Universidade do Minho

Mestrado Integrado em Engenharia Informática

3º ANO, 2º Semestre, 2018/2019

Processamento de Linguagens

Trabalho Prático 3 - YACC

Ana Pereira
A81712

Maria Dias
A81611

June 10, 2019

Índice

1 Introdução

No âmbito da Unidade Curricular de Processamento de Linguagens, foi proposto o desenvolvimento de uma gramática e um reconhecedor léxico e sintático para uma dada linguagem, com recurso ao par de ferramentas geradoras **Flex/Yacc**.

Este trabalho surge, assim, como oportunidade de aplicar conhecimentos adquiridos na unidade curricular, tais como a utilização de ferramentas como Yacc e Flex, bem como o estudo aprofundado de gramáticas, expressões regulares e analisadores léxicos e sintáticos.

Neste relatório são expostos o enunciado e descrição do problema, juntamente com todas as decisões tomadas pelo grupo para o desenho e implementação da solução do mesmo. O presente documento inclui também diversos exemplos de utilização do programa desenvolvido.

De acordo com o critério de atribuição de enunciados, coube ao nosso grupo resolver o enunciado número 5, **"Tradutor YAML - JSON"**.

2 Especificação do Problema

2.1 Enunciado - Tradutor YAML-JSON

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. Observe para isso os exemplos a seguir mas procure as definições dos formatos em causa.

2.2 Descrição do Problema

Para conseguirmos perceber o contexto do problema, devemos primeiro esclarecer alguns conceitos importantes. Em primeiro lugar, interessa saber a definição de *Linguagem de Domínio Específico*, uma vez que é o que nos propomos a construir. Depois, é necessário perceber o conceito de *Gramática Independente de Contexto*, pois será também preciso definir uma.

Tendo estes conceitos aprendidos, a implementação do tradutor baseia-se na criação de uma GIC e de um analisador léxico que consigam interpretar textos na linguagem YAML e traduzi-los para o formato JSON.

2.2.1 Linguagem de Domínio Específico

Uma Linguagem de Domínio Específico (*Domain-Specific Language* - DSL) são os paradigmas e funções, ou códigos específicos, de uma linguagem de programação ou linguagem de especificação em desenvolvimento de software e engenharia de domínio, dedicada a um domínio de problema particular.

Exemplos de linguagem de domínio específico incluem gramáticas YACC para criação de analisadores sintáticos e expressões regulares para analisadores léxicos. O oposto de uma DSL seria uma linguagem de programação de propósito geral, como C, Java ou uma linguagem de modelagem de propósito geral, como UML.

A linguagem que nos propomos a implementar é deste tipo, sendo baseada numa gramática YACC e num analisador léxico em Flex.

2.2.2 Gramática Independente de Contexto

Para o problema em questão, é necessário uma linguagem que permita carregar os dados a processar. Estes dados são obtidos a partir de um ficheiro `*.yaml`.

Como tal, recorremos a uma Gramática Independente de Contexto, GIC, que especifique a linguagem pretendida. Uma GIC G é um quádruplo G = (T, N, S, P), em que:

- T é o conjunto finito não vazio de símbolos terminais;

- N é o conjunto finito não vazio de símbolos não-terminais;
- $S \in N$ é o símbolo inicial, também designado por axioma da gramática;
- $P \subseteq N \times (N \cup T)^*$ é o conjunto finito não vazio de produções ou regras de derivação.

3 Desenho e Implementação da Solução

Antes da implementação da solução, é necessária uma análise dos formatos YAML e JSON. Na linguagem YAML, uma linha começada pelo caractere '#' é um comentário. Em JSON, não existem comentários. Logo, na tradução Y2J, estas linhas devem ser ignoradas.

YAML	JSON
`key: value`	`"key": "value"`

`yaml:`
`  - slim and flexible`
`  - better for configuration`
	`"yaml": [`
	`  "slim and flexible",`
	`  "better for configuration"`
	`]`

`object:`
`  key: value`
`  array:`
`  - null_value:`
`  - boolean: true`
`  - integer: 1`
	`"object": {`
	`  "key": "value",`
	`  "array": [`
	`  {`
	`  "null_value": null`
	`  },`
	`  {`
	`  "boolean": true`
	`  },`
	`  {`
	`  "integer": 1`
	`  }`
	`  ]`
	`  }`

`paragraph: >`
`  Blank lines denote`
 	
`  paragraph breaks`
	`"paragraph": "Blank lines denote\nparagraph breaks\n"`

`content: |-`
`  Or we`
`  can auto`
`  convert line breaks`
`  to save space`
	`"content": "Or we\ncan auto\nconvert line breaks\nto save space"`

3.1 Desenho da Gramática

Depois de uma análise atenta ao formato Yaml, identificamos os padrões que foram expostos na secção anterior.

Para começar a desenhar a gramática, em primeiro lugar, constatámos que um documento Yaml é começado pela sequência de 3 caracteres "-" e, de seguida, surge um conjunto de pares chave/valor. Este conjunto podia constituir de apenas um par chave/valor ou vários, tal como podemos ver na segunda regra da gramática.

```
Programa : START KeyValuePairs     {FILE* file = fopen("Y2J.json","w+");
                                    fprintf(file,"{\n%s\n}\n",$2);}
         ;

KeyValuePairs : KeyValuePair                 {asprintf(&$$, "  %s",$1);}
              | KeyValuePairs KeyValuePair   {asprintf(&$$, "%s,\n  %s",
                                              $1,$2);}
              ;
```

Por sua vez, um par chave/valor pode ser definido de várias formas. Podemos ter:

⇒ `KEY List` em que o valor é uma lista;

⇒ `KEY Object` em que o valor é um objeto, ou seja, uma coleção "aninhada" de pares chave/valor;

⇒ `PARAGRAPH Block` em que o valor é um bloco de texto em que as linhas em branco representam uma quebra de linha;

⇒ `CONTENT contBlock` em que o valor é um bloco de texto em que as quebras de linha são representadas normalmente;

⇒ `KEYVALUE` em que tanto a chave como o valor são texto;

```
KeyValuePair : KEY List     {asprintf(&$$, "\"%s\": [\n    %s\n  ]",$1,$2);}
    | KEY Object            {asprintf(&$$, "\"%s\":{\n  %s\n  }",$1,$2);}
    | PARAGRAPH Block       {asprintf(&$$, "\"%s\": \"%s\"",$1,$2);}
    | CONTENT contBlock     {asprintf(&$$, "\"%s\": \"%s\"",$1,$2);}
    | KEYVALUE              {char* tokens = strtok ($1,": ");
                             char* values[2];
                             values[0] = strdup(tokens);
                             tokens = strtok (NULL, ": ");
                             values[1] = strdup(tokens);
                             asprintf(&$$, "\"%s\": \"%s\"",
                             values[0],values[1]);}
    ;
```

Para o caso em que temos um par chave/valor em que o valor é uma lista, esta define-se da forma que podemos ver a seguir. A lista pode ter apenas um elemento ou ser uma sequência de elementos.

```
List : list         {asprintf(&$$, "\"%s\"",$1);}
     | List list    {asprintf(&$$, "%s\n    \"%s\"",$1,$2);}
     ;
```

No que toca a um valor do tipo objeto, este pode conter vários tipos de conteúdo e, tal como vimos na lista, pode ter apenas um elemento ou vários. Estes elementos podem ser em forma de array ou apenas em forma de correspondência chave/valor, em que ambos os valores consistem em texto.

```
Object : KeyValue          {asprintf(&$$, "  %s",$1);}
       | Object KeyValue   {asprintf(&$$, "%s,\n  %s",$1,$2);}
       ;

KeyValue : OBJECTKEY Array  {asprintf(&$$, "  \"%s\": [\n%s\n    ]",$1,$2);}
       | OBJECTKEY OBJECTVALUE   {asprintf(&$$, "\"%s\": \"%s\"",$1,$2);}
       ;
```

Um elemento em forma de array é definido por uma ou várias entradas, que são da forma "chave: valor", sendo a chave identificada pelo token `ARRAYKEY` e o valor identificado pelo token `ARRAYVALUE`.

```
Array : ARRAYKEY ARRAYVALUE        {asprintf(&$$, "     {\n    \"%s\": %s
                                    \n}",$1,$2);}
      | Array ARRAYKEY ARRAYVALUE  {asprintf(&$$, "%s,\n     {\n    \"%s\"
                                    : %s\n     }",$1,$2,$3);}
      ;
```

Um par chave/valor onde o valor representa um bloco de texto, pode apresentar dois tipos de formatação para esse texto, que tal como acontece para as listas e objetos, consiste numa linha ou várias linhas de texto, sendo cada linha identificada pelos tokens `blockline` e `contLine`.

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

contBlock : contLine            {asprintf(&$$, "%s",$1);}
          | contBlock contLine  {asprintf(&$$, "%s\\n%s",$1,$2);}
          ;
```

O primeiro tipo de formatação, referente ao `Block`, admite linhas em branco, indicativas de novos parágrafos. Enquanto que no segundo tipo, `contBlock`, as quebras de linha serão no ficheiro final representadas pelo caractere de final de linha.

3.2 Especificação da GIC

Conforme a definição de GIC apresentada na secção da especificação do problema e as regras que definimos para a nossa gramática, temos que a GIC que especifica o nosso tradutor Y2J é a seguinte:

```
T = { START, KEY, KEYVALUE, PARAGRAPH, CONTENT, OBJECTKEY, OBJECTVALUE,
      ARRAYKEY, ARRAYVALUE, list, blockline, contLine }

N = { Programa, KeyValuePairs, KeyValuePair, Object, KeyValue, Array,
      List, Block, contBlock }

S = Programa

P = {
    p0: Programa -> START KeyValuePairs

    p1: KeyValuePairs: KeyValuePair
    p2:                | KeyValuePairs KeyValuePair

    p3: KeyValuePair: KEY List
    p4:               | KEY Object
    p5:               | PARAGRAPH Block
    p6:               | CONTENT contBlock
    p7:               | KEYVALUE

    p8: Object: KeyValue
    p9:       | Object KeyValue

    p10: KeyValue: OBJECTKEY Array
    p11:         | OBJECTKEY OBJECTVALUE

    p12: Array: ARRAYKEY ARRAYVALUE
    p13:      | Array ARRAYKEY ARRAYVALUE

    p14: List: list
    p15:     | List list

    p16: Block: blockline
    p17:      | Block blockline

    p18: contBlock: contLine
    p19:          | contBlock contLine
    }
```

3.3 Especificação Flex

```
1  "---"                          { return START; }
2  ^[#][^\n]*                     { ; }
3  ^[A-Za-z]*":"                  { yylval.key = strdup(yytext);
4                                    yylval.key[yyleng - 1] = '\0';
5                                    flag = 0; return KEY;
6                                  }
7  ^[A-Za-z]*":"[^\n]*            { yylval.key = strdup(yytext);
8                                    return KEYVALUE;
9                                  }
10 [-]                            { if(flag==0) BEGIN LIST;
11                                   else BEGIN ARR;
12                                 }
13 [a-zA-Z0-9 ]+":"               { flag=1; BEGIN OBJ;
14                                   yylval.str=strdup(yytext);
15                                   yylval.str[yyleng - 1] = '\0';
16                                   return OBJECTKEY;
17                                 }
18 <OBJ>[\n]                       { BEGIN INITIAL;}
19 <OBJ>[a-zA-Z0-9][^\n]*          { BEGIN INITIAL;
20                                   yylval.str=strdup(yytext);
21                                   return OBJECTVALUE;
22                                 }
23 <LIST>[^\n]*                    { yylval.str = strdup(yytext+1);
24                                   BEGIN INITIAL; return list;
25                                 }
26 <ARR>[^\n]*":"                  { yylval.key = strdup(yytext+1);
27                                   yylval.key[yyleng - 2] = '\0';
28                                   BEGIN ARRVAL; return ARRAYKEY; }
29 <ARRVAL>[^\n]*                   { BEGIN INITIAL;
30                                   yylval.arrayvalue=strdup(yytext+1);
31                                   return ARRAYVALUE;
32                                 }
33 <ARRVAL>[\n]                     { BEGIN INITIAL;
34                                   yylval.arrayvalue=strdup("null");
35                                   return ARRAYVALUE;
36                                 }
37 [A-Za-z]*": >"                   { yylval.str = strdup(yytext);
38                                   yylval.str[yyleng - 3] = '\0';
39                                   BEGIN PAR; return PARAGRAPH;
40                                 }
41 <PAR>[ ]{3}[^\n]*                { yylval.str = strdup(yytext+3);
42                                   yylval.str[yyleng - 1] = '\0';
43                                   return blockline;
44                                 }
45 <PAR>^([\n])                     { yylval.str = strdup("\\n");
```

```
46                                    return blockline;
47                                  }
48  <PAR>[\n]/([A-Za-z])            { BEGIN INITIAL;
49                                    yylval.str = strdup("\\n");
50                                    return blockline;
51                                  }
52  [A-Za-z]*": |-"                 { yylval.str = strdup(yytext);
53                                    yylval.str[yyleng - 4] = '\0';
54                                    BEGIN CONT; return CONTENT;
55                                  }
56  [A-Za-z]*": |"                  { yylval.str = strdup(yytext);
57                                    yylval.str[yyleng - 3] = '\0';
58                                    BEGIN CONT; return CONTENT;
59                                  }
60  <CONT>[ ]{3}[^\n]*              { yylval.str = strdup(yytext+3);
61                                    yylval.str[yyleng - 1] = '\0';
62                                    return contLine;
63                                  }
64  <CONT>[\n]/([A-Za-z])           { BEGIN INITIAL; }
65  (.|\n)                                  { ; }
```

4 Resultados Obtidos

De seguida, apresentamos os diversos textos fontes e respetivos resultados produzidos pelo tradutor implementado.

4.1 Exemplo 1

4.1.1 Texto Fonte

```
---
# <- yaml supports comments, json does not
# did you know you can embed json in yaml?

json:
  - rigid
  - better for data interchange
yaml:
  - slim and flexible
  - better for configuration
object:
  skey: 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
```

4.1.2 Resultado

```
{
  "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\ncan auto\nconvert line breaks\nto save space"
}
```

4.2 Exemplo 2

4.2.1 Texto Fonte

```
json:
  - rigid
yaml:
  - slim and flexible
  - better for configuration
stuff: here
content: |
   We
   love
   Language
   Processing
```

4.2.2 Resultado

```
{
  "json": [
    "rigid"
  ],
  "yaml": [
    "slim and flexible"
```

```
    "better for configuration"
  ],
  "stuff": "here",
  "content": "We\nlove\nLanguage\nProcessing"
}
```

4.3 Exemplo 3

4.3.1 Texto Fonte

```
---

animal: pets
pets:
 - Cat
 - Dog
 - Goldfish

```

4.3.2 Resultado

```
{
  "animal": "pets",
  "pets": [
    "Cat"
    "Dog"
    "Goldfish"
  ]
}
```

5 Guia de Utilização

5.1 Compilar

Como podemos ver na `makefile` apresentada em baixo, para compilar os programas basta correr o comando `make`, que cria o executável `yaml`, capaz de produzir o resultado da tradução de qualquer texto fonte fornecido na linguagem *Yaml*.

```
1 make:
2     flex yaml.l
3     yacc -d -v yaml.y
4     gcc -o yaml y.tab.c
```

5.2 Executar

Depois de compilar, para correr o tradutor basta fornecer-lhe um ficheiro para o seu stdin, seguindo o exemplo que se segue:

```
./yaml < ficheiro.yaml
```

6 Conclusão

Este último trabalho prático da unidade curricular de Processamento de Linguagens foi, sem dúvida, o mais exigente, no sentido em que foi o exercício mais trabalhoso e demorado na sua implementação. A sua execução requereu conhecimentos alargados da ferramentas *Flex* e *Yacc*, sendo que este foi o nosso primeiro contacto com a segunda, fora do contexto das aulas da unidade curricular.

A realização deste trabalho exigiu, portanto, a aplicação de todos os conhecimentos adquiridos e consolidados com o desenvolvimento dos demais trabalhos práticos, em especial o primeiro, que consistia essencialmente na elaboração de filtros de texto em Flex, ferramenta também utilizada neste último trabalho.

Estando este trabalho inserido no tema das linguagens de programação, tema com o qual, enquanto alunos de Engenharia Informática, estamos bastante acostumados, trouxe-nos um contacto com estas completamente diferente ao que já tínhamos tido.

De uma forma geral, terminamos este trabalho prático confiantes de que o seu objetivo foi cumprido, tendo sido um projeto no qual nos empenhamos e nos esforçamos para aprender e trazer à vida o resultado pretendido.