Escola de Engenharia
Universidade do Minho

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Mestrado Integrado em Engenharia Informática
Processamento de Linguagens

Trabalho Prático

3º Exercício

Grupo

Lucas Pereira
A68547

Ricardo Pereira
A73577

Tiago Baptista
A75328

Braga, 10 de Junho de 2019

Conteúdo

1. Introdução

O processamento de linguagens é uma ferramenta extremamente importante em várias áreas da engenharia e, por esse motivo, dá nome a uma das unidades curriculares que integram o plano de estudos do *MIEI*.

Como é expectável, o mundo da programação é totalmente dependente de *softwares* e de várias linguagens de programação, pelo que se torna impreterível adquirir algum conhecimento e agilidade ao trabalhar com os mesmos. Entre os demais encontram-se, os sistemas operativos onde foi realizado o projeto, (*macOS* e/ou *Linux*) que são extremamente práticos e básicos, visto que no terminal se conseguem fazer todas as operações que são pedidas, isto se forem usados corretamente. As expressões regulares são também um "instrumento"que se irá aprimorar com este projeto prático. Quantas vezes não nos deparamos com situações em que queremos, por exemplo, num documento, eliminar pedaços de texto com determinadas características? Essa é uma situação bastante comum que se resolve facilmente se se tiver a noção de como construir estas expressões.

O terceiro trabalho prático envolve ainda o *YACC*, um gerador de analisador sintático, que constroi uma ferramenta que com a ajuda de vários símbolos léxicos e de várias regras gramaticais, permite a execução de várias instruções em linguagem *C* associadas a essas mesmas regras. A ideia é que com este gerador se possa construir algo que ajude a abstrair melhor determinados problemas, como por exemplo, traduzir um ficheiro em formato *YAML* para um ficheiro em formato *JSON*.

Dos seis exercícios que devem ser desenvolvidos, a fórmula $(NAlu\%6)+1$ indica o identificador daquele que está atribuido à equipa autora deste relatório, o quinto exercício ($(75328\%5)+1 = 5$). Essencialmente, o projeto consiste no exemplo dado anteriomente; a tradução de um ficheiro que se encontre no formato *YAML* no respetivo ficheiro em formato *JSON*.

2. Análise e Especificação

O projeto em questão visa a tradução de ficheiros de dados de um determinado tipo em ficheiros de dados de um tipo diferente. Sabe-se que existem várias formas de representar dados em ficheiros informáticos, nomeadamente, ficheiros do tipo *XML*, *JSON*, *YAML*, *CSV*, etc. Ora aquilo que aqui se prentende é que se crie um programa, que dado um ficheiro do tipo *YAML*, gere outro ficheiro, com a mesma informação, mas no formato *JSON*, cuja disposição dos dados é totalmente diferente.

Para o efeito, usa-se um analisador léxico que, aliado a um analisador sintático, permitirá a análise do ficheiro a traduzir e executará determinadas instruções que vão transpondo de forma válida para o novo ficheiro toda a informação presente no primeiro.

Assim, foi necessário analisar o ficheiro *YAML* para que se pudesse ter uma noção da forma como os dados estão dispostos no seu interior. Com a mesma intenção analisou-se também um ficheiro *JSON* que, como os restantes, é construído cumprindo determinadas regras e normas.

Posto isto, após uma análise exaustiva de ambos, conseguimos reunir para o ficheiro *YAML* as seguintes normas:

- **O elemento mais atómico é um conjunto de palavras**, sejam elas separadas ou não por determinados caratéres especiais. De agora em diante chamar-se-á **frase**;
- **As listas** são identificadas pelo seguinte padrão (pela ordem que se descreve):

  1. uma frase (identificador da lista);
  2. o caratére dois pontos e o caratére de mudança de linha;
  3. um número específico de espaços, um hífen e um espaço;
  4. uma/um frase/valor (um elemento da lista) e o caratére mudança de linha.

  O número de vezes que o terceiro e quarto pontos se repetirem, representará o número de elementos da lista.
- **Um valor** é um conjunto que pode conter qualquer uma das estruturas faladas nestes pontos e consegue-se distinguir através das identações, isto é, todas as estruturas que tenham o mesmo número de espaços depois de uma mudança de linha, pertencerão ao mesmo valor;
- **Um par chave/valor** associa uma chave a um valor, em que a chave é uma frase que identifica determinado valor. Estamos na presença de um par chave/valor quando temos (pela ordem que se descreve):

  1. uma frase (a chave);
  2. o caratére dois pontos e o caratére de mudança de linha;

3. um número específico de espaços, o caratére dois pontos e uma frase.

- Pode também acontecer o caso de termos uma chave e apenas uma frase associada, isto é, **um par chave/frase**, e aqui ver-se-á a chave, seguida do caratére dois pontos, do espaço e da respetiva frase.

Por outro lado, para o ficheiro *JSON* o número de espaços e as identações não terão qualquer significado sintático, pelo que reunimos as seguintes normas:

- **O elemento mais atómico é um conjunto de palavras**, sejam elas separadas ou não por determinados caratéres especiais. Sabe-se ainda que os carateres delimitadores são as aspas, há distinção dos tipos (inteiro, *string*, etc) e doravante esta estrutura chamar-se-á **frase**;
- **As listas** podem conter frases e/ou valores e são delimitadas por parenteses retos;
- **Um valor** pode conter qualquer uma das estruturas mencionadas nestes pontos e apresenta-se sempre com esse mesmo conteúdo separado por vírgulas e envolvido por duas chavetas;
- **Os pares chave/valor** surgem com uma frase a representar a chave, seguida de dois pontos e de um valor com um determinado conteúdo;
- Podem também ocorrer **pares chave/frase**, onde após uma chave (representada por uma frase) e do carater dois pontos, aparece uma frase.

3. Desenho e Implementação

3.1 Estruturas de dados utilizadas

Ao contrário dos dois trabalhos práticos anteriores, em que foram sempre usadas estruturas de dados, a intenção neste é fazer a leitura e imediata tradução daquilo que é lido. A verdade é que, estando a gramática bem construida, não é preciso utilizar qualquer estrutura para guardar os dados lidos, uma vez que os ficheiros irão ter os dados dispostos de forma similiar e apenas se fará uma substituição de determinados carateres.

3.2 Expressões regulares

A ferramenta *flex* é um ponto extremamente essencial e indispensável ao nosso projeto, visto que é uma ferramenta que nos permite pesquisar determinados conjuntos de carateres com um determinado padrão, tudo isto fazendo uso das expressões regulares. As expressões regulares permitem encontrar facilmente os padrões falados anteriormente, que depois são tratados pelo *yacc*.

Para a realização deste projeto, foram então usadas oito expressões regulares:

- ***COMMENT*** - Esta expressão regular representa um comentário no código-fonte, isto é, algo que deve ser ignorado no momento da compilação e é compatível com todos os conjuntos de carateres que apresentem a seguinte sequência de formação: a ocorrência de um cardinal seguida de nenhuma ou múltiplas ocorrências de uma letra minúscula, maiúscula, de um espaço e/ou de um algarismo.

```
COMMENT          [#][A-Za-z0-9 ]*
```

Figura 3.1: Expressão regular *COMMENT*.

- ***ATOM*** - Um padrão **ATOM** identifica qualquer parte do ficheiro que se apresente como aquilo a que se chamou frase na secção 2. Tendo em conta essa definição, uma expressão deste género deve conter os seguintes carateres (pela ordem que se refere):
  - uma ocorrência de uma letra maiúscula ou minúscula ou de um algarismo,
  - zero ou mais ocorrências de uma letra maiúscula ou minúscula, de um algarismo e/ou de um espaço,
  - novamente a ocorrência de uma letra maiúscula ou minúscula ou de um algarismo.

```
ATOM                [A-Za-z0-9][A-Za-z0-9 ]+[A-Za-z0-9]
```

Figura 3.2: Expressão regular *ATOM*.

- ***NEWLINE*** - O objetivo desta expressão regular é unicamente o reconhecimento do carátere que representa uma nova linha.

```
NEWLINE             [\n]
```

Figura 3.3: Expressão regular *NEWLINE*.

- ***COLON*** - Similar à anterior, o objetivo desta expressão regular é unicamente o reconhecimento do carátere dois pontos.

```
COLON               [:]
```

Figura 3.4: Expressão regular *COLON*.

- ***HY*** - A expressão regular **HY** reconhece um *hífen*.

```
HY                  [-]
```

Figura 3.5: Expressão regular *HY*.

Temos ainda duas expressões regulares auxiliares uma que identifica a ocorrência de três hífens e outra que identifica a ocorrência de um espaço.

```
[-][-][-]       {;}
[ ]             {;}
```

Figura 3.6: Expressões regulares auxiliares.

3.3 Gramática

Na gramática criada podem-se encontrar as definições sintáticas que foram definidas para o analisador. De acordo com as mesmas, à medida que se vão encontrando sequências de caráteres que coincidam com as expressões regulares especificadas anteriormente, vão (ou não) sendo executadas várias instruções em linguagem *C*. A ideia é que, à medida que se vai lendo o ficheiro a traduzir, se introduzam no novo ficheiro as sequências de caráteres lidos mas com pequenas alterações, pequenas substituições de determinados caráteres, como por exemplo, os espaços que denotam e distinguem os conteúdos dos vários valores que serão substituidos por chavetas (note-se que um ficheiro em *JSON* não tem em conta os espaços).

3.3.1 Símbolos não terminais

A função de um analisador é, com a ajuda do *flex*, reconhecer uma sequência de carateres não pela sua identidade (função do flex) mas pela forma como estão dispostos e é nesse sentido que surge a palavra símbolo, que geralmente denota um sequência de determinados caráteres com uma determinada disposição. Chamam-se símbolos não terminais uma vez que degeneram noutros símbolos terminais ou não terminais. Neste analisador existem os seguintes **símbolos não terminais**:

- *yaml*
- *object*
- *object1*
- *object2*
- *map*
- *mapnull*
- *array*
- *arrayelems*
- *arrayelem*

3.3.2 Símbolos terminais

Constrastando com os símbolos não terminais temos os símbolos terminais, que por ilação têm a função oposta à dos anteriores. Chamam-se símbolos terminais uma vez que não degeneram em qualquer outro símbolo terminal ou não terminal. Neste analisador existem os seguintes **símbolos terminais**:

- *COMMENT* - comentário em *YAML*;
- *ATOM* - frase em *YAML*;
- *NEWLINE* - caráter mudança de linha;
- *COLON* - caráter dois pontos;
- *HY* - caráter hífen.

Posto isto, em baixo explica-se cada um dos grupos de regras gramaticais definidos para que a tradução fosse feita da forma mais correta possível, sendo que para cada grupo de regras atribuir-se-á o mesmo nome que que se atribui ao seu símbolo esquerdo não terminal.

3.3.3 Grupo *yaml*

O grupo *yaml* representa a maioria das funções sintáticas que podem ocorrer na leitura do ficheiro *YAML*. Nomeadamente:

- o *COMMENT*;
- o *object*,
- o *array*;
- o *map*;
- o *mapnull*;
- o *NEWLINE*;
- e o vazio;

É de notar que para todos os símbolos não terminais do lado direito deste conjunto de produções é sempre feita a impressão do conteúdo desses símbolos que é antecedida pela impressão de um espaço e precedida pela impressão de uma mudança de linha. É com estas pequenas alterações das sequências de carateres que o programa vai traduzindo, símbolo a símbolo, o ficheiro *YAML* para um ficheiro *JSON*.

```
yaml: yaml COMMENT                  {;}
    | yaml object                   {fprintf(out,"  %s\n",$2);}
    | yaml array                    {fprintf(out,"  %s\n",$2);}
    | yaml map                      {fprintf(out,"  %s\n",$2);}
    | yaml mapnull                  {fprintf(out,"  %s\n",$2);}
    | yaml NEWLINE
    | /*empty*/
    ;
```

Figura 3.7: Grupo de produções *yaml*.

3.3.4 Grupo *object*

O grupo *object* representa o par "chave/valor"que se definiu na secção 2. Como se pode observar, quando é encontrada a produção *mapnull object1* é criada uma nova *string* com o conteúdo de *mapnull* e com o conteúdo de *object1*, sendo que este último é colocado entre chavetas, tal como aquilo que foi dito anteriormente acerca do "valor". Esta *string* é colocada no símbolo do lado esquerdo que irá preencher algo num nível acima.

```
object: mapnull object1            {sprintf($$,"%s  {\n%s\n  }",$1,$2);}
      ;
```

Figura 3.8: Grupo de produções *object*.

3.3.5 Grupo de produções *object1*

Nesta subsecção são abordadas as produções de outro símbolo não terminal, o *object1*. A primeira produção trata o caso em que existe um valor dentro de um valor, desdobrando *object1* em si mesmo e em *object2*. A segunda produção trata o caso mais simples de um valor com algo dentro que não é um valor, algo que falaremos a seguir. Na primeira, a *string* que passa para o nível superior tem o conteúdo de *object1* e de *object2* e na segunda a string tem só o conteúdo de *object2*.

```
object1: object1 object2           {sprintf($$,"%s\n%s",$1,$2);}
       | object2                   {sprintf($$,"%s",$1);}
```

Figura 3.9: Grupo de produções *object1*.

3.3.6 Grupo de produções *object2*

O grupo do símbolo não terminal *object2* trata os casos em que o valor que tem dentro de si é um *mapnull*, um *array* ou um um *map*. Não é feita qualquer ação em linguagem *C*, contudo, todos eles degeneram noutro símbolo não terminal.

```
object2: mapnull
       | array
       | map
       ;
```

Figura 3.10: Grupo de produções *object2*.

3.3.7 Grupo de produções *map*

Um dos símbolos não terminais usados nas produções da subsecção 3.3.3 e da subsecção 3.3.6 é o *map*. Este grupo tem apenas uma produção, toda ela constituida por símbolos terminais, e trata o caso "chave/frase"em que a seguir à chave se vêm os dois pontos, um espaço e uma frase. A tradução deste caso para *JSON* mantêm todos os caráteres, não alterando nada, pelo que a *string* que passa para o nível superior é a junção do conteúdo dos três símbolos terminais.

```
map: ATOM COLON ATOM                    {sprintf($$,"%s: %s",$1,$3);}
   ;
```

Figura 3.11: Grupo de produções *map*.

3.3.8 Grupo de produções *mapnull*

A produção associada a *mapnull* trata a situação em que associado à chave, está um valor nulo, que em *YAML* é representado pela ausência de caráteres a seguir ao caráter dois pontos (não é condição única), sendo em *JSON* representado pela palavra *null*. Ora, assim a *string* colocada no nível superior é uma concatenação da chave com o caráter dois pontos, um espaço e "*null*".

```
mapnull: ATOM COLON NEWLINE             {sprintf($$,"%s: null",$1);}
       ;
```

Figura 3.12: Grupo de produções *mapnull*.

3.3.9 Grupo de produções *array*

Os *arrays* são conhecidos por serem representados entre parênteses retos e em *JSON* a representação não é diferente. A primeira e única produção que está associada a este grupo engloba três símbolos terminais e um não terminal. Para esta situação, o nível superior da árvore irá arrecadar uma *string* com o conteúdo da chave seguida do carater dois pontos e de

uma *string* que é recebida do nível inferior por intermédio de *arrayelems* e que ficará disposta entre parênteses retos.

```
array: ATOM COLON NEWLINE arrayelems    {sprintf($$,"%s: [%s]",$1,$4);}
     ;
```

Figura 3.13: Grupo de produções *array*.

3.3.10 Grupo de produções *arrayelems*

Na subsecção anterior falou-se do símbolo não terminal *arrayelems* do qual iria emergir uma *string*. Essa *string* não é nada mais nada menos do que a concatenação de todos os elementos da lista que está a ser lida no ficheiro *YAML*. Com a primeira produção, o caso em tratamento pode ser qualquer um dos elementos da lista que não o final, sendo todos eles concatenados à *string* que passará para o nível superior. Com a segunda produção, faz-se o tratamento do elemento final de uma lista, degenerando em *arrayelem*, um caso que trataremos de seguida.

```
arrayelems: arrayelems arrayelem        {sprintf($$,"%s, %s",$1,$2);}
          | arrayelem
          ;
```

Figura 3.14: Grupo de produções *arrayelems*.

3.3.11 Grupo de produções *arrayelem*

Por último, um dos símbolos não terminais usados nas produções da subsecção *arrayelems* e consequentemente na *array* é a *arrayelem*, tal como referimos. Este grupo possui apenas uma produção constituída somente por símbolos terminais e trata de identificar um elemento de um *array*, que em *YAML* se define como um *hífen* seguido de um espaço, de uma frase e de um caráter de mundança de linha (’
n’). Assim, para a *stack* é enviado apenas o conteúdo de *ATOM* (a única informação relevante) que será colocada no *array* como um elemento.

```
arrayelem: HY ATOM NEWLINE      {$$ = $2;}
         ;
```

Figura 3.15: Grupo de produções *arrayelem*.

3.4 *Main*

Nesta parte do *yacc* é feita a chamada pricipal da gramática desenvolvida, é aberto um descritor de ficheiro para guardar o output do parser, é colocado o caráter de início e fim de ficheiro, é chamado o *yyparse* e por fim encerra-se o descritor de ficheiro.

```
int main(){
    out = fopen("out.json","w+");
    fprintf(out,"{\n");
    yyparse();
    fprintf(out,"}\n");
    fclose(out);
    return 0;
}
```

Figura 3.16: Main.

3.5 *Makefile*

A *makefile* elaborada permite agilizar o processo de utilização da solução desenvolvida. Para tal incluimos os comandos de processar o *flex* e o *yacc* assim como a compição do *y.tab.c* que é gerado pelo *yacc*.

Por outro lado, existe o comando *clean* de maneira a remover os ficheiros de output gerados pelos comandos acima descritos e também o ficheiro *JSON* resultado da execução do *run*.

```
make:
	flex flex.fl
	yacc -v yacc.y
	cc -o run y.tab.c

clean:
	rm y.tab.c lex.yy.c y.output run out.json
```

Figura 3.17: Makefile.

4. Demonstração

```
---
hello:world
#comentario teste yaml2json
hello:
  array1:
    -val1
    -val2
    -val5
  array2:
    -val3
    -val4
    -val6
#e necessario um comentario para separar as aguas xD
hello:world
```

Figura 4.1: Ficheiro *YAML* original.

```
rrpereira@ricardo:~/yaml2json$ make
flex flex.fl
yacc -v yacc.y
yacc.y:28.6: warning: empty rule for typed nonterminal, and no action [-Wother]
     | /*empty*/
       ^
yacc.y: warning: 2 shift/reduce conflicts [-Wconflicts-sr]
cc -o run y.tab.c
rrpereira@ricardo:~/yaml2json$ ./run < in.yaml
rrpereira@ricardo:~/yaml2json$
```

Figura 4.2: Comandos necessários para a execução.

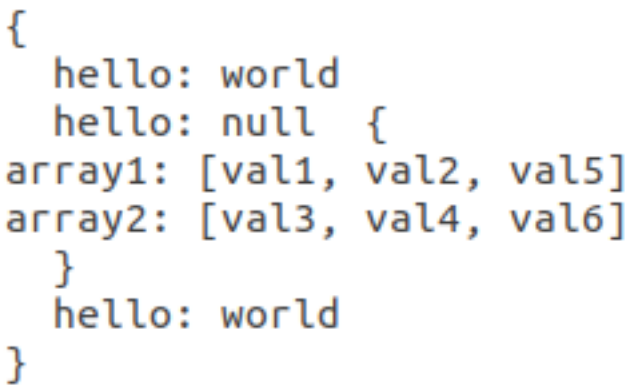

Figura 4.3: Ficheiro *JSON* de *output*.

5. Conclusão

5.1 Desfecho

A realização de um projeto prático numa unidade curricular é sempre benéfica. Ajuda a consolidar toda a matéria que é lecionada e é tão ou mais importante do que as aulas. A partir deste projeto começou-se a ter uma melhor perceção do impacto que o que se estuda tem em sistemas mais complexos.

No ínicio do presente relatório foi falado um exemplo onde o processamento de linguagens é extremamente útil. Na programação, sem nos apercebermos, usa-se muitas vezes esta ferramenta para substituir, por exemplo, o nome de determinadas variáveis, ou determinar onde está um erro que aparece no terminal onde se compila um programa. Imagine-se agora que o intuito é eliminar/encontrar todas as ocorrências de um determinado padrão num ficheiro na ordem das centenas de *megabytes*. Imagine-se ainda que não se dispõem de nenhuma ferramenta do tipo *finder* para encontrar esses padrões, visto que o ficheiro é tão volumoso que não consegue sequer ser aberto pelos editores de texto comuns. Como é que é feita a obtenção de todas as ocorrências? Sem dúvida que, nesta situação, um programa com expressões regulares que consiga fazer a análise do ficheiro é extremamente útil, não haveria outra opção.

Aliado ao conceito de expressão regular que nos é facultado pelo *flex*, temos o *yacc*. Este gerador de analisador sintático provou ser bastante útil na conceção de sistemas que operam sobre a sintaxe e a forma como está disposto o conteúdo de determinados ficheiros. A particularidade de poder selecionar cirurgicamente determinadas partes da informação que está a ser analisada e ao mesmo tempo permitir manipulá-la, faz do analisador gerado pelo *yacc* uma ferramenta extremamente poderosa que pode ser utilizada noutras valências, nomeadamente, análise de código e falhas de segurança, algo que é já uma realidade em várias organizações do setor empresarial.