columns

Processamento de Linguagens

Trabalho Prático nº3 (Yacc)

Formato Biblio::Thesaurus

Relatório de Desenvolvimento

Cesário Perneta	João Gomes	Tiago Fraga
a73883	a74033	a74092

12 de Junho de 2018

Resumo

Neste relatório será abordado a resolução e verificação do terceiro trabalho prático da unidade curricular de Processamento de Linguagens. Será possível encontrar a forma como abordamos o problema proposto bem como uma explicação detalhada do código elaborado. Vamos ainda mostrar uma analise de resultados, de modo, a ser percetível, que os objetivos estabelecidos foram cumpridos pelo grupo.

Conteúdo

1 Introdução **2**

2 Enunciado **3**

3 Analise do Problema **4**

4 Desenvolvimento **5**

4.1 Analisador léxico 5

4.1.1 INITIAL 5

4.1.2 COMENTARIO 6

4.1.3 LANGUAGE 6

4.1.4 INV 7

4.1.5 RELACOES 7

4.1.6 SN 7

4.2 Gramática 7

4.2.1 Estruturas de Dados 7

4.2.2 Símbolos Terminais e Não Terminais 9

4.2.3 Definição da Gramática 9

5 Analise de Resultados **11**

6 Conclusão **13**

Capítulo 1

Introdução

Este trabalho prático foi nos proposto com o intuito, de melhorarmos as nossas capacidades no desenvolvimento de processadores de Linguagens segundo o método da tradução dirigida pela sintaxe, suportado numa gramática tradutora.

O enunciado que nos foi proposto, foi o número quatro, ou seja o **Formato biblio:: Thesaurus**. Neste enunciado, é nos dado como exemplo um ficheiro no formato ***ISO 2788 (T2788)***, este tipo de ficheiro contêm um conjunto de meta-dados e um conjunto de conceitos.

Com este tipo de informação fornecida, é nos pedido que geremos um conjunto de páginas *HTML* para cada conceito presente no ficheiro respeitando o conjunto de meta-dados.

Capítulo 2

Enunciado

Tal como explicado de forma introdutória, o nosso enunciado consiste em criar uma gramática Tradutora, capaz de processar um ficheiro do tipo **Formato biblio:: Thesaurus** no formato ***ISO 2788 (T2788)***. Este tipo de ficheiro é constituído por um conjunto de meta-dados (indicação das línguas, relações externas,inversas das relações, titulo, etc) e um conjunto de conceitos.Este conjunto de conceitos apresenta :

- representante na língua base
- traduções noutras línguas
- relações conceptuais com outros conceitos

Os objetivos deste enunciado proposto pelos docentes, são escrever uma gramática e um analisador léxico capaz de reconhecer o documento no formato especificado, armazenar tudo o que reconhecemos filtrando o que interessa numa estrutura de dados e por último criar uma página *HTML* para cada conceito reconhecido.

Exemplo de um ficheiro **Formato biblio:: Thesaurus**:

```
# directivas / metadados
%language PT EN
%baselang EN
%inv NT BT

# conceitos:

 animal
 PT animal
 NT cat, dog, cow, fish, ant
 NT camel
 BT Life being

 cat
 PT gato
 BT animal
 SN animal que tem sete vidas e meia

 #comentário
```

Capítulo 3

Analise do Problema

Neste capitulo explicamos como abordamos o problema, todas as etapas da resolução deste enunciado. Começamos pela criação do analisador léxico, assim começamos por estruturar o nosso código reconhecendo desde já todos os símbolos importantes e a descartar informação que não nos interessava.

Posto isto criamos a nossa gramática, como já tínhamos o analisador estruturado, permitindo assim a criação da gramática de forma fácil e intuitiva.

Com o analisador e a gramática criada faltava criar as estruturas de modo a guardar a informação apanhada e posteriormente a escrita no ficheiro de modo a criar as páginas *HTML*.

Estes foram os passos tomados, na realização do trabalho prático, no capítulo seguinte apresentamos o código desenvolvido e a explicação do mesmo, mostrando os passos tomados de modo mais detalhado.

Capítulo 4

Desenvolvimento

Neste capítulo vamos apresentar como desenvolvemos cada parte do código de forma detalhada.

4.1 Analisador léxico

O nosso analisador léxico como foi dito anteriormente foi o primeiro a ser desenvolvido, como tal, tivemos de colocar todos os valores de retorno, a medida que o ficheiro era processado de modo a corresponder com a gramática. Vamos apresentar o ficheiro que vai ser interpretado pelo **Flex**.

```
%x COMENTARIO LANGUAGE INV RELACOES SN
```

4.1.1 INITIAL

No inicio do documento temos o nome dado as condições de contexto utilizadas, tal como, aconteceu no Trabalho prático anterior, achamos que as condições de contexto iriam facilitar o processamento do ficheiro.

```
<INITIAL>{
\#                  {BEGIN COMENTARIO;}
\%lang[a-zA-Z]+?    {BEGIN LANGUAGE;}
\%baselang          {BEGIN LANGUAGE;}
\%inv[a-zA-Z]+?     {BEGIN INV;}
[a-z]+              { yylval.c = strdup(yytext); return TITULO;}
SN                  { yylval.c = strdup(yytext); BEGIN SN; return ID;}
[A-Z]+              { yylval.c = strdup(yytext); BEGIN RELACOES; return ID;}
.|\n                {;}
}
```

Aqui temos a condição de contexto inicial onde apresentamos as marcas no ficheiro e iniciamos as condições de contexto declaradas em cima.

```
\#                  {BEGIN COMENTARIO;}
```

Sempre que no ficheiro encontrarmos um #, que representa o comentário no nosso tipo de ficheiro, começamos a condição de contexto **COMENTARIO**, que explicamos posteriormente.

```
\%lang[a-zA-Z]+?    {BEGIN LANGUAGE;}
```

Esta *tag* começada por *%lang* representa no nosso ficheiro todas as línguas admitidas por ele, por este motivo achamos que devíamos iniciar uma condição de contexto, de modo, a apanharmos as linguagens todas do ficheiro.

```
\%baselang          {BEGIN LANGUAGE;}
```

Esta *tag* representa a linguagem de base do ficheiro, como tal iniciamos a mesma condição de contexto acima referida.

```
\%inv[a-zA-Z]+?     {BEGIN INV;}
```

Quando aparece uma *tag* começada por *%inv* quer dizer que estamos perante relações inversas, por isso temos de guardar todas as relações inversas, que aparecem para mais tarde associar com os conceitos. Logo iniciamos uma nova condição de contexto.

```
[a-z]+              { yylval.c = strdup(yytext); return TITULO;}
```

Quando aparece no ficheiro um conjunto de letras em minúsculas, estamos perante um titulo, como tal guardamos o valor, e retornamos para a gramática o símbolo terminal **TITULO**.

```
SN                  { yylval.c = strdup(yytext); BEGIN SN; return ID;}
```

Quando no inicio da linha temos *SN*, estamos perante uma nota, e para tal temos que guardar o texto associado, e iniciar uma condição de contexto de modo a guardar o resto da nota associada. Além disso retornamos o símbolo terminal **ID**.

```
[A-Z]+              { yylval.c = strdup(yytext); BEGIN RELACOES; return ID;}
```

Posto isto falta definir as expressões regulares para apanhar as relações do ficheiro bem como as palavras associadas para tal guardamos a relação, começamos uma condição de contexto e retornamos novamente o símbolo terminal **ID**.

```
.|\n                {;}
```

Por último temos a expressão regular para eliminar linhas em branco porque não têm interesse para o resultado final.

4.1.2 COMENTARIO

```
<COMENTARIO>{
.                   {;}
\n                  {BEGIN INITIAL;}
}
```

Nesta condição de contexto fazemos o tratamento do comentário, como não nos interessa apanhar o comentário sempre que apanhamos o símbolo # eliminamos tudo o que aparece a frente.

4.1.3 LANGUAGE

```
<LANGUAGE>{
[A-Z]{2}                        {yylval.c = strdup(yytext); return LINGUA;}
\n                               {BEGIN INITIAL;}
}
```

Como foi dito anteriormente sempre que estamos perante as *tags* **%lang** e **%baselang** começamos esta condição de contexto. Nesta condição de contexto guardamos as linguagens presentes a frente das *tags* e retornamos o símbolo terminal **LINGUA**, sempre que mudamos de linha voltamos a *tag* inicial.

4.1.4 INV

```
<INV>{
[A-Z]+[\_]?[A-Z]+                    {yylval.c = strdup(yytext); return ID;}
\n                                   {BEGIN INITIAL;}
}
```

Quando apanhamos a *tag* começada por *%inv* temos de guardar as relações inversas, como mostra a primeira expressão regular, apanhamos todo o texto podendo opcionalmente conter _ no meio das palavras e retornamos o símbolo terminal **ID**. Sempre que mudamos de linha voltamos ao inicio.

4.1.5 RELACOES

```
<RELACOES>{
[A-Za-z]+[ ]?[a-z]+              {yylval.c = strdup(yytext);return TERMO;}
\,                               {;}
\n                               {BEGIN INITIAL;}
}
```

De modo a guardarmos posteriormente os conceitos corretamente, temos de guardar toda a informação sempre que aparece uma relação daí esta condição de contexto, como no ficheiro podemos ter relação e um termo, ou relação e vários termos, temos de considerar o carácter **,**. Neste ponto retornamos o símbolo terminal **TERMO**.

4.1.6 SN

```
<SN>{
.*            {yylval.c = strdup(yytext); return TERMO;}
\n            {BEGIN INITIAL;}
}
```

SN é uma relação, mas como exige um tratamento diferente das demais, tivemos que criar uma condição de contexto para a guardar corretamente embora o símbolo terminal que ela retorna seja o mesmo que as outras relações, ou seja, **TERMO**.

4.2 Gramática

Nesta secção vamos apresentar a nossa gramática, bem como as estruturas de dados utilizadas, ou seja, o nosso ficheiro que vai ser interpretado pelo **Yacc**.

4.2.1 Estruturas de Dados

As nossas estruturas de dados baseiam-se em listas ligadas, de modo a guardarmos toda a informação relativa ao ficheiro processado.

```
typedef struct linguas{
        char* nome;
        struct linguas* next;
}Lista_Linguas;

typedef struct inversos{
        char* primeiro;
        char* segundo;
        struct inversos* next;
```

```
}Lista_Inversos;


typedef struct dados{
        Lista_Linguas* linguas;
        char* baseLanguage;
        Lista_Inversos* inversos;
}Lista_Dados;
```

Estas três listas ligadas guardam a informação relativa aos meta-dados do ficheiro.

A primeira estrutura guarda todas as linguagens presentes no ficheiro.

A segunda estrutura guarda todas as relações inversas presentes no ficheiro.

As terceira estrutura possuí um apontador para a cabeça da primeira estrutura e da segunda, e guarda a linguagem de base.

```
typedef struct termos{
        char* termo;
        struct termos* next;
}Lista_Termos;

typedef struct relacoes{
        char* id;
        Lista_Termos* termos;
        struct relacoes* next;
}Lista_Relacoes;


typedef struct conceitos{
        char* titulo;
        Lista_Relacoes* relacoes;
        struct conceitos* next;
}Lista_Conceitos;
```

Estas estruturas de dados servem para guardar os conceitos presentes nos ficheiros.

A primeira estrutura vai guardar todos os termos presentes a frente das Relações.

A segunda estrutura vai guardar todas as relações.

E tal como na estrutura utilizada para a estrutura de meta-dados, guardamos na 3ºestrutura o apontador para a lista de relações por titulo de conceito.

```
typedef struct doc {
        Lista_Dados* dados;
        Lista_Conceitos* conceitos;
}Document;
```

Esta última estrutura serve para guardar o apontador para a lista dos dados e conceitos.

```
    Document* doc;
    Lista_Relacoes* temp;
    int d;
```

Estas são as variáveis globais utilizadas para guardar a informação.

4.2.2 Símbolos Terminais e Não Terminais

Aqui apresentamos os tipos que vamos atribuir aos nossos símbolos presentes na nossa gramática.

```
%union {
        char* c;
        char ch;
        int n;
}
```

Os símbolos terminais são colocados no inicio do ficheiro da seguinte forma:

```
%token<c> LINGUA TITULO ID TERMO
```

Os símbolos não terminais apresentam-se da seguinte forma:

```
%type<c> Thesaurus Metadados Linguagens Base Inversos Inverso
Conceitos Conceito Relacoes Relacao Termos
```

4.2.3 Definição da Gramática

Nesta secção apresentamos a definição da gramática da nossa solução

```
Thesaurus : Metadados Conceitos {asprintf(&$$, "%s : %s", $1,$2);}                    ;
```

No inicio da nossa gramática temos os símbolos não terminais **Theasaurus**,**Metadados** e **Conceitos**. A ação semântica representada guarda os valores de **Metadados** e **Conceitos**.

```
Metadados : Linguagens Base Inversos {asprintf(&$$, "%s ; %s ; %s", $1,$2,$3);}
            ;
```

Aqui temos a definição de **Metadados** que é constituído por **Linguagens**, **Base** e **Inversos**. Esta ação semântica representada guarda os valores de **Linguagens**, **Base** e **Inversos**.

```
Linguagens : LINGUA {asprintf(&$$, "%s", $1);adicionaLingua(doc,$$);}
    | Linguagens LINGUA {asprintf(&$$, "%s %s", $1,$2);d = adicionaLingua(doc,$2);}
  ;
```

Aqui temos a definição de linguagens, que pode conter só uma **LINGUA** que é um símbolo Terminal ou então ser constituído por várias **Linguagens**. Aqui como já aparece o símbolo terminal temos de guardar o valor na estrutura criada.

```
Base : LINGUA {asprintf(&$$, "%s", $1);adicionaBaseLang(doc,$$);}
     ;
```

Aqui temos a definição de **Base** que é referente a língua de base, sendo constituída pelo símbolo terminal **LINGUA**. Como no ponto anterior adicionamos o valor na estrutura criada.

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

Aqui temos a definição de **Inversos** que pode ser constituído só por um **Inverso** ou por vários **Inversos**. Na ação semântica guardamos os valores.

```
Inverso : ID ID {asprintf(&$$, "%s %s", $1,$2);adicionaInverso(doc,$1,$2);}
                ;
```

Na definição de **Inverso**, temos que ele é constituído por dois símbolos terminais **ID**, e como tal adicionamos na Estrutura relativa às relações inversas.

```
Conceitos : Conceito {asprintf(&$$, "%s", $1);}
                  | Conceitos Conceito {asprintf(&$$, "%s / %s", $1,$2);}
                  ;
```

Apresentamos agora a definição de **Conceitos**, que pode ser constituído por **Conceito** ou por vários **Conceitos**. Vamos guardando cada conceito como ateriormente.

```
Conceito : TITULO Relacoes {asprintf(&$$, "%s %s", $1,$2);adicionaConceito(doc,$1);}
                 ;
```

Conceito é definido pelo símbolo terminal **TITULO** e pelo não terminal **Relacoes**. Na ação semântica guardamos o *TITULO* na estrutura.

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

A definição de **Relacoes** consiste em uma **Relacao** ou várias **Relacoes**.

```
    Relacao : ID Termos        {asprintf(&$$, "%s : %s", $1,$2);adicionaRelacao(doc,$1);}
                ;
```

Relacao é definida pelo símbolo terminal **ID** e pelo símbolo não terminal **Termos**. A ação semântica adiciona o **ID** e guarda o **Termos**.

```
Termos : TERMO {asprintf(&$$, "%s", $1);adicionaTermo(doc,$1);}
           | Termos TERMO {asprintf(&$$, "%s , %s", $1,$2);adicionaTermo(doc,$2);}
           ;
```

Por último temos a definição de **Termos** que é constituído pelo símbolo terminal **TERMO** ou por vários **Termos**. A ação semântica guarda o valor de **TERMO**.

Capítulo 5

Analise de Resultados

Neste capítulo apresentamos o output gerado pelo nosso programa.

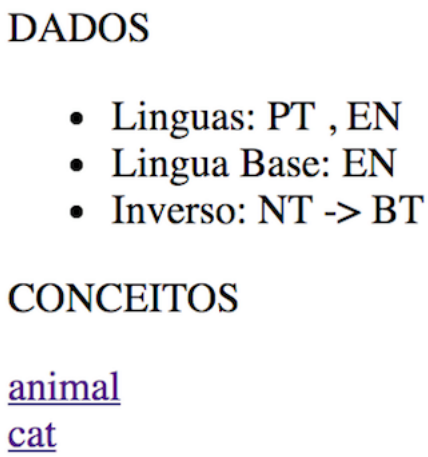

Figura 5.1:

Em cima temos a imagem inicial do output. Agora a página Animal resultante do output. E por último a

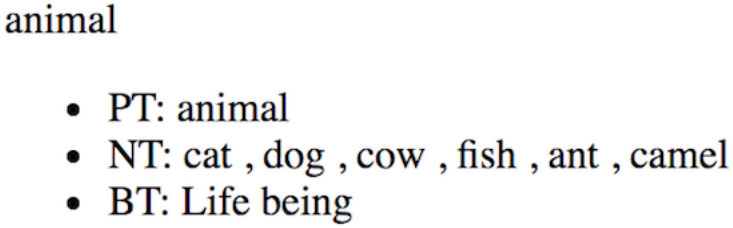

Figura 5.2:

pagina *cat* criada.

Depois de analisarmos os resultados achamos que concluímos todos os requisitos propostos pelo enunciado, tivemos que criar os nossos próprios ficheiros de exemplo, de modo a corresponder com o nosso analisador léxico e com a nossa gramática.

cat

- PT: gato
- BT: animal
- SN: animal que tem sete vidas e meia

Figura 5.3:

Capítulo 6

Conclusão

A elaboração do terceiro trabalho prático desta unidade curricular foi um bom modo de aprofundarmos os conhecimentos obtidos nas aulas acerca de Gramáticas Tradutoras e acerca de *Yacc*.

As principais dificuldades encontradas foram sobretudo na forma como estruturar o nosso pensamento, de modo a criar a gramática, bem como, o analisador léxico.

O trabalho embora que completo e funcional, poderia ter funções extras implementadas. Apercebemo-nos que poderíamos ter criado mais exemplos de forma a ter ficheiros diferentes.

Não obstante, o trabalho cumpre todos os requisitos pedidos de forma eficaz, concluindo então que, embora pudesse estar mais completo trata-se dum trabalho competente.