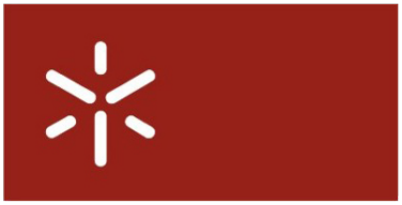

Processamento de Linguagens

Mestrado Integrado em Engenharia Informática

Conversor de TOML para JSON

[YACC/FLex]

A85227 João Pedro Rodrigues Azevedo
A85729 Paulo Jorge da Silva Araújo
A83719 Pedro Filipe Costa Machado

Braga
Junho 2020

Conteúdo

Capítulo 1

Introdução

Este relatório é relativo ao segundo trabalho prático da UC de Processamento de Linguagens que tem como um dos principais objetivos escrever gramáticas independentes de contexto (GIC) que satisfaçam a condição LR(), para criar Linguagens de Domínio Específico (DSL).

Para tal, iremos recorrer a geradores de compiladores como o par **FLex/Yacc** e ainda desenvolver processadores de linguagens regulares capazes de alterarem e transformarem textos segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora (GT).

Este trabalho serve também como a introdução e aplicação de linguagens de programação como o **TOML** e o **JSON** na leitura e representação final dos dados obtidos após reconhecimento de frases válidas da nossa linguagem.

Capítulo 2

Enunciado Escolhido

Como pedido pela equipa docente, para a escolha do enunciado para a realização deste projeto usamos a fórmula *exe = (NGr % 6) + 1* que aplicada ao nosso grupo (nº 69) resultou na escolha do enunciado quatro, um conversor **toml2json** (enunciado 2.4).

Desta forma, foi necessário proceder a uma breve pesquisa com intuito de percebermos o que é a linguagem **TOML** e de que forma se comporta, visto que nenhum dos elementos do grupo tinha trabalhado com este formato de arquivos.

Percebemos então que o **TOML** é um formato de arquivo de configuração criado para ser mais legível para os humanos e que usa uma sintaxe mínima na representação de pares chave-valor.

Assim, de modo a cumprir os objetivos deste trabalho prático, devemos escolher um subconjunto de regras da linguagem **TOML** que o nosso conversor vai seguir. Neste relatório iremos indicar quais as regras que são possíveis de aplicar e como procedemos para a sua conversão em **JSON**.

Capítulo 3

Estratégia Adotada

Com vista a estudar a linguagem **TOML**, o *website* disponibilizado pela equipa docente serviu para retirar informações precisas e claras sobre esta linguagem.

Deste modo, podemos verificar todas as regras que são aplicadas no **TOML** e retirar exemplos para depois os aplicar ao nosso conversor, comparando o resultado com vários conversores *online*.

3.1 Gramática do TOML

Tal como proposto, foi selecionado um subconjunto de regras da linguagem **TOML** a que nos comprometemos a aplicar e depois representar no **JSON** final.

A linguagem **TOML**, no nosso entender, pode ser representada através de um conjunto de tabelas e atribuições. Cada atribuição pode estar "ligada" a uma tabela, comportando-se como um atributo dela. Uma tabela, por sua vez, tem um número ilimitado de atribuições. Devido a este comportamento da gramática do **TOML**, a definição de tabelas, sub-tabelas e atribuições foram os principais focos para o subconjunto considerado.

Desta forma, iremos abordar os vários pontos na definição de uma linguagem **TOML**.

3.1.1 Atribuição

O **TOML** segue a prática de muitas outras linguagens no que toca à atribuição de valores e variáveis. Esta atribuição é caracterizada por uma **Key** e um **Value** da seguinte forma:

Sintaxe: Key = Value

- **Key**

  Uma **Key** pode ser uma ou mais palavras, sendo que para usar várias é necessário colocá-las entre aspas ou plicas.

  Assim, pode-se colocar uma série de caracteres, pois o **TOML** não é restritivo nesse parâmetro. Contudo, os caracteres especiais não são permitidos sem a colocação de aspas/plicas.

  ```
  Exemplos de Keys:
  "maçã", banana, tuti_fruta, "tomate é um fruto", ...
  ```

- **Value**

  Um **Value** pode assumir vários valores e serve para caracterizar a **Key** respetiva.

  Neste conversor foram considerados quase todos os tipos de dados disponíveis no **TOML**, com a exceção das *strings multi-line*, pois apenas se considerou as *strings* clássicas com apenas uma linha, usando caracteres especiais como `\n`, `\t`, ...

  ```
  Número: 3
  String: "morango"
  Boolean: true
  Date-Time: 1979-05-27T07:32:00
  Array: [1,2,3]
  ```

  As strings consideradas no nosso conversor são bastante simples, pelo que não é possível ter plicas ou outras aspas dentro destas.

  Destes *values* há que destacar que os *arrays* do **TOML** precisam de ter para todos elementos o mesmo tipo de variável, ou seja, algo como o seguinte **não é possível**:

  ```
  array = [1, 2, true, "vermelho"]
  ```

  Sendo que um exemplo de *array* válido, seria:

  ```
  cores = ["verde", "amarelo", "vermelho",]
  ```

  A linguagem **TOML** aceita a vírgula no final, embora possa não estar presente. De realçar que também é possível obter *arrays* de *arrays*, sendo que cada elemento precisa de ser sempre um *array*.

  ```
  animal = [ ["raposa", "cão"], ["camarão", "peixe-espada"], [1, 2] ]
  ```

3.1.2 Tabelas

As tabelas no **TOML** servem para agrupar dados de maneira simples. Assim, uma tabela pode ter várias atribuições referenciadas a ela própria através das atribuições "filhas". Contudo uma tabela nunca tem uma atribuição associada a ela mesma!

Na gramática de **TOML** uma atribuição está sempre referenciada à última tabela inicializada, ou seja, à tabela acima, caso exista.

```
[morango] = "vermelho"   # Não é permitido!

[frutos]
morango = "vermelho"
```

Neste exemplo, a **key** *morango* está referenciada na tabela *frutos* com o **value** *"vermelho"*.

As tabelas em **TOML** podem ser definidas através do uso dos parênteses retos ("[" e "]") como se observa no exemplo anterior. Dentro dos parênteses retos aceitam-se palavras com ou sem aspas, pelo que as últimas não podem ser do tipo "" (vazio), ou seja, precisam de conter pelo menos um caractere.

Para além desta definição simples de uma tabela, podemos definir um conjunto de tabelas com uma hierarquia através do ponto:

```
[frutos.morango]
[frutos."maçã"]
[frutos.laranja.casca]
```

Aqui a tabela frutos tem várias tabelas "filho" (morango, "maçã" e laranja), sendo que a última também possui a referência para outra tabela.

Deste modo, é possível definir uma hierarquia de tabelas sendo que cada uma delas pode conter atributos e/ou outras tabelas.

Contudo, na definição das tabelas é preciso ter em consideração que o **TOML** não permite a definição da mesma tabela, ou seja, tabelas com o mesmo nome.

```
[frutos]
morango = "vermelho"
[frutos]            # Não é permitido!
```

A linguagem **TOML** também possui outra forma de inicialização de tabelas hierárquicas com uma atribuição que também foi tido em conta neste conversor:

```
frutos."com caroço"."pêssego" = 2

É equivalente a:

[frutos]
[frutos."com caroço"]
"pêssego" = 2
```

Como podemos observar, temos duas maneiras diferentes de introduzir a hierarquia de tabelas. Neste exemplo, estamos a criar uma tabela *frutos* com uma outra tabela *"com caroço"* que possui uma atribuição.

3.1.3 Comentários

A linguagem **TOML** é muito semelhante a **JSON** e **YAML** visto que todas elas servem para descrever ficheiros de configuração de forma muito simples. **TOML** e **YAML** são muito semelhantes no que toca à simplicidade do ser humano em ler a linguagem, e, por isso mesmo, é possível adicionar comentários no ficheiro[1].

Os comentários em **TOML** são efetuados através do símbolo #, que marca a linha após este como sendo um comentário.

Assim, o nosso conversor ignora qualquer informação na linha após o #, excepto quando se encontra dentro de uma *string*.

```
[frutos]
# A tabela em cima é para frutos
```

3.2 Analisador Léxico, FLex

O analisador sintático gerado pelo **Yacc** requer um analisador léxico que é fornecido externamente através do **FLex** que habitualmente conhecemos, aplicado no primeiro trabalho prático.

Assim, visto que o **Yacc** não lê a partir duma simples entrada de dados, temos a necessidade de implementar um analisador como o anterior para poder fornecer os *tokens* necessários de modo a perceber a semântica introduzida.

Por isto, tivemos que definir uma série de expressões regulares que identificam esses *tokens*, seguindo-se uma breve descrição de alguns exemplos práticos aplicados no analisador léxico:

1. **Reconhecimento de números inteiros ou com vírgula flutuante:**

   ```
   Regex (números): [-+]?[0-9]*(\.)?[0-9]+([eE][-+]?[0-9]+)?
   ```

   A expressão regular anterior permite o reconhecimento de números positivos ou negativos com vários algarismos e, inclusivé, introduz possibilidade de casas decimais e notação científica, por exemplo: 1.02, -93, ou até $94.93e-9$.

   Posto isto, o *token* retornado e previamente definido no **Yacc** é o **NUM**, generalizando, assim, a introdução de *integers* ou *floats* por utilizadores.

   O valor definido para esse número será guardado numa *union* intercalada entre os programas onde o valor de *yylval.number* é definido através da conversão da sequência de *chars* reconhecida, **yytext**, para *float*.

[1] O formato **JSON** não permite comentários, ao contrário de **TOML** e **YAML**

2. **Reconhecimento de valores booleanos:**

```
Regex (bools): (true|false)
```

Os valores booleanos podem ser facilmente confundíveis com o reconhecimento de palavras pelo que são, por isto mesmo, palavras reservadas.

Deste modo, o *token* enviado ao **Yacc** corresponde à *macro* definida para **BOOL** sendo, no entanto, guardado o seu valor como uma simples *string* no campo *yylval.string* da estrutura de dados partilhada.

3. **Reconhecimento de datas:**

```
Regex (datas): [0-9]+-[0-9]+-[0-9]+([T ][0-9]+\:          \\
               [0-9]+\:[0-9]+(Z|([+-][0-9]+\:[0-9]+))?)?
```

As datas são necessariamente uma sequência de caracteres mais comprida de reconhecer, pois seguem os padrões estabelecidos pelo **ISO 8601** para representações universais de datas na *Internet*.

Esta expressão regular aceita assim exemplos como o seguinte:

```
Exemplo: 1994-11-05T08:15:30-05:00
```

Que corresponde a 5 de Novembro de 1994, 8h15m30s da manhã nos padrões "US Eastern Standard Time". Posto isto, o *token* retornado é definido como **DATE** e os *chars* reconhecidos são armazenados no *yylval.string* .

4. **Reconhecimento de chaves (*keys*) e *strings*:**

O reconhecimento das variáveis que serviram de indexação e orientação na construção da estrutura de dados que viria a ser convertida para **JSON**, foi mais um desafio na realização deste trabalho prático.

Na prática, a introdução de chaves ou variáveis é relativamente simples considerando-as como uma sequência de caracteres como na seguinte expressão regular:

```
Regex (chaves normais): [a-zA-Z_0-9-]+
```

Onde o seu intuito é reconhecer o *left-hand-side (lhs)* de uma atribuição (**a** = ...), a inicialização de tabelas ([a]) ou a construção de termos a partir da concatenação de várias chaves ([**a.b**] ou **a.b**).

No entanto, a versatilidade do **TOML** faz com que sejam também permitidas como chave *strings* (não vazias), pelo que os exemplos seguintes são também válidos:

```
"a" = 2  ou "b".c = 3 ou ["d"] ...
//Nota: "" = ... ou [""] são situações inválidas!
```

Esta propriedade interessante levou-nos a introduzir uma nova expressão regular que teria necessariamente de fazer uma série de verificações relativamente ao reconhecimento de *strings*, não esquecendo as possíveis situações de erro, tendo definido para isso um *token* **ERRO** não reconhecido pela gramática:

```
Regex (strings chaves ou valor): ["'][^'"=]*["']
Ação: {

        if (*yytext != *(yytext+strlen(yytext)-1))
            return (ERRO);

        if (IS_RHS)
        {
            yylval.string = strdup(yytext);
            IS_RHS = 0;
            return (STRING);
        }
        else
        {
            if (strlen(yytext)==2) return (ERRO);
            yylval.string = strdup(yytext);
            IS_RHS = 1;
            return (KEY_STRING);
        }
    }
```

Esta expressão regular tanto reconhece *strings* como valores ou como chaves, sendo que, no caso de serem chaves estas têm necessariamente de ter tamanho superior a 2, ou seja, as duas aspas que lá estão sempre mais a *string* em si.

Em ambos os casos, a primeira verificação, isto é, o primeiro *if* garante que o primeiro caractere reconhecido é igual ao último, ou seja, começa com aspa e termina com aspa e caso isso não se verifique envia-se ao **Yacc** o *token* **ERRO**.

De seguida, definiu-se uma variável global ao programa, **IS_RHS** que serviria para identificar se estaríamos na fase de *parsing* do lado esquerdo (valor = 0) ou lado direito (valor = 1) de uma atribuição, porque para o caso de *IS_RHS == 0* então estaríamos a reconhecer uma chave de uma atribuição (ou não) e esta não pode ser vazia (caso *else*), retornando para esse caso um *token* **KEY_STRING**.

Por outro lado, caso *IS_RHS == 1*, estamos no lado direito de uma atribuição e portanto a *string* já pode ser vazia, retornando para esse caso um *token* **STRING**.

Definir se estamos de um lado direito da atribuição é possível de se fazer no caso do reconhecimento do caractere = (e outros), já o caso do lado esquerdo estabelece-se quando estamos no *rhs* e lemos uma valor ou quando encontramos caracteres como o `"."` associados a chaves:

```
Regex (atribuição, parenteses): [\[\]=,] { IS_RHS = 1; ... }
Regex (ponto, comentário): [.#] { IS_RHS = 0; ... }
```

5. **Reconhecimento de outras expressões:**

   Obviamente, para além dos vários símbolos variáveis definidos anteriormente é necessário incluir o reconhecimento de sinais como o caractere do comentário (numa linha) e outros caracteres ignorados como o de mudança de linha, tabulação, etc...estabelecendo, por último, um caso *otherwise* para todos os caracteres desconhecidos reconhecidos pelo **FLex**.

3.3 Analisador Sintático, Yacc

Após a implementação da parte léxica, tivemos de desenvolver o código do analisador sintático (**Yacc**) de forma a receber os símbolos provenientes do **FLex** e, a partir destes, verificar se cumpre a gramática pretendida, sendo neste caso um subconjunto da linguagem **TOML**.

O símbolo não terminal principal do nosso conversor chama-se de Toml e deriva apenas em Language.

De seguida, o Language pode resultar em Atributions que representa uma lista de atribuições ou num Atributions seguido do símbolo Declarations onde se definem todas as declarações de tabelas. A razão pela qual o símbolo terminal Language deriva nestas duas produções deve-se à possibilidade da ocorrência de atribuições não relacionadas a nenhuma tabela e a atribuições sempre relacionadas a tabelas[2].

Desta forma, a primeira produção, visto que o Atributions pode derivar em vazio, indica que podem haver apenas declarações ou atribuições (ou não caso seja vazio) seguidas de declarações. A segunda produção serve para a existência de atribuições sem tabelas posteriormente.

```
Toml : Language
     ;

Language : Atribuitions Declarations
         | Atribuitions
         ;
```

Assim, as declarações e dentro destas, as Definition, são definidas pelas seguintes produções:

```
Declarations : Definition
             | Declarations Definition
             ;

Definition : Table Atribuitions
           ;
```

Como observamos o símbolo não terminal Definition deriva numa tabela (Table) seguida de atribuições referentes a essa tabela.

Com isto, podemos dizer que uma tabela é composta pelos símbolos terminais (“[” e “]”) e pelo símbolo não terminal TermTable:

```
Table : '[' TermTable ']'
      ;
TermTable : Key
          | Key '.' TermTable
          ;
```

Assim sendo, apenas reconhecemos uma tabela através do uso destes símbolos terminais desta forma, bem como do correto uso do símbolo não terminal. Já o TermTable é representado pelo conjunto de Key com o símbolo terminal “.” .

[2]Caso tenhamos uma declaração no início, todas as atribuições seguintes são sempre referentes a uma tabela já definida.

É importante notar que neste símbolo não terminal foi usado recursividade à direita, ao contrário dos restantes símbolos, pela facilidade na inserção dos termos na **Key**, ou seja, como na segunda produção do TermTable, como o TermTable é uma tabela que se encontra dentro da tabela Key, há uma maior facilidade na inserção com este tipo de recursividade.

De seguida, o Key é representado pelos símbolos terminais que indicam todo o tipo de string disponíveis para o nome de uma tabela.

```
Key : KEY_STRING
    | KEY_NORMAL
    | STRING
    ;
```

O Atributions pode ser definido como sendo um conjunto de atribuições representadas pelo Atrib. Nas produções deste símbolo vamos definir todas as atribuições que são depois inseridas na estrutura de dados na tabela ou local correto.

Já o Atrib corresponde à verdadeira atribuição a nível da nossa estrutura de dados do Value, valor a inserir no Term, ou seja, na variável.

```
Atribuitions :
             | Atribuitions Atrib
             ;
Atrib : Term '=' Value
      ;
Term : Key
     | Key '.' Term
     ;
```

O Term é semelhante ao TermTable, tendo como única diferença a forma que se comporta a nível de inserção na estrutura de dados. Este caso acontece quando temos, por exemplo, "a.b = 2", em que o "a.b" se comporta como uma variável, mas na verdade o "a" é uma tabela com o "b" como atribuição.

Por último, o Value indica todas as possíveis variáveis que se pode ter no subconjunto de **TOML** que foi implementado.

```
Value : NUM
      | Array
      | BOOL
      | STRING
      | DATE
      ;
Array : '[' ']'
      | '[' ArrayString EndArray ']'
      | '[' ArrayNum EndArray ']'
      | '[' Arrays EndArray ']'
      | '[' ArrayBool EndArray ']'
      | '[' ArrayDate EndArray ']'
      ;

EndArray :
         | ','
         ;
```

Assim, o Value pode assumir as variáveis de número, *array*, *boolean*, *string* e data, através dos códigos dos seus símbolos terminais enviados pelo analisador léxico.

De seguida, temos de definir o único símbolo não terminal do Value, o Array, que devido à gramática do **TOML** necessita de ter todos os elementos do mesmo tipo.

De realçar que podemos também ter um *array* de *arrays* pela produção 4 do código anterior e que um *array* em **TOML** pode acabar em vírgula e, por isso mesmo, temos o símbolo não terminal EndArray.

```
ArrayString : STRING
            | ArrayString ',' STRING
            ;
ArrayNum : NUM
         | ArrayNum ',' NUM
         ;
ArrayBool : BOOL
          | ArrayBool ',' BOOL
          ;
ArrayDate : DATE
          | ArrayDate ',' DATE
          ;
Arrays : Array
       | Arrays ',' Array
       ;
```

Neste código representamos o *array* de um tipo de dados como sendo um conjunto desse tipo separado apenas pelo símbolo terminal, a vírgula.

3.4 Estruturas de dados

A análise da estrutura e funcionamento da linguagem de programação **TOML** permitiu-nos perceber a sua forte ligação com representação indexada de dados por chaves e valores associados a essas chaves definidos de forma recursiva, ou seja, uma vista muito próxima do resultado final pretendido com o reconhecimento de frases pelo **Yacc**.

O **JSON** é então um dos principais exemplos de estruturas deste tipo, pelo que o mais lógico seria utilizar tabelas de *hashing* na representação dos dados. Com uma *hashtable* final a tradução dos dados para a geração do ficheiro *output* é direta, podendo-se comparar a um simples *toString()* de uma linguagem orientada a objetos.

3.4.1 Indexação por chaves e tabelas de *hashing*

Obviamente este não é um trabalho cujo foco seja a aplicação de algoritmos e criação de estruturas complexas para armazenar dados, pelo que o grupo decidiu utilizar APIs públicas, como a **glib**, para o efeito.

Uma estrutura como uma tabela de *hashing* tem um conjunto de entradas independentes, mapeadas da forma par chave-valor (*key - value*), pelo que o próximo passo seria estabelecer quais seriam as *keys*, ou melhor, qual o formato a adotar para uma *key* associada a uma atribuição ou à definição de uma tabela.

O mais simples será então utilizar uma *string* (*char** do C) para definir a chave e ter alguma forma de guardar também um valor associado a um atributo dentro da própria *key*. Isto levou-nos a definir uma estrutura **general_key** que continha uma série de informações como o tipo de chave, indicação se já está definida ou não e por fim o valor associado à própria chave:

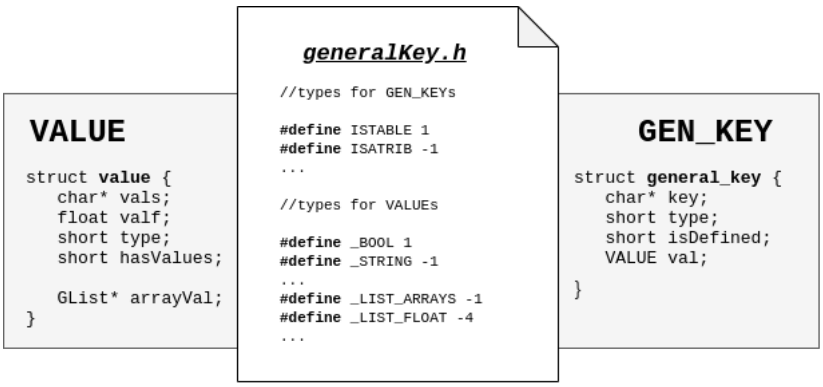

Figura 3.1: Estruturas de dados para chaves e valores.

O valor (**struct value**) da *struct general_key* é novamente uma estrutura dada a variedade de informação que pode estar atribuída a uma chave, tendo para isso definido um conjunto de *macros* para saber que tipo de dados se encontra armazenado na *key*.

Como se pode ver na figura anterior, todas estas estruturas e definições globais encontram-se definidas no *header* **generalKey.h** para facilitar a organização do código C.

As *macros* presentes no caso das **GEN_KEY** permitem perceber se a chave pode ter mais associações (*values*) ou não após terem sido criadas, ou seja, caso uma chave tenha um *type* definido como *ISTABLE* podemos deduzir que não existem valores associados a essa chave diretamente, isto é, apenas pode ter outras chaves filho no campo *value* da *hashtable*.

Já a *struct value* armazena o valor associado a uma dada *key* após uma atribuição, podendo esse valor ser uma *string*, um *float* ou até um *array* de valores.

3.4.2 Exemplo de aplicação

De forma simples, nesta secção, pretende-se mostrar o resultado esperado de um exemplo da linguagem **TOML** e o resultado esperado para a estrutura final de dados obtida:

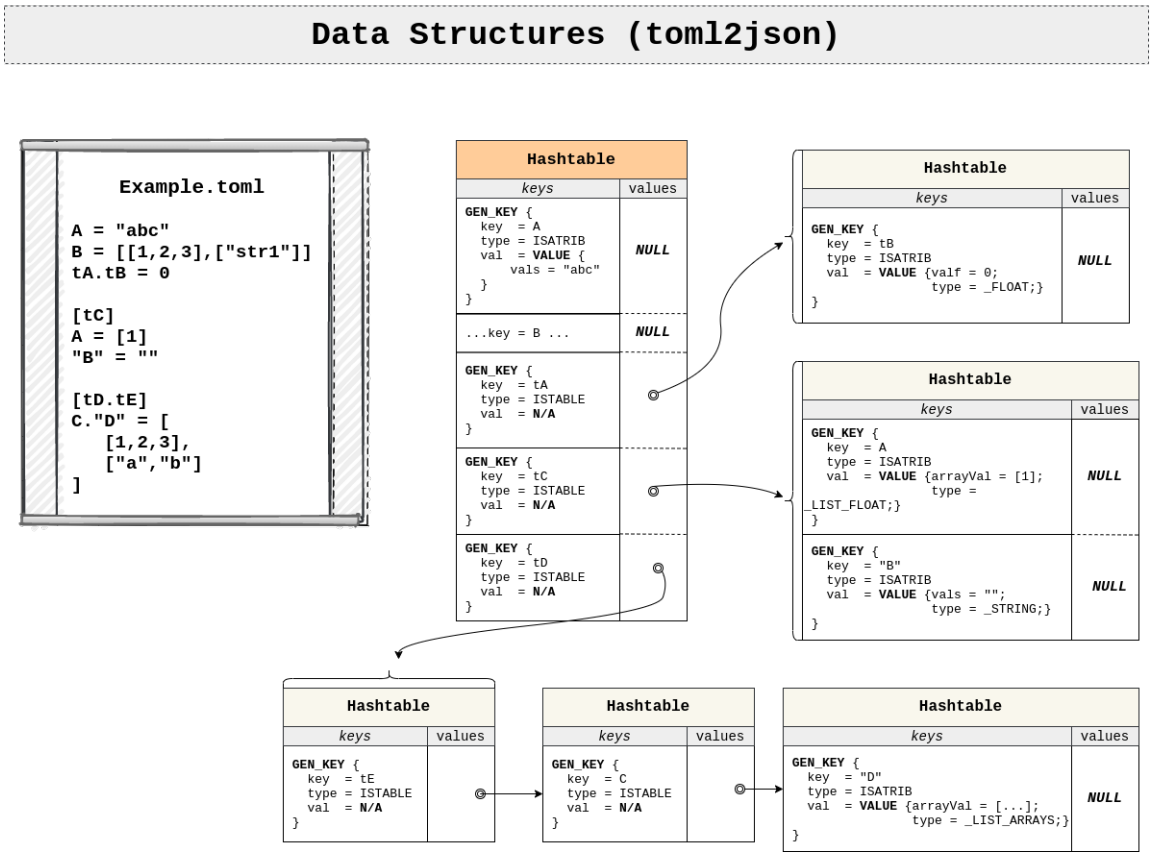

Figura 3.2: Exemplo de aplicação TOML e estrutura final.

Este exemplo cobre, assim, grande parte dos casos válidos para definição de pares chave-valor neste trabalho, pelo que o a tabela final, representada à direita é sucessivamente criada à medida que o analisador sintático faz a leitura e aplicação de algoritmos de *parsing bottom-up*.

Numa próxima secção iremos abordar, de forma breve, como é feita a gestão desta tabela de *hashing* ao nível das adições, controlo de valores duplicados e construção recursiva de tabelas.

Capítulo 4

Estrutura do código

4.1 Árvore de ficheiros e código fonte

A organização do código do programa final é uma parte importante para qualquer programa pelo que não foi ignorada de todo. Temos vários módulos em C com referência a diferentes porções de funções, estruturas e macros utilizadas em várias situações durante a análise sintática do **Yacc**.

Os diferentes ficheiros que compõem o nosso trabalho seguem a seguinte estrutura em árvore:

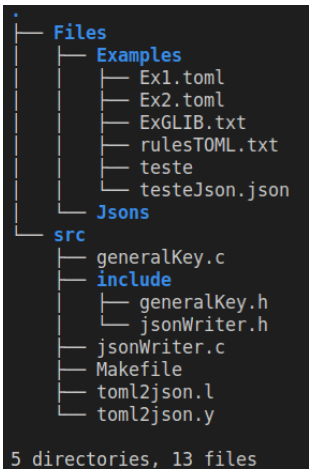

Figura 4.1: Árvore da diretoria raiz do projeto.

Na diretoria *Files* temos diversos exemplos de aplicação de **TOML** que fomos obtendo da pesquisa que foi feita sobre o que se pode e não pode fazer nesta linguagem, assim como alguns *inputs* exemplo ao programa, como o *Ex1.toml*, enquanto que na diretoria *Jsons* ficarão todos os *outputs* do *parser* no formato pretendido.

Por outro lado, foram estabelecidos os *headers* e os ficheiros relativos ao código fonte do projeto, armazenados na diretoria *src*.

Existem, portanto, dois ficheiros principais, o analisador léxico com o código do **FLex** *toml2json.l* e o analisador sintático com o código do **Yacc** *toml2json.y* onde é feito o reconhecimento de todos os símbolos terminais e onde são aplicadas as regras sintáticas manipulando esses *tokens*.

No módulo *generalKey.c* definimos as várias funções que fazem a manipulação dos dados da estrutura final que será convertida para o formato pretendido e também a definição da estrutura das chaves e valores no respetivo *generalKey.h*.

Por fim, o *jsonWriter.c* pretende encapsular as funções relativas à conversão dos dados das tabelas de *hashing* definidas para o ficheiro *output* final segundo as regras impostas por estruturas como o **JSON**.

4.2 Construção das tabelas de *hashing*

Aproveitamos esta secção para descrever, de forma geral, como é feita a inserção e verificação de erros durante o processo de construção das tabelas que servirão de conversão dos dados para o formato final.

A inserção de novas tabelas à estrutura de dados principal é um processo que deve ser feito verificando a existência de chaves já definidas que podem produzir o mesmo *hash* e, por isso, substituir o valor anteriormente associado a uma determinada chave.

Deste modo, quando é reconhecida uma nova chave, através da seguinte semântica:

```
Hipótese 1: [chave1] # definir uma tabela indexada com chave1
Hipótese 2: [chave1.chave2] # definir duas tabelas encadeadas
```

É necessário verificar, deste modo, a existência de duplicados e isso é feito através da seguinte função, dado o seu protótipo **update_table(tableEntrada, tabelaFinal)**.

O algoritmo, de forma geral, recebe uma tabela de entrada, que corresponde à última tabela definida em **TOML** através das hipóteses especificadas acima, correspondendo também ao fim de processamento do símbolo não terminal:

```
Table : '[' TermTable ']'
      ;
```

Onde, após reconhecimento do *TermTable*, o valor de **$2** contém um encadeamento de *GHashTables*[1] com o termo a ser inserido, por exemplo:

```
Exemplo 1: Inserir [chave1] produz a seguinte HT:
    $2 = (HT: key (chave1)) -> NULL
Exemplo 2: Inserir [chave1.chave2] produz a seguinte HT:
    $2 = (HT: key (chave1)) -> (HT: key (chave2)) -> NULL
```

A função introduzida acima tem então o trabalho de verificar se na HT **tabelaFinal** existe um encadeamento de HTs tal que corresponda ao caminho (*path*) produzido pelos exemplos acima, caso exista deve reportar erro **-1** para as demais funções que a chamam.

A função deve ser capaz então de verificar situações como:

```
# inserir encadeamento a -> b -> c -> NULL
# apenas a tabela "c" se encontra definida! "a" e "b" são (Not defined)
[a.b.c]
atribsC = ...

# inserir, de seguida, "a" deve ser possível, mudando o seu estado
# de (Not Defined) para (Defined)
[a]
atribsA = ...
```

[1] Utilizaremos HT como abreviação para *GHashTables* da **glib**.

A nível de atribuições foi necessário implementar funções capazes de atribuir *values* a certos termos capazes de os suportar, ou seja, que fossem do tipo *IS_ATRIB*.

Para isso, a função **add_lastTable** serviu como atualização do *value* do term e consequente inserção no local correto da tabela final.

```
Atrib : Term '=' Value
# É sempre chamado o add_lastTable(...) de forma a inserir o
Value no Term
```

Para a inserção ocorrer da forma esperada, é importante que o símbolo não terminal **Value** resolva e retorne o valor correto para a inserção.

Capítulo 5

Resultados obtidos

De forma a verificar os resultados do nosso conversor, foram desenvolvidos dois exemplos de ficheiros **TOML** a que o nosso programa consegue dar resposta, sendo que um deles é o disponibilizado pela equipa docente.

Para além de escrever a conversão num ficheiro previamente identificado, o conversor indica no terminal durante o tempo de execução as tabelas por ele lidas como forma de *debug* e de perceber como ficará o resultado final.

```
# 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"
]
#c = [["er", "#"], [], [12,2], [[]] ]
```

Figura 5.1: Ficheiro TOML

Assim, podemos ver de seguida um exemplo de aplicação do nosso conversor ao ficheiro **TOML** indicado em cima, dando o resultado final e alguns *prints* durante a execução deste da nossa estrutura de dados.

Figura 5.2: Estrutura de dados durante a execução e resultado final em JSON

A imagem à esquerda representa os *prints* durante o tempo de execução da nossa estrutura de dados, enquanto que a imagem à direita é o resultado da conversão do **TOML** para **JSON**.

De notar que o nosso conversor pode ser executado através do comando:

```
make run < ../Files/Examples/Ex1.toml
```

Capítulo 6

Conclusão

Este trabalho vem complementar a componente teórico-prática fortemente aplicada nesta unidade curricular pelo que serve de um bom teste aos nossos conhecimentos e a forma como podemos lidar com aquilo que é tão básico para um programador como um compilador.

O trabalho em questão foi interessante no sentido em que pusemos em prática conhecimentos relativos a analisadores léxicos como o **FLex** e à importância do mesmo no que toca ao processo de *parsing* e deteção de caracteres de erro para uma linguagem, analisadores sintáticos como o **Yacc** resolvendo por vezes conflitos relativos aos algoritmos de *parsing bottom-up* e a estratégias para os resolver e também outras linguagens de programação como o **TOML** que são usualmente preferidas em situações de configuração de programas e dispositivos simples.

Por outro lado, a aplicação de algoritmos em estruturas de dados complexas serviu de revisão e reforço da sua importância no desenvolvimento de compiladores rápidos e eficientes, reutilizando APIs públicas do C na aplicação prática dos mesmos.

Em suma, achamos que este trabalho teve grande proveito no que toca ao assimilar de conhecimentos relativos ao desenvolvimento de *Processadores de Linguagens Regulares* que filtram e transformam textos, identificando semânticas, tendo em base o conceito de regras de produção *Condição-Ação* e, a nosso ver, cumprimos os requisitos propostos pela equipa docente no que toca a este projeto.