DSL para Cadernos de Anotações em Humanidades Digitais

Eduardo Lourenço da Conceição
a83870

António Manuel Carvalho Gonçalves
a85516

João Pedro Dias Fernandes
a84034

28 de Junho de 2020

Resumo

Para a cadeira de Processamento de Linguagens, do 3º ano da Licenciatura em Engenharia Informática, foi-nos proposto realizar um conversor de um caderno de anotações para HTML. Neste relatório iremos expor todos os passos por que passamos para produzir o resultado final, bem como as dificuldades que tivemos e como as ultrapassamos.

Conteúdo

1 Introdução

Para este trabalho, foi-nos dada a missão de, a partir de um caderno de anotações no ramo das *Humanidades Digitais*, com uma DSL (*Domain Specific Language*), criar um site, com vários documentos HTML, utilizando um conversor com analisadores léxico e sintático.

Neste relatório iremos explicar como procedemos para criar tal conversor, utilizando as robustas ferramentas *Flex* e *Yacc*, para gerar processadores Léxicos e Gramaticais, respetivamente, para esta DSL.

Em primeiro lugar, descreveremos a gramática do DSL, passando depois para explicar a sua implementação com o *Yacc*, juntamente com as ações associadas às várias produções. Depois, passaremos a explicar como procedemos à análise léxica, exclarecendo as várias ações semânticas atribuídas a cada uma das expressões regulares que compoem o léxico da linguagem em questão.

2 Descrição da Gramática

O documento em questão é composto por uma lista de *Pares*, sendo este *Pares* o correspondente ao axioma da gramática. Cada um destes pares é composto por duas secções: *Documento* e *Triplos*. O início de cada um destes *Pares* é sinalizado por um conceito, que aparece depois de três sinais '='. Este *conceito* trata-se de uma palavra, normalmente em *camelCase* ou *snake_case*, antecedido de ':'.

Começando por desconstruir a secção *Documento*, esta será composta por três elementos: um conceito, distinto do *conceito* definido anteriormente, *títulos* e *textos*. Um *conceito*, nesta secção, é a primeira linha, que é marcada por iniciar com "@tit:", seguido por um *conceito*. Depois, teremos vários *títulos*, que são frases começadas por '#', e/ou *textos*, que são conjuntos de frases, não começadas por algum caracter em especial.

Por fim, temos a secção *Triplos*, que será a mais complexa das duas. Esta é iniciada por "@triplos:", e é depois seguida de vários *triplos* (de notar a diferença entre *Triplos*, secção, e *triplos*, elemento da seccção *Triplos*). Cada um destes *triplos* é composto por um *sujeito*, uma *relação* e um ou mais *objetos*, sendo que cada *objeto*, *relação* e *sujeito* são, por si, também conceitos. Cada elemento do *objeto* está separado da próxima *relação* por um ponto e vírgula e, no caso de uma *relação* ter mais do que um *objeto*, os vários *objetos* estão separados por uma vírgula. Para além disto, podemos também ter *objetos* que não são *conceitos*, mas sim *strings* entre aspas. O caso mais notório disto são as imagens, que são sempre antecedidas pela *relação* ":img", logo são fáceis de detetar.

Para além disto, pedido como bónus, existe, no final da lista de *Pares*, uma secção *Meta*, começada por "@meta", onde vai ser guardada uma lista de *triplos*, com a estrutura dos *triplos* em *Triplos*, mas em que a relação é sempre ":inverseOf", para definir relações inversas (isto é, por exemplo, a relação "filho_de"é o inverso de "pai_de").

3 Implementação

Nesta secção trataremos de clarificar a implementação utilizando *Yacc* e *Flex* que nós decidimos utilizar, justificando as várias decisões que tomamos ao longo da realização do projeto.

3.1 Estruturas de Dados

Neste trabalho, ao contrário do primeiro que realizamos para esta cadeira, não há necessidade de guardarmos informação em estruturas de dados mais complexas, uma vez que a maior parte da informação é utilizada no momento em que é obtida.

No entanto, a única estrutura que utilizamos foi um *array* de tamanho variável de *strings* para podermos guardar os *links* que vamos obtendo para os poder no final agrupar num ficheiro "index.html".

```
char **hyperlinks;
int size;
```

A este *array* estão associadas funções de inserção e uma função para podermos no final retirar a memória alocada, de modo a reduzir as *memory leaks*.

Isto não inclui, no entanto, a necessidade de guardar objetos na comunicação entre os processos do *Flex* e do *Yacc*, pelo que utilizamos uma *union*.

3.1.1 União

Uma das ferramentas que o *Yacc* implementa, vinda de C, são as *unions*, que corresponde a um conjunto disjunto de atributos. Em *Yacc*, o ficheiro "y.tab.h"fornece a *union yylval*, que nos permite fazer uma comunicação mais robusta entre o *Yacc* e o *Flex*.

Neste projeto, utilizamos dita *union* para transportar *strings* desde o analisador léxico até ao gramatical, de forma a que a *union* tem a seguinte definição:

```
%union
{
     char* string;
}
```

O atributo *string* será prático para guardarmos os vários símbolos terminais (variáveis ou não) com que o processador léxico terá de lidar.

3.2 Gramática

3.3 *Tokens* e tipagem

Os ***tokens*** que podemos encontrar no programa que desenvolvemos para *Yacc* são os seguintes:

```
%token<string> frase titulo quote con
%token<string> ERROR BEGIN_CON BEGIN_TRIP BEGIN_IMG
               BEGIN_META TRESIGUAIS INVERSEOF
%token<string> CARDINALT VIRGULA PONTOVIRG PONTO A
```

Utilizando a notação que foi convencionada nas aulas práticas desta cadeira, os primeiros *tokens* correspondem a símbolos terminais variáveis, sendo uma "frase"um elemento de um *Texto*, o "titulo"um elemento de cada um dos títulos das secções de um *Documento*, "quote"uma expressão entre aspas e um "con"uma *string* com o sufixo ':'.

Continuando a lógica da notação das aulas, o segundo conjunto corresponde a símbolos terminais não variáveis (exceto o ERROR), sendo todos explicados na secção do léxico.

Fugindo ao convencionado na cadeira, no entanto, o segundo grupo corresponde a símbolos terminais não variáveis de tamanho um. Apesar de os podermos simplesmente definir como caracteres únicos na gramática, que o *Yacc* poderia fazer o reconhecimento, essa implementação apresentava erros, por razões que desconhecemos. Como tal, passamos a procurar estes símbolos utilizando o *Flex*.

Quanto à **tipagem** dos vários *tokens* e produções, é simples de explicar. Uma vez que a *union* apenas tem um tipo de dados, um apontador para *char*, todos os *tokens* e produções que precisam de tipo são do tipo *string*, que é o nome que demos ao atributo da *union* que referimos.

3.4 Produções

Nesta secção iremos expor a implementação da gramática que descrevemos anteriormente. Esta implementação foi feita utilizando a sintaxe do *Yacc*.

- **Axioma da Gramática**

  O axioma da gramática, *Caderno*, e outras produções do mesmo nível, têm a seguinte estrutura:

```
Caderno : Pares Meta
        ;
Pares   : Pares Par
        |
        ;
Par     : TRESIGUAIS ConceitoPar Documento Triplos
        ;
```

Sendo que o *Caderno* é uma lista de *Par*, definida por *Pares*, seguida de uma secção *Meta*. Cada *Par*, elemento mais importante da gramática, é constituído por um título que correponde a três caracteres '=' seguidos (TRESIGUAIS) e um *conceito* (*ConceitoPar*), por um *Documento* e um *Triplos*.

De notar desde já que, apesar de todos os *conceitos* terem a mesma estrutura, reduzindo-se apenas ao símbolo *con*, como têm ações semânticas diferentes, decidimos distinguir os vários tipos que se encontram ao longo do documento.

Como tal, o *ConceitoPar* define-se por:

```
ConceitoPar : con
            ;
```

Que é a mesma definição para as várias produções *ConceitoX* que vamos encontrar na gramática, sendo X $\in \{Par, Triplo, Relacao\}$

- **Documento**

  Na secção *Documento*, vamos ter as seguintes produções:

```
Documento    : ConceitoDoc DocElems
             ;
ConceitoDoc  : BEGIN_CON titulo
             ;
DocElems     : DocElems Bloco
             |
             ;
Bloco        : Titulo Texto
             ;
Titulo       : CARDINALT titulo
             |
             ;
Texto        : frase
             ;
```

  Aqui podemos ver que um *Documento* vai ser um *ConceitoDoc* (distinto dos outros *ConceitoX*) e uma lista de *Blocos* de *Documento*, designada *DocElems*. Cada um destes blocos tem um *Titulo* e um *Texto*, sendo que o *Titulo* pode ser vazio, pois existem casos em que um *Documento* não tem nenhum *Titulo*, ou um cardinal seguido de uma frase, ou *titulo*. O *Texto*, por outro lado, é um apenas uma *frase*. A possibilidade de um *Titulo* ser vazio permite também que o *Texto* seja mais que uma linha.

- **Triplos**

  A secção *Triplos* é a mais complexa em termos de gramática, e, como tal, tem o maior número de produções. Ditas produções são as que se seguem:

```
Triplos         : BEGIN_TRIP TriplosElems
                ;
TriplosElems    : TriplosElems Triplo
                | Triplo
                ;
Triplo          : ConceitoTriplo Relacoes PONTO
                ;
Relacoes        : Relacoes PONTOVIRG Relacao
                | Relacao
                ;
Relacao         : A Objeto
                | ConceitoRelacao Objeto
                | BEGIN_IMG quote
                ;
Objeto          : Objeto VIRGULA con
                | con
                | quote
                ;
```

  Aqui vemos que a secção *Triplos* começa com "@triplos:"(BEGIN_TRIP), e segue-se de uma lista de *Triplo*s chamada *TriplosElems*.

  Cada *Triplo*, por sua vez, tem um *ConceitoTriplo*, um *Relacoes* e termina com um '.' (PONTO). Este *Relacoes* é uma produção recursiva, em que cada elemento é uma *Relacao*, e estão todas separadas por ';' (PONTOVRIG).

  Uma *Relacao* é um 'a' (A) seguido de um *Objeto*, ou um *ConceitoRelacao* seguido de um *Objeto*, ou é uma imagem e, nesse cado, é um ":img"(BEGIN_IMG) seguido de uma expressão entre aspas (*quote*).

  Por fim, um *Objeto* é outra produção recursiva, em que temos uma lista de *conceitos*, um só *conceito* ou uma expressão entre aspas.

- **Meta**

  A última secção cujas produções devemos definir é a *Meta*, onde definimos relações inversas. De notar, uma vez mais, que, apesar de termos definido a gramática e produções para esta área, não existem nenhumas ações semânticas associadas.

  Esta porção do *input* é definida por:

```
Meta      : BEGIN_META MetaElems
          |
          ;
MetaElems : MetaElems TriploM
          | TriploM
          ;
TriploM   : con INVERSEOF con
          ;
```

  O primeiro aspeto notável desta secção é a possibilidade de ser vazia, o que é normal. Caso não seja vazia, ela é iniciada por "@meta:"(BEGIN_META), e possui uma lista de *TriploM*, denominada *MetaElems*. Cada elemento da lista é um *triplo*, com uma relação entre dois *conceitos*. A relação, nesta secção, é sempre a mesma: ":inverseOf"(INVERSEOF), e é feita apenas entre dois elementos, não havendo a possibilidade de serem vários elementos separados por vírgula como nos triplos anteriores.

3.5 Ações Semânticas

As ações semânticas realizadas nas várias produções são variadas em alguns casos, mas, numa boa parte deles, o foco é passar o bloco de texto detetado para a produção de onde viemos, com a devida formatação HTML. Isto é, por exemplo, quando detetamos um *Texto*, uma *frase*, para ser mais específico, simplesmente colocamos dentro de uma *tag* HTML "p"e colocamos no '$$', ou seja, no termo à esquerda, podendo assim a produção que a chamou colocar a informação na sua própria *string* até que reconhecemos um *Par* e, nesse caso, escrevemos o seu conteúdo no ficheiro devido. Este ficheiro será, por exemplo, se estivermos no *Par* identificado por "===:OnórioDL", o ficheiro "OnórioDL.html". O ficheiro é aberto em "a"(*append*) de forma a, caso haja algo escrito nesse ficheiro, não apagarmos o que já lá estava.

Na eventualidade de termos algo escrito já no ficheiro em questão quando estamos na produção *Par*, temos o cuidado de escrever sempre o *Documento* no início do ficheiro, com a função *fbegwrt()*, que passa o conteúdo do ficheiro para um *buffer* temporário, caso o ficheiro já exista, apaga o conteúdo do ficheiro, escreve o texto do *Documento* no HTML e depois escreve o que já estiver no *buffer*. A implementação desta função é a seguinte:

```
void fbegwrt(char* path, char* str)
{

    FILE *fp;
    char *tmp, buf[1024];

    tmp = strdup("");

    fp = fopen(path, "r");
    if(fp){
        while(fgets(buf, 1024, fp)){
            asprintf(&tmp, "%s", buf);
        }
        fclose(fp);
    }

    fp = fopen(path, "w");
    if(fp){
        fprintf(fp, "%s%s", str, tmp);
        fclose(fp);
    }
    free(tmp);
}
```

O outro tipo de ação semântica que devemos ter em consideração é chamada quando encontramos um *conceito*. Uma vez que cada *conceito* deverá ter um

ficehiro dedicado, há que criar o ficheiro certo quando se encontra um novo *conceito*. De forma a não apagar informação, abrimos sempre os ficheiros em modo *append*. Isto garante que a informação é mantida e que, ao adicionarmos mais informação, é no fim de um ficheiro, como deverá ser.

De forma a explicar como escrevemos em ficheiro, sem ser quando estamos na produção *Par*, tenhamos em conta o triplo:

```
( subj , rela, obj )
```

Quando entramos neste triplo, a primeira coisa a fazer é guardar o "subj"numa string, e depois o "rela"noutra. Isto vai-nos ajudar com a escrita, pois, uma vez que acabemos de identificar o "obj", vamos escrever, no ficheiro "rela.html"(assumindo que "rela"é um *conceito*):

```
subj : obj
```

No ficheiro "obj.html", mais uma vez, assumindo que o "obj"é um *conceito*:

```
rela : subj
```

E, por fim, no ficheiro "subj"(que será SEMPRE um *conceito*), iremos escrever:

```
rela : obj
```

Todos estes elementos terão a devida formatação em HTML, e os elementos que vêm de *conceitos* terão o *link* para o respetivo ficheiro sempre que são chamados.

Estas são as ações semânticas mais notórias nas várias produções. Entre cada produção, as ações não são muito distintas, pelo que normalmente a verdadeira distinção passa pela formatação HTML exata, que varia de elemento para elemento.

3.6 Léxico

O léxico desta gramática, que está maioritariamente condensado no ficheiro "cad_anot.l", pelo que na seguinte secção explicaremos os seus conteúdos.

3.6.1 *Aliases*

Em primeiro lugar, de notar que existem as seguintes *aliases*:

```
spec_chars \xc3[\x80-\xbf]
letter    [a-zA-Z]|{spec_chars}

pal       {letter}+(_{letter}*)*
par       [^\n#@]+
```

A primeira das várias *aliases* corresponde a caracteres especiais (letras com acentos, nomeadamente), sendo que a expressão regular foi-nos fornecida pelo corpo docente na primeira fase. A segunda trata-se apenas de uma entensão da primeira, de forma a incluir todas as letras do alfabeto latino acentuadas ou não.
A terceira vai corresponder a uma palavra, que pode ser escrita em *camelCase* ou em *snake_case*, e a última corresponde a uma linha de texto, que acaba com um '\n' ou com os caracteres que iniciam secções especiais na gramática.

3.6.2 Contextos

Os vários contextos exclusivos que encontramos no léxico são referentes às várias secções principais que compôem o ficheiro *input*:

```
%x TRIPLOS CONCEITO TEXTO TITULO META
```

Os contextos TRIPLOS e META correpondem às respetivas secções, enquanto que o contexto CONCEITO serve apenas para a primeira secção de um *Par*, e os contextos TITULO e TEXTO correpondem a elementos da secção *Documento*.
Ao analisarmos o léxico, vamos explicar as várias ações semânticas e expressões regulares dentro de cada um destes contextos, de forma a melhor organizar a informação.

3.6.3 Expressões Regulares

- **Contexto CONCEITO**

  Este contexto serve para identificar o *conceito* inicial de um *Par*, e, como tal, apenas o vai identificar, guardar na *union*, e devolver o respetivo *token*.

```
<CONCEITO>:{pal} { BEGIN TEXTO; yylval.string =
                   strdup(yytext+1); return con; }
```

- **Contexto TITULO**

  No documento, quando estamos no contexto TITULO, apenas temos de encontrar um parágrafo, e passá-lo depois para o processo do *Yacc* para formatação para HTML. Como tal, apenas temos a seguinte ação:

```
<TITULO>{par} { BEGIN TEXTO; yylval.string =
                strdup(yytext); return titulo;}
```

  No final da mesma, temos de devolver um *token titulo*.

  Para além disto, tendo em conta que estaríamos no contexto TEXTO antes de entrarmos em TITULO, temos de voltar a esse mesmo contexto.

- **Contexto TEXTO**

  O outro contexto que encontramos quando estamos na secção *Documento* é o TEXTO. Aqui, teremos de identificar duas situações: o cardinal que simboliza o início de um *titulo*, e uma frase que forma a maior parte do texto. Como tal:

```
<TEXTO>#      { BEGIN TITULO; return CARDINALT; }
<TEXTO>{par}  { yylval.string = strdup(yytext); return frase; }
```

  Como podemos ver, quando encontramos um cardinal passamos ao contexto TITULO e devolvemos o *token* CARDINALT, e quando encontramos uma frase guardamo-la na *union* e devolvemos o *token frase*.

- **Contexto TRIPLO**

  No contexto TRIPLO temos de identificar os vários símbolos terminais não variáveis de tamanho 1, pelas razões que definimos anteriormente, bem como *conceitos*, o início de uma imagem, e uma *string* entre aspas. Nos dois últimos casos, temos de guardar a *string* captada na *union*.

```
<TRIPLOS>\.          { return PONTO; }
<TRIPLOS>;           { return PONTOVIRG; }
```

```
<TRIPLOS>,           { return VIRGULA; }
<TRIPLOS>:img        { return BEGIN_IMG; }
<TRIPLOS>a           { return A; }
<TRIPLOS>\"[^\"]+\"  { yytext[yyleng-1] = '\0';
                       return quote; }
<TRIPLOS>:{pal}      { yylval.string = strdup(yytext+1);
                       return con; }
```

De notar que, quando capturamos uma *string* entre aspas, a *string* que vai ser guardada na *union* não vai ter as aspas. No final de cada ação, como sempre, tem de ser devolvido o *token* correspondente.

- **Contexto META**

  Neste contexto, uma vez que não implementamos as várias ações semânticas no "cad_anot.y", não guardamos nenhuma informação. Assim sendo, apenas devolvemos o *token* necessário de forma a reconhecermos corretamente a gramática no fichieiro *input*.

```
<META>:inverseOf   { return INVERSEOF; }
<META>:{pal}       { return con; }
```

  Como podemos notar, neste contexto apenas reconhecemos *conceitos*, sendo que reconhecemos o *conceito* ":inverseOf"como um caso especial

- **Ações livres de Contexto**

  Para além das várias ações descritas nos contextos expostos até agora, temos também ações *contextless*, isto é, expressões regulares que testamos independentemente do contexto.

```
<*>===       { BEGIN CONCEITO; return TRESIGUAIS; }
<*>@tit:     { BEGIN TITULO; return BEGIN_CON; }
<*>@triplos: { BEGIN TRIPLOS;  return BEGIN_TRIP; }
<*>@meta:    { BEGIN META; return BEGIN_META; }

<*>[ \t\n\r] ;
<*>.         { return ERROR; }
```

  O primeiro grupo destina-se a reconhecer o início das várias secções do documento, enquanto que o segundo serve para filtrar os espaços brancos e detetar, caso todas as outras expressões falhem, erros no documento.

4 *Output*

O *output* do programa vai ser escrito na pasta "out". Para além dos vários documentos HTML com a informação toda que se foi reunindo, também vai ser criado um "index.html", onde reunimos todos os *links* dos ficheiros *output*, para acesso mais fácil. Este ficheiro é criado depois da execução da função *yyparse()* na *main*, e apenas vais escrever os *links* que estão guardados no *array* dinâmico *hyperlinks*, que, depois de isto decorrer, será apagado.

Aqui mostramos um excerto do documento "OnórioL.html", um dos muitos que é gerado a partir do ficheiro *input*, quando aberto com o Brave Browser.

Onório de Lima

Infância no Brasil

Onório de Lima viveu no Brasil.

Ele viveu no Brasil até que deixou de viver

no Brasil.

Percurso cultural no Porto

Entre 1873 e 1930 Onório de Lima viveu no Porto,

Cedofeita e foi parágrafos separados por linhas em branco.

Lorem Ipsum

Triplos

OnórioL:

Person, Mecenas_cultural

Name: Onório de Lima

birth_place: Brasil

viveu_em: Porto, Gerês

Figura 1: Documento "OnórioL.html"

5 Makefile

A Makefile do projeto é simples, tendo apenas as primitivas para poder gerar os ficheiros objeto do *Yacc* e do *Flex*, compilando-os depois juntos. Para além disto, tem a possibilidade de obter o ficheiro *input* caso ele não exista, com o comando *wget* que está pronto a ser utilizado em qualquer *bash* Unix. Mais ainda, para correr o programa e gerar os ficheiros HTML, pode-se fazer simplesmente "make run". Por fim, temos o item "clear"para podermos apagar todos os ficheiros gerados após a execução do programa.

```
run : prog in/ex.caderno
	cat in/ex.caderno | ./prog

prog : y.tab.o lex.yy.o
	gcc -o prog y.tab.o lex.yy.o -ll

y.tab.o : y.tab.c
	gcc -c y.tab.c

lex.yy.o : lex.yy.c y.tab.h
	gcc -c lex.yy.c

y.tab.c y.tab.h : cad_anot.y
	yacc -d cad_anot.y

lex.yy.c : cad_anot.l y.tab.h
	flex cad_anot.l

ex.caderno :
	cd in/
	wget https://natura.di.uminho.pt/~jj/pl-20/TP2/ontologic-wiki/ex.caderno
	cd ..

clean:
	rm -f lex.yy.* y.tab.* prog *.html out/*.html
	clear
```

Figura 2: Makefile

6 Conclusão

Acreditamos ter feito, no final, um bom trabalho. O foco do projeto foi converter o ficheiro *input* no ramo das Humanidades Digitais, com uma gramática e sintaxe específicas, num pequeno site HTML, bastante *bare bones*, utilzando os vários conhecimentos que nos foram passados nas aulas de Processamento de Linguagens sobre o *Yacc*, o *Flex* e as suas complexidades, e nesse aspeto conseguimos alcançar todos os objetivos.

Tivemos alguns problemas no desenvolvimento do *software*, nomeadamente alguns erros de manipulação da gramática que nos obrigaram a utilizar soluções menos comuns, mas soluções eficazes na mesma.

A maior falha no trabalho foi não termos tido a possibilidade de fazer tudo dos requisitos extra, mas, mesmo assim, conseguimos definir as produções que seriam de esperar e com que teríamos de trabalhar caso essa secção tivesse sido completada.

Em suma, acreditamos que, ao todo, fizemos um trabalho bastante bom e que cumpre todos os requisitos pedidos.