# Concretização da Estrutura

digitalHumanity

## 1 Introdução

Anteriormente verificou-se que qualquer potencial estrutura sobre a qual a primitiva *digitalHumanity* assente sobre, deve encapsular uma estrutura que permita acessos eficientes.

Ademais, chegou-se a conclusão de que, qualquer estrutura visada neste processo teria de ter um enorme foco nas relações, repartindo em o sistema em duas componentes, relações reservadas e relações não reservadas. Relações não reservadas são aquelas definidas pelas próprias pessoas. Por exemplo, predicados do tipo `filho_de` ou `pai_de`.

Porém, há uma distinta diferença entre estas. Há relações reservadas só de relações entre relação (`:synonym`, `:inverseOf`), sendo que estas devem surgir nos metadados. Porém, e em contrapartida, existem predicados reservados que são entre conceitos, sendo indistinguives das demais relações não reservadas. Por essa razão, estes dois tipos de relações possuem domínios diferentes, razão pela qual a sua mistura pode levar a mais complicações do que simplificações. Mas mais a frente veremos isso com atenção.

De seguida, surge também a necessidade de imbutir variáveis no sistema, penso que, apesar de parecer complicado, este mecanismo pode ser extremamente fácil de se implementar. Se conseguir definir os metadados das variaveis ao nível do analizador léxico, então só temos de enviar para o *yacc* o conteúdo correspondente daquele conceito, depois de acedida a tabela de hash. Tendo em conta que temos exatamente isto do trabalho anterior, a transicao seria muito *smooth*. Mas será que isto funciona? Acho que não, mas mais a frente analiso com mais detalhe. Porém, acho que é importamente atacar este problema imediatamente, até porque a gramática desenvolvida pode vir a depender diretamente deste mecanismo.

## 2 Como incluir metadados?

Pela leitura dos parágrafos acima, é possível chegar à conclusão de que, neste contexto, metadados correspondente tanto a meta-variaveis, como a estabelec-

imento de relações reservadas (sobre relações). Como tal, podemos definir a gramática dos metadados como sendo simplesmente;

```
Meta : /* Vazio */
     | Meta MetaElem
     ;

MetaElem : Pal '=' PalList
         | ':' Pal ':' Pal ':' Pal
         ;

PalList : Pal
        | PalList Pal
        ;
```

O excerto acima permite indica que é possível definir a secção `Meta` como sendo um conjunto de `MetaElem`, sendo que estes tanto podem ser atribuições como explicitações de relações reservadas.

Portanto, a nossa ideia de obter metadados de acordo com o analizador léxico, vai complementamente por agua abaixo, isto porque, por redução ao absurdo, se tivessemos os metadados armazenados no analisador léxico, então teriamos de ter uma forma de comunicar no sentido contrário ao que é feito normalmente, e transmitir metadados ao analizador léxico, para que este o possa armazenar. Isto é obviamente inviável, pelo que teremos de depender do analisador léxico apenas para o seu objetivo principal, enviar tokens e não receber nada.

Por essa mesma razão, os metadados tem de ser inseridos ao nivel do yacc. Tirando partido de pseudo-código, podemos facilmente estender este predicado e ficar com o seguinte. Assumindo P como sendo o conjunto de palavras que compõem o dicionário de metadados, e M como sendo a tabela de hash que associa a cada relação um tuplo de origem e destino.

De qualquer das formas, nunca nos podemos esquecer que `PalList` deve ter um tipo proprio, se forma a tornar possivel preencher um buffer com a informação desta lista de palavras.

```
Meta : /* Vazio */
     | Meta MetaElem
     ;

MetaElem : Pal '=' PalList  {add ($1,$3) to P;}
         | ':' Pal ':' Pal ':' Pal {add ($2,$6) to M($4);}
         ;

PalList : Pal           {$$ = $1;}
        | PalList Pal   {$$ = concat($$,$1);}
        ;
```

Assim sendo, já temos uma forma de preencher o buffer da lista de palavras, bem como uma forma de inserir os metadados na respectiva tabela.

Com tudo isto em conta, penso que o assunto dos metadados está terminado. Porém, pode ser importante salientar que a tabela de hash M deverá ter como valor uma outra tabela de hash, que evite a duplicação de definições repetidas, que no futuro podem vir a causar grandes danos em termos de eficiência.

Com este problema resolvido, conseguimos obter os seguintes dois objetivos:

Trazer um pouco do trabalho anterior para o presente, apresentando assim um mecanismo agradável de armazenamento de variaveis. Porém, falta ainda perceber de que forma é que tiramos partido dos valores que estão armazenados nesta tabela de metainformação. Como tal, bastará desenvolver uma ideia de `ComplexPal`, que tanto pode ser uma palavra como uma variavel. É de notar que estamos a passar do estado em que consideramos apenas 1 *token* para agora considerar 2 *tokens* com 1 *type*. Com esta metodologia poderemos também muito mais facilmente encapsular erros caso a variavel que tenha sido encontrada não exista armazenada nos metadados. Então, o predicado acima passa simplesmente a ser:

“‘bison Meta : /* Vazio */ | Meta MetaElem ;

MetaElem : Pal ‘=’ PalList {add ($1,$3) to P;} | ‘:’ CPal ‘:’ CPal ‘:’ CPal {add ($2,$6) to M($4);} ;

PalList : CPal {

=

1; | PalList CPal

= concat($$,$1);} ;

CPal : Pal {

=

1; | Var

= look up $1 in P; error if $1 not in P;} ; “‘é transmitida para

Isto implica que a definição de um metadado deve ser uma palavra simples, e não pode ser uma palavra complexa. De agora em diante, de forma a encapsular o comportamente, devemos sempre nos referenciar a um conjunto de palavras como `PalList` e a uma palavra complexa como `CPal`.

Falta ainda teorizar de que forma vamos# Como incluir metadados? lidar com as relações reservadas que ocorrem nos metadados. A minha ideia é tanter simular um mecanismo de *dynamic dispatch* de funções trabalhadoras sobre cada uma das relações afetadas. Porém, só a futura implementação da secção do `Caderno` poderá dizer se esta implementação é ou não viável.

# 3 Como representar Cadernos?

Cadernos são compostos por 0 ou mais pares de (documento, triplos), associando a cada documento respectivos triplos. Cada documento é composto por (conceito, titulo, texto) e cada triplo é composto por (sujeito, relação, objeto). Como tal, podemos muito abstratamente definir esta gramática da seguinte forma, sem nunca esquecer, tal como surge no enunciado do TP. Qe podem de facto existir operações lógicas de disjunção ou junção, como por exemplo:

```
:OnorioL
    a     :Person ,
            :Mecenas_cultural ;
    :Name "Coisas de Limão" ;
    :birth_place "Pombal" ;
    :img "flag.jpg" .

:CarolinaDA
    :filho_de :OnorioL.
```

Há muitas componentes de interesse nesta definição, começando pelo mais simples, verificamos que `"Coisas de Limão"` deve ser tratado como se de uma única palavra se trata-se, sendo responsabilidade do analizador léxico o correto processamento deste. A expressão regular necessária não é muito complexa, simplesmente seria `"[^"]`, omitindo claro os espaços brancos, de seguida o analisador léxico teria também a string correta ao processador, retirando as aspas. Também uma operação simples.

De seguida, um comportamento também de interesse é que um dado conceito pode estar associado a várias relações e objetos simultaneamente. Porém, também se verifica que um dado objeto pode conter vários outros objetos, separados entre si por virgulas. A demais, ou pares (relação, objeto), podem ser separados entre si por ‘;’ para um mesmo sujeito, sendo obrigatorio que se termine a especificação dos pares com a letra ‘.’.

Então resumindo, para cada triplo temos um sujeito que pode estar associado a uma lista de pares (relação, objeto), sendo que o objeto pode ser uma combinação de 1 ou mais objetos separados entre si por ‘,’.

Podemos então, de forma geral, obter o seguinte resultado, em pseudo-código, da especificação da gramática em questão.

```
Caderno : /* Vazio */
        | Caderno Documento Triplos
        ;

Documento : Conceito Titulo Texto
          ;
```

```
Triplos : /* Vazio */
        | Triplos Sujeito ParesRelacao
        ;

ParesRelacao : PRList '.'
             ;

PRList : PRListElem
       | PRList ';' PRListElem
       ;

PRListElem : Relacao ComplexObject
           ;

ComplexObject : Object
              | ComplexObject ',' Object
              ;
```

Esta gramática, apesar de simplificada, é capaz de captar todos os comportamentos essênciais e necessários ao funcionamento adequado. Claro que falta agora especificar de que forma é que esta gramática pode ser utilizada a nosso favor no funcionamento interno.