27T00:32:00-07:00"
    },
    "Integer": {
        "key1": 99,
        "key3": 0,
        "key2": 42,
        "key4": -17
    },
    "Booleans": {
        "True": true,
        "False": false
    },
    "Float": {
        "fractional": {
            "key1": 1.0,
            "key3": -0.01,
            "key2": 3.1415
```

```
    },
    "both": {
        "key": 6.626e-34
    },
    "exponent": {
        "key1": 5e+22,
        "key3": -2E-2,
        "key2": 1e6
    }
},
"dog": {
    "tater": {
        "type": "pug"
    }
},
"x": {
    "y": {
        "z": {
            "w": {}
        }
    }
},
"products": [
    {
        "name": "Hammer",
        "sku": 738594937
    },
    {},
    {
        "color": "gray",
        "name": "Nail",
        "sku": 284758393
    }
],
"String": {
    "basic": "I'm a string. \"You can quote me\". Name\tJos\u00E9\nLocation\tSF.",
    "Multilined": {
        "Singleline": {
            "key1": "The quick brown fox jumps over the lazy dog.",
            "key3": "The quick brown fox jumps over the lazy dog.",
            "key2": "The quick brown fox jumps over the lazy dog."
        }
    },
    "Multiline": {
        "key1": "One\nTwo",
        "key3": "One\nTwo",
        "key2": "One\nTwo"
    },
    "Literal": {
        "winpath": "C:\\Users\\nodejs\\templates",
```

```
        "winpath2": "\\\\ServerX\\admin$\\system32\\",
        "quoted": "Tom \"Dubs\" Preston-Werner",
        "regex": "<\\i\\c*\\s*>",
        "Multiline": {
            "regex2": "I [dw]on't need \\d{2} apples",
            "lines": "The first newline is\ntrimmed in raw strings.\n   All other whit
        }
    }
},
"Array": {
    "key1": [
        1,
        2,
        3
    ],
    "key3": [
        [
            1,
            2
        ],
        [
            3,
            4,
            5
        ]
    ],
    "key5": [
        1,
        2,
        3
    ],
    "key2": [
        "red",
        "yellow",
        "green"
    ],
    "key4": [
        [
            1,
            2
        ],
        [
            "a",
            "b",
            "c"
        ]
    ],
    "key6": [
        1,
        2
```

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

5.4.3 fruit.toml

```
[[fruit.blah]]
  name = "apple"

  [fruit.blah.physical]
    color = "red"
    shape = "round"

[[fruit.blah]]
  name = "banana"
  physical = { color = "yellow" , shape = "bent" }
```

5.4.4 fruit.json

```
{
    "fruit": {
        "blah": [
            {
                "physical": {
                    "shape": "round",
```

```
                    "color": "red"
                },
                "name": "apple"
            },
            {
                "physical": {
                    "shape": "bent",
                    "color": "yellow"
                },
                "name": "banana"
            }
        ]
    }
}
```

5.4.5 hard_unicode.toml

```
# Test file for TOML
# Only this one tries to emulate a TOML file written by a user of the kind of parser write
# This part you'll really hate

[the]
test_string = "You'll hate me after this - #"          # " Annoying, isn't it?

    [the.hard]
    test_array = [ "] ", " # "]      # ] There you go, parse this!
    test_array2 = [ "Test #11 ]proved that", "Experiment #9 was a success" ]
    # You didn't think it'd as easy as chucking out the last #, did you?
    another_test_string = " Same thing, but with a string #"
    harder_test_string = " And when \"'s are in the string, along with # \""   # "and comm
    # Things will get harder

        [the.hard."bit#"]
        "what?" = "You don't think some user won't do that?"
        multi_line_array = [
            "]",
            # ] Oh yes I did
            ]

# Each of the following keygroups/key value pairs should produce an error. Uncomment to th

#[error]   if you didn't catch this, your parser is broken
#string = "Anything other than tabs, spaces and newline after a keygroup or key value pair
#array = [
#         "This might most likely happen in multiline arrays",
#         Like here,
#         "or here,
#         and here"
#         ]     End of array comment, forgot the #
#number = 3.14  pi <--again forgot the #
```

5.4.6 hard.json

```
{
    "the": {
        "test_string": "You'll hate me after this - #",
        "hard": {
            "harder_test_string": " And when \"'s are in the string, along with # \"",
            "test_array": [
                "] ",
                " # "
            ],
            "test_array2": [
                "Test #11 ]proved that",
                "Experiment #9 was a success"
            ],
            "bit#": {
                "multi_line_array": [
                    "]"
                ],
                "what?": "You don't think some user won't do that?"
            },
            "another_test_string": " Same thing, but with a string #"
        }
    }
}
```

5.4.7 hard.xml

```
<object>
    <the type="object">
        <test_string>"You'll hate me after this - #"</test_string>
        <hard type="object">
            <harder_test_string>" And when \"'s are in the string, along with # \""</harde
            <test_array type="list">
                <value>"] "</value>
                <value>" # "</value>
            </test_array>
            <test_array2 type="list">
                <value>"Test #11 ]proved that"</value>
                <value>"Experiment #9 was a success"</value>
            </test_array2>
            <bit# type="object">
                <multi_line_array type="list">
                    <value>"]"</value>
                </multi_line_array>
                <what?>"You don't think some user won't do that?"</what?>
            </bit#>
            <another_test_string>" Same thing, but with a string #"</another_test_string>
        </hard>
    </the>
```

```
</object>
```