Processamento de Linguagens

MIEI - 3º ano - 2º semestre
Universidade do Minho

Trabalho Prático nº3

Linguagem de Domínio Específico

Pedro Freitas
A80975

Francisco Freitas
A81580

10 de Junho de 2019

Conteúdo

1. *Contextualização*

Este relatório é o resultado do terceiro trabalho prático da Unidade Curricular Processamento de Linguagens. Este trabalho teve por base criar uma **Linguagem de Domínio Específico**(DSL). Para tal será preciso definir uma *gramática independente do contexto* assim como uma gramática tradutora, utilizando o par **flex/yacc** para gerar a compilação.

De todos os enunciados possíveis vamos trabalhar sobre o enunciado **Tradutor YAML – JSON**.

Ao longo do relatório vamos explicar o que nos foi pedido assim como as decisões e abordagens ao enunciado para podermos obter o resultado final pretendido.

2. *Tradutor YAML - JSON*

2.1 Enunciado

Construa um tradutor YAML-to-JSON (Y2J) entre estes 2 formatos de representação de informação tão em voga atualmente no seio das comunidades informáticas. Para implementar este tradutor terá de definir, através de uma GIC, o formato de entrada (linguagem Yaml) e depois de ter um parser/analisador-léxico para reconhecer textos-fonte escritos nessa notação deverá juntar Ações Semânticas às produções da gramática para obter uma GT que, com recurso ao Yacc, permita gerar automaticamente o tradutor.

2.2 Descrição do problema

Tal como foi referido anteriormente todo este trabalho baseia-se em criar uma Linguagem de Domínio Específico(DSL) e desenvolver processadores dessa mesma linguagem através do método da *tradução dirigida à sintaxe*,suportado numa *gramática tradutora*(GT). Para gerar compiladores será necessário ter um par flex/yacc.

No nosso caso temos de criar um tradutor entre a sintaxe de YAML para JSON. Assim teremos de definir uma gramática e um processador de texto de modo a suportar esta tradução. Para tal teremos de estudar um pouco tanto a sintaxe de YAML assim como a própria sintaxe de JSON.

2.3 YAML

YAML é um formato de codificação de dados legíveis por humanos e foi inspirado em linguagens como XML,C ou Python. Assim YAML tem uma sintaxe e um contexto muito característico. Ao longo desta secção vamos abordar algumas características deste formato que consideramos importantes.

Início um ficheiro YAML

Neste tipo de ficheiros é possível colocar um "padrão"identificativo de início do documento. Este padrão é de colocação facultativo sendo que quando optada teremos no início:

– – –

Comentários

Os comentários são possíveis neste formato sendo estes possíveis através do caractér '#' como podemos ver no exemplo seguinte:

```
# <- yaml supports comments
# yaml is very flexible
# yaml is nice!!
```

Listas

Para representar listas o YAML usa o hífen e um espaço:

```
---
# A list of tasty fruits
   - Apple
   - Orange
   - Strawberry
   - Mango
...
```

Atribuições Chave:Valor

Também é possível representar simples atribuições de "Chave"e "Valor":

```
#Key and Value
key: Value
```

Atribuições Chave:Valores

Da mesma forma que atribuímos um valor a uma chave também é possível atribuir vários valores:

```
#Key and Values
yaml:
    - slim and flexible
    - better for configuration
```

Dicionários

Chamamos dicionário a todos os valores que surgem no seguinte formato:

```
#An employee record
martin:
    name: Martin Developer
    job: Developer
    skill: Elite
```

Arrays

Sendo este um formato inspirado em várias linguagens de programação também suporta os tão famosos arrays:

```
#Arrys
array:
    - null_value:
    - boolean: true
    - integer: 1
```

2.4 JSON

Enquanto que YAML é um formato de codificação inspirado em XML, C e Python , JSON é uma derivação da notação de objetos de JavaScript.

Início e Fim de um ficheiro JSON

Tods os ficheiros JSON começam e acabam , respetivamente, pelos seguintes caracteres: { }

Chave-Valor

Tal como YAML em JSON também podemos atribuir um valor a uma determinada chabve sendo a sintaxe dessa ação dada por:

```
"key": "value"
```

Coleção

Um coleção em JSON podem ser representados começando por um '[' e terminando com um ']' :

```
"json": [
  "rigid",
  "better for data interchange"
]
```

2.4.1 Array

Um array é dado pela seguinte sintaxe:

```
"array": [
  {
  "json": "rigid"
  },
  {
  "json": "better for data interchange"
  }
]
```

Objetos

Neste tipo de ficheiros os objetos podem ser formados por vários tipos de elementos diferentes como lista, arrays, pares de chaves valores, etc. :

```
"object": {
    "key": "value",
    "array": [
    {
        "json": "rigid"
    },
    {
        "json": "better for data interchange"
    }
  ]
}
```

2.5 Decisões e implementação

Nesta secção vamos falar sobre as decisões tomadas e explicar o porquê das mesmas de forma a conseguirmos construir um tradutor entre YAML e JSON.

Para o podermos fazer tivemos de criar dois ficheiros: um em *flex* e outro em *yacc*.

2.5.1 Considerações Gerais

De realçar que todo este processo da construção do tradutor teve por base e inpiração oe exemplo dado pelo enunciado. Depois de uma pequena pesquisa apercebemo-nos do quão complexo pode ser a representação da informação quer em YAML quer em JSON. Por isso achamos por bem focarmo-nos no exemplo dado para demonstrar de forma mais clara como funciona o processo de tradução entre estes dois tipos de ficheiros.

2.5.2 yaml_to_json.l

Este é o ficheiro responsável por ler os ficheiros em formato yaml e processa-lo.

Devido às características e as funcionalidades do *flex* tivemos de criar estados que tinham comportamentos diferentes consoante as expressões regulares que apanhávamos. Para identificar os estados necessários assim como as expressões regulares que nos permitiam entrar nos mesmos tivemos de ter em conta a sintaxe do YAML, já anteriormente referida e exemplificada.

Tendo isto em conta achamos que seriam necessários nove estados.

YAML

O estado **YAML** é invocado quando lemos do ficheiro de texto a expressão — Apesar de anteriormente termos referido que os ficheiros neste formato podem começar por com este padrão ou não, para o nosso tradutor ser bem sucedido é necessário começar por essa expressão. Assim temos:

```
[-][-][-]       { BEGIN YAML; }
```

LISTAVALUE

Este estado é invocado sempre que estamos no estado YAML e encontramos a seguinte expressão regular:

```
^[a-zA-Z0-9][^:]+
```

, que representa sempre que encontramos alguma palavra escrita no início da linha.

Dentro deste estado podemos ter dois comportamentos diferentes. Se encontramos um representante de uma nova linha voltamos para o estado YAML ou então damos match com a expressão:

```
^[a-zA-Z0-9][^\n]+
```

e além de invocarmos o estado VETOR enviamos essa mesma string para a gramática. Um caso prático disto é:

```
exemplo: vetor
```

VETOR

O estado VETOR é invocado pelo estado LISTAVALUE sempre que encontra uma string depois de ":". Assim neste estado podemos ter três possibilidades diferentes: ou encontramos um **barra n** e enviamos para a gramática uma string apenas com um espaço, ou encontramos vários carateres exceto **barra n** e enviamos essa string para a gramática ou ainda encontramos um **barra n** seguidos de caracteres e retornamos ao estado YAML. Neste caso optamos por introduzir um caracter extra na expressão para não consumir o caracter lido após o **barra n**. As respeticas expressões regulares são dadas por:

```
<VETOR>[\n]   {(...)}
<VETOR>[^\n]* {(...)}
<VETOR>[\n]/[a-zA-Z0-9]     { BEGIN YAML;}
```

LISTA

Este estado é invocado pelo estado YAML apenas quando encontramos um '-' e no caso de uma flag identificativa de valores estiver a 0.

Dentro deste estado sempre que encontramos uma string enviamos para a gramática essa mesma string e voltamos para o estado YAML

ELEMENTOARRAY

Este estado apenas é invocado pelo estado YAML sempre que encontramos um '-' mas neste caso a flag identificatica de valores tem de estar a 1.

Dentro deste estado sempre que encontramos expressões que acabem em ':' enviamos essa expressão para a gramática e entramos no estado MAPPING:

```
<ELEMENTOARRAY>[a-zA-Z0-9][^:]+  {BEGIN MAPPING;(...);}
```

MAPPING

Invocado pelo estado ELEMENTOARRAY temos dois comportamentos. Se encontramos um **barra n** enviamos para a gramática a string "null"e retornamos para o estado YAML ou encontramos uma expressão que acabe em **barra n** e enviamos essa mesma expressão e retornamos também ao estado YAML.

VALUE

Estado invocado por YAML sempre que encontramos uma expressão que acabe em ':' porém esta expressão não tem de estar encostado à coluna mais à esquerda do ficheiro. Esta expressão é dada por:

```
<YAML>[a-zA-Z0-9][^:]+   {flag = 1;BEGIN VALUE;(...);}
```

Neste estado se encontrarmos um separador de nova linha voltamos para o estado YAML ou então encontramos texto e enviamos para a gramática.

PARAGRAFO

Podemos entrar neste estado através do estado YAML sempre que estamos presentes da seguinte expressão regular:

```
<YAML>^[a-zA-Z0-9]+[:][ ]+[>][ ]*[\n]
```

Aqui além de entramos no estado PARAGRAFO removemos todos os espaços da expressão e retiramos tanto os dois pontos como o sinal de maior e enviamos à gramática.

Já dentro deste estado enviamos à gramática sempre que encontramos texto até um **barra n**. Caso encontremos apenas um barra n , enviamos à gramática um string correspondente a um espaço vazio. Caso sejam dois seguidos enviamos uma string correspondente a um barra n para a gramática também. Também temos o caso de encontrarmos um barra n e depois um caracter. Neste caso enviamos o código de barra n para a gramática e voltamos ao estado YAML. Todas estas expressões regulares são dadas respetivamente por:

```
<PARAGRAFO>[a-zA-Z0-9][^\n]+ {(...)}
<PARAGRAFO>[\n]     {(...)}
<PARAGRAFO>[\n][\n]             {(...)}
<PARAGRAFO>[\n]/[a-zA-Z0-9]     {BEGIN YAML;(...)}
```

CONTENT

Este estado pode ser invocado pelo estado YAML de duas maneiras: ou encontramos um texto que termine com a expressão ": —- "ou apenas ": — ". Em ambos os casos voltamos a remover os espaços e os caracteres a partir dos dois pontos, inclusivé.

Dentro deste estado se encontrarmos uma string enviamos para a gramática. Se encontrarmos um barra n enviamos uma string à gramática com esse mesmo barra n. Se encontrarmos um barra n com um caracter na linha seguinte, não o consumimos e voltamos ao estado YAML.

2.5.3 yaml_to_json.y

Neste ficheiro vamos definir a nossa gramática para com a ajuda do flex já definido podermos concretizar a nossa tradução.

Para podermos receber dados do ficheiro flex definimos três variáveis:

```
%union{
    char* key;
    char* atributo;
    char* value;
}
```

Depois definimos os tokens necessários à gramática:

```
%union{
    %token  KEY ATR VAL ELEMENTOVALUE ELEMENTOKEY KEYINSIDE TEXTO PARAGRAPH CONT
}
```

Por último definimos os nossos tipos:

```
%type <key> KEY KEYINSIDE ELEMENTOKEY PARAGRAPH CONT
%type <atributo> b  ATR LISTAARRAY  ARRAYVALUES lista  OBJECT
                 CHAVESVALORESPAR KEY_LIST
%type <value> VAL  ELEMENTOVALUE AUX LISTA ELEMENTO  KEYVALUE TEXTO PARAGRAFO
              LISTATEXTO CONTEUDO
```

Estando isto definido vamos definir a gramática em si e os comportamentos a tomar para cada ação da linguagem.

con

É o nosso programa em si. É o responsável por imprimir a primeira e última chavestas e invocar a ação correspondente ao corpo do ficheiro.

```
conv :  CHAVESVALORESPAR   { printf("{\n\t%s \n}\n",$1);}
     ;
```

CHAVESVALORPAR

Representa o nosso ficheiro YAML na totalidade. Podemos dividir o ficheiro por secções que começam sempre por uma palara no inicio da linha que é a nossa KEY, sendo assim o CHAVESVALORPAR uma lista com todas essas secções:

```
CHAVESVALORESPAR:KEY_LIST                    {$$ =$1;}
                | CHAVESVALORESPAR KEY_LIST   {asprintf(&$$,"%s,\n\t%s",$1,$2);}
                | OBJECT                      {$$=$1;}
                | CHAVESVALORESPAR OBJECT     {asprintf(&$$,"%s,\n%s",$1,$2);}
                | PARAGRAFO                   {$$=$1;}
                | CHAVESVALORESPAR PARAGRAFO  {asprintf(&$$,"%s,\n%s",$1,$2);}
                | CONTEUDO                    {$$=$1;}
                | CHAVESVALORESPAR CONTEUDO   {asprintf(&$$,"%s,\n%s",$1,$2);}
                ;
```

KEY_LIST

Uma KEY_LIST é representado por uma KEY numa linha e depois nas linhas seguintes vir um hífem seguido de uma string que corresponde aos seus atributos. Um exemplo disso será:

```
json:
  - rigid
  - better for data interchange
```

Também pode ser representado por uma KEY seguido de texto nessa mesma linha e seguidas por outras linhas com pelo mesno um espaço e mais texto. Um exemplo disto será:

```
json: example
  - rigid
  - better for data interchange
```

A gramática correspondente será:

```
KEY_LIST : KEY lista {asprintf(&$$,"\"%s\": [\n\t%s \n\t]",$1,$2);}
         | KEY LISTATEXTO {asprintf(&$$,"\"%s\" : \t\"%s\"",$1,$2);}
         ;
```

, onde a lista é uma lista de b (que é do tipo atributo e consequentemente do tipo char*):

```
lista : b {$$=$1;}
      | lista b  {asprintf(&$$,"\t\"%s\",\n \t\t\"%s\"",$1,$2);}
      ;
```

e LISTATEXTO é de facto uma lista de tokens TEXTO que tem o tipo value que também é um char* :

```
LISTATEXTO: TEXTO        {asprintf(&$$,"%s",$1);}
          | LISTATEXTO TEXTO {asprintf(&$$,"%s%s",$1,$2);}
          ;
```

OBJECT

O OBJECT tem uma KEY que é uma palavra que começa a linha e depois tem uma LISTA. Isto acontece porque dentro do OBJECT pode existir arrays ou pares key:value então definimos um tipo AUX em que este representa um array ou um par key:value sendo que a LISTA representa uma lista de AUX podemos ter vários arrays e varios key:value dentro do OBJECT.

```
OBJECT: KEY LISTA {asprintf(&$$,"\t\"%s\": {\n\t%s\n\t}",$1,$2);}
      ;

LISTA: AUX          {asprintf(&$$,"%s",$1);}
    | LISTA AUX   {asprintf(&$$,"%s,\n\t%s",$1,$2);};
    ;

AUX:KEYVALUE        {asprintf(&$$,"%s",$1);  /* */ }
   |LISTAARRAY           {asprintf(&$$,"%s",$1);}
   ;

LISTAARRAY:KEYINSIDE ARRAYVALUES {asprintf(&$$,"\t\"%s\": [\n\t%s\n\t\t]",$1,$2);}
          ;


ARRAYVALUES: ELEMENTO {asprintf(&$$,"%s",$1);}
          | ARRAYVALUES ELEMENTO {asprintf(&$$,"%s,\n\t%s",$1,$2);}
          ;
```

```
ELEMENTO: ELEMENTOKEY ELEMENTOVALUE {asprintf(&$$,
                                     "\t  {\n\t\t\t  \"%s\" \"%s\"\n\t\t}",
                                     $1,$2);}
        ;

KEYVALUE: KEYINSIDE VAL  {asprintf(&$$,"\t\"%s\": \"%s\"",$1,$2);}
        ;
```

O LISTARRAY representa um array dentro de um OBJECT em que KEYINSIDE vem do flex e é o nome dado à chave e o ARRAYVALUES é uma lista com o que esta dentro do array. O ELEMENTO representa cada par key:value dentro do array onde ELEMENTOKEY e ELEMENTOVALUE vem do flex.

PARAGRAFO e CONTEUDO

Por último temos o PARAGRAFO e o CONTEUDO cuja sintaxe é igual, porém tem comportamentos diferentes na tradução.

```
PARAGRAFO: PARAGRAPH LISTATEXTO {asprintf(&$$,"\t\"%s\": \"%s\"",$1,$2);}
        ;
CONTEUDO: CONT LISTATEXTO {asprintf(&$$,"\t\"%s\": \"%s\"",$1,$2);}
        ;
```

2.6 Como obter os resultados

Para compilar e correr o programa basta fazer dois passos.

Primeiro temos de abrir o terminal na diretoria onde estão todos os ficheiros e escrever

```
make
```

Depois de termos compilado para correr o programa precisamos de escrever:

```
./yaml_to_json < YOURPATH/YOUREXAMPLE
```

, onde **YOURPATH** é o caminho para a diretoria onde estará o ficheiro a ser processado e **YOUREXAMPLE** é o nome do ficheiro a ser processado.

Para limpar os executáveis e os ficheiros criados após a compilação basta escrever:

```
make clean
```

2.7 Resultados Obtidos

Nesta secção vamos apresentar o ficheiro de teste (ficheiro yaml) e o ficheiro resultante da tradução.

YAML

```
---
# <- yaml supports comments, json does not
# did you know you can embed json in yaml?
# try uncommenting the next line
# { foo: 'bar' }

json:
  - rigid
  - better for data interchange
yaml:
  - slim and flexible
  - better for configuration
object:
key: value
  array:
    - null_value:
    - boolean: true
    - integer: 1
paragraph: >
   Blank lines denote

   paragraph breaks
content: |-
   Or we
   can auto
   convert line breaks
   to save space
```

JSON

```
{
    "json": [
        "rigid",
        "better for data interchange"
    ],
    "yaml": [
        "slim and flexible",
        "better for configuration"
    ],
    "object": {
        "key": "value",
        "array": [
          {
            "null_value": "null"
          },
          {
```

```
            "boolean": "true"
          },
          {
            "integer": "1"
          }
        ]
    },
    "paragraph": "Blank lines denote\nparagraph breaks\n",
    "content": "Or we\n can auto\n convert line breaks\n to save space\n "
}
```

3. *Apreciação Crítica*

Durante a realização deste terceiro e último trabalho prático desta unidade curricular várias foram as etapas realizadas de forma a obtermos um resultado final fidedigno e consistente.

Tendo em conta as metas deste trabalho e os objetivos propostos no enunciado, achamos que obtivemos um resultado bastante positivo e satisfatório. Estamos também bastante satisfeitos com todos os conteúdos abordados nas aulas desta unidade curricular assim como todos os temas dos trabalhos práticos. Todos estes foram igualmente desafiantes e enriquecidores no que toca a Expressões Regulares, processar ficheiros e definir gramáticas. Como grupo saímos daqui satisfeitos e conscientes que pusemos em prática os assuntos abordados corretamente e que fomos rigorosos connosco mesmos.