



UNIVERSIDADE DO MINHO

PROCESSAMENTO DE LINGUAGENS (3º ANO DE CURSO)

Trabalho Prático 2

RELATÓRIO DE DESENVOLVIMENTO

Mestrado Integrado em Engenharia Informática

Realizado por G09:

Hugo André Coelho Cardoso, a85006

João da Cunha e Costa, a84775

Válter Ferreira Picas Carvalho, a84464

28 de Junho de 2020

Resumo

O segundo trabalho prático realizado por este grupo no âmbito da unidade curricular *Processamento de Linguagens* consistiu na resolução de um conversor de uma parte da linguagem *TOML* (a ser explorada mais à frente) para *JSON*, que se traduziu no desenvolvimento de um programa à base de *Flex* e *Yacc*.

No presente relatório será explicada a forma como o grupo interpretou o problema e envisionou a solução, bem como as adversidades que surgiram durante a sua implementação, de maneira a construir um programa escalável capaz de processar não apenas o exemplo dado no enunciado, como também ficheiros *TOML* bem mais complexos, de maneira a proporcionar mais liberdade ao utilizador final da aplicação.

Os principais objetivos deste trabalho prático passaram por aumentar a experiência de uso do ambiente Linux, de linguagens imperativas (em particular, será focado o C) na implementação de ações semânticas e sintáticas; desenvolver a capacidade de escrita de grámatas independentes de contexto, *GIC*, que satisfaçam a condição LR() para criar Linguagens de Domínio Específico, *DSL*; desenvolver processadores de linguagens através da tradução orientada à sintaxe, suportado numa gramática tradutora, *GT*; utilizar geradores de compiladores, em particular, o *Flex/Yacc*.

Conteúdo

| | | |
|----------|---|-----------|
| 1 | Introdução | 3 |
| 1.1 | Enquadramento e Contextualização | 3 |
| 1.2 | Problema e Objetivo | 3 |
| 1.3 | Resultados ou Contributos | 4 |
| 1.4 | Estrutura do Relatório | 4 |
| 2 | Análise e Especificação | 5 |
| 2.1 | Descrição informal do problema | 5 |
| 2.2 | Especificação do Requisitos | 5 |
| 2.2.1 | Dados | 5 |
| 2.2.2 | Pedidos | 5 |
| 2.2.3 | Relações | 5 |
| 3 | Conceção/desenho da Resolução | 6 |
| 3.1 | Gramática Independente de Contexto | 6 |
| 3.1.1 | Esqueleto da GIC | 6 |
| 3.1.2 | Ações Semânticas na GIC | 9 |
| 3.2 | Analisador Léxico | 16 |
| 4 | Codificação e Testes | 27 |
| 4.1 | Execução do Programa | 27 |
| 4.2 | Alternativas, Decisões e Problemas de Implementação | 27 |
| 4.3 | Testes realizados e Resultados | 27 |
| 4.3.1 | Teste 1 | 28 |
| 4.3.2 | Teste 2 | 28 |
| 4.3.3 | Teste 3 | 30 |
| 4.3.4 | Teste 4 | 31 |
| 4.3.5 | Teste 5 | 35 |
| 5 | Conclusão | 39 |
| A | Código do Programa e Makefile | 40 |
| B | <i>Exemplo de Ficheiros TOML e JSON</i> | 51 |

Lista de excertos de código

| | | |
|---|--|----|
| 1 | Expressão regular que apanha uma chave | 16 |
| 2 | Condição VALORBLOCO | 17 |
| 3 | Condição VALOR | 17 |
| 4 | Condição ASPAS | 18 |
| 5 | Condição NOMBLOCO | 21 |
| 6 | Condição BLOCO | 23 |

Lista de Figuras

| | | |
|---|---|----|
| 1 | Conversão das listas de TOML para JSON | 10 |
| 2 | Conversão de pares chave-valor de TOML para JSON | 12 |
| 3 | Conversão de secções de TOML para JSON | 14 |
| 4 | Bloco de declarações inicial de TOML | 16 |
| 5 | Exemplo de objetos em TOML | 20 |
| 6 | Atributos com chaves compostas e respetiva tradução para JSON | 25 |

1 Introdução

Área: Processamento de Linguagens

No âmbito da unidade curricular *Processamento de Linguagens*, inserida no plano de estudos do Mestrado Integrado em Engenharia Informática da Universidade do Minho, foi proposta a realização de um trabalho prático focado na elaboração de filtros de texto, com recurso à ferramenta *Flex*, assim como a elaboração duma gramática independente de contexto, com recurso ao *Yacc*.

Neste relatório será abordado, em detalhe, todo o processo criativo por detrás da resolução do problema proposto no enunciado nº 4, isto é, **Conversor toml2json**, que foi atribuído ao grupo pela equipa docente, através da fórmula explícita para seleção: $9 \% 6 + 1 = 4$.

1.1 Enquadramento e Contextualização

A linguagem *TOML*, cujo nome significa *Tom's Obvious, Minimal Language*, é uma linguagem que se foca na simplicidade tanto de leitura como escrita para descrever estruturas complexas e hierárquicas de dados, através do sistema chave-valor, secções e comentários, cujo objetivo é ser traduzida diretamente para *hash tables* sem ambiguidades.

Este projeto consiste em criar um programa **toml2json** que transforme uma parte da linguagem *TOML* (arbitrária pelo grupo) em um ficheiro *JSON*.

Seguem-se, então, as regras parciais utilizadas pelo grupo:

- Uma **secção** inicia-se quando é detetado *[secção]*
- As **secções** podem ser hierarquizadas, isto é, podem existir *[secção.foo1]*, *[secção.foo1.foo2]*, etc, para a secção original *[secção]*
- Existem pares *chave = valor* no ficheiro *TOML* dentro e fora de secções;
- Uma **chave** é obrigatoriamente inicializada com letra minúscula e não contém espaços, isto é, a chave *date_time* é válida enquanto *date time* não é;
- Uma **chave** pode estar indentada de forma arbitrária;
- Uma **chave** pode estar definida hierarquicamente, por exemplo *orange.color = ...*;
- Um **valor** pode ser, por exemplo, um booleano, uma lista (indentada arbitrariamente), um inteiro, uma data, uma hora, no entanto, tudo o que seja diferente dos mencionados será tratado como uma *string*;
- Os **comentários** podem ser colocados em qualquer parte do texto desde que inicializados com o carácter *'#'*.

No anexo B é visível um exemplo de um ficheiro com extensão *TOML* onde se podem verificar as regras enunciadas.

1.2 Problema e Objetivo

Como foi referido acima, o ficheiro original *TOML* segue regras concretas (apesar de serem parte das regras globais de toda a linguagem), o que implica o tratamento individual de cada uma das situações.

Como implica tratamento exaustivo, são gerados uma série de problemas nomeadamente com a indentação arbitrária, hierarquização de secções, leitura dos valores e associá-los a cada

chave assim como colocar os pares chave-valor na secção respetiva (se aplicável). Um outro problema é que esta linguagem permite a existência de comentários enquanto que o *JSON* não, o que implica ações especiais nestes casos.

Obtendo uma interpretação satisfatória dos problemas, é necessário proceder à sua resolução, para garantir que o resultado final é um ficheiro *JSON* coerente e pronto a usar sem qualquer tratamento posterior, resultado da tradução direta de qualquer ficheiro *TOML* que se pretenda traduzir.

1.3 Resultados ou Contributos

O grupo procurou ir além dos requisitos presentes no *template* fornecido, com o objetivo de elaborar uma solução mais abrangente e complexa. Desta maneira, o resultado obtido permite, entre outros:

- Indentação totalmente arbitrária para secções e pares chave-valor;
- Hierarquização de secções;
- Listas indentadas arbitrariamente;
- Listas podem ter listas aninhadas;
- Separação (nº de espaços) entre chaves e o caracter '=' arbitrária assim como desse mesmo caracter ao valor respetivo de cada chave;
- As listas têm apenas um elemento por linha no ficheiro *JSON* para uma melhor leitura;
- Hierarquização de pares chave-valor;
- Indentação legível e coerente do ficheiro *JSON* resultante;
- Qualquer erro no ficheiro *TOML* é imediatamente comunicado ao utilizador.

1.4 Estrutura do Relatório

O presente relatório está dividido em cinco capítulos, sendo este o primeiro, cujo propósito é introduzir o projeto, contextualizando-o e fazendo uma breve análise do problema e objetivos, além de tocar levemente nos resultados obtidos.

No capítulo 2 realiza-se a análise e especificação do projeto, descrevendo o problema de maneira mais pormenorizada e identificando os seus requisitos, distinguindo dados, pedidos e relações.

No capítulo 3 é abordada em detalhe a conceção/desenho da resolução, explicando as estruturas de dados usadas e os motivos pelos quais foram escolhidas, os algoritmos elaborados e a função de todos os ficheiros que foram criados.

No capítulo 4 expõem-se as alternativas e problemas de implementação que surgiram, justificando as decisões tomadas, e percorre-se os testes realizados ao programa de maneira a testar a sua operabilidade, bem como os resultados produzidos a partir dos mesmos.

Finalmente, no capítulo 5, termina-se o relatório com uma síntese do que foi dito, as conclusões e o trabalho futuro.

No apêndice do relatório estão ainda disponíveis para consulta o código e *Makefile* desenvolvido e exemplos de ficheiros *TOML* e *JSON* relativos ao enunciado, referenciados ao longo deste relatório.

2 Análise e Especificação

2.1 Descrição informal do problema

Dado um ficheiro seccionado e cujo conteúdo é um texto escrito em *TOML*, é necessário determinar todas as secções e todas as chave-valor existentes, mantendo a coerência original no novo ficheiro *JSON*.

É, portanto, necessário construir uma ordem de hierarquia (implícita ou explícita) que seja mantida na sua tradução para o novo formato e garantir que todas as secções acabam com as mesmas subsecções e/ou pares chave-valor originais.

É fundamental distinguir um valor numa secção (porque implica indentações e sintaxe diferentes), assim como filtrar todos os comentários existentes no ficheiro de entrada, para garantir que o *JSON* é coeso e robusto.

Por fim, é fundamental garantir que se existir um erro no programa, o utilizador é imediatamente informado do porquê e onde ocorreu, relativamente ao ficheiro de entrada.

2.2 Especificação do Requisitos

2.2.1 Dados

Os dados do problema constituem o **conhecimento *a priori*** a partir do qual se elabora a solução e, no que toca a este enunciado, correspondem ao **texto de entrada** que é necessário filtrar e às suas características.

Como tal, pode-se afirmar que há três tipos de dados neste problema:

- Pares **chave-valor** presentes em qualquer parte do texto de entrada;
- **Secções**, opcionais, que indicam a hierarquização de todos os pares chave-valor no ficheiro *TOML*;
- **Comentários** presentes em qualquer parte do ficheiro *TOML*.

2.2.2 Pedidos

Os pedidos correspondem aos **requisitos estabelecidos** no enunciado e os **resultados pretendidos** para o trabalho.

Neste caso, é esperado que a partir do ficheiro *TOML* dado por argumento seja criado um novo ficheiro *JSON* com exatamente a mesma informação do ficheiro original (à exceção dos comentários, uma vez que não existem em *JSON*).

2.2.3 Relações

As relações traduzem-se nas **ligações existentes entre os dados e os pedidos**, ou seja, de que maneira é que obtemos estes últimos partindo dos primeiros. No contexto deste enunciado, são os **mapeamentos ou transformações necessárias** para produzir a saída:

- Todos os comentários serão **descartados**;
- Qualquer secção no formato *TOML* deverá ser transformada no equivalente em *JSON* (consultar [B](#));
- Todos os pares chave-valor serão transformados no seu equivalente em *JSON* (consultar [B](#)).

3 Conceção/desenho da Resolução

O desenvolvimento da solução foi um processo iterativo, ao longo do qual o grupo mudou várias vezes de estratégia, implementando estruturas de dados e algoritmos cada vez mais eficientes. Neste capítulo, será abordada apenas a solução final, deixando a evolução do projeto para o subcapítulo 4.2, onde serão explicadas as limitações das implementações iniciais e a culminação no produto final.

3.1 Gramática Independente de Contexto

3.1.1 Esqueleto da GIC

Será discutida nesta secção o esqueleto da gramática implementada para realizar este processo de conversão, isto é, sem as respetivas ações semânticas que serão discutidas mais à frente com mais detalhe.

Como apontado anteriormente, é estritamente necessário distinguir pares chave-valor que estejam dentro e fora de hierarquias, assumindo sempre que, a partir do momento que se inicializa uma secção nova, todos os pares que apareçam imediatamente a seguir pertencem-lhe obrigatoriamente. Portanto, temos dois casos: pares **antes** de uma secção e **dentro** de uma secção (o grupo decidiu usar o nome bloco na GIC mas refere-se à secção no ficheiro *TOML*), que é o ponto de partida da gramática desenvolvida.

```
Tom1: ChavesValores INITBLOCOS Blocos
      ;
```

Nesta produção, está explícito que um ficheiro *TOML* está definido por pares chave-valor fora de secções seguidos de pares dentro das secções, que era o pretendido. O símbolo terminal *INITBLOCOS* é apenas para sinalizar o início do tratamento de secções, para ser mais simples estabelecer esta divisão.

De seguida, era necessário definir os pares chave-valor na gramática, pelo que foi necessário a inserção de um novo conjunto de produções, garantindo a coerência com a sintaxe *TOML* original, isto é, os pares são separados pelo carácter '='.

```
ChaveValor: chave '=' Valor
            | chave '.' chave '=' Valor ChaveBloco FIMCHAVEBLOCO
            ;
```

Analisando as produções, verificamos que um par é simplesmente definido pelo mencionado previamente ou então é um par chave-valor **hierarquizado** pela chave, pelo que o símbolo terminal *FIMCHAVEBLOCO* indica que terminou a sequência de chaves hierarquizadas (têm de estar escritas sequencialmente para pertencerem à mesma hierarquia). O símbolo *chave* é definido como símbolo terminal variável porque as chaves são totalmente arbitrarias.

Salientando agora o outro conjunto de produções *ChaveBloco* verificamos que é simplesmente definida por uma sequência finita de *ElemChaveBloco*, cuja única produção é um par chave-valor hierarquizado. Estas produções criam a relação sequencial de dependências de chaves hierarquizadas, isto é, se existirem têm de estar escritas imediatamente seguidas para uma mesma ordem de hierarquia.

```
ChaveBloco: ChaveBloco ElemChaveBloco
            |
            ;
```



```
ElemChaveBloco: chave '.' chave '=' Valor
;
```

Quanto aos valores, o desafio está na diversidade de tipos que pode assumir, sejam *strings*, números inteiros, listas, entre outros. Portanto, é necessário ter em consideração todos estes casos, definidos anteriormente com mais detalhe, pelo que é necessário a criação de um conjunto de produções coerentes com o que foi definido.

```
Valor: Aspas
| valor
| Array
;
```

O símbolo terminal variável *valor* define todos os tipos atômicos que no *JSON* não são introduzidos entre aspas, como os booleanos e inteiros, por exemplo.

O conjunto de produções *Aspas* define valores detetados como algo entre aspas, isto é, uma *string* em *TOML*, porque é necessário saber onde começa e onde termina. O símbolo terminal variável *string* é usado para obter esse segmento separado por aspas, para o tratamento posterior em *JSON*, admitindo *strings vazias*.

```
Aspas: '"' string '"'
| "'" "'"
;
```

Por outro lado, o conjunto de produções *Array* dizem respeito à sintaxe permitida para listas, isto é, será considerado uma lista uma sequência iniciada pelo carácter '[', que permite elementos espaçados pelo carácter ',' e terminadas por ']', como por exemplo '[1,2,"abc"]'.

```
Array: '[' ElsArray ']'
| '[' ']'
;
ElsArray: ElsArray ',' Valor
| Valor
;
```

Com esta definição obtemos a sintaxe pretendida e garantimos que poderão existir listas aninhadas (porque *Valor* assume um possível *Array*), assim como os outros restantes tipos compatíveis com *Valor*.

Terminando a análise dos pares chave-valor, será agora focada a análise das secções. Uma secção essencialmente é um conjunto de secções (implica hierarquia) e de pares chave-valor, em que começa pelo carácter '[' e termina em ']', em que no meio deles está o nome (hierarquizado ou não) da secção.

```
Blocos: Blocos Bloco
|
;
Bloco: TagBloco FIMTAG ElsBloco FIMBLOCO
;
TagBloco: '[' chave ']'
;
```

Como é possível analisar, verificamos que a única produção de *TagBloco* é utilizada para definir o nome da secção e *FIMTAG* é um símbolo terminal apenas para definir o final deste nome. Um ficheiro *TOML* é constituído, para além de pares chave-valor, por zero ou mais secções, como está definido nas produções *Blocos*, em que é necessário definir o que é um *Bloco*.

Um *Bloco* (novamente, o termo *Bloco* e *Secção* significam o mesmo no contexto lógico, o grupo é que decidiu utilizar designações diferentes) é um conjunto definido pelo seu nome, já explicado, e pelos seus elementos, sendo marcado o seu fim com o símbolo terminal *FIMBLOCO*.

```
ElsBloco: ElsBloco ElemBloco
        | ElemBloco
        ;
ElemBloco: ChaveValor
        | Bloco
        ;
```

Rapidamente verificamos que os *ElsBloco* são pelo menos 1 *ElemBloco*, ou seja, a secção definida poderá ser vazia (quando o *Bloco* é vazio) ou então tem pelo menos um elemento, que é precisamente o que acontece em *TOML*. Os elementos aceites para cada secção são pares chave-valor ou outras secções, como definido anteriormente.

De seguida está disponível o esqueleto da GIC completa.

```
Toml: ChavesValores INITBLOCOS Blocos
    ;

Blocos: Blocos Bloco
    |
    ;

Bloco: TagBloco FIMTAG ElsBloco FIMBLOCO
    ;

ElsBloco: ElsBloco ElemBloco
    | ElemBloco
    ;

ElemBloco: ChaveValor
    | Bloco
    ;

TagBloco: '[' chave ']'
    ;

ChavesValores: ChavesValores ChaveValor
    |
    ;

ChaveValor: chave '=' Valor
    | chave '.' chave '=' Valor ChaveBloco FIMCHAVEBLOCO
    ;

ChaveBloco: ChaveBloco ElemChaveBloco
    |
```

```

;

ElemChaveBloco: chave '.' chave '=' Valor
;

Valor: Aspas
      | valor
      | Array
;

Array: '[' ElsArray ']'
      | '[' ']'
;

ElsArray: ElsArray ',' Valor
         | Valor
;

Aspas: '"' string '"'
      | '"' ' ' '"'
;

```

3.1.2 Ações Semânticas na GIC

A verdadeira utilidade da definição na gramática está associada às suas ações semânticas. Para efetuar realmente a conversão em *JSON*, é necessário que sempre que seja completada uma das produções acima definidas, algo seja produzido na nova linguagem a partir do *TOML*.

A estrutura *union* definida permite-nos indexar valores semânticos dos símbolos terminais, que neste caso são apenas *strings*. Como pretendemos transformar tudo em *strings* para a conversão imediata para *JSON*, é só necessário este tipo porque todos os restantes iam ter exatamente o mesmo tratamento, evitando definições redundantes.

```

%union{
    char* s;
}
%type <s> valor chave string
%type <s> Blocos Bloco ElsBloco ElemBloco TagBloco ChavesValores ChaveValor
ChaveBloco ElemChaveBloco Valor Array ElsArray Aspas

```

Nesta secção será utilizado como exemplo o ficheiro disponível em [B](#) para os valores resultantes de cada ação semântica para cada produção. Começando por uma produção simples, as *Aspas* definem *strings* no ficheiro *TOML* original.

```

%union{
Aspas: '"' string '"'                                {asprintf(&$$, "\"%s\"", $2);}
      | '"' ' ' '"'                                {asprintf(&$$, "\" \" \");}
;

```

Com estas ações semânticas, está a ser colocado no valor da produção uma *string* em que o seu conteúdo é uma sequência de letras entre aspas.

Por exemplo, no caso da *string* "*TOML Example*", o valor desta produção será exatamente igual.

No que toca às produções *Valor*, apenas anexam ao valor da produção os valores das suas produções atômicas.

Para o caso das listas, o *JSON* define uma sintaxe mais legível que é um elemento por linha e indentando corretamente, que é utilizado nesta GIC para ser mais apelativo ao utilizador.

```

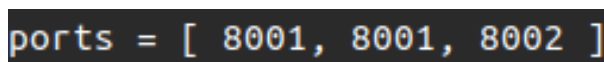
Valor: Aspas      {asprintf(&$$,"%s",$1);}
      | valor     {asprintf(&$$,"%s",$1);}
      | Array     {asprintf(&$$,"%s",$1);}
      ;

Array: '[' ElsArray ']'
      {
        char* indentacao = indent(blocoAtual+arrayAtual);
        asprintf(&$$,"[\n%s\n%s]", $2, indentacao);
      }
      | '[' ']'    {asprintf(&$$,"[]");}
      ;

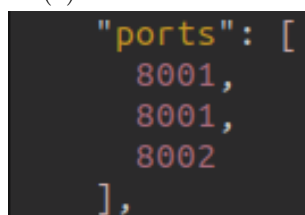
ElsArray: ElsArray ',' Valor
        {
          char* indentacao;
          if ($3[0] == '[') indentacao = indent(blocoAtual+arrayAtual);
          else indentacao = indent(blocoAtual+arrayAtual+1);
          asprintf(&$$,"%s,\n%s%s", $1, indentacao, $3);
        }
      | Valor
        {
          char* indentacao;
          if ($1[0] == '[') indentacao = indent(blocoAtual+arrayAtual);
          else indentacao = indent(blocoAtual+arrayAtual+1);
          asprintf(&$$,"%s%s", indentacao, $1);
        }
      ;

```

Nota-se, também, a presença de um conjunto de variáveis que garantem a indentação correta. Esta indentação essencialmente baseia-se na secção atual e a ordem hierárquica para garantir que todos os elementos estão nivelados corretamente para transmitir ao utilizador uma melhor experiência.



(a) Lista em TOML



(b) Lista em JSON

Figura 1: Conversão das listas de TOML para JSON

Com as listas definidas, é necessário concluir o resultado para os pares chave-valor. Para tal, é tido também em consideração todas as indentações devido às hierarquias e a sintaxe "chave": valor típica do JSON. Para cada linha é anexada, também uma vírgula caso o par não seja o último relativo à hierarquia atual.

```
ChavesValores: ChavesValores ChaveValor
{
    if(strlen($1) > 0) asprintf(&$$,"%s,\n%s",$1,$2);
    else asprintf(&$$,"\n%s",$2);
}
|
;

ChaveValor: chave '=' Valor
{
    char* indentacao;
    if (blocoAtual==0) indentacao = indent(blocoAtual);
    else indentacao = indent(blocoAtual+1);
    asprintf(&$$,"%s\"%s\": %s",indentacao,$1,$3);
}

| chave '.' chave '=' Valor ChaveBloco FIMCHAVEBLOCO
{
    char* ind = indent(blocoAtual);
    char* ind1 = indent(blocoAtual+1);
    char* ind2 = indent(blocoAtual+2);
    if(strlen($6)==0 && blocoAtual==0)
        asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s\n%s}",ind,$1,
        ind1,$3,$5,ind);
    if(strlen($6)==0 && blocoAtual!=0)
        asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s\n%s}",ind1,$1,
        ind2,$3,$5,ind1);
    if(strlen($6)!=0 && blocoAtual==0)
        asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s,\n%s%s}",ind,
        $1,ind1,$3,$5,ind,$6);
    if(strlen($6)!=0 && blocoAtual!=0)
        asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s,\n%s%s%s}",ind1,
        $1,ind2,$3,$5,ind2,$6,ind1);
}

;

ChaveBloco: ElemChaveBloco ChaveBloco
{
    char* ind = indent(blocoAtual);
    char* ind2 = indent(blocoAtual+2);
    if (strlen($2)==0 && blocoAtual==0) asprintf(&$$,
    "%s%s\n%s",$1,$2,ind);
    if (strlen($2)!=0 && blocoAtual==0) asprintf(&$$,
    "%s,\n%s%s",$1,ind,$2);
    if (strlen($2)==0 && blocoAtual!=0) asprintf(&$$,
    "%s%s\n%s",$1,ind2,$2);
    if (strlen($2)!=0 && blocoAtual!=0) asprintf(&$$,
    "%s,\n%s%s",$1,ind2,$2);
}
```

```

    }
    |    {$$ = "";}
    ;

ElemChaveBloco: chave '.' chave '=' Valor
    {
        char* indentacao = indent(blocoAtual);
        if (blocoAtual==0) asprintf(&$$,"%s\\\"%s\\\": %s",
            indentacao,$3,$5);
        else asprintf(&$$,"\\\"%s\\\": %s", $3,$5);
    }
    ;

```

O resultado de todas as ações semânticas prévias está visível como exemplo nas imagens que se seguem.

```

server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

```

(a) Pares chave-valor em TOML

```

"server": "192.168.1.1",
"ports": [
    8001,
    8001,
    8002
],
"connection_max": 5000,
"enabled": true

```

(b) Pares chave-valor em JSON

Figura 2: Conversão de pares chave-valor de TOML para JSON

O que as ações semânticas têm, também, de realizar é a separação pro secções. Em *JSON*, esta separação implica a introdução de chavetas para realizar o fecho de cada secção. Há algumas observações importantes sobre as secções (na sua tradução):

- É colocada a sua designação entre aspas no início seguido do carácter ':';
- Começa e termina com uma chaveta;
- Todos os pares chave-valor que lhes pertencem são colocados dentro das chavetas.

É importante realçar que é realizado, também a indentação completa para estas novas regras, para uma melhor experiência do utilizador.

```

Blocos: Blocos Bloco
    {
        if(strlen($1) > 0) asprintf(&$$,"%s,\\n%s", $1,$2);
        else asprintf(&$$,"\\n%s", $2);
    }
    |    {$$ = "";}
    ;

```

```

Bloco: TagBloco FIMTAG ElsBloco FIMBLOCO
    {
        char* indentacao = indent(blocoAtual);
        asprintf(&$$,"%s%s\n%s",$1,$3,indentacao);
    }

;

ElsBloco: ElsBloco ElemBloco
    {asprintf(&$$,"%s\n%s",$1,$2);}
| ElemBloco
    {asprintf(&$$,"%s",$1);}

;

ElemBloco: ChaveValor
    {
        char* indentacao = indent(blocoAtual);
        asprintf(&$$,"%s",$1);
    }
| Bloco {asprintf(&$$,"%s",$1);}

;

TagBloco: '[' chave ']'
    {
        char* indentacao = indent(blocoAtual);
        asprintf(&$$,"%s\\\"%s\\\": {\n",indentacao,$2);
    }

;

```

A aglomeração de todas as ações semânticas mencionadas até agora demonstram a possibilidade de especificar secções (que estão presentes nas imagens seguintes) ou pares chaves-valor fora de secções em *JSON*.

Todas as situações de hierarquização entre blocos está compacta neste excerto da GIC e coloca as indentações corretas para todos os níveis hierárquicos não só para as secções mas também para todos os elementos dentro de cada secção.

Foi uma parte do projeto bastante desafiante de completar porque era necessário uma visão da estrutura global do ficheiro, no entanto, apenas é possível saber o texto em tempo de interpretação sem saber o que que já tinha sido processado anteriormente, e para conseguir as indentações corretas foi imprescindível o interpretador léxico Flex a ser analisado mais à frente.

```
[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true
```

(a) Secção em TOML

```
"database": {
  "server": "192.168.1.1",
  "ports": [
    8001,
    8001,
    8002
  ],
  "connection_max": 5000,
  "enabled": true
},
```

(b) Equivalente à secção em JSON

Figura 3: Conversão de secções de TOML para JSON

Por fim, a ligação lógica entre as partes não conectadas é realizada pela única ação semântica disponível, que é também aquela que coloca o resultado final num ficheiro *JSON* cujo nome é dado por argumento (se não for dado um nome por argumento é escolhido um por defeito). Esta ação coloca uma chaveta no início e no fim, o que significa que um ficheiro *TOML* vazio originará um ficheiro *JSON* cujo conteúdo são só duas chavetas em linhas diferentes.

```
Toml: ChavesValores INITBLOCOS Blocos
{
  if(strlen($3) == 0) fprintf(yyout,"%{s\n}",$1);
  else if(strlen($1) == 0) fprintf(yyout,"%{s\n}",$3);
  else fprintf(yyout,"%{s,%s\n}",$1,$3);
}

;
```

Para o ficheiro *TOML* mencionado (B), o resultado final completamente filtrado de comentários e convertido em *JSON* é:

```
{
  "title": "TOML Example",
  "owner": {
    "name": "Tom Preston-Werner",
    "date": "2010-04-23",
    "time": "21:30:00"
  },
  "database": {
    "server": "192.168.1.1",
    "ports": [
      8001,
      8001,
      8002
    ],
  },
```




```
    "connection_max": 5000,  
    "enabled": true  
  },  
  "servers": {  
    "alpha": {  
      "ip": "10.0.0.1",  
      "dc": "eqdc10"  
    },  
    "beta": {  
      "ip": "10.0.0.2",  
      "dc": "eqdc10",  
      "hosts": [  
        "alpha",  
        "omega"  
      ]  
    }  
  }  
}
```

3.2 Analisador Léxico

Para complementar a lógica implementada em Yacc, com recurso à Gramática Independente de Contexto desenvolvida, foi também criado um analisador léxico em Flex, responsável por filtrar o input e enviar os respetivos tokens ao analisador sintático.

De início, foram declaradas as seguintes expressões regulares, de maneira a facilitar o seu acesso e a melhorar a legibilidade do filtro Flex.

| | |
|----------|-----------------------------------|
| digito | [0-9] |
| acentos | \xc3[\x80-\xbf] |
| letra | [a-zA-Z] {acentos} |
| palavra | {letra}+ |
| espacos | []* |
| chave | [a-z][^ \, \. \n \t \r \[\] \"]* |
| nullbool | (true) (false) (null) |
| num | (\+ -)?[0-9]+(\.[0-9]+)? |

Um ficheiro escrito na linguagem TOML pode ou não ter, no início, uma série de declarações de variáveis, como no exemplo seguinte:

```
title = "TOML Example"
var1 = true
var2 = false
exampleVariable = "Hello world!"
array = [ 8001, [8002,8003], 8004, "string"]
```

Figura 4: Bloco de declarações inicial de TOML

Como podemos observar, cada declaração consiste num par chave-valor, com o símbolo de atribuição '=' entre eles. Para reconhecer estes casos, começa-se por fazer uso da expressão regular **chave**, ilustrada em ??, colocada na condição geral do programa Flex, para captar o nome da variável: começa obrigatoriamente por uma letra minúscula e não pode ter espaços brancos nem caracteres especiais, especialmente os que têm relevância no resto do trabalho (",[],.), os quais salvaguardamos na expressão regular.

Caso seja reconhecido o nome de uma variável, o programa guarda-o na union do analisador sintático e envia ao mesmo um token **chave**. Por fim, salta para a start condition **VALOR-BLOCO**, que processa o resto da atribuição. A condição no código é relativa a outra parte do trabalho e será explicada mais para a frente.

```
1 <*>{chave} {
2     BEGIN VALORBLOCO;
3     if(chaveBlocoAtual) {
4         free(chaveBlocoAtual); chaveBlocoAtual = NULL;
5         chaveBlocoAtual = strdup(yytext);
6         tokens[2] = -1;
7         return FIMCHAVEBLOCO;
8     }
9     yylval.s = strdup(yytext);
10    return chave;
11 }
```

Listing 1: Expressão regular que apanha uma chave

Nesta nova condição, o programa começa por filtrar o símbolo de atribuição (=), enviando o token do respetivo caractere ao analisador sintático. De seguida, só resta consumir o valor da

variável, que pode ser um dos seguintes: string, encapsulada por aspas; array, encapsulado por parêntesis retos; um valor não encapsulado por pontuação.

```
1 <VALORBLOCO>{espacos}=      {return '=';}
2 <VALORBLOCO>{espacos}\"      {blocoAspas = 1; BEGIN ASPAS; return '\"';}
3 <VALORBLOCO>{espacos}\[      {BEGIN ARRAY; arrayAtual++; return '[';}
4 <VALORBLOCO>{espacos}        {BEGIN VALOR;}
```

Listing 2: Condição VALORBLOCO

No caso de um valor que não seja encapsulado por aspas ou parêntesis retos, o programa salta para a condição **VALOR**, que diz respeito ao símbolo não-terminal **Valor** do analisador sintático.

```
1 <VALOR>{nullbool}
2     {
3         if(chaveBlocoAtual) BEGIN 0;
4         else BEGIN BLOCO;
5         yylval.s = strdup(yytext);
6         return valor;
7     }
8 <VALOR>{num}
9     {
10        if(chaveBlocoAtual) BEGIN 0;
11        else BEGIN BLOCO;
12        if(yytext[0] == '+') yylval.s = strdup(yytext+1);
13        else yylval.s = strdup(yytext);
14        return valor;
15    }
16 <VALOR>[^ \t\n\r#]+
17     {
18        if(chaveBlocoAtual) BEGIN 0;
19        else BEGIN BLOCO;
20        char* text = malloc(strlen(yytext)+2);
21        sprintf(text, "\"%s\"", yytext);
22        yylval.s = strdup(text);
23        free(text);
24        return valor;
25    }
```

Listing 3: Condição VALOR

Nesta condição, o analisador léxico apanha:

- um valor nulo (null) ou booleano (true/false), o qual copia para a estrutura do Yacc tal como é filtrado;
- um número, inteiro ou decimal, e que pode ter sinais - também é copiado tal como é filtrado para a union, tirando, no máximo, o sinal positivo no início, se o houver, porque é redundante;
- caso não seja um dos acima, que são os únicos tipos que não precisam de estar protegidos por aspas em JSON, terá de colocar as aspas no texto apanhado - por exemplo, se apanhar as horas 23:12:05, guardará "23:12:05" na union do Yacc.

No fim, envia um token **valor** ao analisador sintático, que acaba desta maneira por reconhecer um símbolo não terminal **ChaveValor**. O programa retorna então para a condição **BLOCO** (ignore-se para já o if que determina para onde retornará, de facto, o programa - diz respeito a outra parte do trabalho).

Voltando a 2, caso o programa apanhe uma aspa, salta para a condição **ASPAS** e envia um token do caractere `'''` ao Yacc.

```

1 <ASPAS>\ "      {
2                 if(chaveComposta==1) {chaveComposta=0; BEGIN
   VALORBLOCO;}
3                 else {
4                     if(blocoAspas == 1) BEGIN BLOCO;
5                     if(blocoAspas == 2) BEGIN ARRAY;
6                     if(chaveBlocoAtual) BEGIN 0;
7                     blocoAspas = 0;
8                     return '""';
9                 }
10            }
11 <ASPAS>[^"]+    {
12                 yylval.s = strdup(yytext);
13                 if(chaveComposta==1) return chave;
14                 else return string;
15            }

```

Listing 4: Condição ASPAS

Aqui, o programa apanha todo o texto que encontrar até à próxima aspa e copia-o para a estrutura do Yacc, enviando de seguida um token **string**. A condição relativa à variável *chaveComposta* diz respeito a uma situação de TOML diferente da que está a ser explicada.

De seguida, apanha a segunda aspa, retornando o token do respetivo caractere, com o qual o analisador sintático acaba de reconhecer um símbolo não terminal **Aspas**. Por fim, volta para a condição de onde saltou para esta, analisando o valor da variável *blocoAspas* - neste caso, irá para a condição **BLOCO**.

O caso de 2 que falta explicar é o array, que é tratado na condição **ARRAY**. Sempre que entra num array, o programa Flex incrementa a variável *arrayAtual*, para saber dentro de quantos arrays está - uma vez que pode haver arrays dentro de arrays -, e envia o token '[' ao analisador sintático.

```

<ARRAY>,          {return ',';}
<ARRAY>\ "        {blocoAspas = 2; BEGIN ASPAS; return '"";}
<ARRAY>{nullbool} {yylval.s = strdup(yytext); return valor;}
<ARRAY>{num}      {
                   if(yytext[0] == '+') yylval.s = strdup(yytext+1);
                   else yylval.s = strdup(yytext);
                   return valor;
                   }
<ARRAY>[^,\\\"\\[\\] \\n\\t\\r]+
{
    char* text = malloc(strlen(yytext)+2); sprintf(text,"%s\\\"",yytext);
    yylval.s = strdup(text);
    free(text);
    return valor;
}
<ARRAY>\\[        {arrayAtual++; return '[';}
<ARRAY>\\]        {
                   decArrayAtual = 1;
                   if(arrayAtual-1 == 0) BEGIN BLOCO;
                   if(chaveBlocoAtual) BEGIN 0;
                   return ']' ;
                   }

```

Um array pode possuir valores de qualquer um dos tipos mencionados na condição **VALORBLOCO**. Como tal, é preciso fazer a mesma distinção nesta start condition, para saber o que se está à espera de apanhar.

Caso filtre uma aspa, envia o respetivo token ao analisador sintático e salta para a condição **ASPAS**, repetindo o raciocínio que foi explicado acima.

Se encontrar um ']', significa que está perante um array aninhado, pelo que incrementa a variável `arrayAtual` e retorna o respetivo token.

Caso contrário, será um valor não encapsulado por pontuação (por exemplo `null` ou `34.5`), pelo que copia o texto filtrado diretamente para a union do Yacc e retorna o token **valor**.

Um elemento de array é sempre seguido por uma vírgula, caso não seja o último, ou por um ']', caso seja. No primeiro caso, o programa envia o token ',' ao analisador sintático. No segundo, decrementa a variável `arrayAtual`, uma vez que termina este array (na verdade não a decrementa imediatamente, mas sim coloca a variável global `decArrayAtual` a 1 para o programa decrementar na iteração seguinte do `yylex()` - isto tem a ver com a indentação realizada no analisador sintático, que tem em conta a variável `arrayAtual`), e retorna o token ']', saltando, neste caso, de novo para a condição **BLOCO**. Com este último token, o Yacc acaba o reconhecimento de um símbolo não terminal **Array**.

Enquanto estiver na parte de declarações iniciais do ficheiro TOML, o programa reconhecerá sempre a seguinte expressão regular da condição **BLOCO**, que faz o mesmo que a primeira expressão regular apresentada nesta secção do relatório, que se encontra na condição geral do programa:

```
<BLOCO>{chave} {BEGIN VALORBLOCO; yy1val.s = strdup(yytext); return chave;}
```

No caso de encontrar parágrafos e linhas vazias, o programa Flex descarta-os, bem como fará a qualquer caractere que não seja reconhecido nas restantes expressões regulares do programa. Também descarta quaisquer comentários que encontre - em TOML, um comentário começa por # e dura até ao fim da linha -, visto que não têm relevância para a tradução para JSON:

```
<*>#.*      {}
<*>{espacos}\n {}
<*>.        {}
```

Uma vez filtradas todas as declarações iniciais de TOML, podem então surgir objetos no ficheiro - esta ordem é importante: uma declaração solta no início do ficheiro é considerada isolada, mas uma declaração solta depois de um objeto é considerada um atributo desse mesmo objeto. Em TOML, objetos seguem o seguinte formato:

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

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]
[servers.alpha]
ip = "10.0.0.1"
dc = "eqdc10"

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

Figura 5: Exemplo de objetos em TOML

Como é possível observar, um objeto possui uma etiqueta inicial - o seu nome encapsulado em parêntesis retos - seguida dos seus vários atributos, que são declarações de variáveis análogas às que podem aparecer no início do ficheiro.

A novidade aqui é que objetos podem possuir outros objetos dentro de si, à semelhança de arrays aninhados, como é o caso de `servers.alpha` e `servers.beta` na imagem acima. Nesse caso, o nome do "sub-objeto" começará obrigatoriamente pelo nome do primeiro objeto, seguido de `.'` e o nome do novo objeto (por exemplo, `servers.alpha`).

Para o correto processamento de objetos TOML, foram criadas duas estruturas e uma variável global no analisador léxico:

```
int tokens[3] = {0}; //FIMBLOCO, '[', chave
char *blocos[MAXBLOCO];
int blocoAtual = 0;
```

O inteiro **blocoAtual** indica dentro de quantos objetos o programa se encontra num dado momento. Nas declarações iniciais, estará a 0; quando entrar no primeiro objeto, será incrementado para 1; se esse mesmo objeto tiver outro objeto dentro de si, **blocoAtual** passará para 2; e assim sucessivamente.

O array de strings **blocos** guarda o nome dos objetos em que o programa se encontra: estará vazio nas declarações iniciais; quando o programa entrar no primeiro objeto, guardará o nome respetivo na primeira posição do array; se sair desse objeto e entrar para um novo, limpa o nome do objeto anterior do array e escreve o nome do novo na mesma posição; se entrar para um objeto dentro de outro, guarda o nome do sub-objeto na posição seguinte do array. Assim, mantém sempre um registo atualizado dos objetos em que se encontra.

O array de inteiros **tokens** funcionará como uma queue, para alturas em que seja preciso enviar vários tokens de uma vez, que tenham sido acumulados em alturas específicas de execução. A primeira posição do array diz respeito a tokens **FIMBLOCO**, a segunda a tokens `'[` e, por fim, os da terceira são tokens **chave**.

O funcionamento da queue será explicado detalhadamente mais à frente neste relatório, quando for abordado o processamento do fim de um objeto.

A leitura de objetos no programa Flex é despoletada pela seguinte expressão regular, que se encontra na start condition geral:

```
<*>{espacos}\n{espacos}/\[
{
  BEGIN NOMBLOCO;
  if (primeiroBloco == 0) {
    primeiroBloco = 1;
    return INITBLOCOS;
  }
  if (chaveBlocoAtual) {
    free(chaveBlocoAtual); chaveBlocoAtual = NULL;
    tokens[1] = 1;
    return FIMCHAVEBLOCO;
  }
}
```

Ao detetar a abertura de uma etiqueta de objeto ('[') - detetar porque o programa não consome o caractere, apenas verifica se existe com recurso a um lookahead - algures depois de um parágrafo (pode ou não estar indentada), o programa faz um par de verificações:

- se for o primeiro objeto do ficheiro, incrementa a variável `primeiroBloco`, que indica isso mesmo, e retorna o token **INITBLOCOS**;
- se não for, verifica se se encontra numa chave composta, um caso mais complicado de TOML que será explicado mais à frente;
- em qualquer dos casos, salta para a condição **NOMBLOCO**.

| | |
|--|--|
| <pre>1 <NOMBLOCO>\[2 <NOMBLOCO>[^\\]\[\\n]+ 3 4 tokensAcumulados(); 5 6 FIMBLOCO; 7 8 <NOMBLOCO>\] 9 <NOMBLOCO>{espacos}\n{espacos}/\[?</pre> | <pre>{tokens[1] = 1;} { atualizaBlocos(yytext); int token = if (token == 0) return if (token == 1) return '['; } {return ' '}; {BEGIN BLOCO; return FIMTAG;}</pre> |
|--|--|

Listing 5: Condição NOMBLOCO

Nesta condição, a primeira coisa que o programa faz é ler o caractere '['. Neste caso, em vez de retornar diretamente o respetivo token, itera o valor da segunda posição da queue que, como já foi explicado acima, diz respeito a esse tipo de token.

De seguida, o programa apanha o nome do objeto (segunda expressão regular) e invoca a função `atualizaBlocos` para o texto filtrado. Esta função atualiza a variável **blocoAtual** e o array de strings **blocos** de acordo com o novo objeto que está a começar a ler:

- no caso do primeiro objeto do programa, guarda o seu nome na primeira posição de *blocos* e incrementa *blocoAtual* para 1;
- se estivesse anteriormente num objeto que acabou de ler e está agora a entrar noutro no mesmo nível de "profundidade", *blocoAtual* não se altera, mas limpa-se o nome do anterior do array e guarda-se o nome do novo na mesma posição;
- se estivesse, por exemplo, num sub-objeto *servers.alpha* e for agora para um objeto *personnel*, o valor de *blocoAtual* diminuirá (neste caso, de 2 para 1), pelo que todos os valores antigos

guardados no array serão descartados e o nome do novo objeto será guardado na posição adequada.

No fim, esta função coloca em `tokens[0]` o número de tokens **FIMBLOCO** que é necessário mandar - igual ao número de objetos que fechou na transição - e incrementa o valor de `tokens[2]` para 1, pois é necessário mandar o token **chave** que diz respeito ao nome do novo objeto, filtrado na expressão regular.

A este ponto, já há múltiplos tokens à espera na fila, e de vários tipos - têm de ser enviados por uma ordem específica, de acordo com o que a GIC espera rececer. Para este efeito, desenvolveu-se a função *tokensAcumulados*:

```
int tokensAcumulados() {
    if (tokens[0] > 0) { //FIMBLOCO
        if (tokens[0] == 1) {tokens[0] = -1;}
        else {tokens[0]--;}
        return 0;
    }
    if (tokens[0] == -1 && tokens[1] > 0) { //' '
        tokens[1] = 0;
        return 1;
    }
    if (tokens[0] == -1 && tokens[2] > 0) { //chave
        tokens[2] = 0;
        return 2;
    }
    if (tokens[2] == -1) { //chave
        tokens[2] = 0;
        return 3;
    }
    tokens[0] = 0;
    return -1;
}
```

Esta função analisa o array *tokens* e indica o próximo token que deve ser enviado, se houver algum na fila, retornando um inteiro diferente consoante o token em questão.

- os tokens FIMBLOCO são todos enviados primeiro, pois a GIC espera fechar todos os objetos anteriores antes de abrir um novo;

- de seguida, é enviado o token ' ', que corresponde à abertura da etiqueta do novo objeto;

- por fim, é enviado o token chave, que diz respeito ao nome do novo objeto.

Se não houver nenhum token na fila, a função retorna -1, que não resultará no envio de um token; a última condição if diz respeito a outra parte do trabalho que será explicada mais à frente.

Voltando à condição 5, o programa enviará primeiro um FIMBLOCO ou um ' ', dependendo da transição de objetos que executou, pelo que analisa o valor retornado pela *tokensAcumulados* e envia o respetivo token.

Após isto, ainda sobram tokens na queue que é necessário enviar antes de continuar a filtrar o input, para o correto funcionamento da GIC. Ora, uma vez que a função *yylex()* é chamada de novo sempre que a mesma faz um *return* (isto até o programa Flex encontrar o EOF), a função *tokensAcumulados* foi colocada também no topo do bloco de expressões regulares do analisador léxico (indentada, para não ser interpretada como uma regra).

Sempre que *yylex()* é iniciada, corre esse código e, se houver tokens na fila, retorna o próximo, retirando-o da fila. Repete este processo até esvaziar a fila, prosseguindo depois para as expressões regulares:


```

%%
if (decArrayAtual == 1) {arrayAtual--; decArrayAtual = 0;}
if (blocoAtual > blocoDpsTokens) blocoAtual--;
if (blocoAtual < blocoDpsTokens) blocoAtual++;

int token = tokensAcumulados();
switch(token){
    case 0: return FIMBLOCO;
    case 1: return '[';
    case 2: {
        if (chaveBlocoAtual) yylval.s = strdup(chaveBlocoAtual);
        else yylval.s = strdup(blocos[blocoAtual-1]);
        return chave;
    }
    case 3: {
        yylval.s = strdup(chaveBlocoAtual);
        free(chaveBlocoAtual); chaveBlocoAtual = NULL;
        return chave;
    }
}

```

No caso do token *chave*, necessita também de copiar o valor do token para a union do analisador sintático, como se pode observar acima.

O primeiro bloco de condições if está relacionado com o processo de indentação do output na GIC, que tem em conta aquelas variáveis.

Voltando a 5 e continuando a linha de raciocínio que estava a ser seguida, neste ponto já só falta processar o `']'` que fecha a etiqueta do objeto, que é exatamente o que a terceira expressão regular faz, retornando o respetivo token.

Por fim, o programa salta para a condição **BLOCO** para tratar a linha seguinte, através da última expressão regular nesta condição, e retorna o token **FIMTAG** ao analisador sintático, com o qual este acaba de reconhecer um símbolo não-terminal **TagBloco**.

| | | |
|---|-----------------------------------|--|
| 1 | <code><BLOCO>\.</code> | <code>{return '.';}</code> |
| 2 | <code><BLOCO>\"</code> | <code>{chaveComposta = 1; BEGIN ASPAS;}</code> |
| 3 | <code><BLOCO>{chave}</code> | <code>{BEGIN VALORBLOCO; yylval.s = strdup(yytext);</code> |
| | <code>return chave;}</code> | |
| 4 | <code><BLOCO>\[</code> | <code>{tokens[1] = 1; BEGIN NOMBLOCO;}</code> |

Listing 6: Condição BLOCO

Uma vez nesta condição, pode aparecer uma de duas coisas (para além de comentários e linhas vazias): um atributo do objeto, ou seja, um par chave-valor com um caractere '=' entre eles, ou uma nova etiqueta de objeto, caso o primeiro atributo seja também um objeto.

No segundo caso, o programa apanha o caractere '[' de abertura da etiqueta do novo objeto, incrementando o número de tokens desse tipo na fila e saltando para a condição **NOMBLOCO**, para executar o raciocínio já explicado.

Caso contrário, o programa filtra o nome do atributo através da terceira expressão regular, copiando-o para a union do analisador sintático e enviando um token *chave* ao mesmo, saltando por fim para a condição **VALORBLOCO**, que já foi abordada acima, em 2.

As duas primeiras expressões regulares estão relacionadas com chaves compostas e serão explicadas mais à frente.

Desta maneira, o programa Flex acaba por processar todos os atributos do objeto atual, num ciclo entre as condições **BLOCO** e **VALORBLOCO** (e outras intermédias pelas quais pode passar).

Os objetos em TOML não têm nenhuma delimitação específica para o seu fim, pelo que, quando acabar de processar o objeto atual, o programa eventualmente (pode ter linhas vazias e comentários pelo meio) usará a seguinte expressão regular, na condição global (desde que não atinja o EOF):

```
<*>{espacos}\n{espacos}/\[      {
    BEGIN NOMBLOCO;
    if (primeiroBloco == 0) {
        primeiroBloco = 1;
        return INITBLOCOS;
    }
    if (chaveBlocoAtual) {
        free(chaveBlocoAtual);
        chaveBlocoAtual = NULL;
        tokens[1] = 1;
        return FIMCHAVEBLOCO;
    }
}
```

Esta expressão verifica a existência de um '[' - em lookahead, não o filtra - pelo que o programa sabe que está prestes a iniciar a filtragem de um novo objeto. Como tal, salta para a start condition **NOMBLOCO**, enviando o token **INITBLOCOS** ao analisador sintático caso seja o primeiro objeto encontrado. Uma vez nessa condição, segue o raciocínio já explicado acima.

O segundo if diz respeito a chaves compostas, que serão abordadas no fim desta secção.

Quando acabar de processar todos os objetos, o analisador léxico eventualmente chegará ao *End Of File*, que filtra com a seguinte expressão regular:

```
<*><<EOF>>      {
    if (blocoDpsTokens > 0) {
        tokens[0] = blocoDpsTokens-1;
        blocoDpsTokens = 0;
        return FIMBLOCO;
    }
    else if (primeiroBloco == 0) {
        primeiroBloco = 1;
        return INITBLOCOS;
    }
    else yyterminate();
}
```

Para terminar a execução, o analisador léxico verifica quantos tokens **FIMBLOCO** falta enviar, através da variável *blocoDpsTokens* - variável de controlo relacionada com as indentações do analisador sintático. Atualiza a posição respetiva da queue, retornando o primeiro token e fica em ciclo a esvaziar a queue, até voltar a esta condição.

Caso *primeiroBloco* esteja a 0, significa que não havia nenhum objeto no ficheiro TOML, pelo que apenas se retorna o token **INITBLOCOS**, com o qual o Yacc acaba de reconhecer um símbolo não-terminal **Toml**.

Uma vez enviados todos os tokens, o programa invoca a `yyterminate()` e termina a sua execução.

O último aspeto de TOML que falta abordar são os atributos com chaves compostas. Estes são atributos que se comportam como objetos, ou seja, o seu nome é constituído por duas strings, intercaladas por um ponto, com o resto da atribuição à frente. De seguida, é possível observar exemplos disto e a respetiva tradução para JSON:

```
name = "Orange"
physical.color = "orange"
physical.shape = "round"
site."google.com" = true

-----

{
  "name": "Orange",
  "physical": {
    "color": "orange",
    "shape": "round"
  },
  "site": {
    "google.com": true
  }
}
```

Figura 6: Atributos com chaves compostas e respetiva tradução para JSON

Como é possível observar, uma chave composta dá origem a um objeto cujo nome é a primeira parte da chave e que possui um atributo cujo nome é a segunda parte da chave e o valor é o que vem depois do '='.

Além disso, é necessário verificar se as linhas seguintes também possuem chaves compostas e, se sim, se dizem respeito ao mesmo objeto ou não - caso digam, inclui-se o novo atributo no mesmo, não se cria um objeto novo. Outro detalhe a ter em conta é que a segunda parte da chave pode vir entre aspas.

Para este efeito, foram criadas as variáveis globais **chaveComposta** e **chaveBlocoAtual**.

A primeira é incrementada quando o analisador léxico começa a processar a segunda metade de uma chave composta e encontra uma aspa - segunda expressão regular de 6. Neste caso, salta para a condição ASPAS (4), onde envia para o analisador sintático apenas um token *chave*, ao revés dos típicos tokens *string* e *'''* nessa condição, pois é assim que a GIC define a chave. Guia-se pela tal variável *chaveComposta* para executar este controlo.

A string **chaveBlocoAtual** mantém o nome do objeto de chave composta atual, e é limpa quando o programa acaba de processar o mesmo. Não foi considerada recursividade em chaves compostas, ou seja, chaves compostas com mais do que um '.'.

O analisador léxico inicia a filtragem de uma chave composta através da seguinte expressão regular, que apanha uma chave normal e deteta um ponto de seguida:

```
<*>{chave}/\.\s* {
    BEGIN BLOCO;
    if (chaveBlocoAtual) {
        if (strcmp(chaveBlocoAtual,yytext)!=0) {
            free(chaveBlocoAtual); chaveBlocoAtual = NULL;
            chaveBlocoAtual = strdup(yytext);
            tokens[2] = 1;
        }
    }
}
```

```
        tokens[0] = -1;
        return FIMCHAVEBLOCO;
    }
}
else chaveBlocoAtual = strdup(yytext);
yylval.s = strdup(yytext);
return chave;
}
```

O programa verifica se já se encontra no meio do processamento de um objeto de chave composta ou não. Se não for o caso, guarda o nome do objeto em *chaveBlocoAtual* e copia-o para a union do analisador sintático, enviado-lhe um token **chave**.

Caso contrário, tem de verificar se a nova chave composta diz respeito ao mesmo objeto ou não - se for um objeto diferente, atualiza o valor da *chaveBlocoAtual*, copia o valor para a union, adiciona um token *chave* à queue e retorna o token **FIMCHAVEBLOCO**, com o qual o analisador sintático termina de reconhecer o símbolo não-terminal **ChaveValor** correspondente ao objeto de chave composta anterior.

Depois de tudo isto, salta para 6, onde processa a segunda metade da chave, e acaba na condição 3, onde verifica *chaveBlocoAtual* para saber para que condição voltar no fim.

Quando apanha uma chave normal, depois de processar uma composta, limpa a variável *chaveBlocoAtual* e guarda lá o valor da nova chave filtrada temporariamente, que será removido depois ao limpar a queue, e retorna um token **FIMCHAVEBLOCO**, como pode ser observado em 1.

4 Codificação e Testes

4.1 Execução do Programa

O executável é criado pelo *Makefile* para facilidade de uso, mas opcionalmente o utilizador pode escolher fazê-lo sem utilizar estas *macros*.

O grupo decidiu atribuir o nome ao executável **toml2json** e maneira como é invocada é a seguinte:

- `./toml2json <file1>(<file2>)`

Em que o **file1** refere-se ao ficheiro com conteúdo *TOML* e o **file2** refere-se ao ficheiro final (recomenda-se que se utilize de imediato o formato *.json*) onde será colocada a tradução do **file1**. No entanto, este argumento é opcional e caso não esteja presente o ficheiro será criado por defeito com o nome completo do **file1** com a extensão *.json* no fim.

Por fim, o **file2** será criado e aparecerá na pasta em que se encontra o executável.

4.2 Alternativas, Decisões e Problemas de Implementação

Como já foi referido no capítulo anterior, a resolução do problema foi um processo iterativo, pelo que foram tidas em conta apenas uma percentagem das regras da linguagem *TOML*, embora seja uma percentagem elevada não deixa de ser parcial e, portanto, o grupo foi adicionando capacidades ao programa a cada iteração que fosse respeitando o máximo de regras possível.

Os melhoramentos mais significativos foram os seguintes, por ordem em que foram implementadas:

- Remoção dos comentários
- Bastante flexibilidade para os valores
- Indentação do ficheiro *TOML* arbitrário para qualquer elemento
- Hierarquização de secções
- Hierarquização de pares chave-valor
- Listas aninhadas
- Indentação legível no ficheiro *JSON*

O grupo decidiu tentar aprofundar ao máximo este projeto para garantir um trabalho mais coeso e complexo, pensando também na perspetiva do utilizador, que pretende algo funcional e simples de utilizar.

Cada um deles garantia os seus problemas de implementação já discutidos em capítulos anteriores, no entanto, foram implementados corretamente e testados, cujos resultados estão disponíveis na secção seguinte.

4.3 Testes realizados e Resultados

Foram elaborados vários *templates* com o intuito de testar todas as funcionalidades do programa e verificar que se encontrava totalmente operacional, dentro dos limites estabelecidos pelo grupo. Para cada um dos testes foi criado um ficheiro *TOML* e foi gerado o ficheiro *JSON* correspondente e, no final, é verificado para verificar a sua consistência com o texto original.

4.3.1 Teste 1

O primeiro teste foi essencialmente testar que secções simples (não hierárquicas) e pares chave-valor simples (com chaves não hierarquizadas) funcionavam corretamente. Este ficheiro é, por natureza, muito simples mas testa uma grande variedade de problemas que poderiam surgir.

O texto original *TOML* é:

```
title = "TOML Example"

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

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true
```

Em que resulta no *JSON*:

```
{
  "title": "TOML Example",
  "owner": {
    "name": "Tom Preston-Werner",
    "date": "2010-04-23",
    "time": "21:30:00"
  },
  "database": {
    "server": "192.168.1.1",
    "ports": [
      8001,
      8001,
      8002
    ],
    "connection_max": 5000,
    "enabled": true
  }
}
```

Como é possível visualizar, todas as indentações estão corretas e o texto resultante corresponde ao mesmo texto original, numa outra linguagem.

4.3.2 Teste 2

Para este teste o objetivo principal era testar a hierarquização dos vários elementos, a presença de listas e aumentar um pouco a complexidade.

O texto de entrada para o programa:

```
title = "TOML FILE"
```

```
[root]
size = 20
max = 10000000

[home]
dirs = 113
files = 18782631
permissions = "root"

[bin]
[bin.execute]
size = 100
files.names = [ "ls", "rm", "cat" ]
files.permissions = [ "rd", "rdrw", "rd" ]
files.owner = [ "root", "root", "root" ]
[bin.properties]
open = false
writable = true
readable = false
```

Como é possível ver o ficheiro resultante contém:

```
{
  "title": "TOML FILE",
  "root": {
    "size": 20,
    "max": 10000000
  },
  "home": {
    "dirs": 113,
    "files": 18782631,
    "permissions": "root"
  },
  "bin": {
    "execute": {
      "size": 100,
      "files": {
        "names": [
          "ls",
          "rm",
          "cat"
        ],
        "permissions": [
          "rd",
          "rdrw",
          "rd"
        ],
        "owner": [
          "root",
          "root",
          "root"
        ]
      }
    }
  }
}
```

```
    }  
  },  
  "properties": {  
    "open": false,  
    "writable": true,  
    "readable": false  
  }  
}  
}
```

Daqui se retira que o programa está a funcionar corretamente para os casos testados neste ficheiro.

4.3.3 Teste 3

O principal objetivo deste teste foi testar os comentários, indentações arbitrárias e hierarquias complexas, aumentando novamente a complexidade. O texto de entrada fornecido ao programa está disponibilizado de seguida.

```
title = "TOML Example"  
  
[owner]  
name = "Tom Preston-Werner"  
date = 2010-04-23  
time = 21:30:00  
    [database]  
server = "192.168.1.1"  
ports = [ 8001, 8001, 8002 ]  
connection_max = 5000  
enabled = true  
  
    [servers]  
[servers.alpha]  
    ip = "10.0.0.1"  
    dc = "eqdc10"  
[servers.alpha.x]  
    ipx = 1001  
    dcx = 1002  
  
[servers.alpha.y]  
    ipy = 2001  
    dcy = 2002  
  
[servers.beta]  
    ip = "10.0.0.2"  
    dc = "eqdc10"  
  
# Line breaks are OK when inside arrays  
hosts = [  
    "alpha",  
    "omega"  
]
```


O *JSON* resultante é:

```
{
  "title": "TOML Example",
  "owner": {
    "name": "Tom Preston-Werner",
    "date": "2010-04-23",
    "time": "21:30:00"
  },
  "database": {
    "server": "192.168.1.1",
    "ports": [
      8001,
      8001,
      8002
    ],
    "connection_max": 5000,
    "enabled": true
  },
  "servers": {
    "alpha": {
      "ip": "10.0.0.1",
      "dc": "eqdc10",
      "x": {
        "ipx": 1001,
        "dcx": 1002
      },
      "y": {
        "ipy": 2001,
        "dcy": 2002
      }
    },
    "beta": {
      "ip": "10.0.0.2",
      "dc": "eqdc10",
      "hosts": [
        "alpha",
        "omega"
      ]
    }
  }
}
```

Conclui-se que para estes casos também está a funcionar corretamente, com indentações coesas e o conteúdo é igual ao fornecido pelo texto original.

4.3.4 Teste 4

Para o penúltimo teste realizado, o grupo focou essencialmente nas indentações, listas aninhadas, vários tipos de valores distintos, filtragem de comentários e hierarquias complexas. Aqui foi necessário aumentar bastante a complexidade para ter um teste mais real, resultando no texto de entrada seguinte:



```
# This is a TOML document.

title = "TOML Example"

[owner]
  name = "Tom Preston-Werner"
dob =      1979-05-27T07:32:00-08:00 # First class dates
  birthday =      1979-05-23
  married =      false

[database]
server      =      "192.168.1.1"
ports =      [ 8001, 8001, 8002 ]
connection_max      =      5000
enabled      = true

[servers]
# Random Comment
# Indentation (tabs and/or spaces) is allowed but not required
[servers.alpha]
online = false
ip      = "10.0.0.1"
      domain.public =      false
domain.dns =      "dns.google.com"
      domain.owner      =      "N/A"
      dc = "abm2024"
connections = 128000
      files.limit =      100
files."available" =      true

[servers.beta]
online = true
ip = "10.0.0.2"

      domain.public =      true
      domain.owner      =      "N/A"
      dc = "abm2025"
connections = 12000
files.limit = 16
files."available" = true

[servers.gama]
online      =      false
ip = "10.0.0.3"

      domain.public =      false
domain.owner =      "N/A"
      dc = "abm2026"
connections = 6000
files.limit = 16
```

```
files."available" = true

[clients]
data =      [ ["Mars", "Hermes", "Athena"],      [16, 24, 16]      ]
requests.max = [ 16, 16, 16 ]
      timeout  =      1
cache.l1      = 32

      cache.l2 = 64

      cache.l3 =      128

cache.total =      [224, 224, 224]

# Line breaks are OK when inside arrays
hosts = [
  "Mars",
  "Hermes",
  "Athena"
]
```

Que resultou no *JSON*:

```
{
  "title": "TOML Example",
  "owner": {
    "name": "Tom Preston-Werner",
    "dob": "1979-05-27T07:32:00-08:00",
    "birthday": "1979-05-23",
    "married": false
  },
  "database": {
    "server": "192.168.1.1",
    "ports": [
      8001,
      8001,
      8002
    ],
    "connection_max": 5000,
    "enabled": true
  },
  "servers": {
    "alpha": {
      "online": false,
      "ip": "10.0.0.1",
      "domain": {
        "public": false,
        "dns": "dns.google.com",
        "owner": "N/A"
      }
    },
  },
}
```



```
    "dc": "abm2024",
    "connections": 128000,
    "files": {
      "limit": 100,
      "available": true
    }
  },
  "beta": {
    "online": true,
    "ip": "10.0.0.2",
    "domain": {
      "public": true,
      "owner": "N/A"
    },
    "dc": "abm2025",
    "connections": 12000,
    "files": {
      "limit": 16,
      "available": true
    }
  },
  "gama": {
    "online": false,
    "ip": "10.0.0.3",
    "domain": {
      "public": false,
      "owner": "N/A"
    },
    "dc": "abm2026",
    "connections": 6000,
    "files": {
      "limit": 16,
      "available": true
    }
  }
},
"clients": {
  "data": [
    [
      "Mars",
      "Hermes",
      "Athena"
    ],
    [
      16,
      24,
      16
    ]
  ],
  "requests": {
    "max": [
```

```
        16,  
        16,  
        16  
    ]  
  },  
  "timeout": 1,  
  "cache": {  
    "l1": 32,  
    "l2": 64,  
    "l3": 128,  
    "total": [  
      224,  
      224,  
      224  
    ]  
  },  
  "hosts": [  
    "Mars",  
    "Hermes",  
    "Athena"  
  ]  
}  
}
```

É possível verificar que filtra totalmente todos os espaçamentos/linhas novas e apenas retira todo o conteúdo necessário, que é o suposto. Concluimos, assim, que para estas condições o programa funciona corretamente.

4.3.5 Teste 5

Este teste foi o teste mais complexo realizado pelo grupo, que junta tudo o que foi testado anteriormente e eleva ainda mais a complexidade para garantir que tudo funciona corretamente. A principal dificuldade para o programa seria registrar corretamente as hierarquias, pelo que procedemos ao teste com o seguinte texto de entrada:

```
title = "Geometry Project"  
page.color = "rectangular"  
page.shape1 = "square"  
page.shape2 = "rectangular"  
page."shape3" = "square"  
autocorrect = false  
spacing = "14pt"  
  
[about]  
name = "Group 104"  
group.members = [ "Jack", "Joe", "John" ]  
group.registered = 2019-03-21  
group.theme = "Geometry"  
teacher = "ffs"  
  
[delivery]  
email = "ffs@fml.com"
```

```
aliases = [ ["ffS", "FFS", "fFs"] , ["fmL", "FML", "fMl"] ]
minimum_words = 1000
public = true
deadline = 2019-06-30

[assignment]
[assignment.pages]
number = 24
index = 2
max = 30
color = "default"
[assignment.pages.properties]
color = "rectangular"
shape1 = "square"
shape2 = "rectangular"
shape3 = "square"

[assignment.grading]
group.grades = [ 15, 15, 15 ]
group.satisfaction = [ 84, 81, 83 ]
teacher = "ffs"
group.nonepassed = false
group.allpassed = true
available = 2019-07-15

[assignment.grading.comments]
teacher = "Brilliant!"
students = "We could've done better!"

[message]
text = "Hello! We're sending the project!"
anexes = [ "project.pdf" , "project.m" ]
```

Que resulta no *JSON*:

```
{
  "title": "Geometry Project",
  "page": {
    "color": "rectangular",
    "shape1": "square",
    "shape2": "rectangular",
    "shape3": "square"
  },
  "autocorrect": false,
  "spacing": "14pt",
  "about": {
    "name": "Group 104",
    "group": {
      "members": [
        "Jack",
        "Joe",
        "John"
      ]
    }
  }
}
```



```
    ],
    "registered": "2019-03-21",
    "theme": "Geometry"
  },
  "teacher": "ffs"
},
"delivery": {
  "email": "ffs@fml.com",
  "aliases": [
    [
      "ffS",
      "FFS",
      "fFs"
    ],
    [
      "fmL",
      "FML",
      "fMl"
    ]
  ],
  "minimum_words": 1000,
  "public": true,
  "deadline": "2019-06-30"
},
"assignment": {
  "pages": {
    "number": 24,
    "index": 2,
    "max": 30,
    "color": "default",
    "properties": {
      "color": "rectangular",
      "shape1": "square",
      "shape2": "rectangular",
      "shape3": "square"
    }
  }
},
"grading": {
  "group": {
    "grades": [
      15,
      15,
      15
    ],
    "satisfaction": [
      84,
      81,
      83
    ]
  }
},
"teacher": "ffs",
```

```
    "group": {
      "nonepassed": false,
      "allpassed": true
    },
    "available": "2019-07-15",
    "comments": {
      "teacher": "Brilliant!",
      "students": "We could've done better!"
    }
  },
  "message": {
    "text": "Hello! We're sending the project!",
    "anexes": [
      "project.pdf",
      "project.m"
    ]
  }
}
```

Concluindo os testes, verifica-se que o programa realiza corretamente tudo o que foi proposto inicialmente e o faz de forma legível (bem indentada), o que o torna bastante eficiente no que toca a resultados.

5 Conclusão

O desenvolvimento deste projeto permitiu a consolidação da matéria teórico-prática lecionada nas aulas da unidade curricular e a elucidação da verdadeira utilidade e poder de ferramentas como os filtros *Flex*, expressões regulares e regras Condição-Ação, que permitem o fácil processamento e recolha de dados com base em padrões em ficheiros extensos, assim como do *Yacc*, um analisador sintático, para a geração de ações semânticas e geração de GIC robustas e coerentes.

O grupo está bastante satisfeito com o resultado final e considera que não só cumpriu os requisitos propostos no enunciado, como foi para além disso e desenvolveu uma solução mais escalável e dinâmica, que proporciona uma liberdade muito superior ao utilizador, permitindo-lhe gerar ficheiros *TOML* mais complexos e, conseqüentemente, uma melhor tradução para *JSON*.

Além disso, a flexibilidade do produto final também esteve sempre em mente, pelo que se procurou implementar o tratamento de erros e irregularidades no *TOML* onde fosse possível, de maneira a ter em consideração o fator do erro humano, que poderia levantar bastantes problemas e inutilizar a solução, avisando o utilizador onde se encontra o potencial erro.

Concluindo, foi possível obter conhecimentos valiosos através da elaboração deste trabalho prático que, de facto, já se provaram úteis em outros projetos alheios à unidade curricular, e desenvolver uma solução rica e compacta que abrange todos os requisitos propostos, facilitando o trabalho de qualquer pessoa que pretenda a tradução rápida e eficaz de ficheiros *TOML* para *JSON*.

A Código do Programa e Makefile

Lista-se a seguir o código da solução desenvolvida pelo grupo. Os comentários presentes nos ficheiros originais foram removidos e a disposição do código foi ligeiramente alterada a fim de tornar o código mais compacto e legível.

```
_____ toml2json.1 _____
%option noyywrap yylineno
%x BLOCO NOMBLOCO VALORBLOCO VALOR ARRAY ASPAS
%{
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include "y.tab.h"

#define MAXBLOCO 1024

int tokens[3] = {0}; //FIMBLOCO, '[', chave
char *blocos[MAXBLOCO];

int primeiroBloco = 0;
int blocoAtual = 0;
int blocoDpsTokens = 0;

int arrayAtual = 0;
int decArrayAtual = 0;

int blocoAspas = 0;
int chaveComposta = 0;
char* chaveBlocoAtual = NULL;

int tokensAcumulados();
void atualizaBlocos();

%}

digito      [0-9]
acentos     \xc3[\x80-\xbf]
letra       [a-zA-Z]|{acentos}
palavra     {letra}+
espacos     [ ]*
chave       [a-z][^ \. \n \t \r \[ \] \"]*

%%
    if (decArrayAtual == 1) {arrayAtual--; decArrayAtual = 0;}
    if (blocoAtual > blocoDpsTokens) blocoAtual--;
    if (blocoAtual < blocoDpsTokens) blocoAtual++;

    int token = tokensAcumulados();
    switch(token){
        case 0: return FIMBLOCO;
```

```

    case 1: return '[';
    case 2: {
        if (chaveBlocoAtual) yylval.s = strdup(chaveBlocoAtual);
        else yylval.s = strdup(blocos[blocoAtual-1]);
        return chave;
    }
    case 3: {
        yylval.s = strdup(chaveBlocoAtual);
        free(chaveBlocoAtual); chaveBlocoAtual = NULL;
        return chave;
    }
}

<*>#.*      {}

<NOMBLOCO>\[    {tokens[1] = 1;}
<NOMBLOCO>[^\\]\\[\n]+
    {
        atualizaBlocos(yytext);
        int token = tokensAcumulados();
        if (token == 0) return FIMBLOCO;
        if (token == 1) return '[';
    }
<NOMBLOCO>\]    {
    return ']';
}
<NOMBLOCO>{espacos}\n{espacos}/\[?
    {
        BEGIN BLOCO; return FIMTAG;
    }

<BLOCO>\.    {
    return '.';
}
<BLOCO>\"    {
    chaveComposta = 1;
    BEGIN ASPAS;
}
<BLOCO>{chave} {
    BEGIN VALORBLOCO;
    yylval.s = strdup(yytext);
    return chave;
}
<BLOCO>\[    {
    tokens[1] = 1;
    BEGIN NOMBLOCO;
}

<VALORBLOCO>{espacos}=
    {
        return '=';
    }

```

```

    }
<VALORBLOCO>{espacos}\\"
    {
        blocoAspas = 1;
        BEGIN ASPAS;
        return '';
    }
<VALORBLOCO>{espacos}\[
    {
        BEGIN ARRAY;
        arrayAtual++;
        return '[';
    }
<VALORBLOCO>{espacos}
    {
        BEGIN VALOR;
    }

<VALOR>(true)|(false)|(null)
    {
        if(chaveBlocoAtual) BEGIN 0;
        else BEGIN BLOCO;
        yylval.s = strdup(yytext);
        return valor;
    }
<VALOR>(\+|-)?[0-9]+(\.[0-9]+)?
    {
        if(chaveBlocoAtual) BEGIN 0;
        else BEGIN BLOCO;
        if(yytext[0] == '+') yylval.s = strdup(yytext+1);
        else yylval.s = strdup(yytext);
        return valor;
    }
<VALOR>[^ \t\n\r#]+
    {
        if(chaveBlocoAtual) BEGIN 0;
        else BEGIN BLOCO;
        char* text = malloc(strlen(yytext)+2);
        sprintf(text, "\"%s\"", yytext);
        yylval.s = strdup(text);
        free(text);
        return valor;
    }

<ARRAY>,
    {
        return ',';
    }
<ARRAY>\"
    {
        blocoAspas = 2;
        BEGIN ASPAS;
        return '';
    }

```

```

    }
<ARRAY>[^,\[\] \n\t\r]+
    {
        yylval.s = strdup(yytext);
        return valor;
    }
<ARRAY>\[
    {
        arrayAtual++;
        return '[';
    }
<ARRAY>\]
    {
        decArrayAtual = 1;
        if(arrayAtual-1 == 0) BEGIN BLOCO;
        if(chaveBlocoAtual) BEGIN 0;
        return ']';
    }

<ASPAS>\"
    {
        if(chaveComposta==1) {chaveComposta=0; BEGIN VALORBLOCO;}
        else {
            if(blocoAspas == 1) BEGIN BLOCO;
            if(blocoAspas == 2) BEGIN ARRAY;
            if(chaveBlocoAtual) BEGIN 0;
            blocoAspas = 0;
            return '"';
        }
    }
<ASPAS>[^"]+
    {
        yylval.s = strdup(yytext);
        if(chaveComposta==1) return chave;
        else return string;
    }

<*><<EOF>>
    {
        if (blocoDpsTokens > 0) {
            tokens[0] = blocoDpsTokens-1;
            blocoDpsTokens = 0;
            return FIMBLOCO;
        }
        else if (primeiroBloco == 0) {
            primeiroBloco = 1;
            return INITBLOCOS;
        }
        else yyterminate();
    }

<*>{espacos}\n{espacos}/\[
    {
        BEGIN NOMBLOCO;
        if (primeiroBloco == 0) {
            primeiroBloco = 1;

```

```

        return INITBLOCOS;
    }
    if (chaveBlocoAtual) {
        free(chaveBlocoAtual); chaveBlocoAtual = NULL;
        tokens[1] = 1;
        return FIMCHAVEBLOCO;
    }
}

<*>{chave}/\ . {
    BEGIN BLOCO;
    if (chaveBlocoAtual) {
        if (strcmp(chaveBlocoAtual,yytext)!=0) {
            free(chaveBlocoAtual); chaveBlocoAtual = NULL;
            chaveBlocoAtual = strdup(yytext);
            tokens[2] = 1;
            tokens[0] = -1;
            return FIMCHAVEBLOCO;
        }
    }
    else chaveBlocoAtual = strdup(yytext);
    yylval.s = strdup(yytext);
    return chave;
}

<*>{chave} {
    BEGIN VALORBLOCO;
    if(chaveBlocoAtual) {
        free(chaveBlocoAtual); chaveBlocoAtual = NULL;
        chaveBlocoAtual = strdup(yytext);
        tokens[2] = -1;
        return FIMCHAVEBLOCO;
    }
    yylval.s = strdup(yytext);
    return chave;
}

<*>{espacos}\n {}
<*>. {}

%%

int tokensAcumulados() {
    if (tokens[0] > 0) {
        if (tokens[0] == 1) {tokens[0] = -1;}
        else {tokens[0]--;}
        return 0;
    }
    if (tokens[0] == -1 && tokens[1] > 0) {
        tokens[1] = 0;
        return 1;
    }
    if (tokens[0] == -1 && tokens[2] > 0) {

```

```
    tokens[2] = 0;
    return 2;
}
if (tokens[2] == -1) {
    tokens[2] = 0;
    return 3;
}
tokens[0] = 0;
return -1;
}

void atualizaBlocos(char* yytext) {
    int subbloco = 0;
    int blocoInit = blocoAtual;

    if (blocoAtual > 0) {
        char blocoAcima[1024];
        for (int i = blocoAtual-1; i >= 0; i--) {
            sprintf(blocoAcima,"%s.",blocos[i]);
            char* pos = strstr(yytext, blocoAcima);

            if(pos != NULL) { //subbloco
                if (i == blocoAtual-1) {
                    blocos[blocoAtual] = strdup(pos+strlen(blocoAcima));
                    blocoDpsTokens = blocoAtual+1;
                }
                else {
                    for (int j = i+1; j < blocoAtual; j++){
                        free(blocos[j]);
                        blocos[j] = NULL;
                    }
                    blocos[i+1] = strdup(pos+strlen(blocoAcima));
                    blocoDpsTokens = i+2;
                }
                subbloco = 1;
                break;
            }
        }
    }

    if (subbloco == 0) {
        for (int i = 0; i < blocoAtual; i++){
            free(blocos[i]);
            blocos[i] = NULL;
        }
        blocos[0] = strdup(yytext);
        blocoDpsTokens = 1;
    }

    if (blocoInit >= blocoDpsTokens) tokens[0] = blocoInit-blocoDpsTokens+1;
    else tokens[0] = -1;
}
```

```
tokens[2] = 1;
}
```

toml2json.y

```
%{
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#define MAXBLOCO 1024

extern FILE * yyin;
extern int blocoAtual;
extern int arrayAtual;
FILE* yyout;

int yylex();
int yyerror(char* s);
char* indent(int iter);
%}

%token TITLE valor chave string INITBLOCOS FIMTITULO FIMTAG FIMBLOCO
FIMCHAVEBLOCO
%union{
    char* s;
}

%type <s> valor chave string
%type <s> Blocos Bloco ElsBloco ElemBloco TagBloco ChavesValores ChaveValor
ChaveBloco ElemChaveBloco Valor Array ElsArray Aspas

%%

Toml: ChavesValores INITBLOCOS Blocos
    {
        if(strlen($3) == 0) fprintf(yyout,"%s\n",$1);
        else if(strlen($1) == 0) fprintf(yyout,"%s\n",$3);
        else fprintf(yyout,"%s,%s\n",$1,$3);
    }
    ;

Blocos: Blocos Bloco
    {
        if(strlen($1) > 0) asprintf(&$$,"%s,\n%s",$1,$2);
        else asprintf(&$$,"\n%s",$2);
    }
    |   {$$ = "";}
    ;
```



```

Bloco: TagBloco FIMTAG ElsBloco FIMBLOCO
{
    char* indentacao = indent(blocoAtual);
    asprintf(&$$,"%s%s\n%s",$1,$3,indentacao);
}
;

ElsBloco: ElsBloco ElemBloco
{
    asprintf(&$$,"%s,\n%s",$1,$2);
}
| ElemBloco
{
    asprintf(&$$,"%s",$1);
}
;

ElemBloco: ChaveValor
{
    char* indentacao = indent(blocoAtual);
    asprintf(&$$,"%s",$1);
}
| Bloco
{
    asprintf(&$$,"%s",$1);
}
;

TagBloco: '[' chave ']'
{
    char* indentacao = indent(blocoAtual);
    asprintf(&$$,"%s\"%s\": {\n",indentacao,$2);
}
;

ChavesValores: ChavesValores ChaveValor
{
    if(strlen($1) > 0) asprintf(&$$,"%s,\n%s",$1,$2);
    else asprintf(&$$,"\n%s",$2);
}
| {$$ = "";}
;

ChaveValor: chave '=' Valor
{
    char* indentacao;
    if (blocoAtual==0) indentacao = indent(blocoAtual);
    else indentacao = indent(blocoAtual+1);
    asprintf(&$$,"%s\"%s\": %s",indentacao,$1,$3);
}
| chave '.' chave '=' Valor ChaveBloco FIMCHAVEBLOCO

```

```

{
char* ind = indent(blocoAtual);
char* ind1 = indent(blocoAtual+1);
char* ind2 = indent(blocoAtual+2);
if(strlen($6)==0 && blocoAtual==0)
    asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s\n%s}",ind,$1,ind1,$3,$5,
    ind);
if(strlen($6)==0 && blocoAtual!=0)
    asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s\n%s}",ind1,$1,ind2,$3,$5,
    ind1);
if(strlen($6)!=0 && blocoAtual==0)
    asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s,\n%s%s}",ind,$1,ind1,$3,$5,
    ind,$6);
if(strlen($6)!=0 && blocoAtual!=0)
    asprintf(&$$,"%s\"%s\": {\n%s\"%s\": %s,\n%s%s%s}",ind1,$1,ind2,$3,
    $5,ind2,$6,ind1);
}
;

```

ChaveBloco: ElemChaveBloco ChaveBloco

```

{
char* ind = indent(blocoAtual);
char* ind2 = indent(blocoAtual+2);
if (strlen($2)==0 && blocoAtual==0) asprintf(&$$,"%s%s\n%s",$1,$2,
ind);
if (strlen($2)!=0 && blocoAtual==0) asprintf(&$$,"%s,\n%s%s",$1,ind,
$2);
if (strlen($2)==0 && blocoAtual!=0) asprintf(&$$,"%s%s\n%s",$1,ind2,
$2);
if (strlen($2)!=0 && blocoAtual!=0) asprintf(&$$,"%s,\n%s%s",$1,ind2,
$2);
}
|   {$$ = "";}
;

```

ElemChaveBloco: chave '.' chave '=' Valor

```

{
char* indentacao = indent(blocoAtual);
if (blocoAtual==0) asprintf(&$$,"%s\"%s\": %s",indentacao,$3,$5);
else asprintf(&$$,"%s\"%s\": %s",$3,$5);
}
;

```

Valor: Aspas

```

{
asprintf(&$$,"%s",$1);
}
| valor
{
asprintf(&$$,"%s",$1);
}

```

```

    | Array
    {
        asprintf(&$$,"%s",$1);
    }
;

Array: '[' ElsArray ']'
{
    char* indentacao = indent(blocoAtual+arrayAtual);
    asprintf(&$$,"[\n%s\n%s]", $2, indentacao);
}
| '[' ']'
{
    asprintf(&$$,"[]");
}
;

ElsArray: ElsArray ',' Valor
{
    char* indentacao;
    if ($3[0] == '[') indentacao = indent(blocoAtual+arrayAtual);
    else indentacao = indent(blocoAtual+arrayAtual+1);
    asprintf(&$$,"%s,\n%s%s", $1, indentacao, $3);
}
| Valor
{
    char* indentacao;
    if ($1[0] == '[') indentacao = indent(blocoAtual+arrayAtual);
    else indentacao = indent(blocoAtual+arrayAtual+1);
    asprintf(&$$,"%s%s", indentacao, $1);
}
;

Aspas: '"' string '"'
{
    asprintf(&$$,"\"%s\"", $2);
}
| '"' '"'
{
    asprintf(&$$,"\"\"");
}
;

%%

int main(int argc, char* argv[]){
    yyin = fopen(argv[1], "r");

    if (argv[2]) yyout = fopen(argv[2], "w");
    else yyout = fopen(strcat(argv[1], ".json"), "w");
}

```

```

extern char *blocos[MAXBLOCO];
for (int i = 0; i < MAXBLOCO; i++)
    blocos[i] = NULL;

yyparse();
fclose(yyin);
fclose(yyout);
return 0;
}

int yyerror(char* s){
    extern int yylineno;
    extern char* yytext;
    fprintf(stderr, "Linha %d: %s (%s)\n",yylineno,s,yytext);
}

char* indent(int iter){
    char* indentacao = malloc(strlen(""));
    indentacao[0] = '\0';

    if(blocoAtual==0) iter++;
    for (int i = 0; i < iter; i++)
        strcat(indentacao," ");
    return indentacao;
}

```

De seguida está o ficheiro Makefile utilizado na resolução deste projeto.

Makefile

```

a=toml2json

$a : $a.y $a.l
    flex $a.l
    yacc -d $a.y
    cc -o $a y.tab.c lex.yy.c -lm

install: $a
    sudo cp $a /usr/local/bin/

clean:
    sudo rm -f $a
    sudo rm -f /usr/local/bin/$a
    sudo rm -f lex.yy.c y.tab.*

```

B *Exemplo de Ficheiros TOML e JSON*

```
title = "TOML Example"

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

[database]
server = "192.168.1.1"
ports = [ 8001, 8001, 8002 ]
connection_max = 5000
enabled = true

[servers]
[servers.alpha]
  ip = "10.0.0.1"
  dc = "eqdc10"

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

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

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    }
  ],
  "children": [],
  "spouse": null
}
```