YACC

Carlos Gomes A77185 | Helena Martins A82500 | Nuno Silva A78156

(28 de Junho de 2020)

Resumo

Este trabalho prático desenvolvido, visa a geração de árvores geneológicas, através de dados recolhidos de diversas ontologias. Isto irá ser feito com o auxílio de gramáticas independentes de contexto, que serão definidas através do *Yacc* e de expressões regulares, em *Flex*.

1 Introdução

Por solicitação da equipa docente, da unidade curricular Processamento Linguagens, que aprofundássemos todos os conhecimentos adquiridos ao londo deste semestre. Desta forma, foi nos proposto o desenvolvimento de um reconhecedor léxico e sintático para ontologias que descrevem árvores geneológicas. Este desenvolvimento, foi possível graças a ferramentas geradoras Flex/Yacc, e também devido à criação de ações semânticas de tradução até às produções da gramática, por via do Yacc.

Este mesmo, graças as suas exigências, apresenta-se como uma mais-valia, dado que aumenta a experiência de uso do ambiente Linux, na linguagem de programação C, aumenta a capacidade de escrita de gramáticas independentes de contexto (GIC) que satisfaçam a condição LR(), para criar Linguagens de Domínio Específico (DSL) e desenvolvimento de processadores de linguagens segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora(GT).

Assim sendo, posteriormente será apresentado todo o processo de resolução e todas as etapas desenvolvidas para esse fim.

2 Objectivos

Como qualquer trabalho que é nos proposto ao longo do nosso caminho estudante, é importante referir objectivos pretendidos durante o desenvolvimento deste trabalho, sendo elas:

- Aumentar a nossa experiência de uso do ambiente Linux, assim como da linguagem imperativa C;
- Rever e aumentar as nossas capacidades de escrever gramáticas independentes de contexto;
- Desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, suportada numa gramática tradutora;
- Utilizar geradores de compiladores Flex/Yacc.

3 Descrição do Problema

Como já foi referido, foi nos solicitado o desenvolvimento de um conversor de ontologias para lingugagem para grafos, escritas em **DOT**. Aqui será possivel visualizar as relações existentes através de **Graphviz** ou até converter para **SVG**, onde irão ser visualizadas em páginas web. De seguida apresenta-se um breve exemplo de como as ontologias poderão ser apresentadas:

Exemplo de ontologias

```
###  http://www.di.uminho.pt/prc/ontologies/2020/prc-genoa#
     Filomena_Esteves_Araujo_1927
:Filomena_Esteves_Araujo_1927 rdf:type owl:NamedIndividual ,
                                       :Pessoa ;
                              :temMae :Maria_Araujo_1884 ;
                              :temPai :Henrique_Luiz_Araujo_1867 .


###  http://www.di.uminho.pt/prc/ontologies/2020/prc-genoa#
     Flora_Castilho_Couto_Leite
:Flora_Castilho_Couto_Leite rdf:type owl:NamedIndividual ,
                                    :Pessoa .

###  http://www.di.uminho.pt/prc/ontologies/2020/prc-genoa#
     Henrique_Luiz_Araujo_1867
:Henrique_Luiz_Araujo_1867 rdf:type owl:NamedIndividual .
```

Para o desenvolvimento deste conversor, é necessário ter em conta algumas anotações:

- Linhas iniciadas por '#' são comentários e devem ser ignorados;
- Uma ontologia nesta forma é uma lista de triplos: (sujeito, predicado, objeto);
- O sujeito é sempre um identificador (URI) de um indivíduo;

- O predicado é sempre um identificador (URI) de uma relação ou propriedade;
- O objeto pode ser um identifcador (URI) ou então um conceito ou um valor escalar: string, número, etc;
- Um triplo normal termina por ".";
- Um triplo terminado por ","significa que o próximo triplo está abreviado e vai partilhar o mesmo sujeito e predicado;
- Um triplo terminado por "."significa que o próximo triplo está abreviado e vai partilhar o mesmo sujeito;

Com isto em mente, foi nos sugerido seguirmos as seguintes etapas para o desenvolvimento do projecto:

- Estudar o formato, a sua estrutura e alguns casos de estudo;
- Escrever uma gramática para a notação ilustrada para este tipo de ontologias;
- Estudar e definir o formato de saida, compatível com **Graphviz**;
- Construir um processador que consiga reconhecer e validar ontologias genealógica, gerando o **DOT** pretendido;

4 Descrição do Problema

Para começarmos o desenvolvimento de projecto, primeiro é importante analisar o ficheiro *mini-familia.ttl* fornecido pela equipa docente. Ao analisarmos este ficheiro, podemos perceber que se divide em 3 partes: *Object Properties*, *Classes* e *Individuals*, sendo a parte *Individuals* a que nos iremos concentrar para o desenvolvimento do gerador.

4.1 Descrição do gerador

Numa fase preparatória para o desenvolvimento desta solução, concluímos que a nossa gramática tem de ser capaz de construir as multiplas relações descritas nos triplos presentes num ficheiro. Assim sendo, podemos considerar o nosso gerador como um conjunto de triplos.

4.2 Geração da Arvore Geneológica

Como indicado anteriormente, o ficheiro gerado terá de ser um do tipo **DOT**. Após uma análise sobre a documentação fornecida pelo **Graphviz** [1], verificamos que o ficheiro gerado terá o seguinte formato:

Exemplo de uma arvore geneológica

```
digraph familytree{
        forcelabels=true
        node [shape=box]
        graph [rankdir="LR",fontname="helvetica", ranksep=1.5, nodesep
            =1.5, overlap="false", splines="true"]
        size="71,41";
        Helena_Pocas_Martins_1998 -> {
            Maria_do_Pranto_Martins_Goncalves_Pocas_1963[xlabel="temMae
            "]
        Valdemar_de_Jesus_Mesquita_Martins_1947[xlabel="temPai"]};
        Maria_do_Pranto_Martins_Goncalves_Pocas_1963 -> {Irene_Martins[
            xlabel="temMae"]
        Silvino_Goncalves_Pocas_1919[xlabel="temPai"]};

}
```

Como podemos verificar, o ficheiro gerado apresenta notações e sintaxes muito simples e perceptíveis para o utilizador, sendo que as sete primeiras linhas, se destinam para efeitos estéticos. Assim sendo, este exemplo irá reporduzir o seguinte grafo:

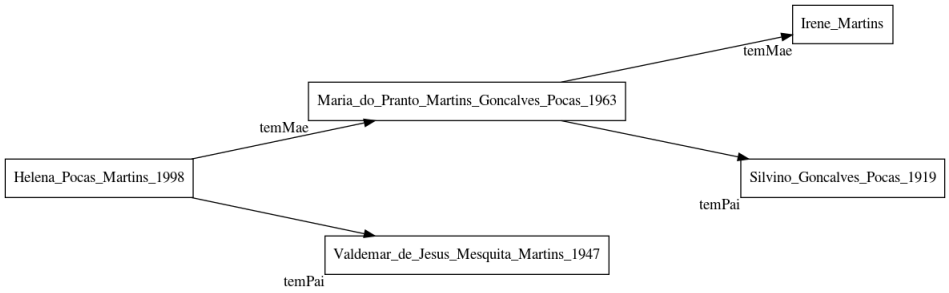

Figura 1: Exemplo de uma arvore geneologica, através de grafos

5 Implementação da Solução

5.1 GIC - Gramática Independente Contexto

No começo da nossa implementação, começamos por definir a nossa GIC, especificando os triplos e as suas possíveis abreviações. Ao longo deste processo, pretendemos que fosse de fácil compreensão, simples e apresentável, com intuito de um utilizador comum consiga entender/desenvolver os triplos pretendidos, surgindo assim a seguinte gramática:

```
T = {SUJEITO, CONCEITO, STRING, NUM, PREDICADO}

N = {Programa, Ontologia, Triplo, ListaDuplos, Objeto}

S = Programa

P = {
p0: Programa -> Ontologia

p1: Ontologia -> Triplo
p2:            | Ontologia Triplo

p3: Triplo -> SUJEITO '.'
p4:         | SUJEITO ',' ListaDuplos

p5: ListaDuplos -> PREDICADO Objeto ',' ListaDuplos
p6:              | PREDICADO Objeto ';' ListaDuplos
p7:              | Objeto ';' ListaDuplos
p8:              | PREDICADO Objeto '.'
p9:              | Objeto '.'

p10: Objeto -> SUJEITO
p11:         | CONCEITO
p12:         | STRING
p13:         | NUM
}
```

5.2 Filtros de Texto

De forma a complementar a gramática desenvolvida e capturar apenas o texto pretendido dos símbolos terminais, foram criadas expressões regulares, sobre o qual iremos explorar de seguida.

5.2.1 Sujeito

Numa análise muito breve sobre as ontologias que nos foram disponibilizadas pela equipa docente, podemos verificar que os nomes de cada pessoa presente seguem-se separados do "_". Em que no final, estes terminam, ou não, com um conjunto de quatro dígitos, sobre o qual o grupo considera ser o ano de nascimento.
Com isto em conta foi desenvolvida as seguintes expressões regulares:

(1) `^:[A-Za-z_]+([0-9]{4})?`

(2) `"_":[A-Za-z_]+([0-9]{4})?`

A primeira expressão, (1), como podemos ver destina-se a capturar o nome do sujeito no início de cada linha, porém, a segunda expressão, (2), é acionada após o estado sobre as relacoes entre sujeitos.

5.2.2 Predicado

Após a identificação do Sujeito, passámos para a identificação dos Predicados. Para estes não foram desenvolvidas nenhumas expressões regulares, por sugestão da equipa docente, simplifica-mos ao máximo a captura destes, indicando-os de uma forma *"hardcoded"*, como podemos ver de seguida.

```
(1)  ":temMae"          (2)  ":temPai"
(3)  ":temFilho"        (4)  ":temAv "
(5)  ":temAv "
```

Após a captura deste predicado, é iniciado um outro estado de Sujeito, tal como indicado anteriormente. Estes predicados, visam a descrever a relacao entre duas entidades, neste caso Sujeitos.

5.2.3 Conceito

Tal como feito para com os Predicados, os Conceitos, foram também indicados de uma forma *"hardcoded"*, como apresentado de seguida:

```
:Pessoa
```

Na maior parte dos triplos identificados, este tipo de Conceito é partilhado entre eles, como podemos ver mais a frente na secção de resultados.

5.2.4 String

Como verificámos anteriormente, um Objeto poderá tomar o seu valor como um Sujeito, Conceito, String, ou valor numérico. Assim sendo, de forma a obter uma String, foi desenvolvida a seguinte expressão:

```
:[A-Za-z]+
```

Através desta expressão é nos possibilitada a captura de qualquer texto inicializado por ":"e que se enquadre a região destinada a Objetos.

5.2.5 Num

Por fim, e como indicado anteriormente, um Objeto poderá tomar um valor númerico, neste caso específico, foi desenvolvida uma expressão que permita, a captura de um ou mais dígitos:

```
:[0-9]+
```

6 Resultados Obtidos

A gramática indicada e as expressões desenvolvidas, possibilitam-nos a criação de árvores geneológicas como poderemos verificar de seguida.

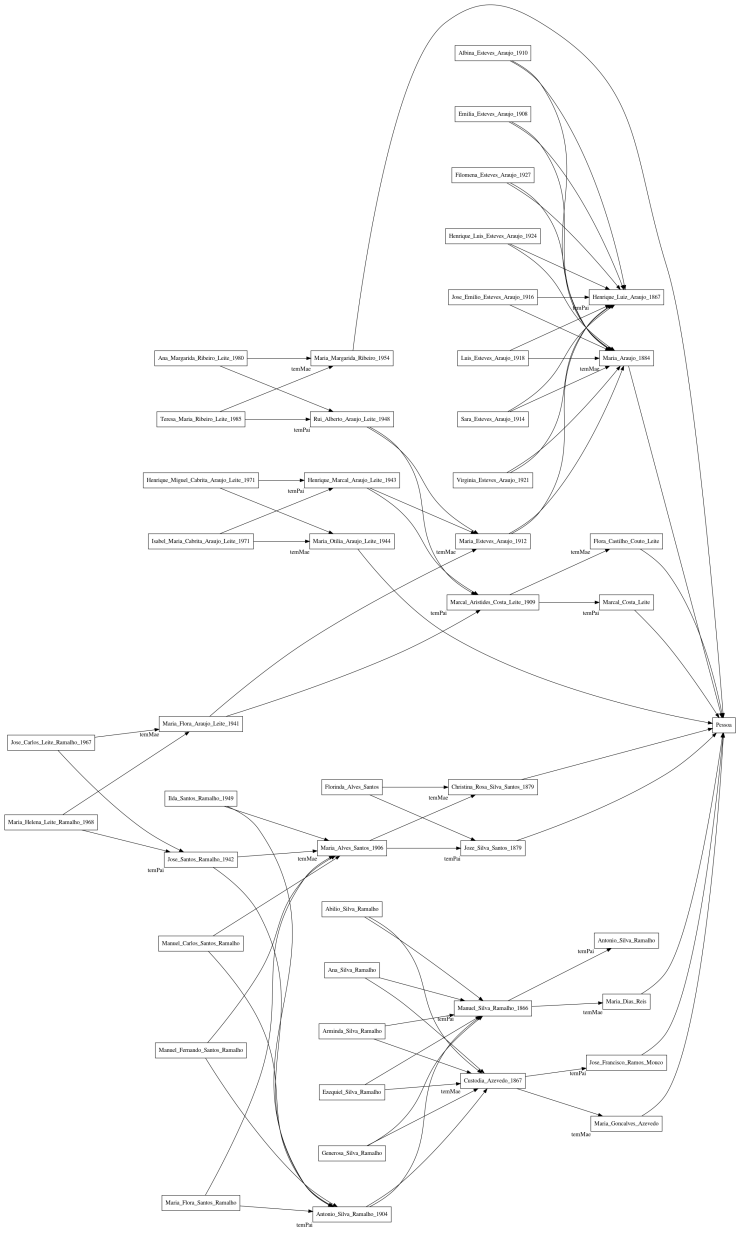

Figura 2: Arvore Geneológica gerada

7 Conclusão

Como sempre, a realização de um trabalho prático traz sempre beneficios e obstáculos, os quais temos que aprender a ultrapassar. Ao longo deste projecto, desenvolvemos um reconhecedor léxico e sintático para ontologias que descrevem árvores geneológicas. Apesar de acharmos que o objetivo pretendido foi alcançado, tivemos algumas dificuldades para o alcançarmos. Uma das dificuldades principais que o nosso grupo teve foi escrever da gramática para uma estrutura onde seriam guardados todos os dados lidos e daí escrever para um ficheiro **DOT**. Como tivemos dificuldades nessa implementação, decidimos mudar o plano inicial e em vez de escrevermos da gramática para a estrutura, acabamos por escrever diretamente para o ficheiro. Posto isto, sentimos que a nossa prestação neste trabalho foi positiva.

Referências

[1] 2020. [online] Available at: https://graphviz.gitlab.io/_pages/pdf/dotguide.pdf [Accessed 28 June 2020].

8 Apêndice

Ficheiro Yacc

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



        int yylex();
        int yyerror(char *s);



%}


%union{
    char* str;
        int value;
}

%token<str> SUJEITO
%token<str> CONCEITO
%token<str> STRING
%token<str> PREDICADO
%token<value> NUM
%type<str> Triplo ListaDuplos Objeto Ontologia



%%

Programa : Ontologia            {FILE* file = fopen("grafo.dot","w+");
                                   fprintf(file,"digraph familytree{\n\
    tforcelabels=true");
                        fprintf(file,"\n\tnode [shape=box]\n\tgraph [
                            rankdir=\"LR\",fontname=\"helvetica\",
                            ranksep=1.5,nodesep=1.5,overlap=\"false
                            \",splines=\"true\"]\n\tsize=\"71,41\";\n%
                            s\n}\n",$1);}
                ;

Ontologia : Triplo              {asprintf(&$$, "%s",$1);}
        | Ontologia Triplo      {asprintf(&$$, "%s %s",$1,$2);}
                ;

Triplo : SUJEITO '.'    {asprintf(&$$, "        %s;",$1); }
       | SUJEITO ',' ListaDuplos  {asprintf(&$$, "      %s -> {%s};\n",
          $1,$3);}
         ;


ListaDuplos : PREDICADO Objeto ',' ListaDuplos    {asprintf(&$$, "%s[
    xlabel=\"%s\"]\n %s", $2,$1, $4);}

            | PREDICADO Objeto ';' ListaDuplos    {asprintf(&$$, "%s[
                xlabel=\"%s\"]\n %s", $2,$1, $4);}

            | Objeto ';' ListaDuplos      {asprintf(&$$, "%s",$3);}

           | PREDICADO Objeto '.'    {asprintf(&$$, "     %s[xlabel=\"%s
                \"]", $2,$1);}
```

```
          | Objeto '.'     {}
            ;


Objeto : SUJEITO          {asprintf(&$$, "%s",$1);}
       | CONCEITO    {asprintf(&$$, "%s",$1);}
       | STRING                  {asprintf(&$$, "%s",$1);}
       | NUM             {asprintf(&$$, "%d",$1);}
         ;

%%


int yyerror(char* s){
    printf("Erro: %s \n",s);
    return 1;
}

int main(){

    yyparse();
    return 0;
}
```

Ficheiro Flex

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "y.tab.h"

%}

%option yylineno
%option noyywrap

%x REL


%%

[\n \t]*                                    {;}

[\.,;]                                       {return yytext[0];}

^@(.|\n)+\#$                                 {;}

\#.*                                         {;}

^:[A-Za-z_]+([0-9]{4})?                          {yylval.str = strdup(yytext
    +1); return SUJEITO;}

:type                                        {;}
:NamedIndividual                             {;}

:Pessoa                                       {yylval.str = strdup(yytext+1);
    return CONCEITO;}

":temMae"                                    {yylval.str = strdup(yytext+1);
    BEGIN REL; return PREDICADO; }

":temPai"                                    {yylval.str = strdup(yytext+1);
    BEGIN REL; return PREDICADO; }
```

```
":temFilho"                                    {yylval.str = strdup(yytext+1);
    BEGIN REL; return PREDICADO; }

":temAv "                                       {yylval.str = strdup(yytext+1);
    BEGIN REL; return PREDICADO; }

":temAv "                                       {yylval.str = strdup(yytext+1);
    BEGIN REL; return PREDICADO; }


<REL>"␣":[A-Za-z_]+([0-9]{4})?                  {yylval.str = strdup(yytext+2);
    BEGIN INITIAL; return SUJEITO;}

:[A-Za-z]+                                     {yylval.str = strdup(yytext+2);
    return STRING;}

:[0-9]+                                        {yylval.value = atoi(strdup(
    yytext+1)); return NUM;}


.           {;}


%%
```