# Dicionários dinâmicos multi-fonte

**José João Dias de Almeida**

Dissertação submetida à Universidade do Minho para obtenção do grau de Doutor em Informática, elaborada sob a orientação de Pedro Rangel Henriques

Departamento de Informática
Escola de Engenharia
Universidade do Minho

Braga, 2003

## Resumo

O tema central desta dissertação é a especificação de dicionários. Defende-se a aproximação de definir consultas a Dicionário com base na activação de consultas a várias fontes (capazes de produzir informação diferente, mas complementar, referente a um termo) e junção dos resultados delas provenientes.

As "fontes a colar" têm por base ferramentas várias de processamento de linguagem natural (funções) e recursos de linguagem natural (constantes).

Chamaremos *fonte de informação* a algo capaz de produzir *informação* associada a um termo. Uma fonte pode ser um simples recurso de processamento de linguagem natural ou envolver um conjunto de ferramentas e recursos.

A especificação formal das ferramentas é um procedimento de abstracção que, para além de permitir entender e discutir as ferramentas, cria uma plataforma comum a todas elas e portanto um domínio sintáctico-semântico onde possam cooperar. Além disso, a abstracção facilita a comunicação e a cooperação.

A especificação formal das ferramentas vai permitir descrever o seu funcionamento e encapsular detalhes menos relevantes. Permitirá também definir uma álgebra de ferramentas, recursos gerais, recursos locais, e um tipo de dados comum.

Para se conseguir juntar as informações há necessidade de usar um tipo de dados comum e definir funções de conciliação que permitam harmonizar informações provenientes das várias fontes.

A definição de funções *estratégia* capazes de descrever o modo como serão consultadas as diversas fontes de informação é também um elemento indispensável para que se possa escolher um compromisso aceitável de custo/qualidade, completude.

Para descrição da interpelação às várias fontes de informação será utilizada a *linguagem de especificação* CAMILA.

Para fazer a adaptação de formatos e encapsular detalhes sintácticos será utilizada *linguagem de scripting* PERL.

Sempre que possível, tentou-se construir e disponibilizar recursos e ferramentas de processamento de linguagem natural de modo a ajudar a colmatar algumas necessidades gerais existentes.

**Palavras chave:** Dicionários, PLN, DSL, ontologias, bibliotecas digitais

# Abstract

The subject of this thesis is the specification of dictionaries. The approach used here is to define dictionary lookup as the consultation of several sources (capable of producing different but complementary information about a term) and merging their results.

The "sources to join" are based on several natural language processing tools (functions) and natural language resources (constants).

We will call *source* to anything that can produce information about a given term. A source can be as simple as a natural language processing resource, or as complex as a result of the collaboration of several tools and resources.

Formal specification of tools is an abstraction process that helps the process of understanding and discussing. At the same time, it builds a common platform, where all the tools share the same syntactic-semantic domain and collaborate, thus helping in communication and cooperation.

The formal specification of the tools builds a description of their behavior but hides their less relevant details. It allows the definition of an algebra of tools, general resources, local resources, and a common datatype.

To join all the results it is necessary to use a common universal datatype and define conciliation functions to merge the information built from all the information sources.

It is crucial to be able to define *strategy* functions capable of describing how the sources should be consulted in order to achieve a trade-off between cost, quality and completeness.

To describe functions and processes, the CAMILA specification language will be used.

To adapt and change formats, the PERL scripting language will be used.

Due to the sparse availability of resources and tools, an effort has been made to open-source all the resources developped during this work.

**Keywords:** Dictionaries, NLP, DSL, ontologies, digital libraries

## Agradecimentos

Ao Pedro Henriques, pela orientação, pelo apoio, pela capacidade de encontrar o lado positivo e pela amizade. Por gostar do Manuel Freire com suas *Pedras Filosofais* e *Raizes do Pensamento*.

Ao Departamento de Informática e aos seus directores Esgalhado Valença, Francisco Moura e Pedro Henriques, por me terem sempre apoiado e proporcionado todas as condições de trabalho.

Ao José Nuno Oliveira, ao Luís Barbosa e ao Luís Neves (núcleo duro do CAMILA), por toda uma criatividade e capacidade de entusiasmo.

Ao Luís Barbosa, por tantas vezes em que me ajudou a assentar ideias, pela imprevisibilidade, por o *Tratado de Estilo* de 1863, o *Compêndio Grammatical da Língua Portugueza* e os *Dicionários* de época nitidamente pré-histórica.

Aos meus colegas Jorge Rocha, João Saraiva e José Carlos do grupo de EPL, por uma série de ensinamentos nas mais diversas áreas.

Ao Alberto Simões, por um conjunto de projectos desenvolvidos em conjunto, pelas páginas culturais que temos mantido, por um constante voluntarismo e capacidade de trabalho, e pelos dióspiros.

Ao José Bernardo, por ser um excelente interlocutor, pela ajuda em vários domínios, pela capacidade de ser sensato defendendo sempre coisas não razoáveis[1].

Ao Rui Mendes, Bacelar, Paulo Novais, Alberto Proença, Olga e Macedo, pelas sugestões e apoio na recta final da escrita da tese.

Ao Ulisses Pinto e ao Ricardo Reis, pela ajuda na implementação de várias ferramentas.

Ao Paulo Rocha, pela sua patológica capacidade de coleccionar línguas.

À equipa dos Alfarrabistas, pela aventura intelectual que me têm proporcionado.

Ao DI, pela capacidade de saber falar muitas línguas, e de permitir a coexistência de Ciência, Engenharia e Arte; por constituir um local onde é agradável estar[2].

Às minhas cachopas Luísa, Mariana e Luisinha e ao meu minorca João, nunca será possível agradecer o apoio, a paciência e o amor.

[1] e por às vezes trazer cá o Manel

[2] e por ter um bar civilizado

À Luísa,

*que tanto me apoiou, e a quem se deve grande parte da pontuação que a tese inclui,*

aos meus pais, Duarte e Isabel

*que são, de longe, os melhores pais do mundo*

# Conteúdo

# Capítulo 1

# Introdução

*I like words more than numbers, and I always did*
*Paul R. Halmos,*
*"I want to be a mathematician"*

Ao longo deste capítulo, tecem-se algumas considerações gerais ligadas à noção de dicionário[1] salientando-se que os dicionários têm uma componente dinâmica e retratam uma visão *termo* $\rightharpoonup$ informação sobre uma realidade distribuída por uma grande variedade de fontes.

Esta visão conduz à necessidade de arranjar uma álgebra capaz de compor um conjunto heterogéneo de recursos e ferramentas. Essa álgebra de composição constitui o centro desta dissertação.

Naturalmente que a existência de recursos e ferramentas é fulcral ao funcionamento da abordagem referida. Neste capítulo, inclui-se um breve introdução às *linguagens de domínio específico* pela sua importância no que refere à especificação e criação de ferramentas.

No final do capítulo, apresenta-se um breve resumo das principais contribuições desta dissertação.

[1] Nesta dissertação *dicionário* designa um conceito mais abrangente que o usual.

## 1.1 Dicionários: conceitos e posicionamento

Nesta secção, pretende-se apresentar algumas características gerais do problema genérico da especificação, construção e uso de dicionários e estabelecer uma posição nesse complexo mapa de conceitos.

Um dicionário estabelece uma correspondência entre termo e informação a ele associada. As noções de termo e de informação, nos dicionários habituais, são uma questão complexa e delicada (Sanromán, 2000). A noção de termo poderá variar muito conforme os casos concretos, podendo ser uma palavra ou algo mais complexo, como lexias compostas, locuções, etc. A noção de informação associada também vai ser muito variada tanto ao nível de complexidade estrutural como de dimensão.

Uma preocupação complexa mas sempre presente é a da qualidade[2] dos dicionários e por consequência a questão da informação em excesso, da informação em omissão e do ruído: uma qualidade máxima e um tentativa de obter um compromisso equilibrado entre as questões nela contidas, são naturalmente os objectivos últimos a atingir.

### 1.1.1 Termos

O uso de termos como domínio do dicionário não é um acaso.

> O uso de termos constitui um bom/natural ponto de entrada na complexa rede de conceitos que constitui o conhecimento que possa ser expresso em palavras, já que tira partido das associações criadas no cérebro quando da aprendizagem dos termos.

Devido ao facto de as estruturas cerebrais, os termos e suas relações e a linguagem natural estarem muito próximas de nós, é por vezes difícil distinguir os campos da representação de conhecimento da linguagem natural.

### 1.1.2 Uso múltiplo

A construção de dicionários é uma tarefa complexa que exige uma enorme disciplina formal dificilmente conseguida sem um projecto elaborado rigorosamente e suportado por ferramentas que ajudem na sua validação. Essa

[2] Apesar da importância desta questão, não será formalmente abordada devido a ter uma importante componente filosófica, que foge ao âmbito deste trabalho.

construção envolve normalmente um grande esforço: grandes equipas, necessidade de tempo, custos elevados.

Paralelamente, a evolução tecnológica tem vindo a diversificar as possibilidades de uso da informação em geral.

Neste contexto, surge como natural a preocupação de permitir que a informação criada possa ser usada para vários fins – multi-uso.

| Para que possa haver multi-uso, é necessário dispor de riqueza estrutural na informação associada aos termos. |
|---|

A estrutura da informação deve ser, tanto quanto possível, reflexo das propriedades semânticas existentes, e deve ser o mais possível independente de um uso específico.

Essa riqueza estrutural, pode ser usada também para melhorar a coerência, já que vai permitir que certas tarefas associadas à formatação, introdução de símbolos especiais, etc., possam ser automatizadas.

| A riqueza estrutural da informação permite melhor coerência. |
|---|

**Exemplo 1:** Discrepâncias

Ao fazer a análise reversa de um dicionário a partir de um texto a enviar para foto-composição, que se sabe ter sido sujeito a um meticuloso processo de revisão, constatou-se que ainda assim aparecem discrepâncias a vários níveis:

– diferentes maneiras de representar graficamente os conceitos (ex: uma vedeta aparece realçada com um tipo de letra de tamanho 15 mas por vezes o tamanho usada é 13 ou 14; As acepções aparecem separadas por um caracter (por exemplo $\diamond$) que por vezes falta, outras vezes aparece de tamanho diferente, outras ainda substituído por outro semelhante);

– assimetrias a nível da quantidade informação fornecida para termos semelhantes;

– assimetrias entre os modos de definir as variadas instâncias da mesma classe;

Isto resulta, naturalmente, de um certo desgaste, de haver pessoas diferentes na equipa de construção, da existência de, por vezes, grandes distâncias temporais entre a produção dos diferentes verbetes.

A existência de estrutura explícita permite reduzir algumas das discrepâncias, permitindo por exemplo, automatizar a produção da apresentação gráfica, reduzindo portanto as incoerências que surjam nessa fase.

### 1.1.3 Dicionário como uma vista sobre o conhecimento

Um dicionário corresponde a uma visão do tipo

$$termo \rightharpoonup inf$$

sobre uma realidade estruturalmente mais complexa.

O facto de o dicionário estabelecer uma correspondência unívoca entre termo e informação não faz com que o conhecimento nele armazenado seja simplesmente esse.

**Exemplo 2:** Dicionário de Camilo Castelo Branco

Observando o exemplo do Dicionário de Camilo Castelo Branco (Cabral, 1988), vê-se facilmente que não se trata de um trabalho de explicação do significado de termos.

Aparentemente, foi feita a colecção de muita informação acerca desse extraordinário escritor; essa informação inclui uma grande variedade de espécies, desde conhecimento acerca de livros, de expressões, de lugares, de eventos, etc.

No entanto uma colecção de informação não é directamente manuseável. Procedeu-se, por isso, à divisão em sub-unidades que seguidamente foram associadas a termos referentes ao seu conteúdo. Esta associação termo-unidade constitui um dicionário.

Por último, foi feita a impressão em livro, ordenada alfabeticamente por termo.

A informação coleccionada não é intrinsecamente um dicionário. A sua apresentação corresponde a uma "dicionarização"da informação.

| |
|---|
| Dicionário não corresponde normalmente à estrutura intrínseca da informação nele contida, mas sim a uma visão sobre ela. |

### 1.1.4 Interdependência da informação

De facto, muita da informação associada a diferentes termos é dependente entre si. Isso advém de haver outras relações estruturais sobre a informação.

Nos exemplos que se seguem, apresentam-se casos de relações que envolvem mais que um termo levando a que as informações a eles associadas não sejam independentes entre si.

**Exemplo 3:** listas de sinónimos

O conjunto $\{gato, erro, gralha\}$ pode ser usado para descrever uma acepção semântica comum para cada um das três entradas (Fellbaum, 1998, wordnet). Por coerência, quando se der a forma de dicionário a esta informação, seria de esperar que as respectivas entradas tivessem a mesma definição semântica e que o conjunto de sinónimos de cada uma delas preservasse a propriedade simétrica normalmente associada à sinonímia.

Ou seja: se houver preocupações de coerência, a forma *dicionário* de mostrar a informação apresenta redundâncias que derivam das propriedades de certas relações, como por exemplo a sinonímia, existentes no conhecimento retratado.

**Exemplo 4:** Conjuntos/sequências fechados de instâncias

$ano = \{Janeiro, Fevereiro, ..., Dezembro\}$ – O facto de *ano* ser constituído por um conjunto de meses vai implicar o aparecimento de muita repetição nas informações associadas aos termos `ano`, `mês`, `Janeiro`, ... `Dezembro`.

Ou seja: se houver preocupações de coerência, a forma *dicionário* de mostrar a informação ligada pela relação classe instância (ordenada ou não), apresenta redundâncias, havendo contudo possibilidade de geração automática dessas repetições.

**Exemplo 5:** Expressões idiomáticas

Nos dicionários habituais, em que os termos são palavras simples, as expressões multi-palavra, como por exemplo a expressão idiomática `vender gato por lebre`, deveriam aparecer replicadas nas informações associadas aos lemas dos várias termos constituintes (`vender`, `gato`, `lebre`).

Mais uma vez, está-se perante repetições sistemáticas e passíveis de serem obtidas automaticamente a partir de uma forma mais abstracta e conceptual.

Da análise destes exemplos, e de muitos outros que se poderia apresentar, fica patente que:

- As informações associadas a diferentes termos, são muitas vezes dependentes entre si.
- Essa dependência reflecte-se na necessidade de repetições.
- A construção manual das repetições cria frequentemente incoerências, incompletudes, dualidade de critérios.

### 1.1.5 Dicionários dinâmicos e multi-fonte

Um dicionário é uma fonte muito importante de informação, mas nunca contém toda a informação útil acerca de um termo.

**Exemplo 6:** Informação associada a ovo

A informação associada a *ovo* (Teixeira, 1996) (aparentemente uma coisa simples), pode incluir:

- óvulos e espermatozóides
- pintos e galinhas
- ovos podres e maus oradores
- omeletes, ovos estrelados, sertãs e seu modo de preparação
- caixas de 12, mercearias, supermercados e preços
- pudim Abade de Priscos, receitas conventuais
- a sociedade inteira...

Como foi referido, o conhecimento existente num dicionário é frequentemente repartido por múltiplas entradas interdependentes, contendo informação frequentemente heterogénea – multi-espécie.

Essas mesmas relações aconselham a que haja uma variedade de fontes de informação a serem juntas para constituírem ou gerarem a informação pretendida.

Fazendo uma comparação (simplista) com as bases de dados relacionais, um dicionário corresponde à relação universal; assumir a existência de várias fontes de informação corresponde a procurar colocar os dados numa forma normal.

As tabelas normalizadas têm algumas vantagens do ponto de vista de manutenção: menor quantidade de redundância, mais fácil verificação de algumas propriedades pretendidas, mais fácil edição de cada tabela. O facto de os dados estarem divididos em várias tabelas, não impede que seja possível vê-los como um dicionário:

- é possível juntá-los numa relação universal (i.e. calcular o dicionário todo a partir das suas partes) ou
- é possível responder a perguntas acerca de um termo específico juntando apenas as partes indispensáveis.

Em vez de tabelas relacionais, pretende-se poder interligar um conjunto multi-espécie de recursos (ex. dicionários, mas também conjuntos de frases exemplo, listas de provérbios) processados e transformados por ferramentas (tanto comandos do sistema operativo, como aplicações em geral, como outras construídas expressamente para o fim em causa).

### 1.1.6 Noção de dicionário nesta dissertação

Como se tem vindo a mostrar, a noção de dicionário usada neste texto é muito abrangente. Para além dos dicionários convencionais – associados à descrição de propriedades das palavras (monolingue ou multilingue) – considerar-se-á, como se disse no início, dicionário como sendo:

> Definição: Dicionário é tudo o que estabelece uma correspondência entre termos e informação associada (estática ou dinamicamente).

Isto inclui:

- dicionários convencionais
- terminologias
- enciclopédias[3]
- thesauri, redes semânticas e ontologias
- analisadores morfológicos – programas que, dada uma palavra, retornam, como informação associada, um conjunto de características correspondentes às análises alternativas dessa palavra

[3] A palavra enciclopédia tem origem idêntica a ciclos – ligações – estando intrinsecamente ligada à vontade de estabelecer relações e ligações entre as coisas.

- comandos do Unix como o `man` – dado um termo (um comando, formato ou biblioteca) devolve o respectivo manual
- qualquer combinação de comandos, filtros e recursos, capaz de, dado um termo, devolver informação a ele associada.

> Neste documento defende-se a possibilidade de especificar e tratar dicionários como um processo de junção de fontes de informação.

Cada fonte de informação é o resultado de uma expressão em que:

- as constantes são recursos (de processamento de linguagem natural (PLN)) – Ex: dicionários tradicionais, ficheiros, bases de dados, páginas HTML, etc.
- as funções são comandos do sistema operativo e ferramentas específicas que usam os referidos recursos, ou outras expressões, como argumentos.

Saliente-se portanto que se está a tomar como preocupação central desta dissertação a teoria e prática da criação de bancadas de construção de dicionários, sendo a noção de dicionário aqui tomada num sentido mais lato do que o habitual.

Neste contexto, pouco do que se apresentará, estará directamente ligado aos dicionários convencionais, o que justifica a pequena dimensão do Capítulo 3 onde se referem apenas alguns trabalhos/marcos históricos ligados aos dicionários convencionais e a léxicos de processamento de linguagem natural.

## 1.2 Linguagens de domínio específico

Referiu-se na secção anterior a vontade de ver as ferramentas e comandos do sistema operativo como funções da álgebra de construção de dicionários. Um importante caso particular destas, são as ferramentas geradoras de ferramentas (por exemplo, geradores de parsers, geradores de analisadores léxicos) que se comportam como funções de ordem superior. De entre as ferramentas geradoras considerar-se-á de um modo especial as que têm a ver com a produção de linguagens (de programação, de especificação) de domínio específico.

### 1.2.1 Definição e conceitos

A definição habitual de **Linguagens de domínio específico** (DSL)[4] (van Deursen, Klint, and Visser, 2000) é:

> "Uma linguagem de domínio específico é uma linguagem de programação ou uma linguagem de especificação executável que oferece, através das noções de abstracções adequadas, poder expressivo focado em (e normalmente restrito a) um domínio específico."

Cada linguagem de domínio específico $L$ (e ferramentas associadas), está a povoar o ambiente de criação de dicionários com:

- uma nova espécie: o tipo documentos escritos em $L$
- constantes: cada documento escrito em $L$
- várias ferramentas (=funções): resultado de processar cada documento da linguagem $L$
- uma ou mais funções de ordem superior (relacionadas com os vários processadores de $L$) que, a partir de um elemento de $L$, produzem uma ferramenta.

A definição de pequenas linguagens de programação para resolução de problemas em domínios específicos, é algo tão usual que passa quase desapercebido. Algumas dessas linguagens são a razão do sucesso de alguns ambientes/sistemas operativos. Imagine-se o que seria o Unix sem as linguagens associadas a, por exemplo, os comandos `make`, `lex`, `yacc` (Bentley, 1986).

No sentido de equipar um ambiente com capacidade para resolver problemas numa área específica, várias abordagens tem sido utilizadas:

**Bibliotecas de funções** – esta abordagem consiste em especificar e construir um conjunto de funções que correspondam às tarefas básicas referentes aos elementos de domínio específico.

**Componentes orientadas aos objectos** – esta abordagem continua a abordagem anterior com uma melhor arrumação das funções (as bibliotecas são um pouco planas) e da sua invocação. Complementarmente, fornece maneiras mais elegantes de compor bibliotecas.

**Linguagens de domínio específico** – nesta abordagem, tenta-se arrumar as funções em álgebras e associar-lhes uma sintaxe concreta, normalmente declarativa, que ofereça bom poder expressivo nesse domínio, escondendo os detalhes da biblioteca de funções subjacente. Muitas vezes, os programas escritos são transformados para invocações de funções.

[4] Neste documento utilizar-se-á a sigla inglesa correspondente a Domain-Specific Languages.

O uso de uma boa DSL permite exprimir conceitos num idioma elegante que deverá ser desenhado de acordo com as maneiras habituais de expressar o conhecimento nesse domínio. Nesse sentido, uma boa DSL permite que um programa nela escrito seja conciso e funcione de documentação.

### 1.2.2 Implementação de DSL

A implementação de processadores de DSL pode usar várias estratégias:

**escrita de um novo compilador ou interpretador** (usando as ferramentas habituais na escrita de compiladores (Aho, Sethi, and Ullman, 1986; Waite and Goos, 1984; Bentley, 1986)). Esta abordagem permite uma máxima liberdade de escolha de notação e da semântica, mas envolve algum custo.

**criação de linguagens embebidas / bibliotecas de domínio específico** – corresponde à criação de uma biblioteca de funções específicas ao domínio e redefinição/extensão à sintaxe concreta da linguagem de base usando a funcionalidade de redefinição de sintaxe que a linguagem de base oferece. Esta abordagem obriga muitas vezes a compromissos entre a linguagem pretendida e a possível/fácil de obter usando as limitações da linguagem de base.

**tradução por pré-processamento ou por processamento de macros** – Nesta abordagem, os novos construtores são traduzidos para a linguagem de base através dum pré-processador, sendo seguidamente executados/interpretados. Esta abordagem é simples embora, por vezes, deficiente, já que a verificação sintáctica e optimizações não são efectuadas no domínio específico.

**escrita de compiladores/interpretadores extensíveis** – Esta abordagem é semelhante à anterior mas a fase de pré-processamento é integrada no compilador, permitindo melhor verificação sintáctica. Exemplo: as extensões ao Tcl (Ousterhout, 1994; Ousterhout, 1998).

Como exercício de especificação que é, o desenho de DSLs e a construção de processadores para DSLs, constitui também um bom meio de reflectir, especificar e sintetizar os conceitos fundamentais de um determinado domínio e como tal constitui um meio de assimilação da área subjacente.

A abordagem das DSL foi amplamente usada ao longo deste trabalho para permitir uma construção eficiente de processadores eficazes.

## 1.3 Estrutura geral deste documento

Ao longo da introdução, procurou-se fazer uma breve apresentação do assunto *dicionários* no sentido de definir um posicionamento perante o problema.

No Capítulo 2, é feita uma apresentação da notação usada na dissertação, referindo-se sucintamente alguns detalhes do ambiente CAMILA – a ferramenta usada no projecto como suporte à especificação, prototipagem e geração de texto matemático. Neste capítulo é definido o tipo universal estrutura de facetas que é, como se disse, um elemento crítico na abordagem apresentada.

No Capítulo 3, faz-se uma breve referência a alguns marcos históricos na dicionarística e em áreas a ela ligadas.

No Capítulo 4, analisa-se a funcionalidade associada aos dicionários (tanto a funcionalidade normalmente disponível como outras menos claras) e descreve-se um primeiro modelo de dicionário.

No Capítulo 5, continuar-se-á a abordagem formal à modelação de dicionários, tipos e funcionalidade, sendo detalhada a noção de estratégia, de filtragem e de conciliação.

Seguidamente, no Capítulo 6, apresentar-se-á algumas contribuições associadas a ferramentas de processamento de linguagens. No capítulo seguinte, 7, detalham-se algumas ferramentas construídas, directamente associadas a dicionários. No Capítulo 8, apresentam-se alguns recursos associados a processamento de linguagem natural.

No Capítulo 9, é mostrada a implementação de um módulo que anima uma série de conceitos referidos anteriormente, constituindo uma ferramenta de ajuda à construção e manuseamento de DDMF.

Por último, o capítulo das conclusões.

Sempre que tal fizer sentido, serão apresentados exemplos de dicionários dinâmicos multi-fonte (distribuídos pelos diversos capítulos).

## 1.4 Tese e contribuições

Neste documento, não se terá a preocupação de definir modos óptimos para construção de dicionários convencionais, nem de construir léxicos para PLN.

Ao longo da dissertação, descrever-se-á um estudo de especificação e tratamento de dicionários envolvendo a utilização cooperativa de várias ferramentas capazes de contribuir com informação associada a termos.

Para que as várias ferramentas possam ser juntas, torna-se necessário definir (formalmente):

- um formato, protocolo, ou tipo de dados comum a todas
- para cada ferramenta:
  - a sua estrutura lógica (modelo formal em CAMILA)
  - funções de tradução (da informação a ser passada à ferramenta e reconhecimento das respostas)
- um conjunto de funções que permita fazer a junção das informações parciais
- um mecanismo de estratégia e conciliação:
  - que active a invocação de ferramentas quando necessário
  - que calcule/reescreva partes de informação com base na informação disponível
  - que calcule de forma incremental a relevância/certeza das sub-informações
- um mecanismo de filtragem

Com esta dissertação espera-se defender o seguinte conjunto de contribuições:

- De um ponto de vista "filosófico":
  - a visão de DDMF como meio de especificar e construir soluções para um conjunto de problemas envolvendo dicionários (tomados numa visão abrangente)
  - a utilização da seguinte estratégia de resolução de problemas: (1) dado um problema $p$ (2) procurar um problema $p'$ relacionado, para o qual haja uma solução (um programa ou módulo) *open source* (3) construir o seu modelo abstracto $m = modelo(p')$ (4) analisar as alterações que é necessário efectuar no modelo $m$ para... (5) determinar como é que essas alterações se vão reflectir no código fonte que resolve $p'$.
- Do ponto de vista de ferramentas e recursos construídos:
  - o CAMILA como plataforma de suporte à especificação (aplicado ao problema abordado nesta dissertação).
  - um conjunto de ferramentas e recursos apresentados nos capítulos 6, 7, 8 e 9.

Dado que a noção de dicionário é resultado de um processo de fusão de informação variada, aproveitou-se a generalidade dos casos de estudo para desenvolver/apoiar a produção de informação cultural (com estrutura explícita) de modo a que haja alguma componente adicional de serviço à comunidade geral. São exemplo desta preocupação vários arquivos culturais no Alfarrábio[5]

[5] `http://alfarrabio.di.uminho.pt`

(alguns dos quais brevemente descritos na Secção 8.4).

# Capítulo 2

# Suporte à especificação

> *Porque a minha invenção é um intérprete das noções, uma bússula que nos guiará no oceano das experiências, um inventário das coisas, um quadro dos pensamentos, um microscópio para examinar as coisas presentes, um telescópio para adivinhar as remotas, um Cálculo Geral, uma escrita que cada um lerá na sua própria língua*
>
> *Leibniz, Carta, 1679*

Este capítulo é dedicado à apresentação da plataforma de especificação formal e prototipagem utilizada nesta tese — o sistema CAMILA — desenvolvido em co-autoria pelo autor da dissertação. A ênfase é colocada na introdução da notação e ferramentas de suporte mais directamente utilizadas na tese. Em particular, detalham-se alguns aspectos técnicos em torno das ferramentas de produção de texto associadas. Note-se, porém, que apenas uma parte das potencialidades do CAMILA são efectivamente exploradas nesta dissertação, a saber, o ambiente de prototipagem parcial de componentes, a sua faceta de suporte à reflexão e análise, uma pequena parte de validação sintáctica, e as ferramentas ligadas à produção de texto matemático (de facto, a quase totalidade do texto matemático aqui apresentado foi produzido usando ferramentas CAMILA).

Além da simples questão prática de introduzir a notação seguida, a apresentação do sistema CAMILA serve simultaneamente para realçar uma das bandeiras que é defendida nesta dissertação (e que realmente esteve na base da abordagem que a sustenta):

$$\textit{resolver problemas} \quad \equiv \quad \textit{criar um modelo matemático, raciocinar}$$

*sobre ele e calcular uma solução.*

Trata-se, no fundo, da estratégia que suporta todo o trabalho em Ciências e Engenharia, a que nos habituamos desde os bancos da escola, mas que, surpreendentemente permanece difícil de incorporar na prática do desenvolvimento de *software*.

A segunda parte do capítulo é dedicada à definição de um tipo de dados abstracto, que designamos por *estrutura tipada de facetas*[1], e que será usado como tipo (quase) universal nesta tese. A sua relevância para o conjunto de noções e ferramentas discutidas nos capítulos seguintes, justificam o destaque que aqui lhe damos.

## 2.1 Camila

As linguagens de especificação formal de *software* têm como preocupação a descrição rigorosa e abstracta, *i.e.*, matematicamente segura e a um nível próximo do raciocínio humano, das partes estruturais dos diversos elementos envolvidos num sistema computacional. Em particular, procuram-se descrever as entidades envolvidas e as relações entre os dados iniciais e os resultados pretendidos ao longo de um processo computacional (ou seja, indicar quais os resultados pretendidos mais do que indicar o modo como os resultados devem ser calculados). Ao contrário, nas linguagens de programação é crucial controlar a maneira como os resultados deverão ser calculados para que a eficiência não fique muito deteriorada.

Uma especificação serve ainda outros objectivos importantes ao funcionar como:

- mecanismo de reflexão independente dos detalhes
- linguagem de comunicação entre os elementos de uma equipa de desenvolvimento
- documentação formal de uma ideia que poderá ser refinada numa linguagem de programação, e simultaneamente servir de *caderno de encargos* para definir o comportamento esperado e para eventuais provas de correcção
- base formal que permita genericamente raciocinar matematicamente sobre o texto da especificação.

Conforme se referiu anteriormente, a componente de especificação formal nesta dissertação é suportada pela plataforma Camila — uma linguagem de

[1] Em inglês, typed feature structure.

especificação e um sistema de prototipagem desenvolvido na Universidade do Minho. À adopção desta linguagem não é alheio o facto do autor ter estado directamente envolvido na concepção, implementação e teste deste sistema, ao longo da preparação do seu doutoramento. Apesar de marginal ao tópico central da tese, o sistema CAMILA é em boa parte uma contribuição do autor que acabaria por marcar (e ser marcado por) a investigação reportada nesta tese.

CAMILA (Almeida et al., 1997a; Almeida et al., 1997b) é uma plataforma experimental para o desenvolvimento formal de *software* que se filia na escola dos métodos de especificação *por modelos* (ou *construtiva*) na qual uma especificação é entendida como um *modelo* matemático do problema em mãos, e não como uma *teoria*, *i.e.*, um conjunto de axiomas numa lógica adequada.

Um dos principais objectivos que presidiu à sua concepção foi a necessidade de, através da noção de *prototipagem rápida*, introduzir este tipo de métodos de desenvolvimento rigoroso de sistemas na prática industrial. Daqui resultou que o projecto fosse concebido como uma colecção de pequenas ferramentas, com interfaces sóbrias, mas facilmente articuláveis com aplicações externas (*e.g.*, bases de dados, geradores de interfaces, processadores de documentos, a própria Web). Por outro lado, o desenvolvimento do CAMILA filia-se na exploração da *programação funcional* como ambiente de prototipagem de modelos formais de software, cuja origem remonta ao trabalho pioneiro de P. Henderson (Hendersen, 1984).

A plataforma tem sido utilizada no ensino de métodos formais e no desenvolvimento de alguns projectos relacionados, nomeadamente, com mecanismos de reutilização de código em sistemas CASE (Oliveira, 1995b), projecto de bases de dados temporais, processamento de documentos (Ramalho, Almeida, and Henriques, 1998), concepção de linguagens de descrição de edifícios em arquitectura (Oliveira, 1995a) e, no contexto da presente dissertação, no projecto de dicionários dinâmicos multi-fonte.

Os artigos (Almeida et al., 1997a) e (Almeida et al., 1997b), assim como o manual de referência da linguagem (Barbosa and Almeida, 1995), constituem as referências básicas da plataforma CAMILA. O desenvolvimento do projecto CAMILA é discutido em (Almeida et al., 1998), (Barbosa and Almeida, 1998) cobre as principais características do sistema com o detalhe de um tutorial, enquanto (Barbosa, Barros, and Almeida, 2000) reporta extensões recentes na área da programação *politípica*. O sistema e respectiva documentação estão disponíveis em `http://camila.di.uminho.pt/`.

Nas subsecções seguintes é sumariamente apresentada a linguagem de especificação usada no ambiente CAMILA e descrita a arquitectura base do sistema. A ênfase é colocada nos tópicos que mais directamente são relevantes nesta dissertação.

### 2.1.1 A Linguagem de Especificação

Na estrutura de um sistema computacional, a Engenharia de Software traça uma distinção fundamental entre *entidades*, que representam as fontes de informação, e as suas *transformações*. As primeiras estão na origem das estruturas de dados, as segundas, dos algoritmos. A observação fundamental subjacente aos métodos de especificação e desenvolvimento formal de programas *orientados por modelos*, tais como VDM (Jones, 1986), Z (Spivey, 1989) or B (Lano, 1996), é de que uma dualidade similar surge na definição de uma álgebra (colecção de conjuntos e funções) ou de uma estrutura relacional (onde as funções se relaxam a relações binárias), tornando-as estruturas adequadas para basearem a *semântica* desses sistemas.

A linguagem de especificação CAMILA pode ser encarada como uma versão executável da notação centenária da *teoria ingénua dos conjuntos* que, na modelação de sistemas, emerge sobretudo como um poderoso *tool to think with*. De facto, o núcleo do ambiente de prototipagem é um calculador de um fragmento (construtivo) dessa teoria e, na prática, é possível tomar a categoria dos conjuntos como o universo semântico das especificações CAMILA[2].

Os tipos de dados (que correspondem às entidades de um sistema de informação) são especificados a partir de um conjunto de tipos primitivos (*e.g.*, inteiros, *strings*, booleanos, etc.) por combinações, eventualmente recursivas, de três construtores de tipos básicos:

- *produto cartesiano* ($A \times B$), que exprime a agregação espacial de informação distinta;
- *alternativa* ou *união disjunta* ($A + B$), que exprime escolha (*i.e.*, agregação no domínio dos tempos) e
- *exponenciação*[3], ou espaço funcional, ($A \to B$) que exprime uma dependência funcional.

Os tipos primitivos incluem ainda o tipo singular (**1**) usado, por exemplo, na modelação de situações de excepção, e um tipo *universal* (**any**) que actua como o elemento de topo do sistema de tipos.

[2] Em rigor, tal categoria é enriquecida por uma ordem parcial para lidar com a parcialidade nas definições e garantir a existência de soluções únicas para equações de tipo.

[3] A exponenciação ou espaço funcional é frequentemente denotada por $B^A$. Nesta dissertação é usada a notação $A \to B$.

Note-se que *funções* são *first class citizens* em CAMILA, podendo, em particular, ser fornecidas e retornadas a outras funções. Similarmente, o tipo função pode surgir na definição de outros tipos, por exemplo para modelar o tipo de um registo cujas componentes são elas próprias operações. As construções canónicas sobre um espaço funcional incluem:

- a composição de funções, $(f \circ g)$,
- *identity* (`id`, definido como $\lambda\, x \,.\, x$ ),
- *curry*,
- o isomorfismo básico entre $C \to (A \times B)$ e $C \to A \to B$),
- a geração de funções identidade (`id`)
- e constantes (`_c v`, denotando $\underline{v} \stackrel{\text{def}}{=} \lambda\, x \,.\, v$, para um valor $v$).

Ao processar as definições dos tipos de dados, o prototipador gera automaticamente os construtores e selectores associados ao tipo produto, assim como os imersores canónicos associados às alternativas. A linguagem assume, ainda, como tipo primitivo o construtor do conjunto potência de um conjunto dado: $set(A)$ é o tipo correspondente ao conjunto de todos os subconjuntos de `A`.

Finalmente, para $A$ e $B$ finitos, a linguagem suporta os seguintes tipos derivados:

- *sequências* finitas de elementos de $A$ $(A^*)$
- *correspondências* ou *funções parciais* de $A$ para $B$ $(A \rightharpoonup C)$.

Todos os construtores de tipos em CAMILA são *functoriais* no sentido em que se aplicam uniformemente tanto a tipos como a funções (primitivas ou definidas pelo utilizador), estendendo o seu efeito de forma estrutural. Por exemplo, para $f : A \longrightarrow B$ função, a expressão $f$`-set` representa uma outra função que aplica $f$ a todos os elementos de conjuntos de $A$. Esta base functorial permite escrever especificações de forma muito concisa e elegante. O leitor interessado é remetido para (Barbosa, Barros, and Almeida, 2000) onde esta base é explorada na codificação em CAMILA de diferentes classes de esquemas recursivos genéricos, típicos das abordagens ditas *politípicas* à construção de programas (Bird and Moor, 1997).

O repositório de operadores em CAMILA é muito rico e estruturado, no sentido em que as álgebras associadas a cada um dos tipos primitivos e construtores referidos acima são directamente exprimíveis no prototipador. Concluímos esta secção apresentando, em forma tabular, para maior concisão e facilidade de referência, as álgebras referidas assim como as expressões de controlo utilizadas na linguagem.

### Construtores de tipos

Notação mais usada na construção de tipos:

| | |
|---|---|
| $set(A)$ | *conjuntos de A* |
| $A \rightharpoonup B$ | *mappings*, correspondências de A para B |
| $A^*$ | *sequências de A* |
| $A \to B$ | *funções de A para B* |
| $A \times B$ | *produtos* |
| $campo1 : A \times campo2 : B$ | *produto* com campos |
| $A + B$ | *alternativas* |
| **any** | *tipo universal* |
| **1** | *tipo singular* |

A definição de um novo tipo pode incluir um predicado sobre os valores de modo a restringir o conjunto portador – *invariante*. Se pretendermos definir um tipo *data*, mesmo na sua versão mais simplificativa, é mais fácil definir um conjunto portador mais amplo (Exemplo produto de três inteiros) e restringir os valores com uma função invariante que verifique a validade do triplo.

$$
\begin{array}{rclcll}
data & = & dia & : & int & \times \\
 & & mes & : & int & \times \\
 & & ano & : & int &
\end{array}
$$

$$\underline{inv}(d) \stackrel{\text{def}}{=} dia(d) > 0 \wedge dia(d) \leq 31 \wedge mes(d) > 0 \wedge mes(d) \leq 12 \wedge ...$$

Algumas funções associadas a tipos:

| **Descrição** | **Notação** |
|---|---|
| tipo de uma expressão ...................................... | $type(e)$ |
| expressão compatível com tipo t ............................. | is-$t(e)$ |

### Funções — $A \to B$

A linguagem Camila dispõem de vários modos de definir funções, podendo estas incluir (ou não) definição de tipos de argumentos e resultado, definição de pre-condição, definição de cláusula de estado. É também possível definir funções anónimas.

| **Descrição** | **Notação** |
|---|---|
| funções compactas ........................................ | $f(x) \stackrel{\text{def}}{=} y$ |
| funções anónimas ........................................ | $\lambda\, x \,.\, f(x)$ |
| funções com assinatura<br>$f : t1 \times t2 \longrightarrow t$<br>$f(x1, x2) \stackrel{\text{def}}{=} g(x1, x2)$ | |
| funções com assinatura e pré-condição<br>$f : t1 \times t2 \longrightarrow t$<br>$f(x1, x2) \stackrel{\text{def}}{=}$<br>$\quad \underline{pre} \quad p(x1, x2)$<br>$\quad \underline{in} \quad g(x1, x2)$ | |
| funções com assinatura e estado<br>$f : t1 \times t2 \longrightarrow t$<br>$f(x1, x2) \stackrel{\text{def}}{=}$<br>$\quad \underline{post} \quad s' = h(x1, x2, s)$<br>$\quad \underline{in} \quad g(x1, x2)$ | |

É possível a definição de funções de ordem superior.

### Construtores genéricos usados nas expressões

A linguagem CAMILA oferece alguns mecanismos habituais em linguagens de especificação, como sejam as expressões *let* (com pattern matching parcial), expressões condicionais (com pattern matching parcial), *or* de linguagem natural.

| **Descrição** | **Notação** |
| --- | --- |
| expressão condicional ........................ | $\begin{cases} c1 & \Rightarrow & v1 \\ c2 & \Rightarrow & v2 \\ \underline{otherwise} & \Rightarrow & vn \end{cases}$ |
| condicional com pattern matching . | $\begin{cases} v1\ is\text{–}< h : t > & \Rightarrow & f(h,t) \\ v2\ is\text{–}<> & \Rightarrow & g \\ v3\ is-\{e : s\} & \Rightarrow & h(e,s) \\ v4\ is-\{\} & \Rightarrow & i \\ \underline{otherwise} & \Rightarrow & j \end{cases}$ |
| expressão let .......................................... | $\begin{array}{ll} \underline{let} & a = e1 \\ & b = e2 \\ \underline{in} & f(a,b) \end{array}$ |
| let com pattern matching .......................... | $\begin{array}{ll} \underline{let} & <a,b> = e \\ \underline{in} & f(a,b) \end{array}$ |
| or - ou de linguagem natural ................................ | $a\ or\ b$ |

O operador `or` (ou de linguagem natural) não é habitual em linguagens de especificação pelo que carece de uma breve explicação. Este operador (próximo do *ou* de linguagem natural) dá como resultado o primeiro argumento excepto se ele for vazio ou indefinido. É usado como operador binário infixo.

$$Or(exp1, exp2) \stackrel{\text{def}}{=}$$
$$\begin{cases} exp1 = \emptyset & \Rightarrow & exp2 \\ exp1 = () & \Rightarrow & exp2 \\ exp1 = <> & \Rightarrow & exp2 \\ indefinido(exp1) & \Rightarrow & exp2 \\ \underline{otherwise} & \Rightarrow & exp1 \end{cases}$$

### Booleanos

A nível dos booleanos, existe em CAMILA as funções lógicas e quantificadores habituais.

| **Descrição** | **Notação** |
|---|---|
| Negação | $\neg a$ |
| Conjunção | $a \wedge b$ |
| Disjunção | $a \vee b$ |
| Implicação | $a \Rightarrow b$ |
| Quantificação universal | $\forall x \in setexp \wedge p(x)$ |
| Quantificação existencial | $\exists x \in setexp : p(x)$ |
| Quantificação existencial unário | $\exists^1 x \in setexp \wedge p(x)$ |

**Mappings, correspondências** — $A \rightharpoonup B$

As correspondências unívocas dispõem das seguintes funções predefinidas:

| **Descrição** | **Notação** |
|---|---|
| Enumeração de mappings | $\left( \begin{pmatrix} a1 \\ b1 \end{pmatrix} \begin{pmatrix} a2 \\ b2 \end{pmatrix} \right)$ |
| Mappings em compreensão | $\begin{pmatrix} f(a) \\ g(a) \end{pmatrix}_{a \in setexp}$ |
| | $\begin{pmatrix} f(a) \\ g(a) \end{pmatrix}_{a \in setexp \wedge p(a)}$ |
| Domínio | $dom(f)$ |
| Contra-domínio | $rng(f)$ |
| Aplicação | $f(x)$ |
| Restrição ao domínio | $f \mid s$ |
| Subtracção ao domínio | $f \setminus s$ |
| Reescrita, reescrever $f$ com $g$ | $f \dagger g$ |

**Sequências** — $A^*$

As sequências de um tipo A dispõem das seguintes funções de base:

| **Descrição** | **Notação** |
|---|---|
| Enumeração de sequências | $<a1, a2, ... >$ |
| Sequências em compreensão | $<f(a) \mid a \in setexp>$ |
| | $<f(a) \mid a \in setexp \wedge p(a)>$ |
| Head | $head(s)$ |
| Tail | $tail(s)$ |
| Elemento a partir da posição | $s(i)$ |
| primeiro elemento | $\pi_1(x)$ |
| segundo elemento | $\pi_2(x)$ |
| Concatenação | $s \frown r$ |
| Acrescentar elemento | $<x> \frown s$ |
| | $<x : s>$ |
| | $s1 \frown s2 \frown ... \frown sn$ |
| Concatenação distribuída | $\frown(<<... >, ... >)$ |
| Conjunto dos elementos | $\{x \mid x \in s\}$ |
| | $elems(x)$ |
| Índices existentes | $inds(s)$ |
| Inversa | $reverse(s)$ |
| Comprimento | $length(s)$ |
| Ordenação | $sort(s)$ |
| Ordenação com função de comparação | $sort2(f, s)$ |

As sequências podem ser heterogéneas (Exemplo $S = \mathbf{any}^*$).

**Conjuntos** — $set(A)$

Os conjuntos dispõem das seguintes funções predefinidas:

| **Descrição** | **Notação** |
|---|---|
| Enumeração de conjuntos | $\{a1, a2, ...\ \}$ |
| Conjuntos em compreensão | $\{f(a) \mid a \in setexp\}$ |
| | $\{f(a) \mid a \in setexp \wedge p(a)\}$ |
| Escolha não determinística | $choice(c)$ |
| Reunião | $c1 \cup c2$ |
| Intercepção | $c1 \cap c2$ |
| Subtração de conjuntos | $c1 - c2$ |
| Pertencer ao conjunto | $e \in c$ |
| Não pertencer ao conjunto | $e \notin c$ |
| Cardinal | $\mathsf{card}\, c$ |
| União distribuída | $\bigcup(\{\{...\ \}, ...\ \})$ |
| Ordenação | $sort(s)$ |
| Ordenação com função de comparação | $sort2(f, s)$ |

Os conjuntos podem ser heterogéneos (Exemplo $S = set(\mathbf{any})$).

## 2.2 O Prototipador

O CAMILA tem sido usado em várias situações e com preocupações que tem variado com o contexto e ao longo do tempo. Daí que se tenha reconhecido a necessidade de constituir um ambiente adaptável, i.e, garantir que haja capacidade de:

- adaptar a linguagem à evolução dos formalismos subjacentes (os quais têm um poder expressivo muito maior),
- adaptar a linguagem/ambiente de modo a poder lidar com objectos muito diversificados – (tanto tipos simples como inteiros, *strings*, como com elementos complexos e por vezes externos como bases de dados, correio electrónico, ou uma rede de emergência dos bombeiros),
- equipar o sistema CAMILA com uma arquitectura versátil que permita que possa ser usado de diversas maneiras (Ex. como biblioteca C, como interpretador, como compilador),
- equipar o ambiente com um conjunto de ferramentas de desenvolvimento e manutenção.

### 2.2.1 Descrição geral da arquitectura interna

Internamente a ferramenta de prototipagem CAMILA baseia-se num conjunto de blocos partilháveis que ligados de diversos modos dão origem a um conjunto de comandos e a um conjunto de bibliotecas.

Os principais blocos partilháveis CAMILA são:

- (1) Parser da linguagem CAMILA com múltiplos blocos semânticos associados:
  - (1.1) semântica S-expression (constrói representações do tipo S-expression)
  - (1.2) semântica pretty-print (constrói termos do tipo LaTeX)
  - (1.3) semântica tags (constrói índices de funções existentes em formato tags)
  - (1.4) transformador S-expression em xmetoo (gera uma representação textual na linguagem xmetoo)

  Pode funcionar como biblioteca C.
- (2) Máquina de S-expressions (calculador).
  - (2.1) Inclui um parser da linguagem xmetoo.
  - (2.2) interpretador (read, evalute and print)
  - (2.3) mecanismos de embedding de C
- (3) Tradutor de xmetoo e de CAMILA para C (semelhante a "inlines"). Entre outras coisas, este tradutor tem sido usado para tornar fácil embutir funções novas no conjunto de funções predefinidas, bastando para tal criar o seu protótipo CAMILA.

Com base nestes blocos foram construídos os seguintes comandos:

- `camila` – interpretador da linguagem CAMILA usado como prototipador. Internamente junta os blocos designados por 1, 1.1, 2, 3 e a biblioteca readline[4]);
- `camilax` – destinado à execução de protótipos em modo não interactivo. Usa um conjunto de blocos semelhantes ao anterior (1, 1.1, 2, 3) excluindo a parte interactiva. Pode ser usado também como biblioteca C e serve de base a um interface perl (funciona como biblioteca perl)
- `xmetoo` – interpretador de S-expressions escritas em formato xmetoo. Usa os blocos 2, 2.1, 2.2, 2.3 e 3, e também pode funcionar como biblioteca C e biblioteca perl
- `seca` – tradutor CAMILA – `xmetoo`. Usa os blocos (1, 1.1, 1.4)
- `campretty` – tradutor CAMILA – LaTeX. Usa (1, 1.2))
- `camtags` – tradutor CAMILA – `tags` de modo a facilitar a edição de texto CAMILA guiado por funções. Usa (1, 1.3).

[4]readline – biblioteca C (GNU) que fornece edição de linha, história e um conjunto de características importantes para qualquer interpretador.

#### *Embedding*

Como foi referido é crucial que uma linguagem de especificação possa abarcar um conjunto variado de mundos, de modo a modelar apenas a janela relevante para o estudo em causa.

Quando há prototipagem, i.e., quando se pretende animar uma parte da especificação, é crucial que se possa comunicar com uma variedade tão grande quanto possível de ambientes.

No ambiente CAMILA existe uma grande variedade de modos de comunicação e *embedding*. Nomeadamente existem mecanismos para:

- incorporar C em Xmetoo
- incorporar Xmetoo em C
- incorporar CAMILA em perl
- incorporar CAMILA em LaTeX
- incorporar CAMILA no próprio C da máquina Xmetoo
- incorporar de comandos externos através de pipes unidireccionais
- incorporar de comandos externos através de pipes bidireccionais
- funcionar como pipe bidireccional
- funcionar como shell (usando camilax)
- incorporação de dados externos (estáticos ou dinâmicos) com injecção automática em tipos CAMILA.

### 2.2.2 Ferramentas associadas

Para além da utilidade relacionada com a capacidade descritiva e com o facto de constituir uma base de discussão matemática e de prototipagem, algumas ferramentas associadas ao ambiente de prototipagem CAMILA forneceram uma preciosa ajuda na produção do texto matemático LaTeX deste documento.

Nesta subsecção far-se-á uma breve descrição da ferramenta CamTEX e do conversor `campretty` para documentar o modo como esta dissertação é produzida e por se considerar um exemplo interessante de *embedding* e de cooperação entre ambientes[5] que normalmente não são ligados.

[5] De facto, aqui interligam-se o ambiente de construção de documentos, o ambiente de construção de protótipos e o ambiente de teste e simulação de protótipos.

### Campretty - conversor Camila LaTeX

O `campretty` é um compilador Camila – LaTeX. Pode ser usado para gerar a totalidade do documento final LaTeX, ou para produzir partes a serem incluídas num documento (neste caso integrado no CamTeX que se descreve abaixo).

Para converter um ficheiro `test.cam` num ficheiro LaTeX `test.tex`, o comando a usar é:

```
campretty -o=test.tex test.cam
```

ou

```
campretty -o=test.tex -t test.cam
```

para produzir um extracto LaTeX a incluir em documentos.

Internamente o `campretty` é um compilador que partilha o nível léxico e o nível sintáctico com o Camila – tanto o comando `camila`, como o `camilax`, como o `camtag`[6], como o `campretty` são extraídos a partir de um único ficheiro de parser Yacc com semânticas emparelhadas.

### CamTeX – LaTeX com inclusão e invocação de Camila

CamTeX é uma ferramenta com funcionamento semelhante ao BibTeX, que permite ligar LaTeX com o ambiente de prototipagem Camila. O CamTeX permite basicamente:

- inclusão de especificações Camila em LaTeX após tradução para a notação matemática (usando `campretty` para essa tradução).
- incorporação de cálculo de expressões Camila em documentos LaTeX

À semelhança do que acontece com o BibTeX, para utilizar CamTeX num texto LaTeX `t.tex`, faz-se a habitual inclusão da definição das macros descritas à frente através de `\usepackage{camtex}`, inclui-se invocação de macros para embeber ou calcular texto Camila. Seguidamente para obter o resultado final, executa-se:

```
latex t.tex; camtex t.tex; latex x.tex
```

O seu funcionamento interno é explicado mais à frente.

[6]Gerador de ficheiros tags (nomes de funções – linha onde começam) para Camila.

### Inclusão de texto Camila

Para incluir texto Camila em LaTeX, duas macros LaTeX podem ser usadas[7]:

**campretty** para incluir directamente um conjunto de tipos e funções Camila.

**Exemplo 1:** Inclusão de texto Camila

O seguinte texto LaTeX

```
\campretty{
    TYPE
      A = termo * dicionario;
      dicionario = fonteInf-set;
      B = inf;
      fonteInf = termo -> inf;
    ENDTYPE

   *func f(t:termo,ds:dicionario):inf
   returns conciliar( {d[t] | d <- ds /\ t in dom(d)});
}
```

dá origem a:

$$\begin{array}{lcl} A & = & termo \times dicionario \\ dicionario & = & set(fonteInf) \\ B & = & inf \\ fonteInf & = & termo \rightharpoonup inf \end{array}$$

$$f : termo \times dicionario \longrightarrow inf$$
$$f(t, ds) \stackrel{\text{def}}{=} conciliar(\{d(t) \mid d \in ds \wedge t \in dom(d)\})$$

**camfpretty** nome – para incluir o texto matemático de um ficheiro contendo um protótipo Camila. Exemplo:

```
\camfpretty{dicionario.cam}
```

### Inclusão de execução/cálculo de expressões Camila

Para executar/calcular em Camila, são fornecidas duas macros LaTeX:

**prototype** nome
para carregar um protótipo onde (tipicamente) se encontram definidas funções e um estado inicial a ser usadas nos cálculos. Em casos reais, é habitual haver um modelo relativamente complexo que irá sendo

[7]Como é de prever, estas funções foram usadas centenas de vezes neste documento.

descrito e demonstrado aos poucos. A inclusão do protótipo, permite garantir que os resultados estejam a ser calculados com a versão actual.

**cameval** expressão
calcula a expressão e insere o respectivo resultado.

**Exemplo 2:** Uso de macros LATEX para cálculo de CAMILA

O seguinte texto LATEX contém invocações a cálculos em CAMILA

```
após inicializarmos a com
 \cameval{do(a <- {2,3,4,5}, a) } e b com
 \cameval{do(b <- {4,5,6,7}, b) }... o cardinal do conjunto
 \campretty{ a U b } é
 \cameval{ #(a U b) } e a sua soma
 \cameval{ add-orio(0, a U b) }.
```

após processado, dá origem a

após inicializarmos a com $\{ \ 2 \ 3 \ 4 \ 5 \ \}$ e b com $\{ \ 4 \ 5 \ 6 \ 7 \ \}$
... o cardinal do conjunto $a \cup b$ é 6 e a sua soma 27 .

O uso deste mecanismo possibilita a inclusão de cálculos com estado no interior do texto LATEX, facilitando a coerência/correcção dos exemplos e simulação de protótipos.

### Funcionamento interno do CamTEX

Como se referiu anteriormente, o modo de utilização do CamTEX é análogo ao `bibtex`:

- executar `latex` que tenta incluir as referências bibliográficas geradas, gera um ficheiro de chaves de referências pedidas
- executar `bibtex` que consulta a lista das chaves de referência pedidas, abre a base de dados das referências, e gera as referencias bibliográficas para serem incluídas na execução seguinte do LATEX
- voltar a executar `latex`

Ou seja,

- o LATEX inclui um ficheiro `texto.cal` se este existir; processa um ficheiro `texto.tex` e produz um conjunto de linhas auxiliares em `texto.aux`
- o CamTEX analisa o ficheiro `texto.aux`, processando as linhas referentes a `campretty` e a `cameval`.:
  - Invoca o comando `campretty` cada vez que haja um extracto CAMILA a converter para latex, guardando o seu resultado.

- Usando o módulo perl `camila.pm`, arranca com uma sessão `camila` (um processo `camilax` em pipe bidireccional, sendo cada `cameval` convertido numa escrita no processo `camilax` e na leitura da respectiva resposta) e calcula e guarda os resultados de cada `cameval`, i.e., de cada simulação.
- gera uma ficheiro `texto.cal` com a traduções LaTeX resultantes dos processamentos dos `campretty` e `cameval`.

– volta a executar LaTeX para poder agora substituir as invocações pelos seus resultados, calculados no ponto anterior.

Saliente-se que o CamTeX ao simular uma sessão CAMILA, executa um processo em pipe bidireccional o que leva a que as definições e alterações de estado sejam conservadas entre as sucessivas invocações a `cameval`.

## 2.3 Especificação de um tipo comum: *Estrutura Tipada de Facetas*

A utilização de um tipo universal para representação interna da informação a manipular, tem sido uma constante de quase todos os ambientes. A escolha de um tipo universal envolve uma série de decisões/acções como sejam:

– usar um sistema tipado ou não (Lamport and Paulson, 1999)
– estudar as consequências a nível do poder expressivo
– estudar as consequências a nível da complexidade de implementação
– estudar as consequências a nível da facilidade de ligação com outros ambientes, outras aplicações – com o mundo exterior.

Foi já apresentada a necessidade/utilidade da existência de um tipo universal que fosse um bom compromisso entre a complexidade na implementação e o poder expressivo.

Cada ambiente vem normalmente equipado com um conjunto de funções de base que facilitam a sua utilização.

Após analisar os tipos de uma série de ambientes:

– associados à programação lógica (termos Prolog (O'Keefe, 1990), $\psi$-terms do Life (Ait-Kaci and Lincoln, 1988; Ait-Kaci, 1990) e seus derivados, *feature structure*)
– associados a sistemas de representação de conhecimento (frames, redes semânticas, *feature structure*),
– associados à programação funcional,
– associados a linguagens de scripting (tipos Perl, e Phyton),

- associados a ambientes de especificação (tipos VDM, CAMILA, Z),
- associados a ambientes genéricos ( Aterms - (van den Brand, Klint, and Olivier, 1999; Pennings, 1994; Saraiva, 1999)),

optou-se por uma versão simplificada de *feature structure* que fosse facilmente conciliável com especificações CAMILA, com implementações em ambiente scripting, e com as necessidade expressivas de PLN. Naturalmente este tipo universal tem semelhanças com quase todos os outros acima referidos.

Este tipo universal é uma variante de *feature structure* (FS).

**FS – *feature structure***

As FS foram introduzidas por Martin Kay (Kay, 1979) e rapidamente se tornaram populares como modo de representar informação em áreas de processamento de linguagem natural.

Posteriormente, surgiram vários enriquecimentos das FS de modo a incorporar uma série de desenvolvimentos em vários aspectos aumentando-lhe bastante o poder expressivo e a complexidade. Um destes formalismos é a typed *feature structure*(TFS) (Krieger and Schäfer, 1994; Krieger, 1995) que se poderia caracterizar por ter:

- estruturas tipadas,
- partilha de sub-estruturas,
- hierarquia de tipos,
- herança múltipla nos tipos.

A elevada capacidade de representação das TFS permitiu-lhes descrever no mesmo formalismo objectos tão variados como entradas léxicas, autómatos, gramáticas, e servir de base a sistemas sofisticados como as *Head-Driven Phrase Structure Grammar*(HPSG)(Pollard and Sag, 1994).

### 2.3.1 Modelo adoptado

O modelo usado nesta tese corresponde a feature structure*s* (FS) simplificadas e serão designadas por **estrutura de facetas**(EF).

As estrutura de facetas não têm partilha de sub-estruturas, mas têm algumas extensões para permitir conjuntos, invariantes e campos que sejam funções (ordem superior).

# Model ef

$$\begin{array}{lcl} ef & = & atomo + (atomo \rightharpoonup ef) + set(ef) + ef^{\star} + (ef \longrightarrow ef) \\ atomo & = & any \end{array}$$

Onde *atomo* é um qualquer tipo atómico, (exemplo: strings, inteiros, etc.). Ou seja o invariante associado é

$$Inv(a) \stackrel{\text{def}}{=} atom(a)$$

Este tipo constitui um compromisso entre poder expressivo e simplicidade de implementação em ambientes variados.

Seguidamente, apresenta-se a definição de uma função *getAll* que extrai todas as ocorrências de valores de um atributo $c$ numa estrutura de facetas $f$ passados como parâmetro. Esta função serve também de exemplo de uso do nível functorial do CAMILA.

$$getAll : ef \times atomo \longrightarrow set(ef)$$

$$getAll(f,c) \stackrel{\text{def}}{=}$$

$$\left\{ \begin{array}{rcl} \text{is-}(set(any))(f) & \Rightarrow & \bigcup(getAll\text{-set}(f,c)) \\ \text{is-}(any^{\star})(f) & \Rightarrow & \bigcup(elems(getAll^{*}(f,c))) \\ \text{is-}(any \rightharpoonup any)(f) & \Rightarrow & \underline{let} \quad a = f(c) \ or \ \emptyset \\ & & \qquad b = \left\{ \begin{array}{rcl} \text{is-}(set(any))(a) & \Rightarrow & a \\ \text{is-}(any^{\star})(a) & \Rightarrow & elems(a) \\ \underline{otherwise} & \Rightarrow & \{a\} \end{array} \right. \\ & & \underline{in} \quad b \cup \bigcup(getAll\text{-set}(ran(f),c)) \\ \underline{otherwise} & \Rightarrow & \emptyset \end{array} \right.$$

Note-se que nesta definição da função *getAll* se usaram alguns functores como seja o functor $f$-set e o functor estrela $f^{*}$ que (de um modo simplificado) transformam funções

$$f : A \rightarrow B$$

em funções $g \stackrel{\text{def}}{=} f\text{-set}$ definidas como:

$$g : set(A) \longrightarrow set(B)$$

$$g(a) \stackrel{\text{def}}{=} \{f(x) \mid x \in a\}$$

ou em funções $h \stackrel{\text{def}}{=} f^{*}$ definidas como:

$$h : A^\star \longrightarrow B^\star$$
$$h(a) \stackrel{\text{def}}{=} {<}f(x) \mid x \in a{>}$$

respectivamente.

Embora torne as especificações mais concisas, para facilitar a leitura, vai-se evitar a utilização deste nível functorial nas especificações que se apresentem neste documento.

### Exemplo de descrição usando EF

Ao descrever informação usando EF há uma necessidade de arrumar a informação em compartimentos que associam um nome de atributo a um valor. Esse valor pode ser atómico ou uma EF.

Considere-se o seguinte exemplo, correspondente à informação ligada à palavra *comprador*.

```
[  name    -> comprador
   género  -> masculino
   cat     -> adjectivo_nomeComum
   número  -> singular
   radical -> [
      name -> comprar
      Cla  -> v. tr.
      Conjugação -> 1
      Fonética   -> kõprár
      FrasesExemplo -> <
           O Miguel comprou um carro novo.
           A Filipa comprou mais um livro.
           O advogado comprou as testemunhas.
            ...>
      Derivadas  -> [
           ...
   ...
```

A informação contida nesta EF está *compartimentada* em vários atributos. Certos atributos têm valores atómicos (exemplo **género**) outros têm valores mais complexos (exemplo **radical**).

# Capítulo 3

# Dicionários: enquadramento histórico e algumas linhas de evolução

***Nuduvas.*** *– significação incerta*
"Diccionario Geral da Lingoa Portugueza de algibeira"
*Lisboa: Impressão Regia 1818*[1].

Nesta secção, apresenta-se uma breve evolução histórica dos dicionários, seguida da referência a alguns projectos e abordagens, que directa ou indirectamente marcaram a visão que hoje se tem deles.

## 3.1 Breve introdução histórica da Dicionarística Portuguesa

De um ponto de vista histórico (Verdelho, 1995), pode-se dizer que os dicionários começaram a desenvolver-se a partir dos finais do séc. XV[2], como modo de apoiar a escolarização humanista do latim (língua da cultura) numa

[1] *Por tres literatos nacionaes. Contem mais de vinte mil termos novos pertencentes a Artes, Officios, e Sciencias, todos tirados de Classicos Portuguezes, e ainda não incluidos em Diccionario algum até ao prezente publicado.* Lisboa: Impressão Regia 1818.

[2] Apesar da existência de manuscritos anteriores contendo listas de palavras latinas e seus correspondentes em Português, como seja o "Dicionário de verbos Alcobacence", códice CDIV/286, Biblioteca Nacional.

sociedade cuja língua do dia-a-dia (língua vernácula) diferia muito do latim. Neste desenvolvimento e para a sua importância foi obviamente crítico a arte da imprensa.

A ciência e a técnica lexicográfica, embora tenham tido importantes antecedentes durante a Idade Média, surgem neste período com um desenvolvimento muito rápido e fecundo. O espanhol António de Nebrija (*Lexicon*, latim-espanhol, Salamanca, 1492), o italiano Ambrósio Calepino (*Dictionarium*, primeiro monolingue, depois plurilingue, 1502) e o francês Robert Estienne (*Dictionarium seu linguae thesaurus*, Paris, 1531) são habitualmente apontados como os pioneiros neste campo.

O termo **dicionário** surge como neologismo introduzido por Jerónimo Cardoso no *Hieronymi Cardosi Dictionarium Iuventuti studiosae admodum frugiferum. Nunc diligentiori emendatione impressum*, Coimbra 1551, naquele que se pode designar pela primeira tentativa lexicográfica portuguesa (um conjunto de 3.600 termos latinos e seus correspondentes portugueses, distribuídos por domínio e não alfabetados). Onze anos mais tarde, o mesmo autor publica o primeiro dicionário português-latim[3], já alfabetado com cerca de 12.000 entradas – o ***Hieronymi Cardosi Lamacencis Dictionarium ex lusitânico e latinum sermonem***, Lisboa, 1562 – apresentando já variada informação acerca dos termos portugueses. Esta obra veio a ter muitas reedições atestando a importância que teve na época.

De entre os dicionários antigos que se seguiram não se poderá deixar de destacar os seguintes trabalhos:

**Agostinho Barbosa** – ***Dictionarium Lusitanico Latinum*** , Braga, 1611 – trabalho de adolescência do canonista Agostinho Barbosa que mais tarde veio a publicar numerosas obras de direito canónico.

**Bento Pereira** – ***Prosodia*** , 1634 e ***Thesouro da Lingoa Portuguesa***, 1647 – que tendo entradas bastante concisas, constituiu um importante dicionário académico, funcionando como normalizador na ortografia do Português; foi um dos pilares do ensino jesuítico em Portugal (sendo mais tarde proibido e mandado queimar pelo Marquês de Pombal).

**Rafael Bluteau** – ***Vocabulário*** , 1713 – um dicionário monumental: um monumento à língua – 8 volumes, feito no reinado de D.João V, sem limitações de custos nem dimensões, muito erudito e baseado num grande quantidade de textos literários, e em recolhas de campo (por falta de documentação escrita acerca de certos áreas, o autor afirma que "*corri*

[3]Há notícia de outros dicionários como por exemplo o *Dictionarium Lusitanum et Latinum* de Francisco Sanches de Castilho que estaria pronto para impressão à data do falecimento do autor (1558) mas do qual não apareceu qualquer exemplar.

*as mais humildes officinas da Republica; passei tardes inteiras em atafonas, entre moegas, & almanjarras, enfarinhado na arte de moer, ...... entrei em forjas de Ferreiros & fundidores, ...... mettime em lagares de vinho, puzme de Gorra ao pê das uvas...*"!!! (Silvestre, 2001).

**António Morais e Silva** – ***Dicionário de Língua Portugueza*** , 1789, – baseado no dicionário de Bluteau, mas reformado e acrescentado por Morais, constituindo o primeiro dicionário monolingue de português.

Mais tarde vão surgindo maiores preocupações meta-lexicográficas (preocupações de especificar a estrutura das definições dos verbetes) – com destaque para o *Dicionário Contemporâneo da Língua Portugueza* de Caldas Aulete, 1881, preocupações de formalizar algum tipo de relações semânticas (ver *Dicionário geral e analógico da Língua Portuguesa* de Artur Bivar, 1948 e o “Dicionário do Dicionário” incluído no *Dicionário do Português Básico* (Vilela, 1991)).

Ao longo deste vasto percurso histórico existe uma infinidade de grandes e pequenos factos fascinantes[4] – que naturalmente não serão aqui referidos – incluindo detalhes técnicos, biográficos, sociais e filosóficos.

Os dicionários correspondem ao esforço de síntese de propriedades de uma ou mais línguas, de palavras ou de conceitos.

Dicionários monolingues, preocupados com propriedades de palavras numa língua, e dicionários bilingues, preocupados com as correspondências entre diferentes línguas, foram-se tornando populares e passaram a constituir objectos habituais em qualquer casa. Este tipo de dicionários é capaz de nos explicar o(s) significado(s) de uma palavra, a sua fonética, algumas propriedades gramaticais.

Um dicionário funcionando como interlocutor de quem o consulta, tem que ter um público em vista para que possa comunicar eficientemente[5]: há necessidade de construir dicionários diferentes para diferentes audiências (por exemplo, um bom dicionário de Língua Portuguesa para quem está a aprender uma língua, não é o melhor dicionário para quem esteja a terminar uma

[4]No *Novo diccionario da Língua Portugueza*, Lisboa, Rollandiana, 1806, aparecem algumas entradas descritas como tendo **significação incerta**!! – o que levou a fortes críticas por parte de certos sectores, mas que constitui uma notável explicitação de informação incompleta.

[5]Numa comunicação eficiente, há necessidade que cada um tenha um modelo correcto dos conhecimentos do interlocutor para que não seja dito o óbvio (interlocutor maçador, enfadonho), nem seja omitida informação necessária à compreensão do que se seguirá (interlocutor inacessível).

licenciatura).

> Historicamente ficou provado que há espaço/necessidade de dicionários de características muito variadas: dicionários académicos, retóricos, monumentais, eruditos, de iniciação, etc., funcionando como meio de satisfazer uma variedade de interlocutores.
> Quando se trata de versões electrónicas, há também utilidade de oferecer várias vistas sobre a informação, e necessidade de escolha das fontes de informação mais adequadas.

## 3.2 Evolução em termos de formato de suporte e ciclo de vida

As alterações em termos do modo como o dicionário está armazenado levaram a uma mudança da funcionalidade exigida/esperada e da maneira como se processa a sua construção e uso.

#### Evolução em termos de formato de suporte

Para além do enorme salto que foi a passagem de dicionários manuscritos para dicionários impressos, após a invenção da imprensa, em termos dos suportes físicos em que o dicionário é construído e publicado, assistiu-se a uma evolução natural: de um dicionários em papel passou-se para um dicionário com idênticas características e funcionalidade mas guardado em computador – os dicionários *machine readable*. Um dicionário em formato electrónico oferece, no entanto, um maior leque de possibilidades a nível de funcionalidade e levantaram uma crescente exigência de interligações – os dicionários *machine readable* deram lugar aos dicionários electrónicos.

O panorama actual evidencia que a informação contida em dicionários tem que ser estruturada dum modo versátil e reutilizável levando à preocupação de separar a definição do uso que lhe vai ser dado (Sperberg-McQueen and Burnard, 1994; TEI, 1995; Ide and Véronis, 1995; Amsler and Tompa, 1988, SGML/TEI), (Ramalho, Almeida, and Henriques, 1998, DAVID).

#### Evolução em termos de modo de construção

Ao nível do modo como é feita a construção dos dicionários, tem-se assistido a uma utilização de ferramentas de suporte de sofisticação crescente – embora a qualidade do produto produzido continue fortemente associada a uma boa base lexicográfica, a uma boa capacidade de observação e empenho *apaixonado* das pessoas ligadas ao processo.

A nível das ferramentas de suporte, tem-se vindo a assistir a uma sofisticação no sentido tecnológico e no sentido dos modelos de base associados:

- dicionário como um conjunto de fichas lexicográficas manuais
- dicionário como um texto processado por um editor de texto
- dicionário como uma base de dados tradicional
- dicionário como uma base de dados lexicográficas com base textual
- dicionário como informação com preocupações de durabilidade, de independência de plataformas e ferramentas; seguindo standards como o SGML
- dicionário como um sistema de representação de conhecimento

**Evolução em termos de utilização e utilidade**

Paralelamente têm-se assistido a um aumento das preocupações no sentido de obter uma máxima utilidade presente e futura:

- a preocupação de possibilidade de reutilização,
- a preocupação de normalizar interfaces e protocolos cliente-servidor de modo a alargar a sua gama de utilização,
- preocupações com a perenidade da informação, independência de ferramentas, independência de sistema operativo (informação mais duradoura do que as ferramenta).

Além das preocupações citadas tem-se vindo a tentar:

- incorporar conceitos comuns a *information retrieval* (IR),
- incorporar conceitos comuns a Ciências Documentais,
- incorporar conceitos comuns a bases de conhecimentos,

de modo a enriquecer a sua funcionalidade e melhor se adaptar a outros domínios.

A evolução tem também pedido aos dicionários capacidade de serem usáveis para fornecer informação capaz de apoiar o seu próprio processo construtivo.

## 3.3 Influência das exigências de PLN

Como é sabido, o uso de *estruturas implícitas* é bom para a comunicação com humanos mas torna mais complexas as tarefas de processamento de linguagem natural (PLN).

A evolução da linguística e a utilização de técnicas das Ciências da Computação, da Inteligência Artificial e das Ciências Documentais vieram salientar a importância de questões como classificação, tipo, normalização e herança.

A organização da informação orientada ao conceito (Miller et al., 1993; Miller, 1990; Fellbaum, 1998, WordNet) e as necessidades das aplicações associadas a processamento de linguagem natural (Krieger and Schäfer, 1994; Krieger, 1995; Pollard and Sag, 1994; Calzolari and Briscoe, 1994; Calzolari, 1993), demonstram a necessidade de poder ver os dicionários como bases de conhecimento a serem usadas como módulos de software capazes de oferecer uma funcionalidade variada em cooperação com outras peças, para em conjunto formarem aplicações.

Nos anos 80, com a evolução da Linguística e do processamento de linguagem natural e como resultado da mútua influência destes dois campos, assistiu-se a um aumento de importância dada ao nível léxico (Briscoe, 1991; Gazdar et al., 1985; Ritchie et al., 1992; Gazdar and Evans, 1989). O interesse da Linguística (computacional) deslocou-se de sistemas mais centrados no nível sintáctico para sistemas mais centrados no nível léxico. Isto veio trazer a necessidade de criar léxicos mais ricos e formais para suporte do processamento de linguagem natural. A ausência de léxicos adequados passou a ser um dos recursos críticos na construção de ferramentas de PLN.

Equacionou-se então o estado da arte ficando saliente que:

- há um enorme esforço necessário para a construção dum dicionário ou de um léxico para PLN (dezenas ou centenas de homens/ano (Briscoe, 1991)).
- a generalidade dos dicionários e léxicos de PLN têm formatos dependentes de um formalismo específico.
- a generalidade dos dicionários e léxicos de PLN são relativamente pouco capazes de cooperar entre si.

Ao mesmo tempo que foram assinaladas as fraquezas dos recursos existentes, assistiu-se ao aparecimento de uma série de propostas de maneiras de codificar o léxico, baseadas em modelos mais sofisticados e formais (Briscoe, 1991; Pollard and Sag, 1994; Popowich, 1993).

Ao contrário dos dicionários em papel, os dicionários para PLN têm necessidade de representação formal e explícita da informação com possibilidade de produzir a informação léxica necessária às várias ferramentas.

Vários projectos surgiram com o intuito de adquirirem informação a partir de dicionários e de criarem novos dicionários com informação mais versátil.

Por outro lado, as aplicações práticas de PLN, mostraram a necessidade de fornecer informação que não aparece (não aparecia) tradicionalmente nos dicionários. Ex: informação quantitativa indicativa de quão usado é determinado termo ou determinada acepção (Briscoe and Carroll, 1993).

Surgiram então vários projectos de construção de léxicos, uns partindo de dicionários tradicionais outros construídos de base. Cite-se, a título de exemplo, algumas iniciativas e projectos dentro desta categoria: Acquilex (Calzolari and Briscoe, 1994), WordNet (Miller et al., 1993; Miller, 1990; Fellbaum, 1998), EDR (Japão, Uchida 90, bilingue, japonês - inglês), Genelex (esprit, Normier e Nossin 90) (GENELEX, 1994a; GENELEX, 1993; GENELEX, 1994b), Multilex (esprit, McNaught 90) (Patrotte and Schumacher, 1993), etc.

Saliente-se o excelente relatório *Preliminary Recomendations on Lexical Semantic Encoding* do projecto EAGLES (Sanfilippo et al., 1999).

### Léxicos para PLN a partir de dicionários tradicionais

De entre os vários projectos que foram surgindo em que se visava construir bases de conhecimento lexical usando informação extraída de dicionários, refere-se a título de exemplo, o projecto Acquilex – ACQUIsition of LEXical information for natural language processing – cujo objectivo consistiu em extrair conhecimento léxico e conceptual a partir de dicionários *machine readable*.

Neste projecto avançou-se com um processo em que, através de uma análise das entradas de dicionários, se procedeu a um trabalho de engenharia reversa de dicionários visando o enriquecimento da estrutura dos dicionários com taxonomias semânticas (parcialmente extraídas das entradas dos dicionários) e com a construção de codificação das entradas lexicográficas como TFS – Typed Feature Structure (Calzolari et al., 1993).

A extracção semântica baseou-se em que cada definição[6] inclui um *genus* (i.e. classe geral) e uma *differentia especifica* (i.e. propriedade que o distingue dos outros elementos dessa classe).

[6]Segundo proposta de Aristóteles!!!

**Exemplo 1:** entrada de dicionário

**cavaquinho** – *n.c.*
Instrumento popular de cordas que tem dimensões pequenas (aprox. 40cm), tem um som muito vibrante e vivo, muito usado no Norte de Portugal.

O *genus* – Instrumento popular de cordas – corresponde à relação de hiponímia.

A *differentia especifica* – tem dimensões pequenas (aprox. 40cm), tem um som muito vibrante e vivo, muito usado no Norte de Portugal – indica que propriedades distinguem este elemento do outros do seu genus. Neste caso, alguns atributos estão instanciados com valores específicos ou diferentes do caso típico.

Este projecto ACQUILEX ajudou também à standardização do formato TEI (Text Encoding Initiative) para intercâmbio de entradas de dicionários em SGML (TEI, 1995), já que a capacidade de fazer intercâmbio de dicionários estava entre as preocupações/necessidades do projecto.

De um modo resumido, no projecto ACQUILEX partiu-se de um dicionário directamente digitalizado[7], procedeu-se à análise sintáctica das definições e extracção de informação semântica. Para tal, extraiu-se e desambiguou-se o **genus** para se efectuar a construção de uma taxonomia. Seguidamente foi filtrada informação da parte da **differentia** e feita a conversão de toda a informação extraída para o sistema de representação TFS.

Deste modo procedeu-se à construção semi-automática de uma base de conhecimento lexicográfica (Copestake et al., 1992).

## 3.4 Uso de corpora na construção de dicionários

É conhecido que um conjunto de exemplos, não permite por si, afirmar a correcção de uma regra que os descreva. Tomar um conjunto particular de exemplos para extrapolar regras, foi usado, desde tempos antigos para provar as coisas mais inconcebíveis, e para fundamentar coisas reconhecidamente falsas.

Um conjunto de exemplos:

[7] Machine readable dictionaries.

- pode não ser significativo para certo fenómeno em estudo,
- pode fazer uma relação de correlação indirecta parecer relação causal
- pode ser facilmente usado para negar certa regra.

Ou seja, o uso de corpora, por si só, é insuficiente para actividades como sejam a extracção de certo tipo de regras. Para além disso, pode ter certos perigos quando usado por quem não saiba interpretar correctamente os factos.

Paralelamente se não se usar corpora, fica-se limitado ao processo de introspecção, que tem também as suas limitações e perigos. No caso particular dos dicionários, é muito difícil arranjar por introspecção:

- o conjunto das K palavras a incluir como vedetas (será que o conjunto inclui as mais usadas? Qual será mais importante a palavra P1 ou a palavra P2?)
- o conjunto de todas a acepções duma palavra
- exemplos reais de uso de uma palavra (no conjunto de exemplos de um excelente dicionário de português fundamental a taxa de ocorrências da palavra gato é cerca de 100 vezes maior que as frequências encontradas num corpus de literatura composto por obras dos séculos XIX e XX; A taxa de ocorrência do nome Fonseca é cerca de 1000 vezes maior que a encontrada no mesmo corpus).

O uso de corpora, para o processo de construção de dicionários, fornece uma fundamentação externa e acaba por constituir o único meio que serve de base à determinação de certo tipo de informação, como seja a frequência de uso de um termo.

O uso de corpora para construção de dicionários não é uma ideia recente. Muitos dos dicionários referidos na introdução à Dicionarística Portuguesa (Secção 3.1), incluíam extractos literários (abonações) como informação complementar das entradas. O dicionário de Bluteau merece realce a este nível pela vastidão do conjunto de obras literárias que foram usadas na sua construção. Em 1755, *Dr Johnson's great dictionary of English* era fortemente baseado em citações extraídas da literatura, em vez de usar simplesmente a introspecção.

Mais recentemente, saliente-se o caso do dicionário COBUILD (Sinclair, 1987) directamente construído como um processo de extrair, rever e sintetizar evidências baseadas em corpus.

## 3.5 WordNet

Um marco histórico da influência do PLN no problema dos dicionários, foi o aparecimento da WordNet.

O projecto WordNet (Miller, 1990; Miller et al., 1993; Fellbaum, 1998) visou a construção de uma ontologia de palavras inglesas equipada com um conjunto rico de relações.

A unidade básica é o conceito, representado por um `synset` – um conjunto de sinónimos. Uma palavra com várias acepções aparece em vários `synset`.

A título de exemplo, apresentam-se alguns dos `synset`s pertencentes ao conjunto dos denominados *componentes semânticos primitivos*[8] que funcionam como conceitos primitivos, e topos da hierarquia de hiperonímia.

| |
|---|
| {acto, acção, actividade} |
| {animal, fauna} |
| {atributo, propriedade} |
| {evento, acontecimento, facto} |
| {sentimento, emoção} |
| {lugar, localização} |
| {pessoa, ser humano} |
| {tempo} |

As relações entre `synset`s incluem:

| Relação | elementos envolvidos | exemplo |
|---|---|---|
| sinonímia | relação base dos synset | |
| antonímia | nominal/nominal | homem/mulher |
| | verbo/verbo | entrar/sair |
| | adjectivo/adjectivo | lindo/feio |
| hiponímia/hiperonímia (isa) | nome/nome | plaina/ferramenta |
| troponímia | verbo/verbo | empurrar/mover |
| meronímia (has part) | nome/nome | mão/dedo |
| holonímia (part of) | inversa da meronímia | dedo/mão |
| causa/éCausado | | |
| derivadoDe | | |
| atributo | nome/adjectivo | tamanho/grande |
| papel/envolvidoEm | | |

Na coluna exemplos substituiu-se cada `synset` por um seu elemento para tornar mais compacta a tabela.

[8] primitive semantic components.

Algumas destas relações têm associados a si notação específica que pode ser usada na definição dos `synsets`.

Uma palavra como *canário*[9] vai estar incluída num `synset` que será relacionado com uma série de conceitos (`synsets`). Ou seja, associado a canário é necessário obter ligações com uma série de conceitos, mais uma vez aqui descritos como um único termo, para maior simplicidade:

- atributo : amarelo, pequeno
- has part : asa, bico, ...
- envolvidoEm : cantar, voar
- hiperonímia : pássaro

A **EuroWordNet** (Vossen, 1999) visou expandir a WordNet para um ambiente multilingue. Participaram na experiência várias línguas europeias. Para além da complexidade inerente à construção das várias WordNet há ainda a complexidade de as juntar. Na realidade, há sérios problemas relacionados com o facto de, por vezes, não existir uma sobreposição total entre os conceitos, levando à necessidade de criar uma arquitectura própria para a EuroWordNet que contemple uma zona comum inter-língua e a criação de algumas relações novas para descrever conceitos quase-sinónimos.

Em relação ao Português está em fase de conclusão uma WordNet (Marrafa, Branco, and Guerreiro, 2001, WordNet-pt) que levantou também um conjunto de problemas específicos, que estão a ser analisados na continuação do projecto.

Na EuroWordNet algumas relações aparecem mais expandidas, como é o caso da relação meronímia/holonímia (PartOf) que aparece dividida nas suas variantes habituais[10] (parte de, membro, porção de, feito de, local).

Tanto a WordNet como a EuroWordNet têm associado software que permite que possam ser usadas e adaptadas a vários contextos, como por exemplo, o módulo Perl `Lingua::Wordnet` (Brian, 2001). As funções[11] disponíveis permitem, por exemplo:

- dado um termo, obter o conjunto de synsets em que ele aparece;
- dado um synset:
    - encontrar os seus termos constituintes,
    - obter a informação geral semântica,
    - encontrar os synsets com ele relacionados e respectiva relação.

[9] Exemplo adaptado de um texto de Miller.

[10] Esta subdivisão aparece também na WordNet a partir da versão 1.7.

[11] Funções aqui referidas em sentido lato: alguns dos interfaces são orientados aos objectos; outros incluem predicados em vez de funções.

Usando o módulo `Lingua::Wordnet`, para escrever a lista dos hiperónimos de *cão*, na sua primeira acepção, podia ser usado o seguinte programa Perl:

```
1 use Lingua::Wordnet;

2 $wn = new Lingua::Wordnet;
3 $synset = $wn->lookup_synset("dog","n",1);
4 while ($synset = ($synset->hypernyms)[0]) {
5   print " " x $i++, "=> {", join(",",$synset->words),"}\n";}
```

Notas:

**linha 3** - Procurar o synset número 1 com categoria `n` – nominal.

**linha 4** - Enquanto houver hiperónimos, "avança"para o primeiro synset pai (pode haver mais que um pai; no caso de haver mais que um pai, os outros estão a ser ignorados).

**linha 5** - Escreve as palavras que constituem o synset.

produzindo a seguinte saída:

```
1 => {canine%2, canid%1}
2  => {carnivore%1}
3   => {placental%1, placental_mammal%1, eutherian_mammal%1}
4    => {mammal%1}
5     => {vertebrate%1, craniate%1}
6      => {chordate%1}
7       => {animal%1, animate_being%1, beast%1, creature%1, fauna%2}
8        => {organism%1, being%2, living_thing%1}
9         => {entity%1}
```

Notas:

**linha 1** - As várias acepções estão numeradas: `canine%2` significa a acepção 2 de canine.

A WordNet para além de constituir um marco histórico ao nível da concepção de dicionários, e para além de constituir um importante recurso disponível (uma rede semântica de várias dezenas de milhares de `synset` ingleses), teve ainda o mérito de levantar uma série de outros desafios (Buitelarr, 1998, CoreLex), tais como:

- definição de distância conceptual entre palavras
- uso de palavras relacionadas em *information retrieval*
- adaptação da WordNet a ambientes multilingue.

## 3.6 Protocolo DICT

Outro importante marco na história dos dicionários foi a proposta de definição de um protocolo para disponibilizar em rede informação de dicionários: o *Dict Protocol* (Faith and Martin, 1998) – um protocolo cliente-servidor para dicionários de linguagem natural – criado por Rik Faith and Bret Martin.

A existência desta proposta de protocolo levou ao aparecimento de implementações públicas de servidores e de clientes para vários ambientes bem como ao aparecimento de dicionários públicos.

Este protocolo propõe uma forma normalizada de disponibilizar em rede a informação referente a termos.

### Servidores

Os próprios autores disponibilizaram um servidor *open-source* – `dictd` – de acordo com o protocolo, bem como um conjunto de programas de suporte.

Posteriormente, foram aparecendo outros servidores cobrindo outros ambientes. Cite-se, a título de exemplo, o servidor – Jiben: the Perl Dict server – por Jay Kominek (Kominek, 1999)[12], um servidor em Perl para o referido protocolo.

### Clientes

Como é usual em arquitecturas cliente-servidor, há necessidade de uma maior variedade de clientes – um único servidor corresponde tipicamente a vários clientes, sendo portanto mais provável a heterogeneidade do lado do cliente.

Para além do cliente criado por Rik Faith, usável na forma de comando de linha, existe actualmente uma grande variedade de clientes DICT como sejam:

- Kdict (Holzer-Klupfel and Gebauer, 1998) – Matthias Holzer-Klupfel, Christian Gebauer – um cliente gráfico para o protocolo Dict, para ambiente KDE, com excelente aspecto gráfico e alguma funcionalidade acrescida,
- Gdict (Hovinen, Papadimitious, and Hughes, 1999) um cliente gráfico para o protocolo Dict, ambiente Gnomo,

[12] Jiben significa dicionário em japonês.

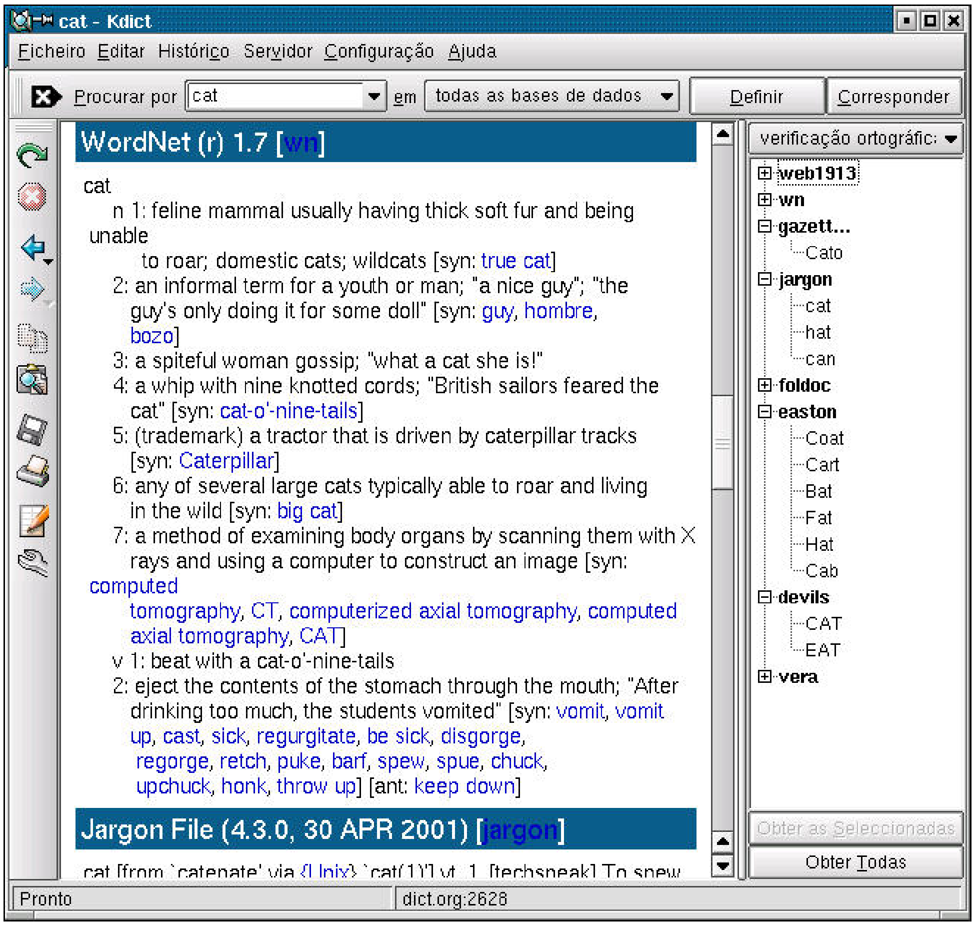



Figura 3.1: Kdict - consulta de *cat*

– o módulo Perl `Net::Dict`(Rubinstein and Bowers, 2000) – dando fácil acesso a partir de programas Perl

**Exemplo 2:** Programa Perl usando Net::Dict

```
1  use Net::Dict;
2  my $dict = Net::Dict->new('natura.di.uminho.pt');
3  $dict->setDicts('eng-por');
4  print "word: ";
5  while(<>){ chomp;
6    $eref = $dict->define($_);
7    for my $entry (@$eref){
8        ($db, $translation) = @$entry;
9        print "------($db)--\n",$translation;}}
10   print "word: "; }
```

Notas:
**linha 2** - Define o servidor dict a usar.
**linha 3** - Define o conjunto de dicionários dentro deste servidor.
**linha 5 a 10** - Aceita palavras, consulta o dicionário e mostra as respostas.

A resposta à palavra *book* é:

```
------(eng-por)--
book [buk]
     livro
     encomendar, pedir, reservar
```

### Dicionários

Para além do sistema proposto, o protocolo DICT deu origem também a um conjunto de dicionários criados especialmente para, ou transformáveis em, dicionários DICT, constituindo assim uma poderosa fonte de informação disponível a toda a comunidade.

Uma das mais conhecidas é a iniciativa FILE – a construção/adaptação de um conjunto de dicionários públicos seguindo o protocolo DICT, disponível para consulta de todos. De entre os dicionários disponibilizados aparecem: o Free OnLine Dictionary On Computer, `FOLDOC` (Howe, 1993); o WEB1913 - Webster dictionary 1913 (**?**); WordNet (Fellbaum, 1998); Gazetter; Elements – um dicionário de elementos químicos; Easton, etc.

Estes dicionários constituem projectos independentes e com diferentes formatos e modos de evolução.

Refira-se também o projecto **FreeDict** (Eyermenn, 1999) – for free bilingual dictionaries – da responsabilidade de Horst Eyermenn, onde se tem construído uma série de dicionários bilingues, aproveitando informação de várias proveniências e cuja qualidade não sendo homogénea, tem vindo a crescer. Presentemente, os dicionários seguem o formato TEI existindo uma ferramenta que produz o dicionário DICT correspondente. Refira–se ainda outros projectos de objectivos idênticos:

- *Quick* agrupando e organizando um conjunto de dicionários públicos (Gühring, 2001)
- *The Internet Dictionary project* (Chambers, 1999).
- *Babylon* (Babylon.com Ltd., 2000) – que tendo fins mais comerciais, disponibiliza uma ferramenta construtora de glossários (babylon builder) e que levou à criação de vários milhares de glossários (de qualidades e tamanhos muito variados) mas usando formatos proprietários.

## 3.7 Influência das Ciências Documentais

### Sistemas de organização de conhecimento

Em arquivos e bibliotecas, costuma designar-se por *sistemas de organização de conhecimento* (Knowledge Organization Systems – KOS) uma variedade de mecanismos de gestão e organização de grandes quantidades de conhecimento.

Os KOS pretendem abarcar vários problemas, incluindo tarefas como métodos de arrumar livros numa estante, acesso por assuntos, ou registos de autoridade.

Já que os KOS são mecanismos de organizar informação, estes sistemas são nucleares em entidades como bibliotecas, arquivos, museus, ou em qualquer situação que envolva grandes colecções.

Os dicionários partilham alguns problemas com os KOS e os KOS são em parte, casos particulares de dicionários.

Normalmente os KOS dividem-se nas seguintes classes:

**KOS baseados em listas de termos**
De entre este conjunto refira-se os ficheiros de **registo de autoridade** que são listas de termos usadas para controlar nomes (possivelmente variantes) de entidades ou valores referentes a um campo particular. São casos habituais destes ficheiros de registo de autoridade as listas de nomes de países, de autores, de entidades, de locais.

Normalmente, este tipo de lista tem estrutura simples.
Neste tipo de KOS podem ainda aparecer glossários e dicionários simples.

**KOS baseados em classificações e categorias**
Este modo de organizar a informação inclui todo um conjunto de **sistemas de classificação, taxonomias e esquemas de categorização** sendo alguns destes sistemas universalmente usados:
Dewey Decimal Classification (DDC) - árvore decimal (de grau 10) de assuntos.
Classificação Decimal Universal (CDU) – estende DDC com algum tipo restrito de facetas e valores particulares de algum aspecto.

**KOS baseados em listas de relação**

**Thesauri** [13] – estabelecem relações entre termos através de um conjunto (normalmente pequeno) de relações semânticas. Ver directivas (ISO 2788, 1986; ISO 5964, 1985).

**Redes semânticas** – estendem os thesauri alargando e diversificando os tipos de relações entre os termos.

**Ontologias** – Estendem as redes semânticas com axiomas e regras. Usam conceitos e tecnologia comum com a área da representação de conhecimento.

### Topic Maps

Com objectivos semelhantes mas vindo de uma comunidade diferente, os **topic maps** (Pepper, 2000; Ahmed, 2000; Wrightson, 2001) pretendem também definir um conjunto de tópicos (representantes dos conceitos, podendo ter um ou mais nomes) e estabelecer relações (associações) entre tópicos, e entre tópicos e documentos (resources) mas incluindo também no mesmo universo do discurso, a noção de faceta, a noção de relação e papel dentro da relação.

## 3.8 Sofisticação da informação semântica

Um outro marco importante na história dos dicionários veio das tentativas de aprofundar até às últimas consequências a descrição das propriedades de cada termo.

[13] Ao longo deste texto, considera-se thesauri como algo mais abrangente que inclua as redes semânticas e as ontologias.

Essas tentativas vieram mostrar que na realidade se está em presença de um problema de complexidade muito grande. A descrição das várias facetas de uma palavra, obrigou ao amadurecimento de modelos léxicos e à compreensão de um conjunto de características de difícil observação devido ao observado e observador estarem demasiado próximos.

Uma série de trabalhos ligados à especificação formal de várias facetas ligadas ao significado (Mel'chuk, 1988) e à combinatória (Mel'chuk, 1984) vieram mostrar que é possível formalizar aspectos que pareciam demasiado vagos através de criar tipologias classificativas descrevendo os diferentes comportamentos relacionados com um determinado aspecto e usar os termos dessa tipologia na construção de descrições lexicográficas.

## 3.9 Notas acerca da construção de dicionários em Portugal

No que diz respeito à realidade portuguesa[14], assistiu-se a um percurso semelhante ao percurso geral, embora por vezes com algum atraso temporal.

Assim, na sequência de um conjunto de dicionários suportados por fichas lexicográficas em papel vindas naturalmente da fase pre-informática:

- surgiram excelentes trabalhos suportados por um simples editor de texto
- assistiu-se ao aparecimento de projectos mais ambiciosos no que diz respeito a conteúdos e estrutura organizativa, como seja o Dicionário de Aprendizagem do Português Básico (Vilela, 1991) – que se salienta pela riqueza e elegância, com inclusão de frases exemplos e pela inclusão de um *dicionário do dicionário*.
- apareceram dicionários suportados por bases de dados desenhadas especialmente para lexicografia, como sejam o dicionário de Crioulo (Abrial, 1999), em que após uma construção usando um editor de texto (e em condições fisicamente desfavoráveis...) se passou por um processo de engenharia reversa para uma base de dados léxica específica;
- procedeu-se à criação de recursos lexicográficos visando aplicações de PLN, como sejam o Palavroso (Medeiros, Santos, and Marques, 1993; Medeiros, 1995) ou o `jspell` (ver Secção 6.1),
- houve envolvimento em projectos internacionais com apresentação de soluções criativas, como sejam o modelo de representação semântica do projecto Genelex (GENELEX, 1994b)

[14] A lista de trabalhos referidos nesta subsecção não pretende de modo algum ser completa, mas apenas sustentar algumas afirmações feitas.

- surgiram preocupações de optimizar o processo de construção. Por exemplo, a equipa da Porto Editora:
  - construiu e forneceu aos seus colaboradores uma ferramenta de construção/edição de bases de dados lexicográficas que produz um formato próximo de SGML,
  - desenvolveu um conjunto de ferramentas SGML que, para além duma muito maior homogeneidade, possibilitou a geração (quase) automática das novas edições,
  - está a estudar modos de inverter dicionários bilingues;
- uma série de dicionários ficaram disponíveis para consulta via Internet.

A falta de recursos publicamente disponíveis constitui um sério obstáculo à investigação e desenvolvimento de PLN envolvendo a Língua Portuguesa: se cada investigador tivesse que começar a sua investigação por construir um léxico novo, seria uma enorme perda de recursos. O mesmo se passa com a construção de corpora, com a construção de ferramentas ligadas à análise morfológica, à fonética, etc.

### 3.9.1 Projecto Natura

Essas necessidades/dificuldades levaram a que, associado ao trabalho que aqui se apresenta, se tivesse tido a preocupação de manter o projecto Natura (Almeida, 1997, `http://natura.di.uminho.pt`), visando construir e coleccionar ferramentas e recursos associados à linguagem natural (sempre que possível de carácter geral, mas nunca esquecendo os casos reais ligados à Língua Portuguesa).

O projecto Natura resultou de um esforço de agrupar num arquivo um conjunto de ferramentas, recursos e documentação construídos para as mais diversas finalidades, envolvendo hobbys, exercícios ligados a várias disciplinas da área de PLN, compiladores, bibliotecas digitais e outras.

No arquivo deste projecto podem ser encontradas várias das ferramentas e recursos referidos nesta dissertação. Como é habitual no mundo *open source*, o projecto mantém uma mailling list, uma árvore CVS (Fogel, 1999) para desenvolvimento cooperativo de ferramentas e recursos e uma estrutura de arquivo digital via Internet.

Sempre que possível, tentou-se obter a colaboração de alunos e de investigadores dentro e fora da instituição.

### 3.9.2 Projecto Linguateca

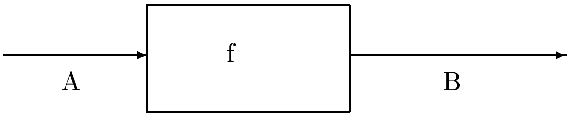
Para além da actividade ligada ao projecto Natura, relativamente a recursos publicamente disponíveis, a situação tem vindo a melhorar: a preocupação de disponibilizar aparece já referida em quase todas as definições de projectos de investigação e nalguns casos vê-se concretamente a sua realização – Saliente-se, por exemplo, a consulta e acesso aos corpora relacionado com o projecto do Centro de Recursos Distribuídos para Língua Portuguesa e com o projecto Linguateca (Rocha and Santos, 2000; Santos, 2000; Santos and Bick, 2000).

# Capítulo 4

# Dicionários: uma visão guiada por modelo

> ***Parachute.*** *– Accessoire des aérostats qui a jui d'un grande vogue à l'époque de son apparition, mais qui est totalement abandonné aujourd'hui. Cet appareil, comme son nom l'indique, etait destiné à parer aux éventualité d'une chute provoquée soit par déchirure du ballon soit par tout autre accident. [...]*
>
> *A. Ganot, Traité de Phisique, 1887*

Classicamente, a resolução de um problema corresponde a calcular os resultados pedidos em função dos dados disponíveis




Infelizmente, se nada soubermos acerca da estrutura de $A$ e $B$ a realização da função $f$ vai ver todos os valores possíveis dos dados como casos especiais.

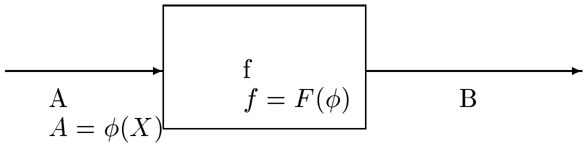
Se conseguirmos encontrar uma estrutura (com boas propriedades matemáticas) nos dados $A$

$$A = \phi(a)$$

a realização da função $f$ poderá vir *guiada* pela respectiva estrutura

$$f = cata(\phi)(x)$$

mais precisamente pelos construtores de $A$ de acordo com o respectivo catamorfismo (Augusteijn, 1999).




Analogamente se encontrarmos uma estrutura em $B$

$$B = \psi(b)$$

a realização da função $f$ poderá vir guiada pela estrutura dos seus observadores (anamorfismo[1])

$$f = ana(\psi)(Y)$$

Pode-se também guiar a definição da função $f$ por uma estrutura nova intermédia

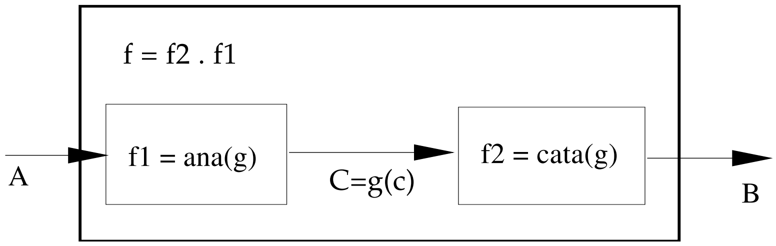
$$C = \Gamma(c)$$

e ver f como a composição de uma função $f_2$ como uma função $f_1$, sendo

$$f_1 = ana(\Gamma)(x)$$

$$f_2 = cata(\Gamma)(y)$$

e $f$ o seguinte hilomorfismo:




Por exemplo, um compilador que transforme um texto $T$ num código executável $E$ tem a sua definição guiada por uma estrutura intermédia (Ex. estrutura semântica intermédia ou estrutura sintáctica) e é a composição de um parser (anamorfismo) com gerador de código (catamorfismo).

O facto das definições das funções serem guiadas por estruturas (existentes nos dados, nos pedidos, ou em objectos intermédios) leva à existência de

[1] Ou, para ser mais genérico, paramorfismo.

semelhanças (estruturais e algorítmicas) entre enredos muito distintos e está também na base da teoria algorítmica em geral (Wirth, 1976; Horowitz and Sahni, 1977).

No caso dos dicionários, a função $f$ poderá ser a consulta, os dados $A$ serão o dicionário e o termo a pesquisar e o resultado $B$ será a informação associada ao termo.

Neste capítulo, começar-se-á por detalhar a funcionalidade normalmente associada ao dicionário. Seguidamente, serão especificados alguns modelos de dicionário. No capítulo seguinte, detalha-se modelos correspondentes a dicionários dinâmicos multi-fonte.

## 4.1 Funcionalidade associada aos dicionários

A funcionalidade oferecida pelos dicionários depende da sua estrutura e depende do seu formato físico.

Um dicionário em papel vai ser capaz de desempenhar um conjunto de funções muito menor do que um dicionário em formato digital com uma boa estrutura conceptual associada. Mesmo um dicionário em papel, quando visto juntamente com os outros elementos constituintes do seu ciclo de criação (ex. Fichas lexicográficas, base de dados léxica, programas de ordenação etc.) tem um conjunto de funções mais amplo do que poderia parecer.

A preocupação central será centrada na funcionalidade esperada para um dicionário digital.

### 4.1.1 Consulta

Uma das capacidades básicas que um dicionário terá de oferecer é dar a informação associada a um termo:

$consulta : termo \times dicionario \longrightarrow inf$
$consulta(t, d) \stackrel{\text{def}}{=}$ ...Dá informação associada a t

Sendo um dicionário algo que estabelece uma correspondência entre termo e informação podemos descrever a consulta como:

$consulta : termo \times dicionario \longrightarrow inf$
$consulta(t, d) \stackrel{\text{def}}{=} d(t)$

Em suporte de papel, esta função costuma ser realizada através de uma pesquisa proporcional sobre um conjunto de páginas com termos ordenados. Quando o domínio dos termos não está equipado com uma relação de ordem total, a realização eficiente desta função é complexa.

**Exemplo 1:** dicionários de Língua Chinesa

A procura de um caracter particular num dicionário de Chinês, constitui uma tarefa complexa, já que não há uma relação de ordem total sobre os caracteres chineses. Em situações como esta, tenta-se recorrer a transformações para domínios com *melhores* propriedades:

- ordenar os caracteres pela correspondente transcrição fonética, escrita em caracteres latinos. (Implica que se conheça bem a respectiva transcrição fonética)
- ordenar segundo um termo que descreva a sequência de acções realizadas para fazer os traços constituintes do caracter (implica saber a ordem *natural* de escrita do caracter).
- recorrer a uma ordem parcial que diminua o espaço de procura...

Se em vez de um termo for possível pedir a informação associada a um padrão sobre termo, teremos um novo tipo de consulta mais alargado, como se discutirá a seguir.

### 4.1.2 Consulta com padrões sobre os termos

A consulta com padrões corresponde a calcular o conjunto das informações associadas aos termos que verificam o padrão:

$$consulta : padrao \times dicionario \longrightarrow set(inf)$$
$$consulta(p,d) \stackrel{\text{def}}{=}$$
$$\{\left(\begin{pmatrix} 'termo \\ t \end{pmatrix}\right) \dagger d(t) \mid t \in dom(d) \wedge verifica(t,p)\}$$

O tipo de padrões aceites nesta função pode ser de maior ou menor complexidade. Exemplo:

- ter como prefixo a seguinte string
- conter uma dada string
- conter ocorrências de uma expressão regular

Esta função pode ainda ser vista como cálculo do sub-dicionário dos termos que verificam o padrão.

$$consulta : padrao \times dicionario \longrightarrow dicionario$$
$$consulta(p,d) \stackrel{\text{def}}{=}$$
$$\begin{pmatrix} t \\ d(t) \end{pmatrix}_{t \in dom(d) \wedge verifica(t,p)}$$

Dado que a implementação natural desta função envolve percorrer a totalidade do domínio do dicionário, esta função pode ser ineficiente para dicionários grandes.

### 4.1.3 Consulta com padrões sobre a informação associada

Quando o padrão de selecção for função da informação associada, temos uma nova variante da consulta – muito parecida com a anterior mas envolvendo verificações sobre a informação associada:

$$consulta : padrao \times dicionario \longrightarrow set(inf)$$
$$consulta(p,d) \stackrel{\text{def}}{=}$$
$$\begin{pmatrix} t \\ d(t) \end{pmatrix}_{t \in dom(d) \wedge verifica(d(t),p)}$$

Esta função tende a ser ineficiente para dicionários de tamanho elevado.

### 4.1.4 Consulta com padrões e filtragem

Na variante que seguidamente se apresenta, permite-se a definição de uma função de filtragem que rejeite uma parte das soluções obtidas.

A função *filtro* restringe um conjunto de respostas

$$filtro \quad = \quad set(inf) \longrightarrow set(inf)$$

$$consulta : padrao \times filtro \times dicionario \longrightarrow set(inf)$$
$$consulta(p,c,d) \stackrel{\text{def}}{=}$$
$$c(\text{...consulta anterior(p,d)...} \quad )$$

ou na versão em que a consulta devolve um sub-dicionário:

$$filtro \quad = \quad dicionario \longrightarrow dicionario$$

$$
\begin{aligned}
&consulta : padrao \times filtro \times dicionario \longrightarrow dicionario\\
&consulta(p, c, d) \stackrel{\text{def}}{=}\\
&\quad c(\begin{pmatrix} t \\ d(t) \end{pmatrix}_{t \in dom(d) \wedge verifica(d(t),p)})
\end{aligned}
$$

Essa função `filtro` pode ter muitas origens. Por exemplo, pode corresponder a um perfil de áreas de interesse: consultar, rejeitando todas as resposta que sejam fora da área de interesse ligada ao perfil.

### 4.1.5 Travessias de dicionários

Embora não seja usual considerar este tipo de funções sobre dicionários, há um conjunto de necessidades que envolvem travessias dos dicionários (ou partes deles) para geração de recursos a ser usados por outras ferramentas. Poderíamos citar várias, como sejam a impressão do dicionário, a extracção de sub-informação fonética para um sintetizador de voz, ou a criação de índices de acesso para um dicionário electrónico.

Abaixo detalha-se, a título de exemplo, duas variantes de travessias para impressão de dicionários em papel.

#### Impressão

$$
\begin{aligned}
&ImprimeDicionario : termo^{\star} \times dicionario \longrightarrow textFormatado\\
&ImprimeDicionario(termos, d) \stackrel{\text{def}}{=}\\
&\quad paginar(\frown(<formata(consulta(t, d)) \mid t \in sort(termos)>))
\end{aligned}
$$

Em muitos casos, os termos a imprimir poderão ser a totalidade dos termos contidos no dicionário (o domínio do dicionário). Há no entanto casos em que esse domínio pode não estar disponível.

Em dicionários cujas vedetas sejam termos multi-palavra, é habitual pretender-se uma visão que associe a cada palavra simples não vazia[2], o conjunto de termos onde ela ocorre e respectiva informação associada.

[2]Por palavra vazia (stop word) entenda-se o conjunto de palavras que mais aparecem numa determinada língua, normalmente preposições, artigos, e conectivos linguísticos vários.

$$
\begin{array}{l}
ImprimeDicionarioMt : termo^{\star} \times dicionario \longrightarrow textFormatado \\
ImprimeDicionarioMt(termos, d) \stackrel{\text{def}}{=} \\
\quad \begin{array}{ll}
\underline{let} & ps = \{<p, t> \mid t \in termos, p \in palavras(t)\} \\
 & ps1 = \{\pi_1(x) \mid x \in ps \wedge nao\_vazia(\pi_1(x))\} \\
 & ter = \begin{pmatrix} x \\ \{\pi_2(y) \mid y \in ps \mid \{x\}\} \end{pmatrix}_{x \in ps1} \\
 & txts = <formata(<t, \begin{pmatrix} x \\ consulta(x, d) \end{pmatrix}_{x \in ter(t)} >) \mid t \in sort(ps1)> \\
\underline{in} & paginar(\frown(txts))
\end{array}
\end{array}
$$

Onde $ps1$ é o conjunto das palavras não vazias contidas nos termos, $ter$ relaciona cada palavra com o conjunto de termos que a contêm e $txts$ é lista dos textos ligados a cada palavra.

### 4.1.6 Transformação de formatos

Há uma grande variedade de formatos de importação / exportação que aparecem envolvidos no ciclo de vida dos dicionários. Alguns deles só são aplicáveis a certos tipos de dicionários; algumas transformações são irreversíveis.

Exemplo: Se pretendermos fazer intercâmbio de dicionários usando o formato TEI, será útil construir um par de funções de tradução (idealmente a sua composição deveria dar a identidade).

$$
\begin{array}{l}
Dic2TeiSgml : dicionario \longrightarrow textTeiSgml \\
Dic2TeiSgml(d) \stackrel{\text{def}}{=} \text{exportar o dicionário em TeiSgml}
\end{array}
$$

$$
\begin{array}{l}
Sgml2Dic : textSgml \longrightarrow dicionario \\
Sgml2Dic(t) \stackrel{\text{def}}{=} \text{importar o dicionário a partir de TeiSgml}
\end{array}
$$

### 4.1.7 Construtores

Sendo a língua um elemento em contínua evolução, naturalmente é de esperar que os dicionários também tenham que sofrer enriquecimentos com novas entradas.

No entanto a utilização de construtores têm um âmbito muito mais largo: suportar o processo de construção de dicionários – o que envolve a capacidade de juntar partes de dicionários, extrair subconjuntos, etc.

Dum modo mais geral é útil a existência de funções referentes:
- ao enriquecimento de dicionários com novas entradas.
- à junção de dicionários.
- à construção de sub-dicionários.

Este grupo de funções é indispensável para a criação de *bancadas de construção de dicionários* – ambientes em que a criação, restauro, adaptação de dicionários não seja feita de modo completamente manual, mas disponha de um conjunto de ferramentas que ajudem a aumentar a eficácia e a qualidade de acabamento.

## 4.2 Dicionário: modelo

Como referimos no início do capítulo, resolver um problema consiste em calcular as respostas aos pedidos a partir dos dados. Fazer uma ferramenta consiste em escrever uma função que calcule as respostas a partir dos dados.

$$B = f(A)$$

Assumindo $f$ = função de consulta de um termo num dicionário, vai-se seguidamente analisar vários modelos de dicionário e as consequentes definições de $f$.

A definição de modelos para dicionário é, conforme dito, importante por várias razões. Por um lado, descreve o que se pretende tendo em vista implementações. Por outro lado os modelos são um elemento de análise, tendo a virtude de facilitar a discussão e as descrições de propriedades pretendidas.

Ao longo desta secção iremos percorrer um processo de sofisticação do modelo dos dados, analisando as características do dicionário final obtido.

Modelo inicial:

$$\begin{array}{lcl} A & = & termo \times dicionario \\ B & = & inf \\ dicionario & = & \text{... armazém específico...} \end{array}$$

$$f : termo \times dicionario \longrightarrow inf$$
$$f(t, d) \stackrel{\text{def}}{=} \text{... solução específica...}$$

Um dicionário *convencional* armazena explicitamente a correspondência entre termos e respectiva informação. Nesse caso, a consulta corresponde a achar a imagem associada ao termo.

Modelo de dicionário convencional:

$$\begin{array}{lcl} A & = & termo \times dicionario \\ B & = & inf \\ dicionario & = & termo \rightharpoonup inf \end{array}$$

$$f : termo \times dicionario \longrightarrow inf$$
$$f(t,d) \stackrel{\text{def}}{=} d(t)$$

O modelo anterior tem uma extensão natural que consiste em ver o dicionário repartido em várias fontes de informação ($fonteInf$), conduzindo a um **dicionário multi-fonte**:

$$\begin{array}{lcl} A & = & termo \times dicionario \\ B & = & inf \\ dicionario & = & set(fonteInf) \\ fonteInf & = & termo \rightharpoonup inf \end{array}$$

$$f : termo \times dicionario \longrightarrow inf$$
$$f(t,ds) \stackrel{\text{def}}{=} conciliar(\{d(t) \mid d \in ds \wedge t \in dom(d)\})$$

sendo a função *conciliar* responsável por produzir uma informação final com base nas várias informações parcelares. Claro que pode haver situações em que as informações se contradigam, ou que tenham relevâncias, custos, etc., diferentes. Para essas situações a função *conciliar* poderá tirar partido de um conhecimento das características gerais de cada fonte de informação. Designaremos por *metaFonteInf* a meta informação acerca da fonte de informação.

### Meta-informação

Um dicionário, para além das entradas, tem toda uma família de decisões, de objectivos, um percurso, uma área, uma época: todo um conjunto de elementos cujo conhecimento pode ser indispensável para uma melhor, mais completa, utilização da sua informação.

Daqui resulta a necessidade de reunir esta informação e armazená-la conjuntamente e a utilidade de dispor de funções referentes à consulta de características gerais dos dicionários, como sejam o seu título, os seus autores, a apresentação, as áreas temáticas, a estrutura interna da informação associada aos termos, a capacidades do dicionário em termos de tipos de consulta, etc.

Esta meta-informação é particularmente importante em contextos onde haja vários dicionários presentes, aplicações distribuídas, interfaces com necessidade de adaptação.

Modelo de dicionário multi-fonte com meta informação acerca das fontes:

$$\begin{array}{lcl}
A & = & termo \times dicionario \\
B & = & inf \\
dicionario & = & set(fonteInf) \\
fonteInf & = & \begin{array}{rcl} m & : & metaFonteInf \\ d & : & termo \rightharpoonup inf \end{array} \quad \times \\
par & = & metaFonteInf \times inf
\end{array}$$

$$f : termo \times dicionario \longrightarrow inf$$
$$f(t, ds) \stackrel{\text{def}}{=}$$
$$\quad conciliar(\{<m(fi), d(fi)(t)> \mid fi \in ds \wedge t \in dom(d(fi))\})$$

$$conciliar : set(par) \longrightarrow inf$$
$$conciliar(s) \stackrel{\text{def}}{=} ...$$

Neste caso, a função *conciliar* está a calcular a informação a partir de conjuntos de pares contendo a informação calculada por cada fonte e as respectivas propriedades gerais ($metaFonte$).

No sentido de prever a evolução e reconstrução das várias fontes de informação, surge a necessidade de incluir funções construtoras de fontes de informação, ou seja, incluir um conjunto de funções capazes de escrever expressões do tipo $fonteInf$.

Por outro lado, dada a interdependência da informação, conforme referido em 1.1.4, nem sempre é exequível ou desejável guardar a informação associada a cada fonte na forma

$$fonteInf = termo \rightharpoonup inf$$

atrás referida.

Uma extensão natural ao modelo anterior é ver as fontes de informação duma forma mais abrangente, incluindo também **fontes dinâmicas**, i.e. contendo para além da meta-informação, uma função

$$f : termo \to inf + \mathbf{1}$$

capaz de calcular a informação associada a um termo.

Essa função indicará o modo como será feito o cálculo da informação associada a um termo com base em ferramentas e recursos disponíveis no sistema (Ex. ferramentas de PLN, comandos do sistema operativo, ficheiros de texto, bases de dados, etc.).

Modelo de **dicionário dinâmico multi-fonte**[3](DDMF):

$$
\begin{array}{lcl}
A & = & termo \times dicionario \\
B & = & inf \\
dicionario & = & set(fonteInf) \\
fonteInf & = & (termo \longrightarrow inf) + (termo \rightharpoonup inf)
\end{array}
$$

$$
\begin{array}{l}
f : termo \times dicionario \longrightarrow inf \\
f(t, ds) \stackrel{\text{def}}{=} conciliar(\{fi(t) \mid fi \in ds\})
\end{array}
$$

$$
\begin{array}{l}
conciliar : set(inf) \longrightarrow inf \\
conciliar(s) \stackrel{\text{def}}{=} \ldots
\end{array}
$$

juntamente com toda uma calculadora[4] de $termo \to inf$ e de $termo \rightharpoonup inf$.

Mais uma vez, a função *conciliar* levanta as mesmas questões referidas para uma função genérica $f$ no início do capítulo: há toda a utilidade em definir um modelo de base para $inf$.

O modelo que se usa para $inf$ será, naturalmente, o tipo universal **estrutura de facetas**, definido na Secção 2.3. Esta decisão trará consequências directas sobre as funções ligadas a $inf$, analisadas no próximo capítulo.

Com fontes de informação reais, cada fonte deixa de ser uma função matemática pura, e passa a haver utilidade em modelar aspectos dinâmico-temporais. Nessa situação, a função consulta poderá ter interesse em escolher uma **estratégia** (de consulta) de modo a evitar consultar certas fontes de custo mais elevado quando não for necessário, ou garantir que certas fontes sejam consultadas só em certas situações de modo a obter um bom compromisso entre custo e completude.

[3]Para simplificar, omitiu-se a parte da meta-informação.

[4]Álgebra cujas espécies incluem $termo \to inf$ e $termo \rightharpoonup inf$, (juntamente com toda uma variedade de elementos do sistema operativo) e cujas funções incluem, para além de funções específicas, comandos do sistema operativo, ferramentas de processamento de linguagem natural.

> Acreditamos que a escrita de um modelo ajuda à compreensão de um assunto, ajuda à explicação/apresentação do mesmo, e guia no processo da implementação. Um modelo ajuda também a tornar clara a maneira como podemos/gostaríamos de ver as outras peças do sistema. Portanto, ajuda a colá-las.

No Capítulo 5 serão analisados outros modelos.

# Capítulo 5

# DDMF: uma abordagem funcional aos dicionários

*There are no answers, only cross references.*

*Weiner's law*

No final do capítulo anterior, foi introduzido o conceito de DDMF baseado em juntar um conjunto de fontes de informação dinâmicas com uma estratégia de conciliação adequada.

Ao longo deste capítulo analisa-se a função `estratégia` (que se pretende ver como parte integrante de cada DDMF) e apresentam-se exemplos de DDMF tentando realçar a potencialidade de ver os dicionários como um exercício de composição de informação distribuída por várias fontes.

## 5.1 Estrutura geral dum DDMF

Decorrente do que se disse atrás e da necessidade de associar cada resultado à identificação da sua fonte de proveniência, o modelo de dicionário dinâmico multi-fonte (DDMF) será:

$$\begin{array}{lcl} ddmf & = & \begin{array}{rcl} estrat & : & estrategia \;\; \times \\ dic & : & fontes \end{array} \\ estrategia & = & termo \times fontes \longrightarrow inf \\ fontes & = & id \rightharpoonup fonteInf \\ fonteInf & = & (termo \longrightarrow inf) + (termo \rightharpoonup inf) \end{array}$$

$$consulta : termo \times ddmf \longrightarrow inf$$
$$consulta(t, d) \stackrel{\text{def}}{=} estrat(d)(t, dic(d))$$

onde *consulta* é uma função de ordem superior que invoca as várias fontes através de uma função estratégia contida no próprio DDMF[1]. No entanto estes modelos não permitiriam compor DDMF com fontes de informação.

No modelo seguinte, incluiremos os DDMF como caso particular de fonte de informação de modo a permitir ter termos complexos em DDMF.

$$\begin{array}{lcl} ddmf & = & ... \\ fontes & = & ... \\ estrategia & = & ... \\ fonteInf & = & (termo \longrightarrow inf) + (termo \rightharpoonup inf) + ddmf \end{array}$$

$$consulta : termo \times fonteInf \longrightarrow inf$$
$$consulta(t, d) \stackrel{\text{def}}{=}$$
$$\left\{ \begin{array}{rcl} \text{is-}ddmf(d) & \Rightarrow & estrat(d)(t, dic(d)) \\ \text{is-}(termo \rightharpoonup inf)(d) & \Rightarrow & d(t) \\ \text{is-}(termo \longrightarrow inf)(d) & \Rightarrow & d(t) \end{array} \right.$$

### Um exemplo introdutório

Neste exemplo define-se um dicionário que calcula informação acerca de termos associados a computação juntando informação proveniente do `foldoc` (Free On Line Dictionary On Computer Science (Howe, 1993)), do `man` do Unix, do comando `apropos` do Unix e de um dicionário local `gloss`.

Primeiro temos que construir as várias fontes de informação.

Consideraremos que o glossário `gloss` é já um dicionário (fonte estática).

A fonte `foldoc` irá consultar o texto `foldoc.txt` (obtido a partir da distribuição oficial) usando para isso o comando agrep (Wu and Manber, 1992) que é análogo ao *grep* do Unix mas com mais funcionalidade (neste caso é importante a possibilidade de definir o separador de registos).

Deste modo `foldoc` é uma fonte dinâmica:

[1] Uma visão equivalente mais natural e elegante seria ver a função estratégia como sendo: $estrategia = fontes \rightarrow (termo \rightarrow inf)$
$consulta : termo \times ddmf \longrightarrow inf$
$consulta(t, d) \stackrel{\text{def}}{=} estrat(d)(dic(d))(t)$

$$foldoc(t) \stackrel{\text{def}}{=}$$
$$fol2ef(agrep(\left(\begin{pmatrix} 'del \\ \texttt{"==="} \end{pmatrix} \begin{pmatrix} 'opt \\ \texttt{"-wi"} \end{pmatrix}\right), t, \texttt{"foldoc.txt"}))$$

$$agrep : sym \rightharpoonup str \times str \times txt \longrightarrow str^{\star}$$
$$agrep(op, t, f) \stackrel{\text{def}}{=}$$
...registos de f contendo o padrão t de acordo com as opções op

A função `fol2ef`, que não será aqui especificada, faz a tradução de cada elemento encontrado por `agrep` para o tipo universal estrutura de facetas (EF), de acordo com o introduzido em 2.3.

De modo análogo, há que garantir que `man` e `apropos` tenham também a assinatura esperada.

$man(t) \stackrel{\text{def}}{=}$ ...executa man t e converte o resultado para uma EF

$apropos(t) \stackrel{\text{def}}{=}$ ...executa apropos t e converte o resultado para uma EF

Seguidamente, vai ser criado um dicionário `d` que junta uma estratégia de consulta `estt` com as fontes criadas:

$$estt(t, fs) \stackrel{\text{def}}{=} \begin{pmatrix} id \\ fs(id)(t) \end{pmatrix}_{id \in dom(fs)}$$

$$\begin{aligned} d' = \underline{let} \quad & fnts = \left(\begin{pmatrix} 'glossario \\ gloss \end{pmatrix} \begin{pmatrix} 'manApropos \\ apropos \end{pmatrix} \begin{pmatrix} 'man \\ man \end{pmatrix} \begin{pmatrix} 'foldoc \\ foldoc \end{pmatrix}\right) \\ \underline{in} \quad & ddmf(fnts, estt) \end{aligned}$$

A estratégia `estt` que se definiu está a consultar a totalidade das fontes, guardando todas as respostas obtidas e etiquetando a informação que cada fonte devolve com o respectivo identificador de fonte.

Esta estratégia de consulta é aplicável a qualquer DDMF. Na próxima secção, analisaremos outras estratégias gerais.

## 5.2 Estratégias de selecção

As várias **estratégias** vão ser constituídas combinando **critérios de selecção** de fontes com **critérios de conciliação** das respostas. Vamos, seguidamente, centrar-nos num conjunto de critérios de selecção que podem ser usados numa grande variedade de casos. Falaremos mais tarde nos modos e critérios de conciliação de informação.

### 5.2.1 Estratégias gerais

#### Consultar todas as fontes

Se dispomos de um conjunto de fontes de informação, a estratégia mais natural consiste em formar um conjunto com todas as respostas por elas calculadas:

$$CompParal : termo \times fontes \longrightarrow inf$$
$$CompParal(t, fs) \stackrel{\text{def}}{=} \{consulta(t, fs(id)) \mid id \in dom(fs)\}$$

Com base nesta função, podemos também definir o construtor de DDMF associado, a que chamaremos `mkCompParal`, do seguinte modo:

$$mkCompParal : fontes \longrightarrow ddmf$$
$$mkCompParal(fs) \stackrel{\text{def}}{=} ddmf(CompParal, fs)$$

Alternativamente, poderíamos associar a cada resposta obtida um identificador *id* de proveniência (estratégia usada no exemplo atrás apresentado):

$$TagCompParal(t, fs) \stackrel{\text{def}}{=} \begin{pmatrix} id \\ consulta(t, fs(id)) \end{pmatrix}_{id \in dom(fs)}$$

Analogamente, o respectivo construtor de DDMF associado seria:

$$\begin{array}{l} mkTagCompParal(fs) \stackrel{\text{def}}{=} \\ \quad ddmf(TagCompParal, fs) \end{array}$$

Outra possibilidade é perguntar às várias fontes e dar a informação cuja qualidade anunciada seja máxima:

$$\begin{array}{l} MaxCompParal(t, fs) \stackrel{\text{def}}{=} \\ \quad \underline{let} \;\; reps = \{consulta(t, fs(id)) \mid id \in dom(fs)\} \\ \quad \phantom{\underline{let}} \;\; qmax = max(\{qualidade(r) \mid r \in reps\}) \\ \quad \underline{in} \;\; choice(\{x \mid x \in reps \wedge qualidade(x) = qmax\}) \end{array}$$

Pode haver mais que uma resposta com qualidade máxima. Na função apresentada está a ser feita a selecção aleatória (*choice*) de uma das respostas. Há várias outras hipóteses de conciliar as melhores respostas.

Estas estratégias baseiam-se na composição paralela das várias fontes ou, dito de um modo diferente, no produto das respostas.

**Consultar por uma ordem**

Outras possibilidades seriam:

1. perguntar sequencialmente às várias fontes até que uma delas dê uma resposta não vazia;
2. perguntar às várias fontes por ordem crescente do custo da consulta até que uma delas dê uma resposta não vazia;
3. perguntar às várias fontes por ordem decrescente da qualidade anunciada até que uma delas desse uma resposta não vazia;
4. perguntar e conciliar as respostas até que se obtenha uma informação de $qualidade \geq x$.

Estas soluções são estratégias que não necessitam de consultar a totalidade das fontes.

As estratégias que implicam consulta às fontes por uma determinada ordem, correspondem a:

$$pergPorOrdem : id^{\star} \times termo \times fontes \longrightarrow inf$$

$$pergPorOrdem(l,t,fs) \stackrel{\text{def}}{=}$$

$$\begin{cases} l\ is\!-\!<> & \Rightarrow\ () \\ l\ is\!-\!< h : ta > & \Rightarrow\ \underline{let}\quad a = consulta(t, fs(h)) \\ & \quad\ \underline{in}\ \begin{cases} a = () & \Rightarrow\ pergPorOrdem(ta, t, fs) \\ \underline{otherwise} & \Rightarrow\ a \end{cases} \end{cases}$$

ou seja, se a lista dos $id$ de fontes a consultar tiver um primeiro elemento $h$ e uma continuação $ta$, consultaremos a fonte $h$ e se ela der uma resposta vazia, continuaremos a consultar as outros fontes.

Na Secção 11, página 79, é apresentado um exemplo dum DDMF usando esta estratégia.

### 5.2.2 Meta-informação acerca das fontes

Conforme referido em 4.2, página 63, um dicionário contém outra informação para além das definições dos termos.

Como é habitual em meta-informação pode incluir-se informação ligada a catalogação, classificação da fonte em causa. Exemplo: título, autores, data, url, domínio temáticos, copyright, etc. Para além deste tipo de informação, é importante conhecer-se:

- as capacidades da fonte (exemplo: ser ou não capaz de pesquisar por padrão; ser ou não capaz de devolver o seu domínio),
- a estrutura das respostas dadas pela fonte,
- a evolução temporal da fonte.

Para que algumas das estratégias atrás referidas sejam possíveis, é necessário que a meta-informação associada à fontes forneça (de modo estático ou dinâmico) as indicações necessárias para a tomada de decisões.

Refira-se como exemplo, alguma meta-informação necessária para tornar possível definir estratégias adaptativas:

**qualidade anunciada de uma fonte** – esta característica deve estimar a qualidade geral de uma fonte (independentemente da pergunta): a riqueza típica das respostas; a probabilidade de saber dar uma resposta. Por exemplo, a qualidade da fonte `man` é elevada (desde que haja resposta).

**medida da qualidade de uma resposta** – estimativa da qualidade da resposta particular (após ser conhecida a pergunta e consultada a fonte). Por exemplo, se tivermos uma fonte `grep` que procure linhas que verifiquem um padrão, poderemos querer dizer que a riqueza das respostas varia conforme o número de linhas encontradas:

$$qualidade(r) \stackrel{\text{def}}{=} \underline{let}\ \ nl = NumeroDeLinhas(r)$$
$$\underline{in}\ \begin{cases} nl \geq 30 & \Rightarrow\ 100 \\ \underline{otherwise} & \Rightarrow\ nl/30 \times 50 \end{cases}$$

**custo da consulta a uma fonte** – dependente da situação concreta, o custo aqui referido pode estar ligado ao tempo necessário para obter a resposta, eficiência em geral, custos monetários, etc.

Algumas estratégias poderão tirar partido de meta-informação ligada à evolução temporal das fontes. Por exemplo, uma fonte que seja constante, que evolua pouco ao longo do tempo, ou que evolua de forma monotónica, pode permitir cópias locais (com ou sem período de validade), caches, diminuindo os custos de consulta.

Alguma meta-informação acerca da evolução temporal da informação poderá também ser útil para geração de certas funções de *house keeping* do dicionário[2].

[2] Este tipo de funções não são abordadas neste texto.

Infelizmente, a criação de alguma desta meta-informação envolve uma certa complexidade. A escolha da meta-informação adequada a uma situação concreta vai ser um compromisso entre o esforço envolvido em criar essa informação e o benefício obtido a nível estratégico e genericamente a nível da qualidade.

### 5.2.3 Estratégias guiadas por sistemas de produções

Os sistemas de produção são uma solução habitualmente usada para definir estratégias em Inteligência Artificial. Poderemos simplificadamente caracterizá-los por um conjunto de regras (as produções do tipo $antecedente \rightarrow consequente$ ou $condicao \rightarrow reaccao$) que são aplicadas sucessivamente a um estado até que não haja mais antecedentes válidos (por cada $condicao$ verdadeiro é realizada a respectiva $reaccao$).

No caso presente o estado é descrito pela informação representada no tipo universal `ef`.

$$\begin{array}{lcrcll} sisProd & = & prod^{\star} & & & \\ prod & = & ant & : & (ef \longrightarrow Bool) & \times \\ & & cons & : & (ef \longrightarrow ef) & \end{array}$$

$$estratSisProd : sisProd \times ef \longrightarrow ef$$
$$estratSisProd(sp, e) \stackrel{\text{def}}{=}$$

...aplica os consequentes das produções enquanto houver
... antecedentes válidos

Esta estratégia consiste em usar as regras do sistema de produção de modo a que, mediante o estado actual (mediante o conhecimento actual acerca dum termo), se vá escolhendo que fontes consultar e decidir a altura própria para considerar a resposta como calculada.

Esta não é bem uma nova estratégia, mas sim um modo de implementar dinamicamente uma estratégia híbrida. Esta estratégia pertence à classe das estratégias adaptativas. Para exemplos de outras estratégias calculadas dinamicamente, ler (Almeida and Henriques, 1998).

## 5.3 Filtragem

Conforme descrito em 4.1.4, a filtragem está associada a diminuir o conjunto de respostas obtidas.

A redução do número de respostas produzidas pode ser incorporado na estratégia (filtragem orientada à fonte de informação), ou pode actuar no conjunto de respostas devolvidas pela função estratégia (filtragem orientada à informação): depois de calculado o conjunto das respostas, a filtragem traduz-se em definir uma função que seleccione um seu subconjunto. Esta situação é um caso particular de função de conciliação.

$$filtra : set(ef) \longrightarrow set(ef)$$
$$filtra(resp) \stackrel{\text{def}}{=} ...$$

### 5.3.1 Filtragem orientada à fonte informação

Este tipo de filtragem corresponde, no fundo, à definição de uma estratégia, e como tal devia estar incluído na secção anterior.

Esta estratégia consiste em:

- reduzir o volume de respostas através de indicação explícita do subconjunto das fontes de informação a ser consultado.
- reduzir o volume de respostas através de indicação de uma expressão sobre a meta-informação que calcule o conjunto das fontes de informação a consultar. (semelhante ao mecanismo de *brocker* usado na área de *information retrieval*)

Este tipo de estratégia, para além de diminuir o volume de respostas, tem consequências directas no desempenho.

### 5.3.2 Filtragem orientada à informação

#### Filtragem com base numa expressão sobre valores de campos

O caso mais simples de filtragem corresponde a definir uma condição de aceitação (ou de rejeição) que exclua informação em função de algumas das suas características.

$$filtra : set(ef) \times (ef \longrightarrow Bool) \longrightarrow set(ef)$$
$$filtra(resp, cond) \stackrel{\text{def}}{=} \{x \mid x \in resp \wedge cond(x)\}$$

Exemplo: excluir as respostas *antigas* – $resp(data) > d$

**Filtragem conceptual**

Este tipo de filtragem implica a existência de uma estrutura classificativa e implica a existência da atributos especiais contendo a classificação.

$$
\begin{aligned}
&filtra : set(ef) \times classe \times thesaurus \longrightarrow set(ef)\\
&filtra(resp, assunto, ec) \stackrel{\text{def}}{=}\\
&\quad \{x \mid x \in resp \land acercaDe(x, assunto, ec)\}
\end{aligned}
$$

Exemplo: obter informação referente a um assunto $a$, numa fonte que usa um thesaurus $ec$ como estrutura classificativa:

$$
\begin{aligned}
&acercaDe : ef \times classe \times thesaurus \longrightarrow Bool\\
&acercaDe(r, a, ec) \stackrel{\text{def}}{=}\\
&\quad \underline{let} \quad interesse = especificos(a, ec)\\
&\quad \underline{in} \quad \exists x \in getAll(\texttt{"class"}, r) : x \in interesse
\end{aligned}
$$

Este tipo de filtragem corresponde a seleccionar os elementos em que haja, pelo menos, uma classificação que corresponde ao assunto pretendido.

Corresponder ao assunto pretendido é um conceito que depende da estrutura classificativa em causa. Exemplo: pertencer ao fecho transitivo segundo a relação de `ser termo específico de`.

Consultas com filtragem deste tipo aparecem descritas na Secção 8.4.

## 5.4 Potencialidades da abordagem: análise de um exemplo

O exemplo que se segue, um dicionário enriquecido, só poderá ser entendido na totalidade após a leitura de vários dos capítulos seguintes. No entanto, optou-se por o apresentar já, para que melhor se entenda o ambiente de criação de dicionários que tem vindo a ser discutido.

Começa-se por descrever as fontes de informação:

**Fonte 1 : dicionário convencional**

A primeira fonte é uma versão estruturada ($term \rightharpoonup ef$) de um dicionário convencional. Esta fonte é o resultado de se ter feito um (complexo) trabalho de engenharia reversa sobre um dicionário de Português Básico[3]: partindo dos detalhes tipográficos, procedeu-se à interpretação da versão electrónica disponibilizada, após o que foi possível obter a estrutura explícita associada a cada entrada.

$dict1' = ...$ resultado de engenharia reversa

### Fonte 2 e 3 : pesquisas morfológicas sobre listas de frases

A 2.ª e 3.ª fontes – `exem()` e `prov()` – são criada com base numa ferramenta de extracção de frases em textos – a função $nlgrep$[4] que dada uma palavra $p$, procura linhas que a contenham, ou que contenham outras que sejam morfologicamente derivadas de $p$.

São duas fontes de informação dinâmicas: uma que devolve conjuntos de frases exemplo, outra conjuntos de provérbios, contendo a palavra em causa.

Os textos são respectivamente `fraExe` – um conjunto de frases exemplo retiradas do dicionário da fonte 1, `dict1`[5] – e `Proverbio` – que contém um conjunto de provérbios.

Assim: $exem(x) \stackrel{\text{def}}{=} nlgrep(x, fraExe)$

$$prov(x) \stackrel{\text{def}}{=} nlgrep(x, Proverbio)$$

### Fonte 4: dicionário de calão e expressões idiomáticas em DPL

A 4.ª fonte é um dicionário de expressões idiomáticas – calao.dic[6] – descrito na linguagem DPL – uma linguagem de domínio específico para descrição de dicionários, que será apresentada em 7.1.

A função $mkdplddmf$ vai compilar um texto na linguagem DPL produzindo

[3] Agradece-se a Diana Santos, a José Carlos Medeiros e ao grupo de Linguagem Natural do INESC a possibilidade de ter usado a versão electrónica do dicionário Asa (Vilela, 1991), para o trabalho investigação referido nesta tese, ao abrigo de uma colaboração com o referido grupo, em 1994.

[4] Pertencente à biblioteca `jspell`, ver Secção 6.1.

[5] $fraExe' = getAll(ran(dict1),$ `"exemplo"`$)$

onde a função $getAll$ (ver Secção 2.3.1) faz a travessia de toda a informação contida em $dict1$, extraindo o conjunto de todas as ocorrências do campo `exemplo`.

[6] Descrita em mais detalhe na Secção 8.2.

um dicionário em formato berkeley DB (Marquess, 1995; Sarathy, 1998).

$$d' = mkdplddmf(\texttt{"calao.dic"})$$

Dado que os termos do dicionário são multi-palavra, aplicou-se seguidamente a função $mkdbddmf$[7], que produz uma fonte de informação capaz de responder a pesquisas por padrão, resultando então a 4ª fonte pretendida.

$$dac' = mkdbddmf(d)$$

### Fonte 5: analisador morfológico `jspell`

Um analisador morfológico é capaz de determinar as possíveis análises de cada palavra. Cada análise é uma estrutura contendo um conjunto de propriedades derivadas em função da composição morfológica da palavra.

O analisador morfológico `jspell`, descrito na Secção 6.1, faz a análise de uma dada palavra, com a função $fea$, constituindo deste modo a 5ª fonte de informação:

$$jspell(x) \stackrel{\mathrm{def}}{=} fea(x, \texttt{"port"})$$

Exemplo: fea("comprador","port") dá:

```
1 { [ CAT -> a_nc
2     rad -> comprar
3     FSEM -> dor
4     N -> s
5     G -> m
6     TR -> t ]
7 }
```

Notas:

**linha 1 a 7** - A resposta é um conjunto (neste caso, singular) de análises. Cada análise é uma estrutura de facetas.

**linha 1** - Classe gramatical: adjectivo ou nome comum.

**linha 2** - Lema: comprar.

**linha 3** - Função semântica de derivação: "dor".

**linha 4** - Número: singular.

**linha 5** - Género: masculino.

**linha 6** - Transitividade: transitivo.

[7] Descrita em mais detalhe na Secção 9.2.4.

### Fonte 6: pesquisa em dicionário após normalização da palavra

Esta fonte calcula as possíveis origens de um termo (usando a função $fea$, de modo idêntico à fonte 5) e seguidamente, enriquece a informação obtida com a consulta ao dicionário $df$ (o dicionário final que junta as várias fontes) quando o lema (forma lematizada do termo procurado) é diferente do termo.

$$
\begin{array}{l}
font6(x) \stackrel{\text{def}}{=} \\
\quad \begin{array}{ll}
\underline{let} & fe = fea(x, \texttt{"port"}) \\
 & r = \{y \mid y \in fe \wedge y('rad) \neq x\} \\
\underline{in} & \{f \dagger \left( \begin{pmatrix} 'infrad \\ consulta(f('rad), df) \end{pmatrix} \right) \mid f \in r\}
\end{array}
\end{array}
$$

Exemplo: font6("comprador") dá :

```
[  name -> comprador
   G -> m
   CAT -> a_nc
   N -> s
   rad -> comprar
   infrad -> [
      Cla -> v. tr.
      name -> comprar
      Fra -> [
         I -> < (pessoa) @ coisa
            1) O Miguel comprou um carro novo.
            2) A Filipa comprou mais um livro.>
         II -> < (fig.) (pessoa) @ pessoa
            1) O advogado comprou as testemunhas.
            ...>]
      conj -> 1
      Der -> [
         Comprador -> [
            Cla -> n. m.
            trad -> O José é o comprador daquela mota.
            ex -> O José comprou aquela mota. ]]
      Fon -> <kõprár>
      Sig -> < (pessoa) @ coisa
               (fig.) (pessoa) @ pessoa>
      Sem -> [ 1 -> ...
               2 -> ... ]]
]
```

Notas:

**linha 1 a 5** - Idêntico ao exemplo anterior.

**linha 6 a 26** - Atributo infrad – informação (estrutura de facetas) adicional acerca do lema *comprar* calculado pelo dicionário `dict1` (dicionário de Português Básico)

### Estratégia

Uma vez apresentadas as 6 fontes que vão ser usadas, é altura de definir a estratégia que, juntando essas fontes, dê origem ao DDMF pretendido.

Como referido na Secção 5.2, o meio mais simples de juntar as várias fontes é a sua composição paralela, ou seja, é juntar as respostas que todas as fontes forem capazes de produzir.

$$
\begin{aligned}
df' = \underline{let} \quad & f = \left( \begin{pmatrix} 'asa \\ dict1 \end{pmatrix} \begin{pmatrix} 'asa2 \\ font6 \end{pmatrix} \begin{pmatrix} 'exem \\ exem \end{pmatrix} \begin{pmatrix} 'prov \\ prov \end{pmatrix} \begin{pmatrix} 'morf \\ jspell \end{pmatrix} \begin{pmatrix} 'calao \\ dac \end{pmatrix} \right) \\
\underline{in} \quad & mkCompPar(f)
\end{aligned}
$$

No entanto, em muitos casos, as respostas obtidas contêm muita redundância (o que piora a qualidade da resposta). Se, por exemplo, o termo questionado pertencer ao dicionário `dict1`, a resposta produzida pela 1ª fonte já contém informação morfológica e exemplos, ou seja, já contém uma versão resumida da informação que outras fontes (por exemplo a 2ª e a 5º) produzem.

Recordando a Secção 5.2, uma estratégia melhor consiste em:

- perguntar à fonte mais rica: dict1
- se dict1 não souber responder, perguntar à fonte 6
- se a fonte 6 não souber responder perguntar a todas as outras ao mesmo tempo.

Usando os construtores associados a cada estratégia geral, pode-se construir o dicionário final `df` como:

$$
\begin{aligned}
df' = \underline{let} \quad & fts = \left( \begin{pmatrix} 'exem \\ exem \end{pmatrix} \begin{pmatrix} 'prov \\ prov \end{pmatrix} \begin{pmatrix} 'morf \\ jspell \end{pmatrix} \begin{pmatrix} 'calao \\ dac \end{pmatrix} \right) \\
& d = mkCompPar(fts) \\
& ordem = <'f1,'f2,'f3> \\
\underline{in} \quad & mkPergPorOrdem(ordem, \left( \begin{pmatrix} 'f1 \\ dict1 \end{pmatrix} \begin{pmatrix} 'f2 \\ font6 \end{pmatrix} \begin{pmatrix} 'f3 \\ d \end{pmatrix} \right))
\end{aligned}
$$

### Exemplos de consultas

Seguidamente, será feita a consulta da palavra "rebentamento".

A palavra inquirida não existe na fonte $dict1$ pelo que é consultada a fonte $font6$; esta fonte calcula a sua análise morfológica e dado que o lema é diferente da palavra, volta a consultar o dicionário *df* com `rebentar`. A palavra `rebentar` também não existe em $dict1$ e $font6$ também não responde devido ao lema ser igual à palavra pelo que é consultada a fonte $daux$, ou seja é feita a consulta paralela de "exem", "prov", "jspell", "dac".

A resposta seguinte não corresponde exactamente às fontes descritas, mas sim a uma versão mais sofisticada com reescrita (que aparece descrita na Secção 9.5) que, por exemplo, acrescentou o campo `Sem` (semântica) com o conteúdo calculado em função do sufixo usado – `dor`!

```
1  { name => rebentamento
2    Sem => [ Acção de rebentar.]
3    TR => t,i
4    rad => rebentar
5    infrad => {
6      name => rebentar
7      Fra => [
8        Na Primavera rebentam os botões das flores.
9        A corda rebentou com o excesso de peso.
10       Quando o corredor entrou no estádio, rebentou uma...
11       A câmara fez cortar as ... a rebentar o pavimento.
12       Rebentou uma conduta da rede de distribuição de água à cidade.
13       O prédio ficou todo inundado, quando a canalização rebentou.]
14     idiomáticas => [
15       { name => rebentar pelas costuras}
16       { name => rebentar a escala
17         sem => ter uma classificação excelente}
18     ]
19   }
20   G => m
21   CAT => nc
22   Gra => pode reger a preposição 'de' seguido do elemento
23          que se vai rebentar.
24   N => s}
```

Notas:

**linha 2,22** - A informação relacionada com os atributos *Sem* e *Gra* foram calculadas pelo sistema de reescrita com base no facto de ter sido usado o prefixo de derivação `dor`.

**linha 7 a 13** - Resultado de procurar ocorrências de palavras derivadas de `rebentar` no conjunto das frases exemplo[8].

[8]Apesar do termo `rebentar` não existir no dicionário $dict1$, aparecem palavras a ele associadas em frases exemplo construídas para outros termos.

**linha 14 a 17** - Resultado da pesquisa no dicionário de calão e expressões idiomáticas.

Como se pode notar nesta resposta, é possível obter informação útil acerca de um termo mesmo quando ele não consta dos dicionários tradicionais de que dispomos.

# Capítulo 6

# Funções: ferramentas de processamento de linguagens

*Que a imaginação te engorde e a matemática te emagreça*
*Agostinho da Silva*

Conforme referido na introdução, nesta dissertação pretende-se ver as ferramentas como funções que aumentem o poder expressivo do nosso ambiente de construção de dicionários: que enriquecem a álgebra dos DDMF.

Esse enriquecimento pode ser obtido de vários modos:

- através de programas que sejam capazes de devolver informação,
- através de programas ou funções capazes de extrair informação de alguns recursos existentes,
- através de programas ou funções que façam as necessárias conversões/-adaptações de tipos para que outras ferramentas/recursos possam ser usadas no ambiente.

Nesta secção, falaremos de algumas ferramentas construídas referentes a processamento de linguagem natural e de outras que têm o objectivo mais geral de facilitar a importação e transformação de recursos.

Algumas das ferramentas que se apresentam não estão convenientemente exploradas no problema concreto dos dicionários aqui em estudo. Por vezes poderão até parecer um pouco excessivas ou deslocadas na situação presente. Isto deve-se um pouco à vontade que se teve de construir coisas que pudessem ter um leque mais vasto de utilização, e ao mesmo tempo estudar uma área mais abrangente[1].

As várias ferramentas que de seguida são referidas, não são independentes entre si. Como se pode ler no Apêndice A, quase todas as apresentadas são usadas em mais que um sítio.

Sempre que possível tentou-se recorrer à criação de *domain specific languages* como meio de arranjar modos de especificação que ajudassem a realçar os conceitos base subjacentes (como meio de ajudar a assimilar conceitos).

Nas secções seguintes discutir-se-ão as seguintes ferramentas:

- `jspell` – um analisador morfológico (indispensável em PLN),
- `nllex` – um gerador de analisadores léxicos para PLN (usa `jspell`),
- YaLG – um ambiente para escrita de gramáticas lógicas para PLN (usa `nllex`, `jspell`),
- Ptc – um processador de termos multi-palavra (usa `jspell`),
- EMS – um etiquetador morfo-sintáctico (usa `jspell`),
- XML::DT – um processador de textos estruturados XML,
- MKTEXTRR – um gerador de sistemas de reescrita textuais,
- Library::* – um processador de ontologias, catálogos e de bibliotecas digitais.

Ao analisar algumas delas, serão apresentados alguns exemplos que acabam por ser também ferramentas.

[1] Ou seja, ainda que por vezes parecendo reinventar a roda, pretendeu-se experimentar vários ambientes e várias estratégias de solução, e sempre que possível construir elementos que pudessem ser usados didacticamente.

## 6.1 Analisador morfológico `jspell`

Esta secção surge na tese devido a:

- o `jspell` constituir um recurso de utilidade geral
- constituir um exemplo relevante de construção de software por abstracção, enriquecimento, refinamento
- constituir em si um DDMF
- constituir um função PLN directamente utilizável em DDMF

### 6.1.1 Introdução

Grande parte das aplicações em linguagem natural necessitam de um mecanismo léxico de classificação que, dada uma palavra, obtenha várias informações a seu respeito. Pode ser relevante saber se a palavra existe ou não no dicionário, mas não menos importante é conhecer a sua Origem ou a Categoria gramatical a que pertence. Essas características bem como o Género, o Número, etc, constituem informação indispensável ao funcionamento dos níveis sintáctico e semântico.

| |
|---|
| Um analisador morfológico é imprescindível a uma grande parte das aplicações de processamento de linguagem natural |

A enumeração exaustiva de todas as palavras e desinências é, não apenas muito morosa mas mesmo virtualmente impraticável, dada a grandeza dos números em questão. Uma forma mais racional de obter o mesmo resultado é armazenar apenas uma lista com as raízes das palavras e ter regras de formação de desinências, devendo cada uma das palavras da lista ter associado o conjunto de regras de formação que lhe podem ser aplicadas.

A morfologia está ligada ao estudo da constituição das palavras.

$$palavra = f(radical^*, afixos^*)$$

A estrutura morfológica e a sua complexidade variam muito com a língua.

#### Tipologia

Habitualmente, (Crystal, 1997) as línguas costumam ser divididas em:

- línguas isolantes (ex. chinês, vietnamita) que no extremo seriam caracterizadas por:

    - terem palavras monossilábicas e invariantes
    - não terem inflexão (verbal, número, etc.)
    - terem frequentemente com alguma complexidade tonal
- línguas aglutinantes (ex. groenlandês, finlandês, turco) – caracterizadas por:
    - anexação de sufixos à palavra-base (flexão e um série de outros fenómenos).

```
nunarsuami (KL) = num grande país
   nunaq=país
   -suaq=grande
   -mi=em
talossani (FI) = de minha casa
   talo=casa
   -ssa=em
   -ni=minha
```

- línguas flexivas (nas quais pode ser enquadrado o Português) apresentando palavras polissilábicas:
    - flexionadas de acordo com tempo, número, género, etc.

```
daremos
   dar + emos
   verbo + 1ª pessoa plural, futuro
livros
   livro + s
   subst.+plural
```

A maior parte das línguas está num estado intermédio e move-se ao longo da história. Por exemplo a Língua Portuguesa pode ser descrita como flexiva mas tem um componente aglutinante com certos sufixos como o *mento* (rebentar/rebentamento).

De um ponto de vista de complexidade, o número de palavra que uma palavra pode criar é muito variável sendo o Português pelo menos uma ordem de grandeza mais complexo que o Inglês[2] e o Finlandês duas ordens de grandeza mais complexos que o Inglês[3].

Enquanto a Língua Inglesa tem uma estrutura morfológica relativamente simples, levando a que em alguns estudos seja simplesmente ignorada, o

[2] Aproximadamente duas ordens de grandeza se considerarmos os clíticos verbais.

[3] Não será por acaso que há vários finlandeses a assinar trabalhos importantes na área da morfologia... (Koskenniemi, 1983; Karttunen, 1991; Grefenstette and Tapanainen, 1994)

mesmo não sucede com as línguas latinas como o Português e muito menos com o Finlandês (Koskenniemi, 1983).

As diferentes características das línguas levam a que não seja fácil a construção de modelos e de analisadores morfológicos de âmbito universal. O analisador morfológico `jspell` encaixa-se na zona das linguagens flexivas. Presentemente, existe publicamente disponível um dicionário e respectivas regras morfológicas para Português e Inglês.

### 6.1.2 Interface de `jspell`

De um modo compacto apresenta-se a seguir as funções principais da biblioteca `jspell`

**State** $J : jspellDic$

$jspellDic$ = $regras \times dic$
...a detalhar mais à frente
$ef$ = ...Feature structures
$idRegra$ = ...Identificador de esquema de derivação;
também referido como flag
$txt$ = ...Texto
$filert$ = ...Texto em que se acrescentou a cada linha as
formas lematizadas

$fea : palavra \longrightarrow set(ef)$
$fea(t) \stackrel{\text{def}}{=}$
... Função principal de análise de palavra.
Dá conjunto de feature structures correspondentes às várias análises possíveis de t

$der : palavra \longrightarrow set(palavra)$
$der(t) \stackrel{\text{def}}{=}$ ... o conjunto das palavras derivadas de t

$rad : palavra \longrightarrow set(palavra)$
$rad(t) \stackrel{\text{def}}{=}$ ... as palavras de que t pode derivar

$nlgrep : opt \times palavra \times set(txt + filert) \longrightarrow set(linha)$
$nlgrep(o,t,fs) \stackrel{\text{def}}{=}$
... todas as linhas (ou registos) dos ficheiros que contêm uma palavra derivada de t; as opções permitem escolher a variante de funcionamento pretendida.

As opções permitem, por exemplo, definir o número máximo de soluções pretendidas, ou definir propriedades dos textos onde vai ser feita a procura: escolher o separador de registo, ou indicar o tipo de ficheiros.

$flags : palavra \longrightarrow set(idRegra)$
$flags(t) \stackrel{\text{def}}{=}$
... conjunto de identificadores de esquema de derivação associados a t no dicionário

$mkradtxt : txt \longrightarrow filert$
$mkradtxt(f) \stackrel{\text{def}}{=}$ ... texto com as palavras e seus radicais. Ver nlgrep

$jspell\_dict : filename \longrightarrow jspellDic$
$jspell\_dict(id) \stackrel{\text{def}}{=}$ ... Carrega para o estado J o dicionário contido em id

$set\_mode : Tmodo \longrightarrow$
$set\_mode(modo) \stackrel{\text{def}}{=}$ ... Selecciona o modo de funcionamento pretendido

Funções de gestão de dicionários jspellDic:

$add\_flag : jspellDic \times idRegra \times set(palavra) \longrightarrow jspellDic$
$add\_flag(d,id,ps) \stackrel{\text{def}}{=}$ ... acrescenta esquemas de derivação a palavras

$rem\_flag : jspellDic \times idRegra \times set(palavra) \longrightarrow jspellDic$
$rem\_flag(d,id,ps) \stackrel{\text{def}}{=}$
... remove esquemas de derivação associadas a palavras

$add\_word : jspellDic \times palavra \times ef \times set(idRegra) \longrightarrow jspellDic$
$add\_word(d,p,cl,regras) \stackrel{\text{def}}{=}$
... dada uma palavra, a sua classificação e os esquemas de derivação, acrescenta-a a um dicionário

$$rem\_word : jspellDic \times palavra \longrightarrow jspellDic$$
$$rem\_word(d,p) \stackrel{\text{def}}{=} \text{... remove palavra de um dicionário}$$

**Exemplo 1:** Análise de palavras usando `jspell`

Segue-se um pequeno exemplo de análise morfológica de palavras simples usando o dicionário de Português. Cada palavra analisada pode ter zero ou mais análises, sendo cada análise descrita através duma estrutura de facetas simples:

```
1  > gatinho
2       { N => s
3         P => 1
4         CAT => v
5         rad => gatinhar
6         T => p
7         TR => i}
8       {
9         N => s
10        G => m
11        GR => dim
12        CAT => nc
13        rad => gato}

14 > molhador
15      { N => s
16        G => m
17        FSEM => dor
18        CAT => a_nc
19        rad => molhar
20        T => inf
21        TR => t
22        unknown => 1}
```

Notas:

**linha 2 a 7** - Primeira análise de `gatinho`.
**linha 2** - Número singular.
**linha 3** - $1^{\text{a}}$ pessoa.
**linha 4** - Classe gramatical: verbo.
**linha 5** - Radical (forma lematizado): gatinhar.
**linha 6** - Tempo: presente indicativo.
**linha 7** - Transitividade: intransitivo.
**linha 8 a 13** - Segunda análise de `gatinho`.
**linha 12** - Classe gramatical: nome comum.
**linha 15 a 22** - Primeira (e única) análise de molhador.

**linha 17** - Função semântica de derivação: `dor`.

**linha 18** - Classe gramatical: adjectivo ou nome comum.

**linha 22** - Tipo de certeza: palavra desconhecida mas possível de obter usando uma regra morfológica que não pertence às oficializadas para o radical em causa. Através da escolha adequada do modo de funcionamento, pode-se *desligar* ou alargar este tipo de análise menos garantida.

### 6.1.3 História do desenvolvimento do `jspell`

Para desenvolver um analisador morfológico com um pequeno dispêndio de recursos, optou-se por:

- procurar uma ferramenta *open-source* que tivesse semelhanças (`ISpell`)
- escrever o seu modelo formal sumário
- escrever o modelo formal pretendido e analisar as diferenças
- analisar as alterações a realizar no código existente com base nessas diferenças
- escrever ainda as partes que faltassem
- analisar até que ponto se poderiam usar características do `ISpell` que não tinham sido pedidas para o analisador morfológico.

Deste modo, o desenvolvimento do `jspell` foi fortemente baseado no corrector ortográfico internacional `ISpell` (Kuenning, 1993). O nome `jspell` surgiu como sendo ("i"+1)spell, ou seja, como resultado de aplicar um processo de enriquecimento ao `ISpell`.

Apesar de se tratar de um problema muito diferente, o modelo do `ISpell` internacional tem semelhanças grandes com o modelo pretendido.

Dum modo muito simplificado, o `ISpell` inclui um dicionário e um conjunto de regras. O dicionário associa a cada palavra, um conjunto de identificadores de regras $idRegra$. A cada identificador $idRegra$ está associado um conjunto de regras simples[4].

[4]Embora o modelo mais natural fosse

$$regras \quad = \quad idRegra \rightharpoonup set(regra)$$

vamos usar $regras = set(regra)$ em que cada regra contém informação do respectivo identificador de esquema de derivação a ela associado. Este novo modelo torna mais simples a explicação das funções de análise de palavra que se vão seguir.

$$
\begin{array}{lcl}
ispellDic & = & regras \times dic \\
regras & = & set(regra) \\
dic & = & palavra \rightharpoonup set(idRegra) \\
regra & = & \begin{array}{rcll} guarda & : & padrao & \times \\ s1 & : & str & \times \\ s2 & : & str & \times \\ id & : & idRegra & \end{array}
\end{array}
$$

Uma palavra é considerada válida se existe directamente no dicionário ou se existe uma palavra no dicionário que a consiga derivar usando as regras a ela associadas.

Em `ISpell`, uma regra de derivação é constituída por um padrão de terminação de palavra, uma terminação a retirar, um sufixo a acrescentar e um identificador de classe de regra (esquema).

**Exemplo 2:** Regra de plural de substantivos ou adjectivos terminados em 'l'

Uma das regras de definição de plurais para a Língua Portuguesa, inclui uma série de regras simples como por exemplo a regra que transforma `animal` em `animais`:

| guarda | suf1 | suf2 | exemplo |
|---|---|---|---|
| al | l | is | animal/animais |
| il | l | s | funil/funis |
| ol | ol | óis | anzol/anzóis |

ou seja, quando uma palavras termina na guarda, será retirado o sufixo suf1, e acrescentado o sufixo suf2.

Continua-se a descrição do `ISpell` com a descrição de uma das suas funções principais: a função `valida` que indica se uma palavra é ou não válida.

Começaremos por definir algumas funções auxiliares. A função `quebra` vai ser usada para determinar o conjunto das possíveis partições da palavra em radical-sufixo[5]

$quebra : palavra \longrightarrow set(palavra \times palavra)$
$quebra(p) \stackrel{\text{def}}{=} \{<\text{p(1..n)} \ ,\text{p(n+1..)} \ > | \ n \in \{ \ 1 \ .. \ \text{len(p)-1} \ \}\}$

Seguidamente, podemos calcular o conjunto das regras aplicáveis (regras que podem ter sido aplicadas para produzir uma palavra) como sendo aquelas

[5]Em Prolog costuma-se chamar `append` a um predicado usado com igual finalidade.

que produzem uma terminação (sufixo 2 ou s2) compatível com a palavra a analisar.

$$regrasaplicaveis : palavra \times set(regra) \longrightarrow set(regra \times palavra)$$
$$regrasaplicaveis(p, rs) \stackrel{\text{def}}{=}$$
$$\{<r, \pi_1(ppp) \frown s1(r)> \mid ppp \in quebra(p), r \in rs \wedge s2(r) = \pi_2(ppp)\}$$

Na realidade esta função não só determina o conjunto das regras que poderiam ter sido utilizadas, mas também qual a palavra inicial que lhe teria dado origem.

Para além de conhecer o conjunto de regras que poderiam ter sido aplicadas para produzir uma determinada palavra, é necessário determinar aquelas que são válidas: aquelas que estejam registadas como legais na informação associada à palavra inicial (de acordo com o registo no dicionário).

$$regraval : regra \times palavra \times ispellDic \longrightarrow Bool$$
$$regraval(r, p, i) \stackrel{\text{def}}{=}$$
$$\underline{let} \quad <regras, dic> = i$$
$$\underline{in} \quad p \in dom(dic) \wedge id(r) \in rs(dic(p))$$

Uma palavra é válida se existe no dicionário ou se existe pelo menos uma regra aplicável e válida que a deriva:

$$valida : palavra \times ispellDic \longrightarrow Bool$$
$$valida(p, i) \stackrel{\text{def}}{=}$$
$$\underline{let} \quad <dic, regras> = i$$
$$\quad poss = regrasaplicaveis(p, regras)$$
$$\underline{in} \quad p \in dom(dic) \vee \exists po \in poss : regraval(\pi_1(po), \pi_2(po), i)$$

A estrutura interna `ISpell` reflecte algumas preocupações morfológicas, mas não contém a totalidade da informação necessária para funcionar como analisador morfológico: dado que não existe registo de propriedades morfológicas associadas a palavras e a regras, não há possibilidade de construir uma análise morfológica de palavra.

A estrutura do `jspell` pode ser vista como um enriquecimento da estrutura anterior:

- a cada palavra do dicionário fica também associada uma classificação (descrita através de um conjunto de propriedades (*feature structure*))
- a cada regra simples, para além de indicar a maneira como se altera a ortografia de uma palavra para obter a sua derivada, contém também

a definição das alterações paralelas (Koskenniemi, 1983) a fazer sobre a sua classificação morfológica, sintáctica e semântica.

– cada esquema (conjunto de regras da mesma classe) tem alguma informação indicativa do tipo de regra e outras pré-condições necessárias à utilização da regras em causa. (Nos modelos que descreveremos abaixo, não incluiremos esta faceta para facilitar a leitura das especificações apresentadas).

**Exemplo 3:** Regra de derivação de nominais *mento* a partir de verbos

Neste exemplo, apresenta-se uma parte da regra de definição de nomes derivados de verbos.

As regras de definição têm agora uma segunda dimensão: a das propriedades associadas à palavra, podendo estas ser de natureza morfológica, sintáctica ou semântica.

| | |
|---|---|
| propriedades iniciais | CAT=v,T=inf |

| padrão | suf1 | suf2 | propriedades | exemplo |
|---|---|---|---|---|
| [ai]r | r | mento | CAT=nc,G=m,N=s, fsem=mento | julgar/julgamento |
| er | er | imento | CAT=nc,G=m,N=s, fsem=mento | bater/batimento |

ou seja, quando uma palavra termina de acordo com a guarda, será retirado o sufixo $suf1$ e acrescentado o sufixo $suf2$, e as suas propriedades são alteradas (reescritas) nas facetas descritas na coluna *propriedades*.

O modelo `jspell`, após os enriquecimentos atrás descritos fica:

$$
\begin{array}{lcl}
jspellDic & = & regras \times dic \\
regras & = & set(regra) \\
dic & = & palavra \rightharpoonup infPal \\
infPal & = & \begin{array}{rcl} cl & : & classificacao \quad \times \\ rs & : & set(idRegra) \end{array} \\
classificacao & = & ef \\
regra & = & \begin{array}{rcll} guarda & : & padrao & \times \\ s1 & : & str & \times \\ s2 & : & str & \times \\ cl & : & classificacao & \times \\ id & : & idRegra & \end{array}
\end{array}
$$

O processo de ver se uma palavra é válida é semelhante ao existente no `ISpell`. Para funcionar como analisador morfológico, o quantificador existencial da função válida dá lugar ao cálculo do conjunto de *feature structure* resultantes da aplicação das regras.

A função que faz a análise morfológica de uma palavra, `fea`, vai calcular o conjunto de regras aplicáveis determinando, seguidamente, aquelas que são válidas: aquelas que estejam registadas como legais na informação associada à palavra inicial (de acordo com o registo no dicionário).

$$
\begin{array}{l}
fea : palavra \times jspellDic \longrightarrow set(fs) \\
fea(pal, j) \stackrel{\text{def}}{=} \\
\quad \underline{let} \quad <d, rs> = j \\
\quad \qquad poss = regrasaplicaveis(pal, rs) \\
\quad \qquad infmorf(par, d) = \underline{let} \quad <r, p> = par \\
\quad \qquad \qquad \qquad \qquad \quad \underline{in} \quad \left( \begin{pmatrix} 'rad \\ p \end{pmatrix} \right) \dagger cl(d(p)) \dagger cl(r) \\
\quad \underline{in} \quad \{ infmorf(x, d) \mid x \in poss \wedge regraval(\pi_1(x), \pi_2(x), j) \}
\end{array}
$$

A informação morfológica devolvida, é calculada com base na junção da informação da palavra original com as alterações associadas à regra morfológica usada.

A condição de concordância com o dicionário `regraval`, que indica a validade da aplicação de certa regra a certa palavra com base no dicionário, corresponde neste caso a verificar que a palavra exista no dicionário e a verificar se a classe associada à regra (`id(regra)`) está contida no conjunto das regras válidas a aplicar à palavra.

A função `setmode(...)` permite alterar a condição de concordância e o domínio das propriedades a determinar, de acordo com as necessidades específicas do problema concreto.

Como `ISpell` se encontra disponível em código fonte, optou-se por o tomar como ponto de partida e por lhe alterar/acrescentar a funcionalidade necessária.

A opção de construir o `jspell` como abstracção/enriquecimento/refinamento do `ISpell`, teve vantagens a vários níveis:

- Diminuiu consideravelmente o tempo de desenvolvimento da nova ferramenta (reaproveitou-se cerca de 80% do código C do `ISpell`).
- Aumentou a fiabilidade (dado que a maior parte do código já está tes-

tado).
– Facilitou a utilização da nova ferramenta por pessoas já familiarizadas com ISpell.
– Permitiu gerar ambos os dicionários a partir de um texto comum, dadas as suas semelhanças estruturais.
– Facilitou a manutenção dos dicionários: a partir de um único dicionário são extraídos os dicionários ISpell e jspell[6].

### 6.1.4 Descrição sucinta do jspell

Como se especificou acima, o jspell baseia-se num dicionário e num conjunto de regras (ficheiro dos afixos) externos.

Tem vários modos básicos de utilização:

1. como corrector ortográfico interactivo com menus e opções (usando termcap) – detectando erros, dando sugestões de correcção e permitindo inserir novas palavras num dicionário pessoal.
2. como interpretador de linha – o utilizador (ou um programa utilizador) introduz uma palavra (ou um comando) e recebe a informação sobre essa palavra (ou sobre a execução desse comando). Este modo foi construído de modo a ser fácil o seu uso em *pipes* bidireccionais.
3. como comando de linha (não bufferizado) – escreve a lista de palavras não incluída no dicionário (comportamento idêntico ao comando spell do Unix) (ver exemplo 5).
4. como biblioteca C.
5. como biblioteca Perl (Almeida and Simões, 2002) (ver exemplos 4 e 5). Este modo usa internamente o interpretador de linha referido no ponto 2, em pipe bidireccional[7].
6. como um interface ao Prolog.

A existência de uma biblioteca C é muito importante para a construção de outras camadas usando o jspell, tanto em C como noutras linguagens.

À semelhança do que acontece no ISpell, há um programa chamado jbuild que analisa os ficheiros de definição das regras (normalmente com a extensão .aff) e o dicionário, e gera uma estrutura optimizada que é carregada para memória sempre que se inicializa o analisador morfológico.

[6]Este dicionário gera também o dicionário português do corrector ortográfico ASpell (Atkinson, 2000).
[7]Ver manual (man) do módulo Perl IPC::Open3.

O `jspell` oferece mecanismos para tratamento de excepções, abreviaturas de *feature structure* de classificação, e um conjunto de características que não serão abordadas neste documento.

Apresenta-se, seguidamente, alguns pequenos exemplos que pensamos serem úteis à compreensão da ferramenta:

- capacidade de usar análise morfológica em programas, tipicamente não interactivo e por vezes envolvendo milhares de análises.
- possibilidade de usar ferramentas de análise morfológica para produzir novos recursos.

**Exemplo 4:** Determinar ocorrências indexadas ao radical

No exemplo que se segue, o `jspell` é usado para calcular uma fonte de informação que associe a cada palavra (normalizada) o seu número de ocorrências e respectiva confiança (devido a potenciais ambiguidades).

O ponto de partida é um ficheiro que contém o número de ocorrências de cada forma. Exemplo: primeiras linhas de ficheiro de ocorrências no corpus CETEMPúblico (Rocha and Santos, 2000):

```
8176954 de
5661841 a
4345946 que
3953982 o
3794334 e
2982635 do
2920996 da
1928486 em
1594221 um
1579580 para
1558714 os
1379398 uma
1349397 com
1221289 não
...
300 abafar
206 abafado
137 abafada
8 abafos
7 abafo
6 abafá-lo
...
```

Formas como *um, uma, uns, umas* correspondem à mesma forma normalizada (um).

No entanto, *a* tanto pode corresponder ao artigo/pronome (forma normalizada *o*, como ser preposição (forma normalizada *a*).

O seguinte algoritmo descreve o modo de usar a função `rad`

```
1  use jspell;
2  jspell_dict("port");

3  for (numoco,pal) in sdtin
4     w = rad(pal);
5     duvida = length(w) > 1;

6     for p in (w)
7        if(duvida){ oco[p][duv] += numoco;}
8        else      { oco[p][gar] += numoco;} }

9  for p in (sort dom(oco))
10    total = oco[p][duv] + oco[p][gar];
11    conf  = oco[p][gar]/total ;
12    print p, total, conf;
```

Notas:

**linha 1** - Importação do módulo `jspell`.

**linha 2** - Identificação do dicionário a ser usado.

**linha 4** - Cálculo das palavras normalizadas que podem ter dado origem à forma `$pal`.

**linha 5** - Há dúvidas se houve mais que uma origem.

**linha 6-8** - Adição ao contador `gar`(antido) ou ao contador `duv`(idioso) associado à palavra.

**linha 9-12** - Escrita das conclusões.

Produzindo a seguinte saída:

```
                 a    6382213 (conf=0%)
                 à     799466 (conf=100%)
               aba        234 (conf=100%)
           abacate         31 (conf=100%)
           abacaxi         12 (conf=100%)
             ábaco         28 (conf=100%)
             abade        571 (conf=100%)
            abadia        270 (conf=100%)
            abafar       1103 (conf=98%)
             abafo         15 (conf=53%)
           abaixar      10754 (conf=3%)
            abaixo      10381 (conf=0%)
   abaixo-assinado       2207 (conf=100%)
           abalada        585 (conf=0%)
         abalançar        122 (conf=100%)
```

```
16              abalar       3912 (conf=68%)
17            abalizar         36 (conf=100%)
18               abalo        867 (conf=30%)
19           abalroar         238 (conf=100%)
20              abanão        160 (conf=100%)
21              abanar       1103 (conf=91%)
```

Notas:

**linha 10** - O substantivo `abafo` é ambíguo com a forma verbal `abafo` do verbo abafar. O correspondente plural `abafos` não é ambíguo. O valor de confiança obtido mede vagamente a fiabilidade da coluna anterior (n.º de ocorrências).

**linha 9** - O verbo `abafar` apresenta uma confiança elevada porque de entre as suas formas verbais encontradas, poucas são ambíguas.

**linha 11** - O verbo `abaixar` apresenta um baixa confiança porque a quase totalidade das suas formas verbais encontradas são ambíguas com o advérbio `abaixo`. Na realidade, a quase totalidade das ocorrências de `abaixo` são advérbios, e portanto a taxa de ocorrências de abaixar está muito inflacionada.

Uma fonte de informação como esta pode ser usada para ajudar a seleccionar que palavras incluir num dicionário, ou pode ainda ser usada para calcular níveis de frequência a incluir como informação em cada entrada de um dicionário.

Quando aplicado às palavras do corpus CETEMPúblico com ocorrências maiores ou iguais a 3, obtivemos as seguintes medidas[8]:

| | |
|---|---|
| número de linhas do texto de entrada | 339650 |
| número de linhas do programa Perl | 20 |
| número de linhas do texto produzido | 27276 |
| tempo gasto | 58 segundos |

Uma script Perl com uma versão enriquecida deste exemplo, existe disponível no arquivo do projecto Natura com o nome de `freqnormpt`.

**Exemplo 5:** Enriquecimento de dicionário

Dado um conjunto de palavras, pretende-se determinar uma lista de plurais não conhecidos, isto é, não existentes no dicionário de trabalho `jspell`, e criar um ficheiro capaz de alterar esse dicionário, acrescentando a flag respectiva aos radicais das palavras cujos plurais foram encontrados.

Neste exemplo, o `jspell` é usado para inferir novas regras para acrescentar a certas palavras do seu próprio dicionário.

[8] Num pentium III, em carga, 600MHz, 128Mbytes de RAM, Linux.

O algoritmo que seguidamente se apresenta, cria um ficheiro com instruções de inserir a flag de plural em certos radicais, com base em evidências de corpora. O ficheiro de entrada é o mesmo que no exemplo anterior.

```
1  use jspell;
2  jspell_dict("port");
3  setmode("+flags");

4  f = shift;

5  for (pal) in  pipeopen("jspell -l -d port < f |");
6     fl = fea(pal);
7     for f in (fl){
8        if ( verif({flags=>"p",CAT=>"nc"}, f)){
9             print  "\ndict->add_flag('p','f[rad]'); # ....",pal;}
10    }
```

Notas:

**linha 3** - O modo `+flags` faz com que os identificadores das regras morfológicas utilizadas na derivação (flags) sejam acrescentados às propriedades devolvidas pela função `fea`. Por omissão, as flags não são incluídas na informação devolvida por `fea`.

**linha 5** - O comando `jspell -l -d port` devolve a lista de todas as palavras em `f` que não foram reconhecidas pelo dicionário `port`. Cada palavra `pal` saída desse comando (i.e. as palavras desconhecidas) é processada seguidamente.

**linha 7** - Calcula a lista das possíveis análises de pal.

**linha 8** - Para cada análise `f` que seja obtida com a flag `p` (flag de obtenção de plurais regulares) correspondente a um nome comum (`CAT=nc`)

**linha 9** - Imprimir uma linha de programa correspondente a inserir no dicionário a referida flag no radical, escrevendo ainda em comentário a palavra plural obtida para facilitar uma subsequente análise manual.

Segue-se um extracto da saída produzida (à qual se juntou algum código inicial de modo a constituir um programa Perl):

```
use jspell_dict;
dict = init("dicionario.dic");

dict->add_flag('p','pataca'); #....patacas
dict->add_flag('p','juiz'); #....juízes
dict->add_flag('p','abaixo-assinado'); #....abaixo-assinados
dict->add_flag('p','passarinho'); #....passarinhos
dict->add_flag('p','co-autor'); #....co-autores
dict->add_flag('p','franco-atirador'); #....franco-atiradores
dict->add_flag('p','soberania'); #....soberanias
dict->add_flag('p','acrobata'); #....acrobatas
dict->add_flag('p','cachecol'); #....cachecóis
```

```
dict->add_flag('p','locatário'); #....locatários
dict->add_flag('p','letreiro'); #....letreiros
dict->add_flag('p','guarda-chuva'); #....guarda-chuvas
dict->add_flag('p','egoísmo'); #....egoísmos
dict->add_flag('p','soprano'); #....sopranos
dict->add_flag('p','paiol'); #....paióis
dict->add_flag('p','avioneta'); #....avionetas
dict->add_flag('p','gare'); #....gares
```

Esta saída (provavelmente após um verificação manual) constitui um programa Perl que quando executado, faz a correspondente actualização do dicionário `jspell`.

Quando aplicado às palavras do corpus CETEMPublico com ocorrências maiores ou iguais a 3, obtivemos as seguintes medidas[9]:

| | |
|---|---|
| número de linhas do texto de entrada | 339650 |
| número de linhas do programa Perl | 12 |
| número de linhas do texto produzido | 1889 |
| tempo demorado | 21.2 segundos |

### 6.1.5 Morfologia em `kspell`

O modo de definir as regras morfológicas em `jspell` é repetitivo: o facto de se associar a cada palavra um conjunto de identificadores de regras simples, obriga a escrever cada derivação morfológica num único passo e portanto a escrever muitas regras.

Um método mais *cómodo* e de mais alto nível consiste em usar esquemas de derivação que possam ser definidos por regras simples, reunião de esquemas, composição de esquemas.

No dicionário, é definido o conjunto de esquemas considerados válidos para cada palavra.

Simplificadamente, e referindo mais uma vez apenas os sufixos, um esquema corresponde a:

[9]Num pentium III, em carga, 600MHz, 128Mbytes de RAM, Linux

$$
\begin{array}{lcl}
esquema & = & regra + reuniao + composicao + alternativa \\
reuniao & = & set(esquema) \\
composicao & = & esquema^{\star} \\
alternativa & = & regra^{\star} \\
regra & = & \begin{array}{rcll} guarda & : & padrao & \times \\ s1 & : & str & \times \\ s2 & : & str & \times \\ cl & : & classificacao & \end{array}
\end{array}
$$

Este método é semelhante à definição de uma gramática para uma linguagem *regling* tendo como alfabeto o conjunto das regras simples: definir um esquema de derivação é descrever em compreensão o conjunto de sequências de regras que podem ser aplicados.

$$regling \quad = \quad set(regra^{\star})$$

De um modo simplificado, o método kspell de definição de esquemas, pode ser visto como uma gramática sobre um alfabeto de regras simples, estendida com *alternativas*.

A linguagem $L$ associada a uma alternativa de regras simples, é equivalente a ter uma única regra $regra_i$, em que $regra_i$ é a primeira regra que for possível aplicar. Essa possibilidade de aplicação depende da palavra sobre a qual o esquema vai actuar.

O construtor de alternativa não era imprescindível mas a sua utilização torna as definições de esquema mais simples e compactas.

**Exemplo 6:** Esquemas para geração de plurais e de femininos

```
pl  = alternativa(
              regra("ão", "ão", "ões", [n=p]),
              regra("r" , ""  , "es" , [n=p]),
              regra(""  , ""  , "s"  , [n=p]));

fem = alternativo(
              regra("or","","a",[n=f])
              regra("o","o","a",[n=f]));
```

A função *expande* que determina a linguagem definida em extensão é:

$$expande : esquema \longrightarrow regling$$

$$expande(e) \stackrel{\text{def}}{=}$$

$$\begin{cases} \text{is-}regra(e) & \Rightarrow & \{<e>\} \\ \text{is-}alternativa(e) & \Rightarrow & \underline{let} \quad e1 = \text{... primeira regra aplicável} \\ & & \underline{in} \quad \{<e1>\} \\ \text{is-}reuniao(e) & \Rightarrow & \bigcup(\{expande(e1) \mid e1 \in e\}) \\ \text{is-}composicao(e) & \Rightarrow & comp(<expande(e1) \mid e1 \in e>) \end{cases}$$

$$comp : regling^{\star} \longrightarrow regling$$

$$comp(ls) \stackrel{\text{def}}{=}$$

... semelhante à concatenação de linguagens usada na definição das gramáticas habituais

**Exemplo 7:** particípios passados de verbos, nominalização em *dor*

Considere-se agora o seguinte extracto com definição de esquemas de flexão de particípios passados de verbos, nominalização em "dor", e flexão em género-número:

```
1   gn  = id
2       | fem
3       | pl
4       | fem pl;

5   id  = regra("", "", "",[]);

6   dor = regra("r","r","dor",[fsem=dor, cat=a_nc]) gn

7   pp  = basepp gn;

8   basepp = alternativa( regra("ar","r" ,"do", [tempo=pp]),
9                         regra("er","er","ido",[tempo=pp]),
10                        regra("ir","r" ,"do", [tempo=pp]))
```

Notas:

**linha 1 a 4** - Definição do esquema de flexão em género-número (gn) – é a união de esquema identidade, com o feminino (fem), com o plural (pl), com o feminino plural (composição do esquema `fem` com o esquema `pl`).

**linha 6** - Nominalização em `dor` é definida como aplicar a regra simples que transforma, por exemplo, *apresentar* em *apresentador*, e seguidamente faz a composição com `gn` dando origem às quatro variantes esperadas (apresentador, apresentadora, apresentadores e apresentadoras).

Note-se que agora existe maior facilidade de re-utilização: os esquemas como *gn* para além se de poderem utilizar directamente em algumas famílias de palavras (por exemplo em adjectivos), ajudam agora a definir outros (por exemplo `pp` e `dor`).

Dado que:

- uma sequência de regras simples pode ser substituída por uma (ou zero) regra simple[10]
- em Português o número de afixos usado em cada palavra é normalmente pequeno,

torna-se possível calcular em extensão o conjunto das regras simples correspondente aos esquemas mais habituais do português, ou seja, consegue-se obter uma definição semelhante à usada no `jspell`.

$simplifica : regling \longrightarrow set(regra)$
$simplifica(rs) \stackrel{\text{def}}{=}$
... substitui cada sequência de regras por uma regra equivalente

`kspell` é uma ferramenta desenvolvida no âmbito do projecto Natura, que aceita regras de estrutura semelhantes às que descrevemos e que oferece uma função *expande* que, dada um conjunto de definição de esquemas, calcula as regras simples correspondentes. `kspell` transforma uma definição morfológica por esquemas em regras `jspell`.

[10] A composição de regras com certo tipo de padrões complexos, ficaria muito difícil de definir, pelo que o tipo de padrões usados é bastante restrito.

## 6.2 NLlex - gerador de analisadores léxicos para linguagem natural

Nesta subsecção, descreve-se a ferramenta `nllex` (Almeida, 1996) que é um gerador de analisadores léxicos para linguagem natural.

À semelhança do `lex` (Lesk and Schmidt, 1975; Johnson and Lesk, 1978) do sistema operativo `Unix` em que é inspirado, o `nllex` gera um módulo C que, após linkado com um analisador morfológico (libjspell, descrito em 6.1) e eventualmente com outros módulos, dará origem a um processador de linguagem natural.

O `nllex` aceita uma notação próxima da utilizada pelo `lex` enriquecida com análise morfológica e outros elementos associados a linguagem natural.

Como caso particular, o `nllex` pode gerar módulos que funcionem como:

- processador léxico simples
- analisador léxico-morfológico (a ser usado conjuntamente com módulos sintácticos como os que o `yacc` (Johnson, 1975), `NLyacc` (Masayuki Ishii, 1994), `btyacc` (Dodd and Maslov, 2002) geram, ou com quaisquer outros que dele necessitem.)

Para além de constituir uma camada adaptável de interface com os dicionários, o `nllex` pode ser usado para lidar com os *elementos não literários*. Designaremos por *elementos não literários*(ENL) todo um conjunto de elementos existentes nos textos e que não são palavras literárias habituais. Normalmente, os ENL correspondem a uma notação própria e constituem sub-linguagens embutidas dentro do texto. Neste conjunto de ENL, estamos a incluir por exemplo:

- sub-linguagens ligadas a linguagens de anotação tal como as etiquetas HTML, as entidades XML, os comandos LaTeX, etc.;
- sub-linguagens ligadas a medidas numéricas tal como medidas de ângulos (ex: 13°27'), datas (ex: 12/4/2001), horas (ex: 12h30m), resultados desportivos (ex: 5-3), etc.;
- URLs, endereços de correio electrónico;

Os ENL são muito frequentes em quase todos os textos reais.

Esta secção aparece nesta dissertação devido à grande relevância de aumentar a adaptabilidade e flexibilidade dos recursos léxicos, dos analisadores morfológicos dos scanners a diferentes problemas e situações. Para além disso também é relevante o modo como a ferramenta está organizada e construída:

- trata-se de uma DSL, partilhando a filosofia e notação do `flex`,

– funcionando como pré-processador para `flex`,
– partilhando vários princípios de construção com `flex`.

**Motivação**

Um dos problemas habituais na construção de processadores de linguagem natural resulta da necessidade de:

– existir um dicionário
– esse dicionário ter a estrutura conveniente (ter os atributos pretendidos e no formato esperado pelo processador)
– esse dicionário ser capaz de comunicar com o processador.

De um modo informal, constata-se que mesmo quando os componentes existem, não é fácil pô-los a colaborar uns com os outros.

No caso particular dos dicionários, há que ter em conta que a sua dimensão, torna penosa e dispendiosa qualquer tipo de adaptação manual.

Perante este panorama, sentiu-se a necessidade de criar um modo de programar as adaptações pretendidas, ou seja sentiu-se a necessidade de criar uma DSL para geração de analisadores léxicos para PLN.

### 6.2.1 História do desenvolvimento do `nllex`

Como se referiu, um dos principais objectivos do `nllex` é constituir um nível onde se possa definir adaptações do *scanner* e do analisador morfológico de modo a ser possível ligá-los às necessidades concretas de processadores e ultrapassar especifidades dos corpora.

Para lidar com linguagens formais, sistemas operativos como o `Unix`, oferecem um conjunto de ferramentas baseadas em expressões regulares (ER).

Para a criação do `nllex`, tomou-se como base de inspiração a ferramenta `flex` (Paxson, 1995). O `flex` é uma DSL capaz de gerar C a partir de uma especificação que associa ER a reacções descritas em C embutido.

Para processar linguagens naturais é natural estender as ER (ER+) de modo a incluir a noção de palavra e suas características, e poder descrever padrões associados a:

– features (Cat, Género, Tempo verbal, ...)
– capitalização (começar ou não por letra maiúscula; ser constituído só por maiúsculas)

- estar ou não definida em dicionário
- ao lema da palavra

Ao desenhar o `nllex`, optou-se por fazer um compilador que, a partir de uma especificação em que se associa ER+ a reacções descritas em C, gerasse `flex` onde implicitamente fosse feito uso de um analisador morfológico (`jspell`) para calcular as propriedades morfológicas das palavras quando tal seja necessário.

Ao definir a sintaxe concreta, tentou-se sempre que possível, seguir uma notação próxima da notação `lex/flex`.

Na geração do código `flex` usou-se um mecanismo de esqueleto com parâmetros de modo a facilitar a escrita de variantes.

### 6.2.2 Breve descrição

Deste modo o `nllex` permite:

- definir *acções* a serem executadas quando são encontradas palavras com determinadas características morfológicas. As características morfológicas são descritas através de pares atributo-valor.
- definir e controlar o comportamento face a *elementos não literários*, como por exemplo:
  - símbolos especiais
  - elementos de *markup*
  - datas, resultados desportivos, números, medias, ...
  - palavras chave
- definir heurísticas para enfrentar os problemas de palavras indefinidas, através de regras envolvendo:
  - propriedades de ser ou não maiúscula
  - aplicação de regras morfológicas não oficialmente permitidas
  - contexto

O `nllex` tem primitivas para lidar com ambiguidade léxica.

Uma especificação `nllex` (ver Fig. 6.1) consiste em:

- Um cabeçalho contendo definições referentes ao dicionário e ao conjunto de atributos usado.
- Um bloco de pares ER+ e as correspondentes acções C – código C a executar quando o correspondente padrão ER+ for encontrado. Neste bloco há notação para programar estratégias para lidar com palavras indefinidas. As acções C têm acesso aos atributos ligados à palavra em análise.
- Outro código C.

### Estrutura de um texto `nllex`

Definições gerais

[código C opcional]

%initnlex
%features

%%

Expressões regulares e acções

<ER> {código C}

%word

<ER+> {código C} Processamento de palavra

«EOW»

%undef

<ER+> {código C} Palavras indefinidas

«EOW»

%%

Figura 6.1: estrutura geral de uma especificação `nllex`

Uma especificação `nllex` enriquece uma especificação `lex` com as directivas:

- `%initnlex options` – para definição de dicionário a utilizar e respectivas opções
- `%features feat-list` – para declaração da lista de *features* que poderão ser utilizadas nos padrões (ER+) e nas acções.
- `%word` – nesta zona (que não aparece em lex) são colocados pares compostos por padrão morfológico/acção correspondente. Os padrões morfológicos correspondem a um conjunto (limitado) de restrições sobre:
  - atributos declarados na directiva `%feature` (exemplo `CAT=prep`)
  - o lema da palavra em análise (exemplo `ROOT=ser`)
  - o tipo de capitalização (exemplo `UC` (palavra com maiúscula))
  - a própria palavra (exemplo `nltext=jj`)
- `<<EOW>>` – acção (opcional) a ser executada depois de processadas todas as análises de uma palavra.

- `%undef` – nesta zona (opcional) definem-se condições/acções referentes ao tratamento de palavras desconhecidas, i.e. palavras que não constam do dicionário `jspell` usado. Mesmo quando uma palavra é desconhecida do analisador morfológico, é possível que este consiga propor alguma informação a ela associada.
- `{P}{S}{W}` – Expressões regulares pré-definidas (para separador de parágrafo, espaços brancos e para palavra, respectivamente)
- features – Os nomes das features declaradas bem como `ROOT` (o lema), `LETT` (tipo de capitalização) e `nltext` (a própria palavra) podem ser usados nas acções semânticas.

### 6.2.3 Exemplos de uso

Para melhor dar a ideia do uso do `nllex`, apresenta-se seguidamente 3 exemplos. No primeiro, o módulo gerado pelo `nllex` é uma aplicação independente e autónoma. No segundo exemplo, o `nllex` está a gerar um analisador léxico para funcionar juntamente com um analisador sintáctico (neste caso gerado pelo nlyacc). No terceiro exemplo, o `nllex` vai gerar um módulo C para ser carregado a partir do Sicstus Prolog.

#### Exemplo de uso independente do `nllex`: um etiquetador simples

Apresenta-se seguidamente um exemplo de utilização do `nllex` como gerador de uma ferramenta independente: um etiquetador morfo-sintáctico sem ligar a contexto[11].

```
%{
int pr=1;
%}

%initnlex port

%feat CAT G N TR P T SEM I Prep Art PFSEM FSEM C Pdem AP AN Ppes ...

ID    ([a-zA-Z0-9_]+)
email ({ID}\@{ID}(\.{ID})*)

%%
```

[11] Ver exemplo de etiquetador morfo-sintáctico com desambiguação por contexto na Secção 6.5.

```
[A-Z]\.                { printf("(abre)%s\n",yytext);}
\$[^$]\$               { printf("(np m)<%s>", yytext);}
{email}                { printf("(np ms)%s\n",yytext);}
[0-9]+                 { printf("(ncard)%s\n",yytext);}
[0-9]+\%               { printf("(nprec)%s\n",yytext);}
[0-9]+º               { printf("(nord m)%s\n",yytext);}
[0-9]+h[0-9][0-9]m?    { printf("(nc fp)%s\n",yytext);}
[0-9]+(cm|mm|m|km)     { printf("(nc mp)%s\n",yytext);}
[;:,]                  { printf("(p1)%s\n",yytext);}
[.?!]                  { printf("(p0)%s\n",yytext);}
«[^»]*»                { printf("(quote)%s\n",yytext);}
{P}                    { printf("(parag)\n\n");}

%word

{*CAT=nc*}             { printf("(nc %s%s[%s])",G,N,ROOT);}
{*CAT=np*}             { printf("(np)");}
{*CAT=prep*}           { printf("(p)");}
{*CAT=v,T=i*}          { ;}
{*CAT=v,ROOT=ser*}     { printf("(ser %s%s%s[%s])",T,N,P,ROOT);}
{*CAT=v,ROOT=ter*}     { printf("(ter %s%s%s[%s])",T,N,P,ROOT);}
{*CAT=v,T=ppa*}        { printf("(vpp %s%s[%s])",G,N,ROOT);}
{*CAT=v*}              { printf("(v %s%s%s[%s])",T,N,P,ROOT);}
{*CAT=art*}            { printf("(art[%s])",ROOT);}
{*CAT=pind*}           { printf("(pind %s%s[%s])",G,N),ROOT;}
{**}                   { printf("(%s[%s])",CAT,ROOT);}

<<EOW>>          { if(pr){printf("%s\n",nltext);}else{pr=1;}}

%undef

{*CAT=v*}             { printf("(v? %s%s%s[%s])",T,N,P,ROOT);}
{*CAT=nc*}            { printf("(nc? %s%s[%s])",G,N,ROOT);}
{*CAT=adv*}           { printf("(adv?[%s])",ROOT);}
{*CAT=adj*}           { printf("(adj?-%s%s[%s])",G,N,ROOT);}
{*nltext=pt*}         { printf("(np ms)Portugal\n");pr=0;}
{*UC*}                { printf("(np?)");}
{*AUC*}               { printf("(abre?)");}
{**}                  { printf("(?)");}

<<EOW>>          { if(pr){printf("%s\n",nltext);}else{pr=1;}}

%%

main(){ yylex(); }
```

Notas:

**linha 2** - `pr` variável booleana que diz se se pretende imprimir a palavra em causa.

**linha 4** - Usar o dicionário Português.

**linha 5** - Declaração das features utilizadas.

**linha 6, 7** - Definição de padrões flex (email é posteriorment utilizado na linha 11).

**linha 9 a 20** - Tratamento de *elementos não literários*.

**linha 22 a 32** - Tratamento de palavras com base em features; sempre que tal faça sentido, tenta-se escrever a categoria geral, as features associadas e o lema.

**linha 22** - Quando a palavra analisada pode ser um nome comum, escreve-se a etiqueta "nc", seguida do valor dos atributos G (género) e N (número) e o respectivo lema.

**linha 25** - Ignorar as análises verbais correspondentes ao modo imperativo (esta linha obviamente só faz sentido em tipos muito específicos de textos).

**linha 26 a 29** - Tratamento de verbos; atribui-se etiquetas específicas às formas do verbo "ser", às formas do verbo "ter", e aos verbos no particípio passado.

**linha 32** - Regra por omissão para tratamento de análise de palavra: escrever a categoria CAT e o lema.

**linha 33** - Acção a executar após tratamento de todas as análises da palavra: escrever a palavra gráfica analisada e mudar de linha.

**linha 35 a 43** - Tratamento de palavras desconhecidas.

**linha 40** - Palavras desconhecidas começadas por maiúscula são prováveis nomes próprios.

**linha 43** - Acção de fim de palavra desconhecida.

O programa gerado quando aplicado ao seguinte texto:

> *O comboio das 17h30 andou tartarugamente 300km.*
> *Depois de $3^o$ julgamento, Lubbers foi libertado.*
> *Podes enviar mensagens para jj@di.uminho.pt.*

produz a seguinte saída

```
(art[o])(ppes[o])O
(nc ms[comboio])comboio
(cp[do])das
(nc fp)17h30
(v pps3[andar])andou
(adv?[tartaruga])tartarugamente
(nc mp)300km
(p0).
(parag)

(adv[depois])Depois
(p)de
(nord m)3°
(nc ms[julgar])julgamento
```

```
14 (p1),
15 (np?)Lubbers
16 (v pps3[ir])(ser pps3[ser])foi
17 (vpp ms[libertar])libertado
18 (p0).
19 (parag)

20 (v ps2[poder])(v pcs2[podar])Podes
21 (v inf[enviar])enviar
22 (nc fp[mensagem])mensagens
23 (p)para
24 (np ms)jj@di.uminho.pt
25 (p0).
26 (parag)
```

Notas:

**linha 4** - Apesar de não ser uma *palavra normal*, **17h30** está a ser classificado como nome comum, masculino plural. Um fenómeno idêntico aparece na linha 7.

**linha 6** - **tartarugamente**, embora seja uma palavra desconhecida, pode ser obtida por utilização de uma regra morfológica sobre uma palavra existente no dicionário. A etiqueta associada é, de acordo com a definição nllex anterior, **adv?** para ser distinguível dos advérbios *oficiais*.

**linha 15** - Apesar de **Lubbers** não aparecer no dicinário, a heurística de tratamento de palavras desconhecidas propôs uma classificação **np?** devido ao termo começar com letra maiúscula.

**linha 16** - **foi** tanto pode ser uma forma do verbo **ser** como uma forma do verbo **ir**; de acordo com a especificação apresentada, isto dá origem a dois tipos de etiquetas diferentes (v=verbo, ser=forma do verbo ser).

**linha 24** - Um email comporta-se como um nome próprio masculino singular.

### Exemplo de uso dependente: um analisador léxico para NLyacc

Neste exemplo, o `nllex` comporta-se como gerador de um módulo de análise léxica para funcionar em cooperação com um analisador sintáctico *generalized LR* (GLR) (Tomita, 1987; Tomita, 1991; Briscoe, 1991) gerado pelo NLyacc (Masayuki Ishii, 1994). NLyacc é um gerador de reconhecedores sintácticos GLR experimental que é um superset do Yacc (embora sem um tratamento de erros) capaz de lidar de um modo determinístico com ambiguidade sintáctica e léxica.

### Especificação do analisador léxico (nllex)

O funcionamento geral do analisador léxico que se pretende gerar, corresponde a reconhecer o elemento em causa, guardar (com **yyset**) as suas análises e retornar 0.

```
1  %{
2  #include <string.h>
3  #define yyset(x) yysetvalue(x)  // modo do NLyacc lidar
4                                  // com a ambiguidade
5  extern YYSEMTYPE yysval;
6  extern YYSTYPE yylval;
7  %}

8  %initnlex port

9  %feat CAT G N TR P T SEM
10 %%
11 [0-9]+  {yyset(INTE); return 0; }
12 [A-Z]\. {yyset(ABR);  return 0; }
13 ...
14 <<EOF>> {yyset(EOF); return 0; }
15 {P}     {yyset(EOF); return 0; }     // parágrafo retorna EOF
16 \"{W}\" {yyset(PN);  return 0; }     // palavra entre aspas --> NP

17 %word
18 {*CAT=n*}           { yyset(NOUN);}
19 {*CAT=v,T=inf*}     { yyset(INF);}   // verbo no infinitivo
20 {*CAT=v,T=ppa*}     { yyset(PPA);}   // verbo particípio passado
21 {*CAT=v,ROOT=ter*}  { yyset(TER);}   // verbo auxiliar TER
22 {*CAT=v*}           { yyset(VERB);}  // verbo nos outros casos
23 ...
24 {*CAT=a_nc*}        { yyset(NOUN); yyset(ADJ);}//retorna 2 valores!
25 ...
26 <<EOW>>             { return 0 ;}
27 %undef
28 {**}                { fprintf(stderr,"(Undefined %s) \n", yytext);
29                        yylval.str=strdup(yytext);
30                        yyset(UNDEF);}
31 <<EOW>>             {  return 0 ;}
32 %%
```

Notas:

**linha 3** - `yysetvalue` é definido no NLyacc para permitir múltiplos valores léxicos.

**linha 14** - Nesta especificação, um parágrafo está a gerar o mesmo tipo de símbolo terminal que o fim de ficheiro.

**linha 23** - Associado à categoria *adjectivo ou nome comum* (**a_nc**), retornam-se duas classes: adjectivo ambiguidade léxica resulta de uma palavra poder ter mais que uma análise léxica.

### Analisador sintáctico (NLyacc)

O analisador sintáctico que aqui se apresenta é apenas um extracto exemplificativo[12] da possibilidade de utilização do `nllex` em colaboração com outros módulos (e geradores de módulos).

A linguagem de especificação de analisadores sintácticos NLyacc é, como se disse, muito semelhante à do yacc.

Neste exemplo só os símbolos **UNDEF** estão a ter um valor semântico associado (o correspondente yytext).

```
%{
typedef  union {char * str;} t  ;
#define YYSTYPE t
%}

%type <str> UNDEF

%token NOUN VERB DET PUNCT UNDEF PPA ADJ INTE ABR PN TER INF
%%
SS : S PUNCT  { puts("S "); }
   | NP PUNCT { puts("NP ");} ;

S : NP VP ;

NP : NP1 | PIND | DET op_inte NP1 ;

op_inte : | INTE;

NP1 : NOUN  | c_PN | ADJ NP1 | INF oNP ;

c_PN : ABR c_PN
     | PN c_PN
     | UNDEF {printf("<<%s/CAT=pn>>/n", $1);} c_PN
     | PN ;
```

[12]Este exemplo concreto não está a fazer nada de interessante.

```
18 VP : VERB oNP | TER PPA oNP ;

19 oNP : | NP ;
20 %%
21 #include "lex.yy.c"                          /* gerado pelo NLlex */
22 main()
23 {   yyinitialize();
24     yyparse();
25     yyterminate(); }
```

Notas:

**linha 2** - Definição dos tipos a utilizar na semântica.

**linha 5** - Neste caso apenas o símbolo terminal UNDEF tem valor semântico associado.

**linha 8 a 19** - Produções da gramática.

**linha 14 a 17** - Nomes próprios complexos.

**linha 16** - Uma palavra indefinida no meio de um nome próprio complexo, deve ser um nome próprio: escrever essa conclusão.

**linha 21** - Incluir o analisador léxico gerado pelo `nllex`.

### Exemplo: interface a Prolog

Neste exemplo, mostra-se como se pode estabelecer ligação entre código gerado pelo `nllex` e Prolog. Esta ligação foi implementada para Sicstus Prolog embora pudesse ser adaptada a outros sistemas Prolog.

Para construção da ligação Prolog – `nllex`, optou-se por, para além de uma especificação `nllex`, criar dois ficheiros que facilitem o interface Prolog – C:

– o ficheiro C (nllexinter.c) que mantém o *estado* e contém as funções que devolvem as análises uma a uma. Estas funções são chamadas a partir do Prolog (e contêm partes dependentes do tipo de Prolog (neste caso Sicstus Prolog)). Este ficheiro é sempre constante.
– o ficheiro Prolog (nllexinter.pl) (sempre o mesmo, do qual se apresentará um extracto) que carrega o ficheiro gerado pelo `nllex`, constrói tipos estruturados usando o módulo **charsio** e devolve as várias análises por *backtracking*.

Este interface oferece (entre outros) os seguintes predicados Prolog definidos em "nllexinter.pl":

– `lex(+palavra,-cat,-sem)` para obter a categoria e semântica de uma palavra com a capacidade de obter as análises todas através de backtracking.

- `set_string(+string)` para definir a frase que irá ser analisada pelo predicado `lex2`.
- `lex2(-word,-cat,-sem)` extrai a próxima palavra, categoria e semântica.
- `set_file(+string)` para definir o ficheiro donde vão sendo sucessivamente extraídas as frases a analisar pelo predicado `lex2`.

O predicado `lex/3` invoca uma função gerada pelo `nllex` para obter uma lista de possíveis análises sendo estas devolvidas posteriormente uma de cada vez.

Cada análise tem uma parte semântica (tipicamente em função do lema associado à palavra analisada) uma categoria gramatical (um termo que aglutina o conjunto das features relevante). O `nllex` apenas constrói listas de strings contendo termos Prolog `sem` e `cat`, sendo estas strings convertidas no ficheiro nllexinter.pl para termos Prolog, usando o módulo `charsio`.

No exemplo que se segue, o processamento de cada análise corresponde a preencher uma posição dos arrays `sem` e `cat` com o termo Prolog mais adequado. Dado que se trata apenas de um exemplo, só serão utilizadas duas features.

Esta especificação `nllex` vai definir o modo como o Prolog vai ver o léxico e poderá ser adaptado cada vez que se pretenda uma riqueza maior (por exemplo mais features, mais categorias gramaticais, etc.).

```
%{
#include <string.h>
#define s1 s->sem[s->n_sol]
#define c1 s->cat[s->n_sol]
#define nlnext (s->n_sol++)

#define YY_DECL yylex(State *s)
%}

%initnlex port

%feat     CAT G N TR P T SEM I Prep Art FSEM
%%
%word
{*CAT=nc*}             { sprintf(s1,"%s.",ROOT);
                         sprintf(c1,"nc(%s).",G);
                         nlnext;}
{*CAT=v,ROOT=ser*}     { sprintf(s1,"%s.",ROOT);
                         sprintf(c1,"v(ser,%s).",T);
                         nlnext;}
```

```
18 {**}                  { sprintf(s1,"%s.",ROOT);
19                         sprintf(c1,"%s.",CAT);
20                         nlnext;}
21 <<EOW>>               { printf(".\n"); return 0 ;}

22 %undef

23 {**}                  { sprintf(s1,"undef(%s).",nllex);
24                         sprintf(c1,"X.");
25                         nlnext;}
26 <<EOW>>               { printf(".\n"); return 0 ;}

27 %%
```

Notas:

**linha 6** - Neste exemplo a função yylex gerada passa a receber um estado `s` como parâmetro. Esse estado é preenchido de modo a conter a lista de análises( guardadas nos arrays `sem` e `cat` contidos no estado.

**linha 15 a 17** - Tratamento de formas do verbo ser. Por exemplo, a categoria da palavra `sendo` seria `v(ser,g)` e a semântica `ser`.

O extracto do ficheiro nllexinter.pl que se segue, é apenas um detalhe técnico que não será aqui comentado.

```
:- module(nllex,[lex/3,lex2/3,set_string/1,set_file/1]).
:- use_module(charsio).

%lex(+Palavra,-Categoria,-Semantica)
lex(Palavra,Categoria,Semantica) :-
        set_string(Palavra),
        init_lex(Ref),
        repeat,
        get_lex(Ref,Cat,Sem,Flag),
        (Flag=1 -> (
            name(Sem,Seml),
            read_from_chars(Seml,Semantica),
            name(Cat,Catl),
            read_from_chars(Catl,Categoria))
          ; !, fail).

foreign_file('nllex.o',[init_lex,get_lex,set_string]).

foreign(set_string,c,set_string(+string)).
foreign(init_lex,c,init_lex(-address('State'))).
foreign(get_lex,c,get_lex(+address('State'),
```

```
                           -string, -string, [-integer])).

:-load_foreign_files('nllex.o',['-ljspell']).
```

A execução do Prolog com este interface, permite aceder ao analisador morfológico de acordo com a vista definida.

```
1  | ?- set_string("era uma vez").
2  | ?- lex2(W,A,B).
3  W = era, A = nc(f),     B = era ? ;
4  W = era, A = v(ser,pi), B = ser ? ;
5  no

6  | ?- lex2(W,A,B).
7  W = uma, A = art,       B = um ? ;
8  W = uma, A = card,      B = um ? ;
9  W = uma, A = pind,      B = um ? ;
10 no

11 | ?- lex2(W,A,B).
12 W = vez, A = nc(f),     B = vez ? ;
13 no
```

Notas:

**linha 1** - Definição da frase em análise: `era uma vez`.

**linha 2** - Análise da primeira palavra.

**linha 3** - Primeira análise obtida por unificação com W, A e B da palavra, classificação sintáctica e semântica.

**linha 4** - Obtenção da segunda análise por *backtracking*.

**linha 5** - Não há mais análises.

**linha 6** - Análise da próxima palavra.

**linha 7 a 10** - Análise da palavra `uma`.

**linha 11 a 13** - Análise de `vez`.

Uma variante ligeiramente mais rica (com uma especificação `nllex` bastante mais complexa) deste interface é usado pelo YaLG (ver secção 6.3).

## 6.3 YaLG – estendendo as DCG para PLN

Em secções anteriores apresentou-se ferramentas de processamento centradas no nível léxico.

A ferramenta que se apresenta nesta secção, contempla o processamento a nível sintáctico e eventualmente semântico e pragmático da linguagem.

O YaLG[13](yet another logic grammar) é uma ferramenta da família das gramáticas de unificação:

- com possibilidade de utilização de dicionários/analisadores morfológicos externos[14],
- com possibilidade de tomar como entrada ficheiros,
- funcionando como uma *domain specific language* sobre Prolog[15],
- com possibilidade de escrita de gramáticas de ordem superior.

Com a ferramenta YaLG pretende-se constituir um ambiente que permita descrever modos de inferir propriedades acerca de termos, através da análise de textos. Essa análise abarca um leque potencialmente muito lato de problemas, como por exemplo:

- análise de exemplos de uso de um termo para inferir o seu padrão de utilização (Exemplo: o verbo gosta tem como padrão de utilização *NP gosta de NP* ou *serVivo gosta de NP*)
- análise de definições de dicionários/glossários para extrair uma informação associada com uma estrutura mais rica (Exemplo: procurar genus/diferencie em definições de dicionários)
- desambiguar, por contexto, algumas situações contendo ambiguidades léxicas, sintácticas ou semânticas

As gramáticas lógicas (Abramson and Dahl, 1989) são conhecidas pela sua elegância e poder expressivo em relação a especificar a sintaxe e semântica de uma linguagem.

Como é sabido, o funcionamento das ferramentas sintácticas, depende da capacidade de escrever gramáticas de cobertura suficiente para os exemplos em análise. A escrita de uma gramática que cubra convenientemente uma língua como o Português, é uma tarefa muito complexa.

[13] O nome original – NYAGNLPLG(not yet another good natural language processing logic grammar) foi rejeitado por dificuldades fonéticas.

[14] Na versão actual usando obrigatoriamente o `jspell`.

[15] Na versão actual usando o Sicstus Prolog.

Historicamente, após uma grande período em que se fizeram numerosos estudos e sistemas centrados no nível sintáctico, assiste-se presentemente a uma tendência a deslocar esse centro de atenção para o nível léxico[16]. No entanto o uso de gramáticas para processamento de ilhas textuais (partes de textos, partes de frases), constitui um modo poderoso de extrair informação a partir de frases.

Esta secção aparece na dissertação devido à importância óbvia de poder utilizar (embora pouco explorada neste documento) mecanismos de base sintáctica para apoio à construção e manuseamento de dicionários, através da extracção de conhecimento lexicográfico, ou de qualquer outro processo que envolva níveis mais ricos de processamento de linguagem natural.

Nesse sentido, a utilização de gramáticas lógicas com a possibilidade de usar ambientes equipados com suporte a inferência, unificação, backtracking e reescrita, é muito importante para aprendizagem automática ou assistida de propriedades acerca de termos.[17]

### 6.3.1 Introdução

Em ambiente Prolog, as `DCG` (Pereira and Warren, 1980) são muito populares e são, simultaneamente, simples de aprender e de ensinar. No que diz respeito a linguagem natural, as `DCG` são adequadas para pequenos problemas mas de difícil aplicação em situações envolvendo grandes dicionários, morfologia complexa, ou textos de grandes dimensões.

A ferramenta YaLG (Almeida, 1994; Almeida, 1995) redefine as `DCG` de modo a:

1. a nível léxico:

   – permitir usar um analisador léxico externo, tendo por base os dicionários `jspell`, controlando a sua adaptação ao Prolog através do `nllex`, que permite também a especificação de detalhes de funcionamento do scanner, incluindo o tratamento de *elementos não literários*(ENL) e tratamento das convenções específicas do texto em causa (relembre-se a noção de ENL apresentada na Secção 6.2 e no exemplo da pg.108).

[16] Ou seja, esta secção está ligeiramente fora de moda...

[17] Para além disso, as gramáticas lógicas têm propriedades muito interessantes do ponto de vista formativo: na tentativa de se arranjar um mecanismo com espaço para descrever de um modo agradável os vários níveis da linguagem, acaba-se a clarificar uma série de conceitos.

- permitir que símbolos terminais possam ser variáveis.
- permitir ver cada símbolo terminal como: uma palavra simples `[ortografia]`; um par categoria gramatical, semântica `[cat-sem]`; ou um triplo ortografia, categoria gramatical, semântica `[cat-sem+ort]`.

2. a nível sintáctico:

- suportar gramáticas de ordem superior, i.e., possibilitar a definição de meta-operadores e símbolos que tenham atributos do tipo símbolo (por vezes designados por combinadores).
- possuir alguns combinadores predefinidos: combinador `?` (optional), combinador `*` (0 ou mais repetições), combinador `perm` (permutação de símbolos terminais ou não terminais), etc.
  A definição de combinadores envolve não só a parte sintáctica mas também a parte semântica[18].

3. a nível semântico:

- possibilitar a utilização de regras de reescrita semântica.
- possibilitar a activação de reescrita automática do símbolo semântico no momento de cada redução.

4. a nível de pragmática:

- possibilitar a definição de reescrita pragmática com base num ou mais conjuntos de regras de reescrita.

### Funcionamento com YaLG

O desenvolvimento de uma aplicação usando YaLG, envolve:

1. construção de um dicionário jspell (ou utilização de um dicionário já existente). Um dicionário contém palavras classificadas e regras morfológicas associadas, como se disse em 6.1.4;
2. construção de uma vista Prolog sobre esse dicionário (feito através duma descrição `nllex`, descrito em 6.2);
3. escrita de uma gramática.

Os elementos 1 e 2 são normalmente reutilizáveis para uma série de exemplos.

[18] O atributo semântico do operador opcional `?` tem que ser definido mesmo quando derivar em string vazia. Neste caso aparece unificado com `nil`. Ver detalhes da definição de combinadores em (Almeida, 1994).

#### Módulo de interface YaLG com o analisador morfológico externo

O interface de ligação entre o Prolog e o analisador morfológico externo está feito usando uma versão ligeiramente estendida do exemplo descrito na Secção 6.2, página 114, e utiliza (de modo transparente para o utilizador) a ferramenta `jspell` com o dicionário Português, permitindo comutar entre análises de ficheiros textuais externos e strings.

As principais dificuldades que a implementação de YaLG teve de resolver (ver detalhes em (Almeida, 1995)), referem-se às questões ligadas às diferenças de tipos e de mecanismos existentes nos dois mundos a ligar (Prolog, C), nomeadamente:

– backtracking sobre as múltiplas análises de uma palavra (a nível do C é devolvido um array de análises que um módulo Prolog de interface se encarrega de guardar na base de dados interna do Prolog e sobre a qual é feito backtracking);
– fluxo de atributos sintácticos e semânticos complexos (o `nllex` está a construir uma string com o termo Prolog pretendido e o módulo Prolog `charsio` está a ser usado para, com base nessa string, construir os atributos);
– representação da entrada (ficheiro ou string) de modo a permitir backtracking.

O módulo de interface em causa exporta (entre outras) as seguintes funções:

```
lex(+Word,-Category,-Semantic)
analisa uma dada palavra

lex2(-Word,-Category,-Semantic)
lê e analisa uma palavra a partir da string ou do ficheiro actual

set_string(+String)
define a entrada actual como sendo a string parâmetro

set_file(+Filename)
define a entrada actual como sendo ficheiro passado como parâmetro
```

Estas funções são usadas internamente no YaLG.

### 6.3.2 Arquitectura geral do YaLG

A implementação do YaLG usa, como foi já dito, uma interface C/Prolog de modo que é dependente da versão Prolog usada. Presentemente, o YaLG depende do Sicstus `Prolog` (Carlsoon et al., 1991), embora possa ser facilmente migrada para outros Prologs com um bom interface ao C.

A Figura 6.2, mostra a arquitectura da ferramenta YaLG que estende a capacidade de processamento das `DCG`, sendo composta pelos seguintes módulos:

- Prolog com gramáticas YaLG
- nível de interface – composto por:
  - nllexinter.pl - parte do interface escrita em Prolog,
  - nllexinter.c - escrita em C
- Scanner + analisador morfológico `nllex.c` (gerada a partir duma especificação `nllex`) e usando a biblioteca `jspell.a` para fazer a análise morfológica com base num dicionário `jspell`

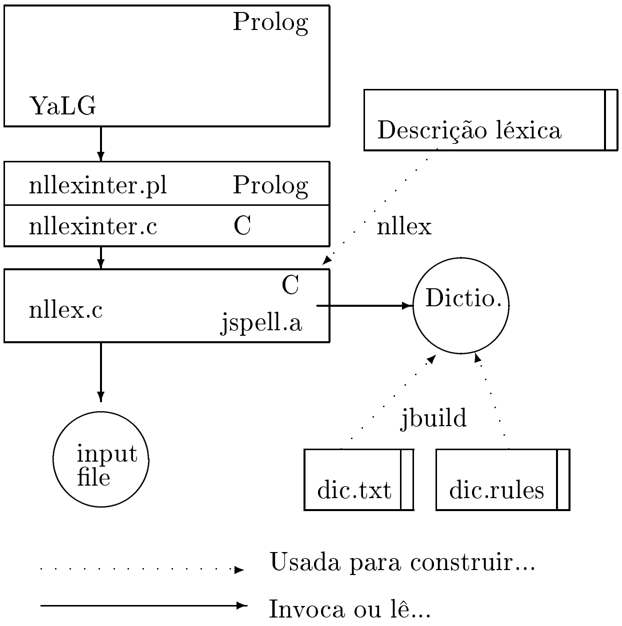



Figura 6.2: Arquitectura do YaLG

### 6.3.3 Comparação YaLG / `DCG`

Ao nível dos símbolos terminais, as principais diferenças entre YaLG[19] e as `DCG` são uma maior variedade de notações envolvendo ortografia, categorização

[19] As gramáticas YaLG basearam-se nas gramáticas GPC (Almeida, 1994) que são `DCG` estendidas com um ambiente de ordem superior e com reescrita automática de semântica. Estas propriedades não são relevantes para este secção.

estruturada e semântica, conforme se sistematiza na tabela seguinte:

| | |
|---|---|
| `[Ortografia]` | símbolo terminal idêntico ao usado nas DCG |
| `[Cat-Sem]` | pseudo-terminal definido por uma categoria e atributos |
| `[Cat-Sem+Ortog]` | junção dos dois casos anteriores |
| `-+->` | símbolo de produção sem reescrita semântica |
| `--->` | símbolo de produção com reescrita semântica automática |

Considere-se o seguinte exemplo de uma DCG que constrói uma árvore sintáctica baseada numa gramática da Língua Portuguesa extremamente simplificada:

```
1  :- op(600,xfy,'::').

2  s(s(SN,SV))         --> np(SN), vp(SV).
3  np(np(SA,SN))       --> art(SA,G,N), nc(SN,G,N).
4  vp(v3(SV,SN1,SN2))  --> v(SV), np(SN1), [a], np(SN2).
5  vp(v2(SV,SN))       --> v(SV), np(SN).
6  vp(v1(SV))          --> v(SV).

7  art(o,m,s)          --> [o].
8  ...
9  art(um,f,p)         --> [umas].

10 nc(_::gato,   m,s)  --> [gato].
11 nc(_::rato,   m,s)  --> [rato].
12 nc(_::galinha,f,s)  --> [galinha].

13 v(mia)              --> [mia].
14 v(come)             --> [come].
15 v(vende)            --> [vende].

16 :- s(S,[o,gato,come,uma,galinha],[]), write(S).
```

Notas:

**linha 1** - Definição Prolog de um operador de ligação classe-instância.

**linha 2 a 6** - Gramática propriamente dita.

**linha 7 a 15** - Léxico (dicionário) no qual se baseia a gramática; os símbolos não terminais *v*, *nc* e *art*, são frequentemente designados por pseudo-terminais. Os símbolos terminais são escritos entre `[ ]`.

**linha 3** - Os atributos G,N estão a ser usados para garantir a verificação de concordância género, número entre artigo e nome.

**linha 16** - Um exemplo de teste envolve fornecer a entrada em forma de lista (ou construir um predicado Prolog *scanner* que faça essa construção).

O mesmo exemplo em YaLG será escrito da seguinte maneira:

```
1 s(s(N,V))        ---> np(N), vp(V).
2 np(np(SA,SN))    ---> [art(G,N)-SA], [nc(G,N)-SN].
3 vp(v3(V,N1,N2)) ---> [v(_,_)-V], np(N1), [a], np(N2).
4 vp(v2(V,N))      ---> [v(_,_)-V], np(N).
5 vp(v1(V))        ---> [v(_,_)-V].

6 :- set_string('o gato come uma galinha').
7 :- s(S), write(S).
8 :- set_file('f').
9 :- s(S), write(S).
```

Notas:

**linha 3** - Os símbolos terminais da gramática, incluem tanto os casos anteriormente usados nas `DCG` (exemplo `[a]`) como os anteriormente designados por pseudo-terminais (exemplo `[art(G,N)-SA]`), sendo estes descritos por pares atributos morfo-sintácticos – atributos semânticos.

**linha 6** - Análise de um exemplo lido de uma string.

**linha 7** - `s(S)` vai despoletar a invocação do analisador morfológico externo que analisa (usando um dicionário real e um conjunto de regras morfológicas também reais) as palavras devolvidas pelo scanner que as retirou da frase em causa.

**linha 8** - Análise de um exemplo lido de um ficheiro.

**linha 9** - Neste caso `s(S)` analisa morfologicamente um ficheiro.

Tipicamente, pretende-se que uma gramática faça mais do que a construção de árvores sintácticas. No exemplo seguinte, acrescentou-se algumas regras de reescrita semântica (versão bastante naïf) para calcular um conjunto de factos elementares (infons):

```
s(s(N,V))        ---> np(N), vp(V).
   s(X-S,v1(V))            ==> [infon(V,[X]),S].
   s(X-S1,v2(V,Y-S2))      ==> [infon(V,[X,Y]),S1,S2].
   s(X-S1,v3(V,Y-S2,Z-S3)) ==> [infon(V,[X,Y,Z]),S1,S2,S3].
np(np(SA,SN))    ---> [art(G,N)-SA], [nc(G,N)-SN].
   np(o,X)            ==> X.
   X::Classe          ==> X-infon(instancia,[Classe,X]) .
vp(v3(V,N1,N2)) ---> [v(_,_)-V], np(N1), [a], np(N2).
vp(v2(V,N))      ---> [v(_,_)-V], np(N).
vp(v1(V))        ---> [v(_,_)-V].
```

Notas:

**linha 2 a 4** - Definição de regras de reescrita semântica para verbos com 1,2 ou 3 argumentos.

**linha 6** - Regra que indica como simplificar nominais quantificados com o artigo `o`.

**linha 7** - Regra para definir o modo de traduzir para infons os nomes.

Este exemplo quando aplicado à string "o gato come a galinha" produz a seguinte saída:

```
[ infon(comer,[_1343,_2779]),
  infon(instancia,[gato,_1343]),
  infon(instancia,[galinha,_2779])
]
```

### 6.3.4 Exemplos

Como foi dito anteriormente, a escrita da uma gramática para a Língua Portuguesa é complexa. No entanto é possível a escrita de pequenas gramáticas para análise de contextos mais pequenos, e estruturalmente muito mais simples do que o *texto*, como se vai ilustrar nesta subsecção.

**Exemplo 8:** Desambiguador de "a"

No próximo exemplo, estuda-se um dos casos mais críticos da etiquetação em Língua Portuguesa: a ambiguidade entre a preposição "a" e o artigo feminino "a" (e já agora entre o pronome "a").

Essa ambiguidade tem especial importância devido a estas palavras estarem entre as que mais ocorrem em Português.

A descrição gramatical que se apresenta, não pretende descrever frases, orações relativas, etc., pretende apenas analisar a quantidade mínima que permita desambiguar os *a* com base em contexto reduzido.

```
sp :- write(' ').

aaa -+-> [QQ],[a],(
               [nc(f,s)-_+V2];
               [nord(f,s)-_+V2];
               [ppos(P,f,s)-Y+V2];
               [adj(f,s)-_+V2]),
             !,{sp,write(QQ), write(' a/art '),write(V2)}, aaa.
aaa -+-> [QQ],[a],(
               [nc(_,_)-_+V2];
               [nord(_,_)-_+V2];
               [ppos(_,_,_)-_+V2];
```

```
12                 [adj(_,_)-_+V2];
13                 [np(m,_)-_+V2];
14                 [v(_,inf)-_+V2];
15                 [ncard-_+V2];
16                 [pind-_+V2];
17                 [ppes(_,_,_,_)-_+V2];
18                 [art(_,_)-_+V2];
19                 [pdem-_+V2]),
20             !,{sp,write(QQ), write(' a/prep '),write(V2)}, aaa.
21  aaa -+-> [QQ],[a],([v(_,_)-_+V2];
22                      [_-que+V2]),
23             !,{sp,write(QQ), write(' a/pind '),write(V2)}, aaa.
24  aaa -+-> [QQ],[a],[X],
25             !,{sp,write(QQ), write(' a/naosei '),write(X)}, aaa.
26  aaa -+-> [eof],
27             !,{nl,write('EOF'),nl }.
28  aaa -+-> [QQ],
29             !,{sp,write(QQ)}, aaa.

30  :- set_file(f1).
31  :- aaa.
```

Notas:

**linha 2** - `[QQ]` – Um símbolo terminal qualquer (a possibilidade de uso de variáveis nos símbolos terminais é muito importante); `[a]` – a palavra ortográfica `a`;

**linha 3,4,5** - `[a-b+c]` – A palavra $c$ com semântica $b$ e categoria gramatical $a$.

**linha 2 a 7** - `a` é provável artigo se estiver antes de um nome comum feminino singular, ou antes de um adjectivo f.s., ou antes de um pronome possessivo f.s. ou antes de um numeral ordinal f.s.

**linha 8 a 20** - `a` É provável preposição se estiver antes de ...

**linha 21 a 23** - `a` É provável pronome se estiver antes de um verbo não infinitivo ou antes de um `que`.

**linha 24,25** - Situação em que há dúvidas acerca da classificação de `a` (ver exemplo abaixo).

**linha 28,29** - Escreve as palavras não envolvidas nas regras anteriores.

Quando aplicado a um ficheiro *f1* contendo:

> *Eu vi a minha tia. Vou pedir a meus pais. A bom entendedor meia palavra basta. A mim, parece-me que vai chover. Devemos isso a Miguel Torga. Foi assim que a vi. Vou dar isto a uma amiga. Devemos isto a Sofia. Eu nasci a 21 de Fevereiro.*

A saída produzida é:

> *Eu vi a/art minha tia . Vou pedir a/prep meus pais . a/prep bom entendedor meia palavra basta . a/prep mim , parece-me*

*que vai chover . Devemos isso a/prep Miguel Torga . Foi assim que a/pind vi . Vou dar isto a/prep uma amiga . Devemos isto a/naosei Sofia . Eu nasci a/prep 21 de Fevereiro .*

Este programa[20] demorou cerca de 38 segundos a etiquetar os `a` num texto com cerca de 100k palavras (uma dimensão semelhante ao livro "Pupilas do Senhor Reitor").

Note-se que este programa está cheio de *commits* "!" (tornando-o quase imperativo), e levando a que possa processar textos reais. Se ao escrever as gramáticas YaLG para processar textos reais, não se tiver algum tipo de preocupações deste género, a eficiência será muito comprometida.

**Exemplo 9:** Extractor simplificado de Frases Nominais

São vários os problemas em que há necessidade de fazer a extracção do grupo nominal. Em extracção de terminologia, por exemplo, a detecção de grupos nominais tem importância.

```
:- begin(gram).

varre -+-> np(N), { writelist(N) }, !,varre.
varre -+-> [eof],! .
varre -+-> [QQ],!,varre.

np([A,B,C,D,E|Spp]) -+->
   ?[art(_,_)-S1+A],
   ?[ppos(_,_,_)-Sa+B],
   ?[adj(_,_)-Sad1+C],
   [nc(_,_)-Sn+D],
   ?[adj(_,_)-Sad+E],
   opp(Spp).

np([A,B,C,D,E]) -+->
   ?[cp-S1+A],
   ?[ppos(_,_,_)-Sa+B],
   ?[adj(_,_)-Sad1+C],
   [nc(_,_)-Sn+D],
   ?[adj(_,_)-Sad+E],
   opp(Spp).

opp([de|L]) -+-> [de],      np(L).
opp([H|L])  -+-> [cp-S1+H], np(L).
```

[20]Num Celeron em carga, 700MHz, 64Mbytes de RAM, Linux

```
21 opp([])     -+-> !, [].
22 :- end(gram).

23 test('o meu gato e a minha avó dormem. A vaca do meu tio fugiu').
24 test('minha gata e a grande casa fria fazem sombra').
25 :- test(A),set_string(A),varre.

26 :- set_file(exemplo),varre.
```

Notas:

**linha 2** - `writelist` é um predicado (que não apresentamos) que escreve uma lista de átomos não vazios.

**linha 2 a 4** - O predicado `varre`, varre um texto, escrevendo todos os segmentos `np` que encontrar.

**linha 6** - `?` É o meta-operador opcional.

**linha 23 a 25** - Teste tendo como base strings.

**linha 26** - Teste tendo como base um ficheiro `exemplo`.

A saída correspondente às linhas 23 a 25 é:

```
o meu gato
a minha avó
A vaca do meu tio
a minha gata
a grande casa fria
sombra
```

### 6.3.5 YaLG e DSL

A implementação de YaLG foi essencialmente um exercício de *domain specific language* sobre Prolog. Quando se pensa em definir uma nova DSL, é importante a escolha de uma notação que siga de perto a notação usada pelos peritos da área. No caso das gramáticas, há várias comunidades que usam notações muito diferentes. Por isso optou-se pela inclusão de um mecanismo de adaptação de notação[21] de modo a permitir fácil acomodação a mais que uma notação.

No exemplo que se segue, as produções têm um identificador de produção, sendo automaticamente construído/acrescentado o atributo semântico em todos os elementos presentes na gramática (excepto naqueles prefixados com o operador "-").

[21] Pré-processamento por reescrita contextual.

**Exemplo 10:** Extracto de gramática da Língua Portuguesa

No próximo exemplo, apresenta-se uma gramática mais complexa, escrita em notação produções identificadas: cada produção tem um identificador; a semântica de cada produção é descrita por regras de reescrita.

```
:- begin(gram).

%% Gramática básica de português
%% lambda(X,Pro) -------> X-Pro
%% sentence s
%%
s -(s)--> np, vp.
s -(iq)--> [prel], np, vp, -[?].
s -(iq2)--> [prel], np, -[?].
      s(npq(D,N,A),N-Fv) ==> q3(D,N,A,Fv) .
   iq(_,npq(o,N,A),N-Fv) ==> q3(i,N,A,Fv) .
   iq2(_,npq(o,N,A))     ==> q3(i,N,A,nil) .
%
%% noun phrases
%% np =  npq(quant,elemento,anteced)
np -(np)--> det, *adjp, n, *n_pcompl.
  np(npq(D,N,Ant),[])        ==> npq(D,N,Ant1):-rewrite(Ant,Ant1).
  np(npq(D,N,Ant),[N-A|R])   ==> np(npq(D,N,and(Ant,A)),R).
  np(npq(D,N,A),[p(Pr,Np)|R])==> np(npq(D,N,and(A,SemPrep)),R):-
          mod(Pr,Np,N) =*> SemPrep.
  np(D,A,N-Pred,P)           ==> np(npq(D,N,Pred),T):-append(A,P,T).

%% determinante
%% det = {o, um, todo, no, N}
det -(artnum)--> ?[art(_,_)], ?[card].
   artnum(nil,nil) ==> o.
   artnum(X,nil)   ==> X.
   artnum(_,Y)     ==> Y.

%% noun
%% n = elemento::classe - pred
n -(cn)-->   [nc(_,_)].
n -(pn)-->   [np(_,_)].
n -(cnpn)--> [nc(_,_)], [np(_,_)].
    cnpn(Cl::_,Cl2::I) ==> (Cl3::I)-nil :- glb(Cl,Cl2,Cl3).
    cn(X::Y)  ==> (X::Y)-nil.
    pn(X::p)  ==> (Z::pessoa)-infon(chamado,[Z::pessoa,X::nome]).
    pn(X::cid)==> (Z::cidade)-infon(chamado,[Z::cidade,X::nome]).
    pn(X::Y)  ==> (Z::Y)-infon(chamado,[Z::Y,X::nome]).
```

```
%% adjectivo
%% adjp = (elemento -> factos_mod)
adjp -(adj)-->  [adj(_,_)].
adjp -(adj1)--> [a_nc(_,_)].
    adj(P)     ==> X-infon(P,[X]).
    adj1(X::Y) ==> X-infon(Y,[X]).

%% complementos após nome
%% n_pcompl = lambda(elemento,factos_mod) | p(prep,NP)
n_pcompl > (adjp ; pp ; rel).

%%  frases preposicionais
pp -(p)--> [prep], np.

%% relativa simples
%% rels
rels -(que)-->  -[_-que], vpr.
rels -(cujo)--> -[_-cujo], np, vpr.
    que(V)                  ==> V.
    cujo(npq(_,N,Ant),N-V)  ==> Z-q3(o,N,T,V) :-
          and(infon(de,[Z,N]),Ant) =*> T.

rel > rels.
rel -(preprel)--> [prep], rels.
%    preprel(P,R) ==>

%% vpr -- frases verbais com falta de um np
vpr -(fcomp)--> np, *[adv], [v(_,_)], *vcompl.
vpr > vp.
   fcomp(npq(Q,N,Ant),[],V,[]) ==> Y-q3(Q,N,Ant,infon(V,[N,Y])).

%% frases verbais
vp -(v)--> *[adv], [v(_,_)], ?np, *vcompl.
   v(V,nil,C)          ==> X-infon(V/C,[X]).
   v(V,npq(D,E,A),C)   ==> X-q3(D,E,A,F)
           :- rewrite(infon(V/C,[X,E]),F).
   v(C1,V,N,C2)        ==> v(V,N,C) :- append(C1,C2,C).

   infon(X/[],Ar)      ==> infon(X,Ar).
   infon(X/[p(P,npq(Q,N,Ant))|R],Ar) ==>
            q3(Q,N1,Ant,infon(Y/R,[N1|Ar])) :- relpp(X,P,N,Y,N1).
   infon(X/[Adv|R],Ar) ==> infon(Y/R,Ar)
                          :-  reladv(X,Adv,Y).

relpp(X,em,N,X,em(N)).
reladv(X,A,X-A).
```

```
%% complementos de verbo / advérbios
vcompl > ([adv] | pp).
```

A reescrita semântica do exemplo anterior está a construir árvores ternárias de quantificadores (?) (um dos casos clássicos de representação semântica intermédia), tomando como elementos de base infons (inspirado em semântica de situação (Barwise and Cooper, 1991; Barwise and Moss, 1991)), ou seja, ao analisar um texto contendo:

*o João vê um velho galo.*

A (uma das) semântica obtida é:

```
q3(o,
   _A::pessoa,
   infon(chamado,[_A::pessoa,'João'::nome]),
   q3(um,
      _B::galo,
      infon(velho,[_B::galo]),
      infon(vê,[_A::pessoa,_B::galo])))
```

Essa representação semântica pode ser usada com uma variedade de intenções. Seguidamente, vamos considerar duas pragmáticas diferentes (cada uma definida por um sistema de reescrita) que convertam a representação semântica para:

1. uma notação próxima do Prolog
2. numa lista de assinaturas das relações existentes nos infons.

**Exemplo 11:** Gramática com reescrita pragmática

Sistema de reescrita pragmática para conversão para uma notação próxima do Prolog:

```
final ! infon(Rel,Args) ==> T :- atom(Rel), T =.. [Rel | Args].
final ! q3(o,_,X,Y)     ==> Z :- rewrite(and(X,Y),Z,final).
final ! q3(todo,X,Y,Z)  ==> all(X,Y1,Z1)
           :- rewrite(Y,Y1,final), rewrite(Z,Z1,final).
final ! q3(um,X,Y,Z)    ==> exist(X,Y1,Z1)
           :- rewrite(Y,Y1,final), rewrite(Z,Z1,final).
final ! q3(i,X,Y,Z)     ==> quest(X,Y1,Z1)
           :- rewrite(Y,Y1,final), rewrite(Z,Z1,final).
final ! q3(K,C::X,Y,Z)  ==> exist(C-set::X,Y1,Z1)
```

```
          :- integer(K),
             rewrite((infon(card,[C-set::X,K]),Y),Y1,final),
             rewrite(Z,Z1,final).
final ! and(A,B)        ==> (A1,B1)
          :- rewrite(A,A1,final),rewrite(B,B1,final).
```

Quando aplicado ao teste (texto) anterior dá origem a:

```
  chamado(_A::pessoa,'João'::nome),
  exist(_B::galo,
        velho(_B::galo),
        ver(_A::pessoa,_B::galo))
```

Esta reescrita pode ser usada para guardar em base Prolog conhecimento extraído de um texto.

**Exemplo 12:** Extracção de assinaturas

Sistema de reescrita pragmática para extracção de assinaturas das várias relações encontradas:

```
sig ! q3(_,_,Y,Z) ==>  L :-
    rewrite(Y,Y1,sig), rewrite(Z,Z1,sig), append(Y1,Z1,L).
sig ! nil         ==> [].
sig ! and(B,nil)  ==> B.
sig ! and(nil,B)  ==> B.
sig ! and(A,B)    ==> L :-
    rewrite(A,A1,sig), rewrite(B,B1,sig), append(A1,B1,L).
sig ! infon(Rel,[_::C]) ==> [sig(Rel,[C])].
sig ! infon(Rel,[_::C1,_::C2]) ==> [ sig(Rel,[C1,C2]) ].
sig ! infon(Rel,[_::C1,_::C2,_::C3]) ==> [sig(Rel,[C1,C2,C3])].

ex(1,'o João come um velho galo').
ex(2,'o João vê um velho galo que alegremente canta em casa').
ex(3,'o cão que ladra não morde').

make1(A,I,J) :- ex(A,I),set_string(I),s(J).
make2(A,I,K) :- ex(A,I),set_string(I),s(J),rewrite(J,K,final).
make3(A,I,K) :- ex(A,I),set_string(I),s(J),rewrite(J,K,sig).
```

Quando aplicado a

*A raposa cuja pegada o João vira comeu quatro gordas galinhas.*
*O João vê um velho galo que alegremente canta em casa.*

dá origem a:

```
[
 sig(de,[raposa,pegada]),
 sig(chamado,[pessoa,nome]),
 sig(ver,[pessoa,pegada]),
 sig(gordo,[galinha]),
 sig(comer,[raposa,galinha]),
 sig(chamado,[pessoa,nome]),
 sig(velho,[galo]),
 sig(cantar/[alegre],[galo]),
 sig(em,[casa,galo]),
 sig(ver,[pessoa,galo])
]
```

Note-se que uma representação como a que acabámos de extrair, pode ser usada para:

- determinação de domínios típicos de cada um dos papéis referentes a verbos, adjectivos, ...
- determinação de relações típicas em que cada *classe* anda envolvida,
- determinação de co-ocorrência de *classes* num mesmo infon,
- desambiguação de sentido de certas relações com base nos argumentos,

ou seja, para determinar algumas propriedades importantes acerca de palavras e conceitos.

## 6.4 PTC – Processador de Termos Compostos

Em Linguagem Natural são frequentes as situações em que certas sequências de palavras tem um significado diferente daquele que seria inferível a partir dos significados das suas partes constituintes.

Estas situações correspondem a diferentes fenómenos linguísticos (Xavier and Mateus, 1992; Baptista, 1994) como idiomatismos, lexias complexas, locuções, etc., que por simplicidade serão aqui designados por **termos compostos**[22].

Em processamento de linguagem natural, há frequentemente necessidade de pré-agrupamento das palavras em termos compostos para um tratamento correcto destes.

Nesta secção, apresenta-se uma ferramenta PTC, para detecção e tratamento automático de termos compostos baseada num dicionário externo.

Na sua versão actual o PTC permite:

- controlar o tipo de flexões possíveis e a que palavras podem ser aplicadas,
- devolver informação morfo-sintáctica associada ao termo composto. Essa informação resulta do dicionário e das flexões encontradas,
- tratar as ambiguidades léxicas,
- funcionar como etiquetador: percorrer textos e anotar os termos compostos encontrados,
- tratar sobreposições de termos compostos.

Uma descrição da estrutura interna de representação bem como a arquitectura geral da ferramenta será apresentada, juntamente com exemplos de utilização.

Esta secção aparece nesta dissertação por ser imprescindível dispor de mecanismo de descrição e de processamento de termos multi-palavra. Adicionalmente, este assunto é relevante para a criação/definição de dicionários porque:

- oferece um modo formal de descrever flexão de termos multi-palavra,
- enriquece a álgebra de processamento de linguagem natural com analisadores léxicos multi-palavra – o que em muitas situações, é crucial.

[22] Agradecemos a Álvaro Sanromán pelo esforço de clarificação das questões terminológicas envolvendo *colocações, frases feitas, fraseologia, idiomatismos, lexemas, lexias, locuções, modismos, solidariedades léxicas, etc.*

### 6.4.1 Introdução

#### Terminologia

Neste documento, usaremos **termos compostos** para designar genericamente as sequências de palavras que têm características não dedutíveis a partir das características das partes constituintes

$$TC \neq f(P_1, P_2, P_n)$$

Alguns exemplos de termos compostos são:

- Ideomatismos (Ex.: *pezinhos de lã*)
- Lexias complexas (Ex.: *máquina de escrever*)
- Locuções (Ex.: *para com, de modo que*).

#### Motivação

São várias as áreas em que o processamento de linguagem natural necessita do pré-agrupamento das palavras em termos compostos. Cite-se por exemplo:

**tradução automática** – a generalidade dos termos compostos não pode ser traduzido *à letra*. No processo de tradução há vantagem em existir um pré-processamento que encontre os termos compostos, para então se fazer uma correcta produção de equivalentes. Naturalmente, ninguém pretende ver *short circuit* tratado como *circuito pequeno...*

**localização de informação** (information retrieval) – para aceder selectivamente a partes de uma grande colecção de documentos, é necessário normalizá-los e indexá-los. Por vezes, não faz sentido que as palavras constituintes de um termo composto sejam tratadas como palavras isoladas.

**Exemplo 13:** memória de elefante

> a palavra *elefante* no composto *memória de elefante* não está relacionada com o mamífero elefante. Quem pesquisar informação sobre elefante não tem interesse em encontrar a entrada relativa a *memória de elefante.*

Na pesquisa pode haver utilização de termos compostos, havendo por isso a necessidade de existência de entradas indexadas dos mesmos.

**Exemplo 14:** base de dados

Para encontrar textos sobre *base de dados* interessa também *bases de dados*, mas não interessa frases contendo apenas *base* e *dados* sem formarem o termo composto.

**tratamento sintáctico de linguagem natural** – a não consideração de certas locuções, leva a situações em que a análise sintáctica falha ou em que há necessidade de uma escrita de analisadores muito mais complexos. (Ex.:... *para com* ...)

**tratamento semântico de linguagem natural** – no tratamento semântico o caso é ainda mais grave já que (por definição) o significado do composto não é dedutível a partir da semântica das palavras que o constituem, a única hipótese de tratamento semântico correcto desses casos é ter a indicação explícita da semântica do termo composto. (Ex.:*Apesar de tudo fomos embora*).

### Questões básicas a considerar

O reconhecimento de um termo composto não se limita evidentemente a uma simples comparação de *strings*, já que as próprias palavras que compõem o termo composto podem ser (parcialmente) flexionadas. Em alguns casos é importante:

- restringir o tipo de flexões que podem ser feitas a partir de cada palavra.
- dar a possibilidade de indicar *features* para o termo definido (de outro modo a sua utilidade em gramáticas (por exemplo) seria muito limitada)
- permitir a herança de *features* de algumas das palavras que a constituem.

**Exemplo 15:** classificação associada a termos compostos

a análise do termo composto *acertei em cheio* deve retornar informação sobre o tempo, o número e a pessoa, sendo esta informação herdada de *acertar* `[CAT=v,T=preper,N=s,p=1]`

No desenho de um processador de termos compostos, é importante ter em conta:

**Ambiguidade** – Há problemas no armazenamento dos termos compostos devido a ambiguidades léxicas dos termos constituintes. O reconhecimento levanta também problemas porque uma palavra pode ter mais que um radical diferente o que obriga a seguir simultaneamente vários caminhos diferentes na árvore de termos compostos.

**Categorização e semântica** – Aos termos compostos são associadas categorias gramaticais e respectivas semânticas, não sendo, em grande parte dos casos possível atribuir uma categoria atómica como *adjectivo, nome comum*, etc., correspondendo-lhes por vezes categorias gramaticais de nível superior - *frase nominal, frase verbal, frase,* etc.

### 6.4.2 Arquitectura do sistema Ptc

O objectivo básico do Ptc é obter sequencialmente os termos do texto. A função elementar responsável por essa tarefa é `get_next_lexia`, que a partir de uma posição (`pos`) num texto (ou *string*), retorna um conjunto de pares `(nova posição, lexia)`.

$$
\begin{array}{rcrll}
analise & = & lexia & : & \text{...termo (simples ou composto) e sua análise} \quad \times \\
 & & pos & : & \text{...ref. para a palavra seguinte a processar}
\end{array}
$$

$get\_next\_lexia : txt \times pos \longrightarrow set(analise)$
$get\_next\_lexia(t,p) \stackrel{\text{def}}{=}$
...dada uma posição no texto de entrada,
dá as várias análises (simples ou compostas)

Ou seja, para uma dada posição num dado texto, `get_next_lexia` dá o conjunto das possíveis lexias[23] e respectivas posições de continuação correspondendo portanto às várias hipóteses de análise.

Internamente, uma aplicação criada pelo utilizador invoca o módulo Ptc para obter termos (simples ou compostos).

Para realizar essa tarefa, o módulo Ptc implementa uma autómato baseado numa árvore de termos compostos. Essa árvore é o resultado da compilação de um dicionário contendo a descrição dos termos compostos.

O módulo Ptc invoca o analisador morfológico `jspell` para obter as palavras do texto de entrada e respectivas análises. Cada transição no autómato realizada pelo Ptc é feita tomando por base os lemas obtidos.

O módulo `jspell`, precisa naturalmente dum dicionário `jspell`, gerado pelo comando `jbuild`. Ver Figura 6.3.

[23] lexia, neste contexto, está a designar um termo simples ou composto.

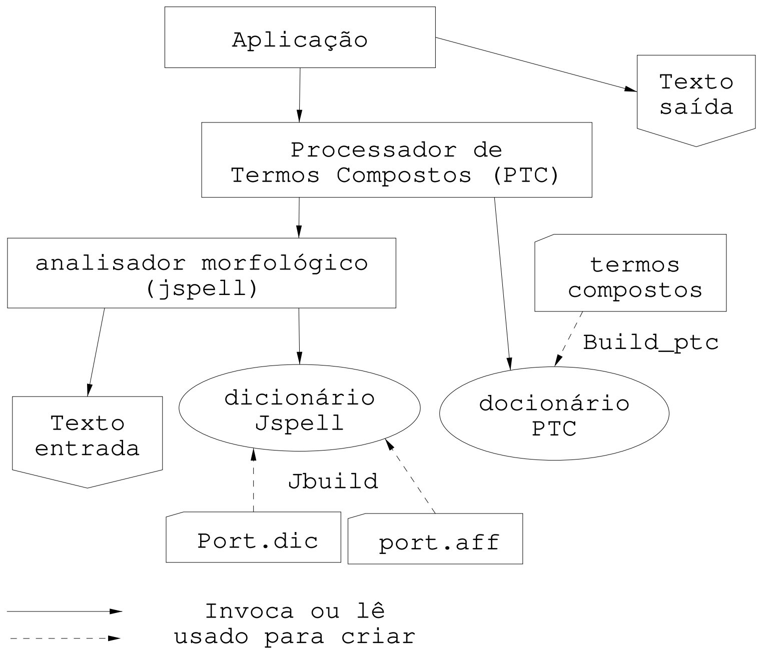



Figura 6.3: Arquitectura de PTC

### 6.4.3 Dicionários PTC

#### Formato de definição de termos compostos

Uma das principais dificuldade na implementação dum sistema deste tipo é o tratamento de flexões. Por ex. *andar à nora* deve incluir *andei à nora*, *andou à nora*, etc. mas não *andei às noras*. Isto significa que devemos dar a indicação de quais as palavras passíveis de serem flexionadas, o que pode ser expresso na seguinte notação:

`andar* à nora`

que tem o significado de

*[todas as flexões associadas a* `andar`*]* `à nora.`

Normalmente, é útil associar ao composto as suas *features* que podem ser indicadas explicitamente ou herdadas de uma das palavras que a compõem. O operador `*` indica flexão universal com herança de *features*.

No caso de existirem duas palavras flexionáveis no mesmo termo composto

(Ex: *acesso directo* no plural é *acessos directos*), devem associar-se dois operadores de flexão. O operador `%` é usado para flexão universal sem herança e o operador `&` para herança sem flexão.

**Exemplo 16:** `andar* depenado%G-N`

Ambas as palavras são flexionáveis: a palavra *andar* é flexionável e é a base para extracção das *features* do termo composto; *depenado* é flexionável em Género (G) e Número (N). Se pretendermos indicar que este termo composto é uma frase verbal (FV) optaríamos por:

`andar* depenado%G-N/CAT=FV`

A anotação `/CAT=FV` explicita *features* (neste caso a categoria) a adicionar (reescrever) ao termo composto.

Sendo assim, podemos definir uma entrada no dicionário que suporte o Ptc como:

```
entrada   → (palavra feaP)+ def_feat
def_feat  →   | '/' feat '=' valor (',' feat '=' valor)*
feaP      →   | sinal | sinal nfeatures
sinal     → '*' | '%' | '&'
nfeatures → feat ('-' feat)*
```

### Estrutura interna dos dicionários Ptc

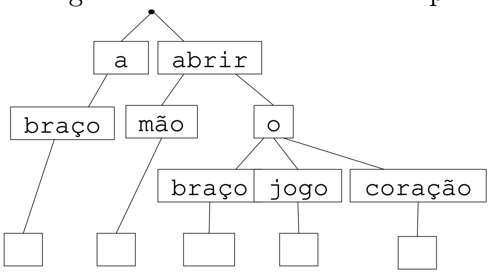
As palavras são armazenadas sob a forma de uma árvore. Cada nó tem uma quantidade variável de células, (igual ao número de termos cujas palavras anteriores são comuns e diferem neste nível). Por razões de eficiência, optou-se por guardar nos nós da árvore a forma normalizada das palavras, sendo acrescentadas nas folhas da árvore as condições que restringem as flexões de acordo com o dicionário original.

**Exemplo 17:** Representação interna dum dicionário Ptc

O dicionário de termos compostos:

```
a braços
abrir mão
abrir o coração
abrir o jogo
abrir os braços
```

fica armazenado na seguinte árvore de termos compostos:




Na árvore aparecem apenas os lemas (*braço* e não *braços*, já que *braços* não é forma normalizada) e a árvore funciona como índice de acesso à informação contida nas folhas.

Nas folhas há informação indicativa de:

– as *features* finais do termo composto,
– as *features* de restrição sobre cada uma das palavras que o constituem,
– palavra a partir da qual será feita a herança de *features*.

O algoritmo PTC corresponde a um autómato guiado pela árvore descrita.

### Grafo como resultado da análise de termos compostos

A extracção de um termo durante a análise pode conduzir a uma situação ambígua que iremos representar por um grafo acíclico (ver esquema abaixo).

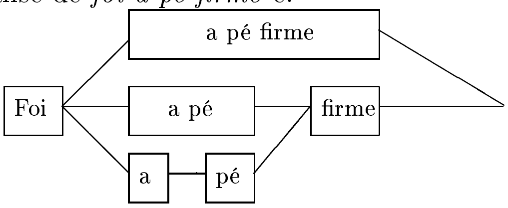
Para uma sequência de palavras que formem um termo composto, pode existir uma subsequência de palavras que também seja um termo composto. Pode ainda acontecer que as palavras devam ser interpretadas separadamente.

**Exemplo 18:** grafo de análise

Considere-se o seguinte dicionário de suporte PTC:

```
a pé
a pé firme
```

O grafo de análise de *foi a pé firme* é:



### 6.4.4 Modo de utilização do Ptc

Depois de definido um dicionário Ptc, é necessário compilá-lo para uma representação interna (uma variante de autómato). Seguidamente, a análise de textos fica acessível de dois modos:

- via programa etiquetador,
- via biblioteca C – o utilizador cria uma aplicação que invoca funções da biblioteca C.

#### Compilação de um dicionário Ptc

A utilização do sistema Ptc passa primeiro pela criação de um ficheiro intermédio que contém as expressões num formato mais manuseável do que o ficheiro de texto inicial que continha a definição de compostos dada pelo utilizador. Mais tarde, o ficheiro criado será usado no reconhecimento de termos compostos num texto.

```
build_ptc expr-por.txt port.hash expr.out
```

`expr-por.txt` – dicionário Ptc textual contendo a lista de termos compostos
`port.hash` – dicionário morfológico (criado pelo jbuild/`jspell`)
`expr.out` – o dicionário Ptc compilado.

#### Utilização como programa etiquetador

Neste modo de utilização, o programa recebe um texto de entrada gerando um novo texto em que cada um dos termos compostos encontrados é marcado com chavetas e com a respectiva categoria.

A invocação do programa para etiquetação das expressões será:

```
ptc port.hash expr.out < input > output
```

**Exemplo 19:** *A Joana abriu os cordões à bolsa alegremente*

tem como resultado

*a Joana {abriu os cordões à bolsa}[CAT=v,T=pp,P=3,N=s] alegremente*

**Exemplo 20:** Anotação de texto

resultado da anotação dum pequeno texto[24] (sem classificação):

*O João {{a pé}firme}, foi com os {gatos pingados} {a altas horas} {abaixo de Braga} {abanar o capacete}, {abriu o apetite}, indo a uma {casa de tias}, já que {a carne é fraca}, entrou {à bruta} e {ficou entregue à bicharada}.*

*{Andou por maus caminhos}, {armou-se em parvo} e levou um {balde de água fria}. Para {salvar a pele} pôs-se a mexer, o {pobre diabo} que era {pessoa de bem} {perdeu a cabeça} e {passou um mau bocado}.*

*{Fez {o diabo a quatro}}, descobriram-se-lhe {os podres} e as coisas foram {de mal a pior}. {Deu {tudo por tudo}}, {correu riscos} desnecessários, ficou {para sempre} a {ovelha ranhosa}, {vendeu caro a vida} mas ele no fundo até {tinha pinta}.*

### Utilização como biblioteca C:

A existência desta biblioteca permite utilizar o módulo de reconhecimento de termos compostos como componente de outros programas.

Como se disse, a função básica desta biblioteca é `get_next_lexia`, que a partir de uma posição num texto (ou *string*), retorna um conjunto de pares `(nova posição, lexia)`.

```
get_next_lexia(char *pos, comp_sol solutions[]);
```

A estrutura `comp_sol` é definida da seguinte forma:

```
typedef struct comp_sol {
   char *lex_end;
   char is_comp;   /*indica se se trata de um termo composto*/
   sol_type sol;
} comp_sol;
```

```
void arv_from_disk(char *filename);
```

[24]Exemplo retirado de (Pinto and Almeida, 1996)

### 6.4.5 Análise de resultados

Na concepção do programa Ptc para processamento de termos compostos, houve alguma preocupação de eficiência.

Num teste realizado com um dicionário de termos compostos de frases de tipo idiomáticos com cerca de 4500 frases aplicado a um sub-corpus com cerca de 3,6 milhões de palavras (retirado do corpus Natura-Público (Almeida, 1997)), foram encontrados 13K termos idiomáticos[25] em menos de 9 minutos[26]:

| | |
|---|---|
| Termos compostos no dicionário | 4482 |
| Tamanho do texto a etiquetar | 3 631 Kpal (22Mbytes) |
| Número de termos compostos encontrados | 13 754 termos |
| Tempo gasto na etiquetação | 8m40.550s |

[25] O número de lexias complexas e de locuções existentes nesse texto seria muito maior que o número de termos compostos de tipo idiomático. As frases idiomáticas foram escolhidas por serem estruturalmente mais complexas e assim constituírem um teste mais rico ao poder expressivo do sistema

[26] Tempos medidos num Celeron em carga, 700MHz, 128Mbytes de RAM, linux

## 6.5 EMS – Etiquetador morfo-sintáctico

Nesta secção apresenta-se a ferramenta EMS (Reis and Almeida, 1998) que é um Etiquetador Morfo-sintáctico para o Português.

Esta ferramenta é referida nesta dissertação por constituir uma ferramenta útil para juntar à álgebra dos dicionários e simultaneamente ser também um exemplo de utilização de DDMF.

O etiquetador EMS é uma variante do Etiquetador Dirigido por Regras de Eric Brill (Brill, 1992; Brill, 1993; Brill, 1994): um etiquetador dirigido por regras (contextuais e lexicais), que faz aprendizagem dessas regras a partir de corpora previamente etiquetado.

O corpus necessário para essa aprendizagem é volumoso: na documentação associada ao etiquetador de Brill, sugerem-se quantidades como 500 000 palavras...

A construção deste etiquetador constituiu um exercício de utilização de DDMF como meio de reduzir a quantidade de material de base necessário para o processo de aprendizagem: sempre que uma palavra é desconhecida, é usado um DDMF externo que fornece a informação que conseguir juntar, permitindo que o etiquetador faça uma aprendizagem aceitável com um corpus etiquetado muito mais pequeno[27].

### 6.5.1 Introdução

O EMS é um etiquetador para Português que é uma variante do Etiquetador de Brill, com algumas modificações:

- utilização adicional de um nível de pré-processamento (opcional),
- utilização de um DDMF para tratamento das palavras desconhecidas,
- utilização de um nível de análise sintáctica (opcional, e que não será aqui referido).

O DDMF de tratamento de palavras desconhecidas, recorre a um analisador morfológico (jspell) e a outras estratégias de modo a fornecer, o mais cedo possível, a informação que se consiga juntar acerca de palavras desconhecidas e destina-se a facilitar o processo de aprendizagem, bem como a facilitar o trabalho de preparação de um *Corpus* de treino inicial.

[27]De acordo com as experiências realizadas, estimamos que o corpus necessário com este método seja cerca de 50 vezes mais pequeno

O conjunto de etiquetas usado é formado por agregação dos vários traços (características) morfológicos: Categoria Gramatical, Subcategorias (quando exista), Género, Pessoa, Número, etc. Tentou-se, sempre que possível, que cada etiqueta tenha uma leitura hierárquica das características que a compõem. Este conjunto de etiquetas foi criado de raiz com a intenção de se adaptar bem à Língua Portuguesa.

Para a aprendizagem foi usado um *corpus* de perto de 10 000 palavras, extraído do *corpus* "Natura-Público", etiquetado por *bootstrapping* [28]

### 6.5.2 Arquitectura do EMS

O EMS é composto por um conjunto de módulos que se apresenta no esquema abaixo, sendo alguns destes de uso opcional, e podendo outros depender do exemplo em análise.

O bloco de *tokenização* (ver figura 6.4) e pré-processamento são dependentes da sintaxe concreta usada no texto a etiquetar.

Os blocos de *Start-state* e *Final-state* correspondem à etiquetação propriamente dita. O bloco *Final-state* leva em conta um contexto de cerca de 4 palavras para detectar eventuais alterações a fazer às etiquetas inseridas pelo *Start-state Tagger*.

O bloco correspondente à análise sintáctica[29] é opcional e não será abordado neste texto.

[28] Ou seja, etiquetar uma pequena porção de texto manualmente, e repetir:

- usar o corpus etiquetado disponível como *corpus* de treino,
- etiquetar mais uma porção de texto usando as regras deduzidas,
- corrigir os erros manualmente e juntar ao corpus disponível,

até se atingir um volume de texto suficiente.

[29] Este analisador (ainda experimental) é gerado pelo yacc (Johnson, 1975), baseia-se numa gramática desenhada especificamente para este efeito, e em que se forçou algumas interpretações de frase como meio de eliminar certas ambiguidades (como sejam ambiguidades ligadas aos sintagmas preposicionais). Para mais detalhes ver (Reis and Almeida, 1998).

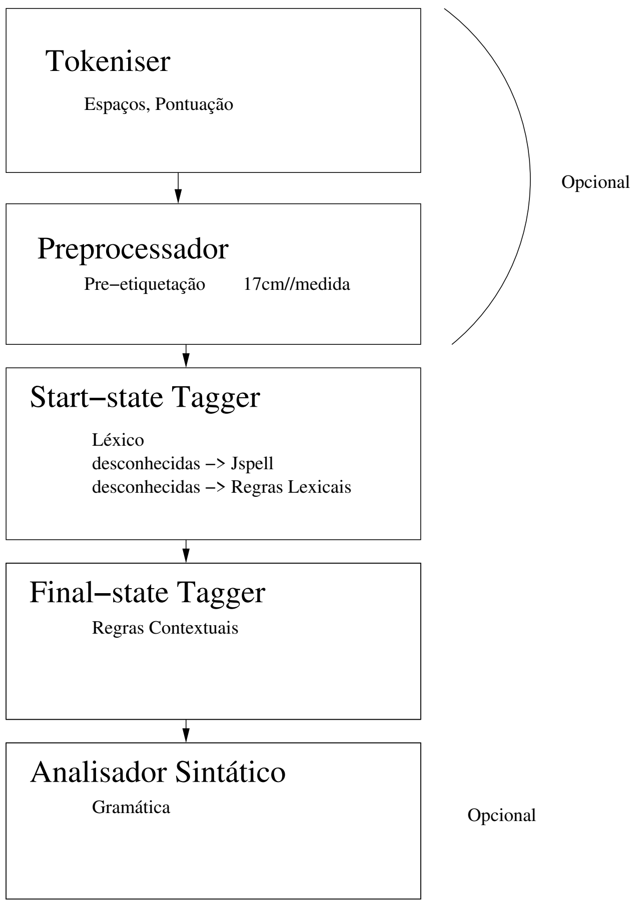



Figura 6.4: Arquitectura de EMS

### 6.5.3 Pré-processadores

#### Bloco de tokenização

A separação de uma frase nos seus elementos constituintes, é uma operação que está dependente do tipo de notação usada, das convenções da língua em análise, etc.

Esta separação não é tão simples como possa parecer, obrigando a análise de contexto de modo a tentar tratar convenientemente as abreviaturas, sinais de pontuação compostos, etc.

Algumas expressões científicas e/ou matemáticas requerem também tratamento especial, nomeadamente aquelas em que os pontos finais fazem parte do formato padrão – vírgulas decimais, pontos de separação de classes numéricas[30], factoriais, etc.

Na sua versão actual, o separador funciona em cascata, implementando os pontos referidos acima e permitindo blocos definidos pelo utilizador.

### Pré-etiquetação

O etiquetador de Brill aceita texto (parcialmente) pré-etiquetado, sendo nesse caso as etiquetas mantidas inalteráveis, isto é, sendo imunes às regras de transformação do Final-state tagger.

As vantagens de um andar de pré-etiquetação tem a ver com:

- tratamento de abreviaturas,
- tratamento de números, horas, medidas, resultados desportivos, etc.,
- conversão de anotações do texto original,
- utilização de processadores externos para etiquetação de fenómenos locais que possam recomendar um tratamento específico.

**Exemplo 21:** Pré-etiquetação

Considere-se o seguinte texto fonte:

```
Às 18h00m do dia 7, o Salgueiros empatou 2-2 com o
Benfica mantendo a tradição.
```

Neste tipo de situação, pode fazer sentido definir um pré-processador que faça a etiquetagem de alguns elementos. Considere-se o seguinte texto flex (Paxson, 1995) do pré-processador:

```
...
[0-9]+          {ECHO;printf("//DNCNP");}
[0-9]+h[0-9]+m {ECHO;printf("//HORA");}
[0-9]+-[0-9]+  {ECHO;printf("//RESUL");}
...
```

A aplicação deste pré-processador juntamente com o etiquetador produz o seguinte texto etiquetado:

[30] Em Portugal, o caracter de separação entre a parte inteira e a parte decimal de um número fraccionário é a vírgula. O ponto, por outro lado, pode ser empregue para separar as classes (unidades, milhares, milhões, ...) embora hoje em dia seja mais usual separá-las com um ligeiro espaço.

```
Às/\&FP 18h00m//HORA do/\&MS dia/NCMS 7//DNCNP ,/,
o/DADMS Salgueiros/NCMP empatou/VIP3S 2-2//RESUL
com/P o/DADMS Benfica/NPMS mantendo/VG a/DADFS
tradição/NCFS ./.
```

A explicação do significado das várias etiquetas será feito na próxima subsecção.

### 6.5.4 Etiquetas utilizadas

Sendo a Língua Portuguesa de origem latina, ela tem uma relativa complexidade morfológica. A definição de um conjunto de etiquetas para o Português é uma tarefa relativamente complexa, e corresponde a um compromisso entre a precisão na descrição e a viabilidade do processo de aprendizagem das regras.

No desenho do conjunto de etiquetas a usar, decidiu-se seguir alguns princípios funcionais:

- tentar ser o mais preciso possível;
- tentar seguir modelos já usados em experiências com outras línguas;
- definir uma estrutura hierárquica para a etiqueta – i.e., cada campo da etiqueta teria um significado específico que refinava o significado do campo anterior;
- não aprofundar as etiquetas para além do conjunto de coisas que possam vir a ser obtidas automaticamente;
- construir etiquetas de tamanho reduzido;
- permitir que possam ser usadas (por pré-etiquetação) etiquetas específicas para casos concretos – tipicamente para elementos não palavra.

A maior parte dos campos constituintes de uma etiqueta, tem uma só letra.

**Exemplo 22:** Análise da etiqueta de *empatou*

```
empatou/VIP3S
     V  = Verbo
     IP = Pretérito perfeito do indicativo
     3  = 3.a pessoa
     S  = singular
```

Podemos assim definir uma etiqueta com a seguinte estrutura:

```
Etiqueta: Categoria Subcategorias Género Pessoa Número
Categoria:  D | P | N | J | V | ADV | C | I | & | ? | QUE | SE
```

referentes a Determinantes, Pronomes ou Preposições, Nomes, adJectivos, Verbos, ADVérbios, Conjunções, Interjeições, Contracções, palavras desconhecidas, **que**, **se**.

Saliente-se que se optou por não classificar as palavras *que* e *se*, devido à complexidade que tal tarefa acarretaria.

A `Categoria` é o único campo obrigatoriamente preenchido. As subcategorias vão depender de cada categoria.

No caso de preposições ou interjeições, a categoria é o único valor.

Flexões comuns a muitos tipos:

**género** — M - masculino; F - feminino;
**número** — S - singular; P - plural;
**pessoa** — 1, 2, 3 - $1^a$, $2^a$ e $3^a$;

Etiquetas:

**Determinantes** — podem ser artigos ou numerais; flectem em tipo, género e número; Etiqueta: Dcategoriatipogéneronúmero
- **categoria** — DA - artigo; DN - numeral;
- **tipo-A** — DA: D - definido; I - indefinido;
- **tipo-N** — DN: C - cardinal; O - ordinal;

**Pronomes** — podem ser pessoais, relativos, possessivos, demonstrativos, interrogativos ou indefinidos; flectem em caso, género, pessoa e número; Etiqueta: Pcategoriacasogéneropessoanúmero;
- **categoria** — PS - pessoal; PR - relativo; PP - possessivo; PD - demonstrativo; PI - interrogativo; PF - indefinido;
- **caso** — N - nominativo; A - acusativo; G - genitivo; D - dativo;

**Nomes** — podem ser comuns ou próprios[31]; flectem em género e número; Etiqueta: Ncategoriagéneronúmero;
- **categoria** — NC - comum; NP - próprio;

**Adjectivos** — flectem em género e número; Etiqueta: Jgéneronúmero;

**Nomes ou Adjectivos** — flectem em género e número; esta categoria é provisória, devendo as regras contextuais resolvê-la em NC ou J; Etiqueta: Xgéneronúmero;

[31] Outras categorias, como colectivos, são integradas nos comuns.

**Verbos** — flectem em tempo, modo, género (particípio passado); pessoa e número; esta flexão não é total, pelo que agrupámos as combinações de tempo e modo numa composta, designada categoria; Etiqueta: Vcategoriagéneropessoanúmero

- **categoria** — formas nominais: N - infinitivo impessoal; PP - Particípio Passado; G - Gerúndio; modo Indicativo: IH - presente (Hoje); IP - pretérito Perfeito; II - pretérito Imperfeito; IM - pretérito Mais-que-perfeito; IF - Futuro; modo conjuntivo (Se): SH - presente (Hoje); SI - pretérito Imperfeito; PI - Futuro (ver Infinitivo Pessoal ou Presente); modo iMperativo: MH - presente (Hoje); modo Condicional: CH - presente (Hoje); modo Infinitivo (Pessoal ou Presente): PI;

**Preposições** — fixas; Etiqueta: P;

**Advérbios** — fixos; Etiqueta: ADV;

**Conjunções** — fixas; Etiqueta: C;

**Interjeições** — fixas; Etiqueta: I;

**Contracções** — flectem em género e número; Etiqueta: &géneronúmero

**Casos Particulares** — as palavras *que* e *se* têm etiquetas iguais a elas próprias: QUE e SE.

O conjunto actual de etiquetas encontra-se parcialmente incompleto: há certas categorias que não estão suficientemente refinadas, como as conjunções e os advérbios.

No caso destes últimos, a sua etiqueta deverá ser mudada para `A` em vez da actual `ADV`.

### 6.5.5 Processo de aprendizagem

Como se referiu, o etiquetador de Brill tem as seguintes características:

**É dirigido por regras** — ou seja, usa regras para determinar a etiqueta de cada palavra, (por oposição à classe de etiquetadores estocásticos, que são dirigidos por tabelas de probabilidades).

**Faz aprendizagem das regras** — extrai de um *corpus* etiquetado a informação morfológica e (proto-)sintáctica necessária para o seu funcionamento, guardando-a sob a forma de regras. Esta fase denomina-se *aprendizagem*.

**Usa regras lexicais** — as regras lexicais servem para determinar a etiqueta de palavras desconhecidas (não presentes no léxico).

**Usa regras contextuais simples** — as regras contextuais servem para corrigir possíveis erros das anteriores. Para tal, guiam-se pela vizinhança (um contexto reduzido) da palavra em questão, analisando etiquetas ou palavras até uma vizinhança de 4 palavras.

Quanto maior for esta base de treino, melhor, porque mais exactas e abrangentes, serão as regras deduzidas.

Os *Corpora* utilizados para este fim, em outras línguas, ascendem a, no mínimo, 500 000 palavras.

Infelizmente, dado que não dispúnhamos deste recurso na Língua Portuguesa, houve necessidade de criar um corpus etiquetado por *bootstrapping* (conforme descrito anteriormente).

### 6.5.6 Estratégia de redução do esforço de aprendizagem

Para reduzir o esforço envolvido, optou-se por alterar o processo de aprendizagem: usou-se um *corpus* de aprendizagem muito mais pequeno (± 10 000 palavras), recorrendo-se a um dicionário/analisador morfológico externo em caso de palavras desconhecidas.

Deste modo, quando o etiquetador encontra uma palavra desconhecida, isto é, uma palavra que não consta do léxico de base, antes de recorrer às regras lexicais consulta um DDMF `guesser`.

A implementação actual abre um *pipe* bidireccional para um programa DDMF escrito em Perl que calcula a informação referente à palavra e traduz para a notação de etiqueta a respectiva resposta (`guesser`).

#### DDMF de palavras desconhecidas

Para determinação de etiquetas a atribuir às palavras desconhecidas, usou-se a composição série de três dicionários :

$$
\begin{array}{rl}
guesser \stackrel{\text{def}}{=} \underline{let} & dnomes(x) = DicionarioNomesProprios(x) \\
& dmorf(x) = jspell\_tags(x)(0) \\
& daux(x) = \begin{cases} maiuscula(x) & \Rightarrow \texttt{"NP"} \\ \underline{otherwise} & \Rightarrow \texttt{"NC"} \end{cases} \\
& fontes = \left( \begin{pmatrix} d1 \\ dnomes \end{pmatrix} \begin{pmatrix} d2 \\ dmorf \end{pmatrix} \begin{pmatrix} d3 \\ daux \end{pmatrix} \right) \\
\underline{in} & mkPergPorOrdem(<d1, d2, d3>, fontes)
\end{array}
$$

ou seja, o dicionário `guesser` procura sucessivamente em *dnomes, dmorf e daux* até haver resposta:

- se a palavra existe no Dicionário de Nomes próprios, consulta-se esse Dicionário de nomes e toponímia.
- se é conhecida do analisador morfológico jspell, consulta o analisador morfológico jspell com o dicionário Português e selecciona-se a primeira resposta.
- se a palavra começa por maiúscula tenta-se adivinhar género e número devolvendo *nome próprio*.
- senão devolve-se *nome comum*.

### 6.5.7 Avaliação e Resultados

Numa análise superficial, após o processo de aprendizagem com um corpus jornalístico de 10 000 palavras extraído do Natura-Público, obteve-se: uma taxa de acerto de cerca de 96% quando aplicado a outro texto jornalístico; uma taxa de acerto de 92% quando aplicado ao prólogo do livro "*Eurico o Presbítero*".

Algumas etiquetas de super-categorização, como *adjectivos ou nomes comuns* (`X__`), foram introduzidas pelo *guesser*, no intuito de serem resolvidas pelas regras contextuais, uma vez que não se encontram no *Corpus* de treino. Algumas não o foram devido à insuficiência contextual.

### 6.5.8 Alguns problemas pendentes

Na versão actual, há problemas com palavras cuja análise deveria ser uma etiqueta composta. Isso acontece, por exemplo, no caso das contracções e no caso dos clíticos. Em ambos os casos, o ideal seria dividir cada palavra composta (contracção, conjugação pronominal, etc.) nos lexemas constituintes, etiquetando-os individualmente. As contracções estão todas aglutinadas numa só categoria, o que será revisto em futuras versões, quando na realidade a existência dessa mesma categoria é questionável.

## 6.6 XML::DT – módulo de processamento de documento XML

Esta secção aparece nesta dissertação porque se considera muito importante a capacidade de equipar a bancada de construção de dicionários com ferramentas capazes de processar, transformar ou gerar textos com estrutura rica. De entre os textos com estas características, a família dos documentos anotados em XML (XML, 1998; Ramalho, 2000) constitui, sem dúvida, um domínio de relevância grande, tanto pelo facto de ser um standard, como pela sua grande divulgação actual.

O XML::DT, que aqui se apresenta, é um módulo Perl desenhado para possibilitar um processamento estrutural de documentos XML. No seu desenho, teve-se em conta a intenção de permitir a construção de especificações de processadores tão compactas quanto possível.

Nesta secção será feita um breve descrição do módulo; seguidamente, será feita uma sucinta análise formal do problema genérico de processar XML, sendo colocada ênfase especial na função `dt` – a que mais directamente está ligada ao processamento estrutural de XML. Depois são apresentados alguns exemplos que mostram a sintaxe concreta e que dão uma visão mais exacta do modo de utilização do referido módulo.

O XML::DT[32] permite também processar ficheiros HTML permitindo funcionar como ferramenta para engenharia reversa de HTML e para extracção de conhecimento a partir de páginas HTML (ver detalhes em (Almeida and Simões, 2003)).

### 6.6.1 Breve descrição do XML::DT

No processamento de documentos XML, são usados processadores estruturais, isto é, processadores guiados pela estrutura do documento. De um modo simplificado, quando se usa uma linguagem de *marcas* para anotar os documentos, cada processador é definido por uma correspondência entre as várias etiquetas e uma função de processamento.

Neste módulo estão incluídas funções para processamento estrutural de ficheiros, de strings, usando ou não XPATH[33] (XPATH, 1999), funções para

[32] Nas versões 1.24 e superiores

[33] De um modo muito simplificado, XPATH permite descrever padrões sobre caminhos

construção de esqueletos de processadores[34], com base em ficheiros exemplo, funções para calcular DTDs[35]

Para além destas, existem no módulo um conjunto de funções que facilitam a estrita das funções de processamento.

### 6.6.2 O algoritmo da função `dt`

O algoritmo que seguidamente se apresenta é uma ligeira simplificação do programa real, feita no sentido de tornar mais fácil o seu entendimento.

Um texto XML pode ser visto como uma árvore generalizada, etiquetada em cada nodo com uma marca (tag), contendo atributos identificados e uma sequência de descendentes, que podem ser árvores ou textos (pcdata).

$$\begin{array}{lcl} XML & = & elem \\ elem & = & \begin{array}{rcll} name & : & idEle & \times \\ atr & : & atributos & \times \\ cont & : & filho^{\star} & \end{array} \\ atributos & = & id \rightharpoonup str \\ filho & = & txt + elem \end{array}$$

#### Processamento estrutural

O processamento de um documento XML, deverá naturalmente, ser guiado pela respectiva estrutura. No caso do módulo XML::DT,

$$\begin{array}{lcl} funProEle & = & idEle \times atributos \times anyb \longrightarrow anya \\ funProFilhos & = & filhoVal^{\star} \times processador \longrightarrow anyb \\ filhoVal & = & idEle \times anya \\ processador & = & \begin{array}{rcll} pe & : & idEle \rightharpoonup funProEle & \times \\ pf & : & idEle \rightharpoonup funProFilhos & \end{array} \end{array}$$

um processador é constituído por:

- um conjunto de funções *funProEle* que define como é feito o processamento de cada tipo de elemento, em função dos seus atributos e dos

dentro da árvore documento. Ex: `/documento/seccao/titulo` corresponde aos elementos *títulos* mas apenas no contexto *documento/secção*.

[34] A construção de esqueletos de processadores corresponde a, analisando documentos a processar, gerar partes dum programa processador, a ser completado.

[35] Como é óbvio, com base em exemplos, consegue-se apenas inferir um possível DTD que funcionará como esqueleto para ser editado manualmente de modo a constituir o DTD pretendido.

seus filhos,
- um conjunto de funções processa filhos (*funProFilhos*) que (para cada tipo de elemento) define como agrupar num único valor a lista de resultados devolvidos pelo processamento dos seus filhos (exemplo: concatená-los ou constituir uma lista com eles).

A função de processamento `dt`, usa o módulo XML-Parser (Wall and Cooper, 2000), ou o módulo XML::LibXML (Sergeant and Glahn, 2001) para construção de uma árvore, que representa (corresponde) ao documento de entrada, a qual é processada pelo função `proc`.

A função $pe(p)(\texttt{"-end"})$ é usada para permitir armazenar no processador uma função que faça um pós-processamento do resultado obtido.

$$
\begin{array}{l}
dt : string \times processador \longrightarrow anya \\
dt(file, p) \stackrel{\text{def}}{=} \\
\quad \begin{array}{ll}
\underline{let} & tree = Parse(file) \\
 & f = pe(p)(\texttt{"-end"}) \ or \ id \\
\underline{in} & f(proc(tree, p))
\end{array}
\end{array}
$$

$$
\begin{array}{l}
proc : filho \times processador \longrightarrow anyb \\
proc(x, p) \stackrel{\text{def}}{=} \\
\quad \begin{array}{ll}
\underline{let} & <pEle, pFil> = p \\
\underline{in} & \left\{ \begin{array}{lll}
\text{is-}elem(x) & \Rightarrow & \begin{array}{ll}
\underline{let} & <ele, atr, fs> = x \\
 & pe = pEle(ele) \\
 & m = pFil(ele) \\
 & fiseq = <<name(f), proc(f, p)> \mid f \in fs> \\
\underline{in} & pe(ele, atr, m(fiseq, p))
\end{array} \\
\underline{otherwise} & \Rightarrow & x
\end{array} \right.
\end{array}
\end{array}
$$

A versão que a seguir se apresenta, corresponde a uma alteração da anterior de modo a permitir usar:

- uma função de processamento de elemento `-default`
- uma função de processamento de `pcdata`
- estratégias pré-definidas de processamento dos $filhos$ (fs).

Para simplificar o modelo, não se abordará:

- `-inputencoding -outputenconding`
- possibilidade de alterar estado e efeitos laterais
- acesso ao contexto (sequência de etiquetas correspondentes ao *path* do elemento actualmente em processamento)
- atributos herdados.

Altera-se portanto a noção de processamento dos *filhos* de modo a que se indique o nome da estratégia a usar (por omissão a estratégia `STR`) passando a existir uma tabela de estratégias. Cada estratégia define o modo de processar os filhos e juntar os seus resultados da maneira mais conveniente.

$$\begin{array}{lcl}
processador & = & \begin{array}{lcl} pe & : & idEle \rightharpoonup funProEle \quad \times \\ pf & : & idEle \rightharpoonup idfunProFilhos \end{array} \\
funProEle & = & idEle \times atributos \times anyb \longrightarrow anya \\
funProFilhos & = & filhoVal^\star \times processador \longrightarrow anyb \\
estrategias & = & idfunProFilhos \rightharpoonup funProFilhos \\
filhoVal & = & idEle \times anya
\end{array}$$

$$estrategia' = \left( \begin{pmatrix} \texttt{"STR"} \\ proFilhosSTR \end{pmatrix} \begin{pmatrix} \texttt{"MAP"} \\ \dots \end{pmatrix} \begin{pmatrix} \dots \\ \dots \end{pmatrix} \right)$$

$$\begin{array}{l}
proc : filho \times processador \longrightarrow anyb \\
proc(x,p) \stackrel{\text{def}}{=} \\
\quad \underline{let} \;\; <pEle, pFil> = p \\
\quad \underline{in} \left\{ \begin{array}{lcll}
\text{is-}elem(x) & \Rightarrow & \underline{let} & <ele, atr, fs> = x \\
 & & & pe = pEle(ele) \; or \; pEle(\texttt{"-default"}) \; or \; id \\
 & & & m = estrategia(pFil(ele) \; or \; \texttt{"STR"}) \\
 & & & fiseq = <<name(f), proc(f,p)> \mid f \in fs> \\
 & & \underline{in} & pe(ele, atr, m(fiseq, p)) \\
\underline{otherwise} & \Rightarrow & \underline{let} & pe = pEle(\texttt{"-pcdata"}) \; or \; id \\
 & & \underline{in} & pe(x)
\end{array} \right.
\end{array}$$

Na versão que se apresenta seguidamente, a informação do processador foi junta num único *mapping* criando-se para tal a entrada `-type` que guarda a tabela das estratégias.

$$\begin{array}{l}
proc : filho \times processador \longrightarrow anyb \\
proc(x,p) \stackrel{\text{def}}{=} \\
\quad \left\{ \begin{array}{lcll}
\text{is-}elem(x) & \Rightarrow & \underline{let} & <ele, atr, fs> = x \\
 & & & f = p(ele) \; or \; p(\texttt{"-default"}) \; or \; id \\
 & & & m = p(\texttt{"-type"})(ele) \; or \; \texttt{"STR"} \\
 & & & fiseq = <<name(fi), proc(fi,p)> \mid fi \in fs> \\
 & & \underline{in} & f(ele, atr, m(fiseq, p)) \\
\underline{otherwise} & \Rightarrow & \underline{let} & f = p(\texttt{"-pcdata"}) \; or \; id \\
 & & \underline{in} & f(e)
\end{array} \right.
\end{array}$$

### Estratégias de processamento

As estratégias de processamento dos filhos podem ser muitas. De entre as várias estratégias de processamento dos filhos, há algumas cuja generalidade foi reconhecida[36] e por isso estão pré-definidas no módulo XML::DT, como é o caso das que estão apresentadas no resto da subsecção.

### Estratégia STR

A estratégia `STR` calcula os valores associados aos filhos e faz a sua concatenação como strings. Para que este funcionamento possa fazer sentido, o resultado de processar cada filho deverá ser do tipo string.

Esta é a estratégia por omissão e a mais simples de entender. Muitos dos utilizadores do módulo têm resolvido uma imensa quantidade de problemas usando apenas esta estratégia.

$procseq : filhoVal^{\star} \times processador \longrightarrow string$
$procseq(fs, p) \stackrel{\text{def}}{=}$
  $\underline{pre}$ valores de cada filho é tipo string
  $\underline{in}$ $strcat - red(\texttt{""}, <\pi_2(x) \mid x \in fs>)$

O operador *-red* ou também designado por *-orio*, constrói funções como o somatório ($add - red(0, s)$), produtório ($mul - red(1, s)$) e neste caso *concatenatório* de strings – i.e. concatenação de todas as strings da lista recebida.

As estratégias que se seguem, são normalmente usadas quando há utilidade em mapear partes do documento XML em sub-árvores (Perl), com valores complexos, de modo a podê-las processar como um todo. Essas estratégias são especialmente interessante em partes de documento que sejam regulares, i.e., que tenham estrutura nítida.

### Estratégia SEQ

A estratégia `SEQ` calcula os valores associados aos filhos e constrói uma sequência com eles.

$procseq : filhoVal^{\star} \times processador \longrightarrow anya^{\star}$
$procseq(fs, p) \stackrel{\text{def}}{=} <\pi_2(x) \mid x \in fs>$

[36]Como era de esperar, não é por acaso que essas estratégias estão ligadas a tipos abstractos de dados.

Esta estratégia é útil para processar sequências homogéneas.

### Estratégia MAP

A estratégia `MAP` calcula os valores associados aos filhos e constrói uma correspondência entre a etiqueta de cada filho e o respectivo valor. Todos os filhos deverão ter etiquetas diferentes.

$$\begin{array}{l} procseq : filhos \times processador \longrightarrow (string \rightharpoonup anya) \\ procseq(fs, p) \stackrel{\text{def}}{=} \\ \quad \underline{pre} \quad \text{tags únicas em todos os filhos } \wedge \text{no máximo 1 pcdata como filho} \\ \quad \underline{in} \quad \begin{pmatrix} \pi_1(x) \\ \pi_2(x) \end{pmatrix}_{x \in fs} \end{array}$$

Esta estratégia é boa para processar directamente partes de texto XML que sejam correspondências $string \rightharpoonup any$.

### Estratégia MULTIMAP

A estratégia MULTIMAP consiste em agrupar todos os filhos que tenham igual etiqueta ($idEle$).

$$\begin{array}{l} procseq : filhoVal^{\star} \times processador \longrightarrow (idEle \rightharpoonup anya^{\star}) \\ procseq(fs, p) \stackrel{\text{def}}{=} \\ \quad \underline{let} \quad seqPares = <<\pi_1(x), \pi_2(x)> \mid x \in fi> \\ \quad \underline{in} \quad juntaDir - red((), seqPares) \end{array}$$

$$\begin{array}{l} juntaDir : (\text{idEle * anya}) \times (idEle \rightharpoonup anya^{\star}) \longrightarrow (idEle \rightharpoonup anya^{\star}) \\ juntaDir(par, ac) \stackrel{\text{def}}{=} \\ \quad \underline{let} \quad <nome, valor> = p \\ \quad \phantom{\underline{let}} \quad antValor = ac(nome) \text{ } or <> \\ \quad \underline{in} \quad ac \dagger \left( \begin{pmatrix} nome \\ antValor \frown <valor> \end{pmatrix} \right) \end{array}$$

Esta estratégia é boa para processar directamente partes de texto XML que sejam correspondências $string \rightharpoonup any^{*}$.

### Estratégia MMAPON

Esta estratégia é uma mistura entre a estratégia MAP e a estratégia MULTIMAP: é dado um conjunto de elementos em que pode haver repetições e que são tratados com a estratégia MULTIMAP, sendo todos os outros processados como MAP. Os resultados são seguidamente reunidos.

Esta estratégia é boa para processar directamente partes de texto XML que sejam correspondências $string \rightharpoonup any^* + any$.

### Estratégia ZERO

Nesta estratégia e na seguinte, usa-se uma assinatura diferente para a função `procseq`: em vez de receber o resultado e o nome dos elementos filhos ($filhoVal^*$), recebe-se a lista dos filhos ($(txt + elem)^*$) antes de serem processados, de modo a permitir que se possa tomar a decisão de não os processar.

A estratégia ZERO consiste precisamente em não processar os filhos e retorna sempre o valor `""`. Isto torna-se importante por questões de eficiência e de tratamento de efeitos laterais.

### Estratégia SEQH

$$
\begin{array}{l}
procseq : filhos \times processador \longrightarrow string \rightharpoonup anya^\star \\
procseq(fs, p) \stackrel{\text{def}}{=} \\
\quad < \left( \begin{pmatrix} \texttt{"-q"} \\ name(x) \end{pmatrix} \begin{pmatrix} \texttt{"-c"} \\ proc(x, p) \end{pmatrix} \right) \dagger\, atr(x) \mid x \in fs >
\end{array}
$$

De um modo mais formal,

- A função *proc* corresponde a um catamorfismo sobre a árvore guiado pelo processador $p$.
- A função *procseq* corresponde a um catamorfismo sobre a sequência de filhos.
- As estratégias correspondem a monades.

### 6.6.3 Descrição do XML::DT através do seu uso

Nesta subsecção, incidir-se-á essencialmente na função `dt`.

Conforme referido, esta função recebe como parâmetros um ficheiro XML e um processador. Na implementação concreta Perl, o processador é uma correspondência entre etiquetas de elementos e funções (neste caso funções anónimas – `sub` do Perl). O processador pode ainda incluir uma função de processamento genérico (associada à chave `-default`) a ser utilizada para elementos cuja etiqueta não exista no domínio, uma definição de estratégias de processamento de filhos (associada à chave `-type` e que será exemplificada mais à frente). O processador pode ainda incluir definição de alguns valores de parametrização.

Para tornar mais compacta a definição das funções de processamento associadas aos vários elementos, cada função tem acesso às seguintes variáveis globais ( na realidade isto não é mais que um método de passagem de parâmetros):

- `$q` – o nome da etiqueta do elemento,
- `$c` – o conteúdo do elemento (content), ou seja o resultado do processamento dos seus filhos,
- `$v{atrName}` – os atributos associados ao elemento – um array `v` indexado pelo nome do atributo.

#### Exemplos introdutórios

**Exemplo 23:** Alteração de um documento XML

Enunciado: dado um documento XML alterar todas as ocorrências do atributo `at` existentes no elemento `doc`, de modo a ficarem em caracteres minúsculos.

```
1 use XML::DT;
2 print dt("file.xml",
3          ( doc => sub{ $v{at} = lc($v{at}); toxml()})
4         );
```

Notas:

**linha 2** - Aplicação da função `dt` ao ficheiro `file.xml` usando o processador descrito nas linhas seguintes.

**linha 3,4** - Definção do processador: apenas há definição de função de processamento para o elemento `doc` (todos os outros ficam sujeitos à função por omissão – função indentidade). O atributo `at` é substituído pelo respectivo valor em minúsculas. No final, a função `toxml()` devolve a reconstrução em XML, feita com base nas variáveis globais atrás referidas (`{$q, $c, $v{...}`)

Dado que não há especificação de estratégias de processamento dos filhos (não há `-type => ...`) a estratégia usada é STR (estratégia por omissão).

Neste exemplo, consegue-se observar que o facto de se estar a actuar num ponto localizado da árvore do documento, torna a resolução deste problema extremamente simples e independente dos detalhes sintácticos do XML.

**Exemplo 24:** Uso de estado

Num documento XHTML, substituir `<contents>...</contents>` por um índice calculado com base nos elementos `h1, h2 e h3`.

```
1   %handler=(
2      h1       => sub{ $index .= "\n$c";     toxml();},
3      h2       => sub{ $index .= "\n\t$c";   toxml();},
4      h3       => sub{ $index .= "\n\t\t$c"; toxml();},
5      contents => sub{ $c = "__CLEAN__";     toxml();},
6      -end     => sub{ $c =~ s!__CLEAN__!<pre>$index</pre>!g; $c});

7   print dt($filename,%handler)
```

Notas:

**linha 1 a 6** - Definição do processador.

**linha 2 a 4** - Sempre que se encontra um elemento h1, h2 ou h3, acrescenta-se o seu conteúdo a um `index` com alguma formatação.

**linha 4** - Sempre que aparece um elemento `contents`, limpa-se o seu conteúdo substituindo-o pela string `CLEAN`.

**linha 5** - No final, substitui-se todas as strings `CLEAN` pelo `index` calculado.

Deste exemplo, conclui-se que o uso de variáveis de estado, embora envolvendo certos riscos, pode simplificar algumas tarefas.

### `mkdtskel` – um programa para gerar processadores XML::DT

Para simplificar a tarefa de escrever processadores XML::DT, criou-se um programa **`mkdtskel`** que gera o esqueleto de um processador a partir de um ficheiro XML exemplo.

Dado que este programa gerador foi escrito em XML::DT vai aqui ser usado para demonstrar outro tipo de utilização do módulo.

Neste caso usar-se-á a acção por omissão (`-default`) para registar os elementos encontrados e respectivos atributos e a acção final (`-end`) para escrever esse dados recolhidos em formato Perl/XML::DT.

```
1  sub mkdtskel{
2   my $file = shift;
3   my %mkdtskelhand=(
4       '-default' => sub{$element{$q}++;
5                         for (keys %v){$att{$q}{$_}=1 }; ""},

6       '-end' => sub{
7           print "#!/usr/bin/perl
8                  use XML::DT ;
9                  my \$filename = shift;
10                 \%handler=(";
11          for $name (keys %element){
12              print "# '$name' => sub{\"\$q:\$c\"},",
13              print "# remember \$v{",
14                  join(",",keys %{$att{$name}}), "}" if $att{$name};
15              print "# $element{$name}\n";
16          }
17          print ");\n print dt(\$filename,\%handler);\n"
18      }
19  );

20  dt($file,%mkdtskelhand)
21 }
```

Notas:

**linha 1** - A função `mkdtskel`

**linha 2** - Recebe um ficheiro como parâmetro.

**linha 3 a 19** - Definição do processador a aplicar ao ficheiro recebido

**linha 4** - Função de processamento por omissão – a aplicar a todos os elementos.

**linha 4** - Conta quantas vezes apareceu este elemento.

**linha 5** - Conta quantas vezes aparece cada atributo.

**linha 6 a 19** - Escreve um texto Perl que é o esqueleto do programa de processamento típico para o tipo de ficheiro recebido.

**linha 7 a 10** - Escreve a parte inicial do processador (sempre igual).

**linha 11 a 16** - Para cada elemento, escreve a associação etiqueta, função, escreve a lista dos atributos encontrados, e o número de vezes que ele ocorreu no ficheiro exemplo. A escrita de um comentário com o número de ocorrências de cada elemento bem como a lista dos atributos encontrados no(s) ficheiro(s) exemplo, destina-se a documentar algumas estatísticas para facilitar o trabalho seguinte.

**linha 20** - Processa o ficheiro exemplo com o processador definido.

### Exemplo envolvendo estratégias diferentes de STR

Neste exemplo, partir-se-á de um ficheiro XML, como aquele que se exemplifica abaixo, contendo informação acerca de uns *proceedings*. Pretende-se que o processador a construir crie um dicionário a partir da informação contida nesse tipo de ficheiros XML.

Descrição dos *proceedings* em XML (exemplo):

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<proceedings>
  <title>The XML Europe 99</title>
  <chair>Pam</chair>
  <abstract>
      Once upon a time in Granada ...
  </abstract>

  <article>
     <title>The XML Down Translator</title>
     <author>J. João Almeida</author>
     <author>J. Carlos Ramalho</author>
     <tool>XML::DT</tool>
     <keys>
       <keyword>XML</keyword>
       <keyword>processamento de documentos</keyword>
       <keyword>Perl</keyword>
     </keys>
     <abstract>
       Era uma vez,
     </abstract>
     ...
  </article>

  <article>
     <title>The XML Parser</title>
     <author>Clark Cooper</author>
     <author>Larry Wall</author>
     <tool>XML::Parser</tool>
     <keys>
        <keyword>XML</keyword>
        <keyword>Perl</keyword>
     </keys>
     <abstract>
       Era outra vez,
     </abstract>
     ...
```

```
  </article>

  <article>
     <title>Perl, the first postmodern computer language</title>
     <author>Larry Wall</author>
     <keys>
        <keyword>linguística</keyword>
        <keyword>Perl</keyword>
        <keyword>linguagens de programação</keyword>
     </keys>
     <abstract>
       Obviously I should have made up a scarier title...
     </abstract>
     ...
  </article>

</proceedings>
```

Dado que este caso se aproxima de um exemplo real, a primeira questão que se vai pôr é determinar o conjunto dos elementos existentes. Para isso, vamos usar o comando `mkdtskel`, atrás apresentado, que constrói um esqueleto de processador a partir de um (ou mais) ficheiro exemplo.

O resultado de aplicar `mkdtskel` ao ficheiro XML apresentado acima, é o seguinte:

```
#!/usr/bin/perl
use XML::DT ;
my $filename = shift;

%handler=(
#     'article' => sub{"$q:$c"},# 3
#     'keys' => sub{"$q:$c"},# 3
#     'keyword' => sub{"$q:$c"},# 8
#     'title' => sub{"$q:$c"},# 4
#     'proceedings' => sub{"$q:$c"},# remember $v{year} 1
#     'author' => sub{"$q:$c"},# 5
#     'abstract' => sub{"$q:$c"},# 4
#     'tool' => sub{"$q:$c"},# 2
#     'chair' => sub{"$q:$c"},# 1
);
print dt($filename,%handler);
```

Este esqueleto seria usado como base para a escrita do processador (tipicamente descomentar os elementos a transformar e escrever as respectivas funções de transformação). A indicação dos nomes de todos os elementos e respectivos atributos, bem como o número de ocorrências de cada elemento, ajuda normalmente a obter uma solução mais rapidamente. Para o problema presente, ignorar-se-á este esqueleto.

O processador que se segue, pode ser usado para carregar (de modo controlado) o ficheiro XML numa estrutura (árvore generalizada) Perl.

```
%handler=(
    '-outputenc' => 'ISO-8859-1',
    '-type' => { proceedings => MMAPON("article"),
                 article     => MMAPON("author","tool"),
                 keys        => "SEQ",
               },
    '-default'   => sub{$c},
);
$out = dt($filename,%handler);
```

Ou seja, faria com que `$out` ficasse com o valor:

```
{ title => 'The XML Europe 99',
  abstract => ' Once upon a time in Granada ...  ',
  chair => 'Pam'
  article =>
   [ { title => 'The XML Down Translator',
         author => [ 'J. João Almeida', 'J. Carlos Ramalho' ],
         tool => [ 'XML::DT' ],
         keys => [ 'XML', 'processamento de documentos', 'Perl' ]
         abstract => ' Era uma vez, ',
         -pcdata => ' ...  ',
       },
       { title => 'The XML Parser',
         author => [ 'Clark Cooper', 'Larry Wall' ],
         tool => [ 'XML::Parser' ],
         keys => [ 'XML', 'Perl' ]
         abstract => ' Era outra vez, ',
         -pcdata => ' ...  ',
       },
       { title => 'Perl, the first postmodern computer language',
         author => [ 'Larry Wall' ],
         keys => [ 'linguística', 'Perl', 'linguagens de programação' ]
         abstract => 'Obviously I should have made up a scarier title...',
         -pcdata => ' ...  ',
       }
   ],
};
```

Seguidamente, junta-se uma função de processamento de cada artigo.

```
1  %handler=(
2      '-outputenc' => 'ISO-8859-1',
3      '-type' => { proceedings => MMAPON("article"),
4                   article     => MMAPON("author","tool"),
5                   keys        => "SEQ",
6                   abstract    => "ZERO",
7                 },
8      '-default'   => sub{$c},
9      'article' => sub{ processaArtigo($c) },
10 );

11 dt($filename,%handler);

12 sub processaArtigo{ my $a=shift;

13  for ( @{$a->{author}}){
14    $dict{$_}{isa}{Investigador}=1;
15    push(@{$dict{$_}{obra}{artigos}}, $a->{title});
16    push(@{$dict{$_}{obra}{ferramentas}}, @{$a->{tool}}) if $a->{tool};
17    push(@{$dict{$_}{"áreas de interesse"}}, @{$a->{keys}}); }

18  for ( @{$a->{keys}}){
19    $dict{$_}{isa}{"área"}=1;
20    push(@{$dict{$_}{artigos}}, $a->{title}); }

21  for ( @{$a->{tool}}){
22    $dict{$_}{isa}{ferramenta}=1;
23    push(@{$dict{$_}{artigos}}, $a->{title});
24    push(@{$dict{$_}{autor}},   @{$a->{author}}); }
25 }
```

Notas:

**linha 3 a 7** - Tal como no exemplo anterior, está a ser construída uma árvore generalizada com certas partes do documento, guiada pela definição de estratégia a usar para alguns dos elementos.

**linha 9** - No processador dos artigos, é invocada a função de reacção `processaArtigo` passando-se a sub-árvore correspondente a um artigo como argumento.

**linha 11** - A invocação do processamento de `filename` com o processador `handler`, vai como efeito lateral construir um dicionário `dict`.

**linha 12 a 25** - Definição da função `processaArtigo` (atrás invocada) que vai enriquecer um dicionário `dict` com base em informação de metadata do artigo.

Ao aplicar o processador ao ficheiro de proceedings XML, apresentado no início, obtém-se o seguinte dicionário `dict`:

```
'Clark Cooper' => {
    'isa' => { 'Investigador' => 1 },
    'áreas de interesse' => [ 'XML', 'Perl' ],
    'obra' => {
       'artigos' => [ 'The XML Parser' ],
       'ferramentas' => [ 'XML::Parser' ] } },
'J. Carlos Ramalho' => {
    'isa' => { 'Investigador' => 1 },
    'áreas de interesse' => [ 'XML',
                              'processamento de documentos',
                              'Perl' ],
    'obra' => {
        'artigos' => [ 'The XML Down Translator' ],
        'ferramentas' => [ 'XML::DT' ] } },
'J. João Almeida' => {
    'isa' => { 'Investigador' => 1 },
    'áreas de interesse' => [ 'XML',
                              'processamento de documentos',
                              'Perl' ],
    'obra' => {
        'artigos' => [ 'The XML Down Translator' ],
        'ferramentas' => [ 'XML::DT' ] } }
'Larry Wall' => {
    'isa' => { 'Investigador' => 1 },
    'áreas de interesse' => [ 'XML',
                              'Perl',
                              'linguística',
                              'linguagens de programação' ],
    'obra' => {
        'artigos' =>
          [ 'The XML Parser',
            'Perl, the first postmodern computer language' ],
        'ferramentas' => [ 'XML::Parser' ] } },
'XML' => {
    'artigos' => [ 'The XML Down Translator',
                   'The XML Parser' ],
    'isa' => { 'área' => 1 } },
'XML::DT' => {
    'artigos' => [ 'The XML Down Translator' ],
    'isa' => { 'ferramenta' => 1 },
    'autor' => [ 'J. João Almeida',
                 'J. Carlos Ramalho' ] },
'XML::Parser' => {
    'artigos' => [ 'The XML Parser' ],
    'isa' => { 'ferramenta' => 1 },
```

```
    'autor' => [ 'Clark Cooper',
                 'Larry Wall' ] },
'linguagens de programação' => {
    'artigos' => [ 'Perl, the first postmodern computer language' ],
    'isa' => { 'área' => 1 } },
'linguística' => {
    'artigos' => [ 'Perl, the first postmodern computer language' ],
    'isa' => { 'área' => 1 } },
'Perl' => {
    'artigos' => [ 'The XML Down Translator',
                   'The XML Parser',
                   'Perl, the first postmodern computer language' ],
    'isa' => { 'área' => 1 } },
'processamento de documentos' => {
    'artigos' => [ 'The XML Down Translator' ],
    'isa' => { 'área' => 1 } },
```

Com estes exemplos pretendemos concluir que:

> A existência de capacidade de programar (de modo compacto) processadores estruturais de documentos XML (genéricos), permite que facilmente estes possam ser vistos como dicionários, ou que possam produzir toda uma variedade de tipos de documentos/recursos que sejam posteriormente processados por outras funções.

O facto de todas as funções de processamento estarem a ser definidas em Perl, permite ainda que seja possível a utilização de um imenso poder expressivo de linguagem e de vários milhares de módulos disponíveis.

Por exemplo, basta acrescentar algumas linhas para estabelecer a ligação entre o dicionário `dict` e uma árvore B+ (Marquess, 1995; Sarathy, 1998), de modo a que este passe a ser não volátil e capaz de armazenar milhões de entradas com boa eficiência.

```
use MLDBM qw(DB_File);
use DB_File;
tie %dict, MLDBM, "dicionario.db", O_RDWR|O_CREAT, 0644, $DB_BTREE

untie %dict;
```

### Outros características do XML::DT

Embora não aparecesse nos exemplos anteriores, é possível construir vários processadores sobre o mesmo documento XML. Deste modo, uma tarefa complexa de processamento pode ser obtida através da junção de diferentes vistas, obtidas por diferentes processadores.

O XML::DT oferece uma variedade de funções para que em cada processador de elemento, se possa:

- ter acesso ao nome e atributos do elemento pai, avô etc.
- se possa alterar os atributos do elemento pai, avô etc.
- se possa definir condições acerca do contexto actual.

O XML::DT oferece também possibilidade de definição de processadores para `pcdata`.

O XML::DT tem funções (exemplo `dtpath`) em que se permite que as funções de processamento de elementos possam ser associadas a expressões XPATH.

O código deste módulo e mais detalhes acerca dele podem ser obtidos a partir da distribuição via CPAN (Comprehensive Perl Archive Network) (Koenig, 1995), ou da árvore CVS do projecto Natura.

## 6.7 MKTEXTRR – gerador de sistemas de reescrita textual

De entre os vários problemas ligados ao texto, há alguns que podem ser vistos como situações de transformação textual. A transformação textual, num conjunto importante de casos, pode ser descrita por regras, permitindo que se possa dividir o problema associado em duas partes:

- um processador genérico de regras de reescrita textual
- um conjunto de regras específicas do problema concreto a resolver,

de acordo com uma estratégia habitual em *domain specific languages*.

A ferramenta MKTEXTRR é um tradutor que aceita um (ou mais) conjuntos de regras de reescrita textual e gera uma (ou mais) função que faz a reescrita associada.

Este sistema contempla vários tipos de regras:

- regras simples,
- regras sensíveis ao contexto,
- regras condicionais,
- regras de inicialização,
- regras de terminação,
- regras envolvendo cálculo de expressões,
- regras que são executadas no final do processo de reescrita.

O MKTEXTRR aparece incluído nesta dissertação porque:

- mostra que a construção de uma pequena *domain specific language* (i.e. cujo processador é muito pequeno e simples) pode simplificar a resolução de uma série de problemas de manipulação textual: resolver um problema = definir regras de transformações;
- a ferramenta em si é útil;
- alguns dos exemplos com ele construídos constituem funções úteis para a álgebra de processamento de documentos.

### 6.7.1 Descrição do funcionamento de MKTEXTRR

Como se referiu, o MKTEXTRR recebe um conjunto de regras de reescrita, e gera uma função que aplica as regras até que nenhuma mais possa ser aplicada ou, até que uma regra de terminação explícita seja executada. É respeitada a ordem das regras (cada regra $r_i$ só é aplicada se nenhuma $r_j$ com $j < i$, tiver antecedente verdadeiro).

Qualquer conjunto destas regras pode originar um sistema de reescrita mas se não houver algum cuidado, o processo de reescrita pode não terminar.

Por cada conjunto de regras de reescrita é gerada uma função $\varphi$ com a seguinte assinatura:

$$\varphi : string \longrightarrow string$$

Torna-se portanto, possível a composição funcional de sistemas de reescrita gerados por MKTEXTRR.

Ao nível da implementação concreta, o MKTEXTRR gera uma ou mais função Perl (podendo alternativamente gerar módulos Perl).

O texto de definição das regras é um ficheiro Perl comum em que cada conjunto de regra de reescrita aparece entre uma linha `RULES` e `ENDRULES`[37]. A linha `RULES` deve conter o nome do sistema de reescrita, dando origem a uma função com esse nome.

Os vários tipos de regras usam a seguinte sintaxe concreta:

| | |
|---|---|
| padrão ==> substituição | regra simples |
| padrão =e=> expressão | regra com execução |
| padrão ==> substituição !! cond | regra simples condicional |
| padrão =e=> expressão !! cond | regra com execução condicional |
| =b=> expressão | regra de inicialização |
| padrão =last=> | regra de terminação |

### 6.7.2 Exemplos de utilização

**Exemplo 25:** divisão em sílabas

No exemplo que se segue define-se um conjunto de regras de reescrita textual para dividir um texto em sílabas,

```
$v = '[aeiouáéíóúàãõâêôäëïöü]';
$f = '[çdfgjkqtv]';
$ac = '[áéíúãõâêô]';

while(<>){print divide($_)}

RULES divide
```

[37]Esta pode ser omitida desde que apareça outra directiva RULES ou no fim de ficheiro.

```
6  ($v|[bclnprsx])($f)==>$1/$2
7  ($v)([bclmnprsx]$v)==>$1/$2
8  ([lmnrsx])([bclmnprs]$v)==>$1/$2
9  ($v|[lmnrsx])([bcp][lr]$v)==>$1/$2
10 ($v)([nlc]h)==>$1/$2

11 ([au])(i[ru])==>$1/$2
12 ([io])([ao])==>$1/$2
13 ([ie])e==>$1/e
14 ($v)($ac)==>$1/$2
```

Notas:

**linha 1 a 3** - Definição de algumas expressões abreviatura a ser usadas nas regras.

**linha 4** - Programa principal: dividir em sílabas (usando a função `divide` gerada a partir das regras) e imprimir os resultados.

**linha 5** - Define-se um sistema de reescrita chamado `divide`.

**linha 6 a 15** - Regras de reescrita que fazem a divisão em sílabas, e que vão dar origem à função `divide`. Cada regra simples, denotada por `==>`, tem como lado esquerdo uma expressão regular Perl (que define o padrão a procurar no texto de entrada), e como lado direito uma expressão que constrói o texto de substituição (a colocar na saída quando a entrada condiz com o padrão). Nas expressões de substituição utiliza-se a variável `$1`, (`$2`, ...) para referir o texto que fez *matching* com o primeiro par de parêntesis (segundo, etc).

Cada regra, vai introduzir um caracter "/"a separar sílabas.

Para compilar para Perl este ficheiro (seja ele chamado `divide.rr`) executa-se:

```
mktextrr divide.rr divide.pl
```

ficando o ficheiro `divide.pl` com uma função denominada `divide` que faz as transformações desejadas.

Este programa quando aplicado ao texto:

> *O Carnaval brasileiro é famoso pelas suas festas exuberantes. No Carnaval, há sempre muitas festas e bailes para as pessoas se divertirem.*

dá o seguinte resultado:

> *O Car/na/val bra/si/lei/ro é fa/mo/so pe/las suas fes/tas e/xu-/be/ran/tes. No Car/na/val, há sem/pre mui/tas fes/tas e bai/les pa/ra as pes/so/as se di/ver/ti/rem.*

Embora a cobertura deste divisor em sílabas não seja total, saliente-se a grande simplicidade da solução.

No próximo exemplo, será enriquecido o divisor com detecção de sílaba e vogal tónicas.

**Exemplo 26:** Divisão em sílabas com cálculo de sílabas tónicas

No exemplo que se segue enriquece-se o exemplo anterior para cálculo de acentuação.

```
1  while(<>){
2    s/(\w+)/vowelaccent(wordaccent(divide($1)))/ge;
3    print;
4  }

5  RULES divide
6  ...

7  RULES wordaccent

8  "=last=>
9  (\w*$ac)==>"$1
10 (\w*([zlr]|[iu]s?))$==>"$1
11 (\w+/\w+)$==>"$1
12 (^\w+$)==>"$1

13 RULES vowelaccent

14 :($ac)==>$1:
15 "(\w*?($v|[yw]))==>$1:
16 ([gq]u):($v|[yw])==>$1$2:
17 "==>
18 :=last=>
```

Notas:

**linha 1 a 4** - Programa principal: em cada linha, substitui-se cada palavra marcando a divisão em sílabas (usando a função `divide`), detectando a sílaba tónica (usando a função `wordaccent`) e marcando a vogal acentuada (usando a função `vowelaccent`).

**linha 8** - Regra de terminação (denotadas por `=last=>`): termina a reescrita logo que tenha sido detectada uma sílaba tónica (ou seja quando já foi inserida uma " ).

**linha 9 a 15** - Sistema de reescrita para detectar a sílaba tónica (marcando-a com o caracter " pre-fixo).

**linha 9** - Uma sílaba contendo um caracter acentuado é tónica.

**linha 10** - Uma palavra terminada em *ar, az, al, ais, ...* é aguda.

**linha 11** - Se não tiver outros indícios, a palavra é grave.

**linha 12** - Se a palavra tiver apenas uma sílaba então ela é aguda.

**linha 16 a 20** - Sistema de reescrita para detectar a vogal tónica (marcando-a com o caracter : pós-fixo; a existência do : é também usada activar a regra de terminação).

Note-se que estes três sistemas de reescrita dão origem a três funções, que podem ser compostas do modo que se pretender.

Quando aplicado ao texto anterior, obtem-se o seguinte resultado:

*O Car/na/va:l bra/si/le:i/ro é: fa/mo:/so pe:/las su:as fe:s/tas e/xu/be/ra:n/tes. No: Car/na/va:l, há: se:m/pre mu:i/tas fe:s/tas e: ba:i/les pa:/ra a:s pe/sso:/as se: di/ver/ti:/rem.*

No próximo exemplo, mostra-se um sistema de reescrita com regras para fazer a execução de expressões.

**Exemplo 27:** Regra com execução de expressões

Dado um texto, substituir as partes que contenham expressões (simples) sobre constantes inteiras, pelo seu valor.

```
\d+(\s*[+*/-]\s*\d+)+=e=> eval($&)
```

Este sistema de reescrita transforma o seguinte texto

```
o volume do tanque é 12*12*5 metros cúbicos e a sua área 12*12
```

em

```
o volume do tanque é 720 metros cúbicos e a sua área 144
```

**Exemplo 28:** tradutor muito naïf

Neste exemplo, a reescrita vai fazer uma tradução baseada num dicionário DICT e num dicionário pessoal.

Há dois sistemas de reescrita: um para fazer a tradução geral (`trad`), e outro (`posproc`) para pós-processamento final que trate de algumas concordâncias e detalhes de contracções de preposição com artigo.

```
$word='\w+';                 # regular expression for word
$m='___';                    # marc of the currente position
loaddict("DICT","pessoal"); # load dictionaries from file

RULES trad

=b=>$_ = "$m$_";
```

```
6  $m$==>
7  $m($word)\s+($word)\b==>$dict{"$1 $2"}$m!!    defined($dict{"$1 $2"})
8  $m($word)\b==>$dict{$1}$m!!                   defined($dict{$1})
9  $m($word)\b=e=> ucfirst($dict{lc($1)}) . $m !!defined($dict{lc($1)})
10 $m($word)\b=e=> procPalInd($1) . $m         !!not defined($dict{$1})
11 $m(.)==>$1$m

12 RULES posproc

13 \bde ([ao]s?)\b==>d$1
14 (o|um)#1\s+($word)=e=> artAgree($1,$2). " $2"
```

Notas:

**linha 1** - Definição de uma expressão regular para palavras.

**linha 2** - Definição de marca (`$m`). Esta marca vai sendo deslocada para a direita, à medida que a frase vai sendo traduzida.

**linha 3** - Carrega os dicionários DICT e o dicionário pessoal.

**linha 4 a 11** - sistema de reescrita para tradução geral.

**linha 5** - Regra de iniciação: é colocada uma marca no início do texto a reescrever.

**linha 6** - Quando a marca está junto ao fim da frase a reescrever, retirar a marca.

**linha 7** - Se a sequência das duas próximas palavras existe no dicionário, fazer a sua substituição e avançar a marca. Esta é uma regra do tipo condicional.

**linha 8** - Se a próxima palavra existe no dicionário, fazer a sua substituição e avançar a marca.

**linha 9** - Regra condicional com execução: se a palavra existe no dicionário após convertida para minúscula, usar como tradução a maiúscula correspondente a essa tradução.

**linha 10** - Se apesar de tudo a palavra não consta do dicionário, substituí-la por o resultado da execução da função `proPalInd` (processa palavra indefinida) que poderia por exemplo perguntar interactivamente ao utilizador qual a tradução a usar e inserir essa tradução no dicionário pessoal.

**linha 11** - Se a marca estiver antes de outro caracter (não incluído em $word$), avançar a marca.

**linha 12 a 14** - Sistema de reescrita para pós-processamento.

**linha 13** - Transforma `de o` em `do`

**linha 14** - transforma `o#1 computação` em `a computação`, ou seja invoca a função artAgree (que para simplificação não foi aqui apresentada) que usando o analisador morfológico jspell, determina a forma do artigo que concorda com a palavra que se lhe segue.

O ficheiro DICT contém o dicionário de base incluindo expressões multipalavra e algumas com marcações para facilitar pós-processamento. Exemplo:

```
driver=motorista
natural language=linguagem natural
computer graphics=computação gráfica
computer=computador
```

```
5 the=o#1
6 of=de
7 car=carro
```

Notas:

**linha 2, 3** - Exemplo de traduções multipalavra.

**linha 5** - Exemplo de tradução com marcação para posterior pós-processamento.

O resultado da execução do tradutor naïf quando aplicado a

```
the computer graphics and the world of the computer moves fast.
```

interage com o utilizador ao encontrar algumas palavras desconhecidas e produz:

```
a computação gráfica e o mundo do computador move-se rapidamente.
```

ao mesmo tempo que enriquece o dicionário pessoal com as palavras novas (permitindo que uma segunda execução do mesmo exemplo conheça já as palavras anteriores).

Sistemas de reescrita com marca deslizante (com é o caso das regras `trad` do exemplo anterior) podem ser descritas de modo mais simples com a cláusula `RULESM` (RULES + Marca)

```
RULESM trad
($word)\s+($word)==>$dict{"$1 $2"}!!     defined($dict{"$1 $2"})
($word)==>$dict{$1}!!                    defined($dict{$1})
($word)=e=> ucfirst($dict{lc($1)})!!     defined($dict{lc($1)})
($word)=e=> procPalInd($1)        !! not defined($dict{$1})
```

que acrescenta automaticamente as marcas deslizante a cada regra.

Na Secção 8.6, será descrito uma versão mais aceitável de um tradutor, baseado neste sistema de reescrita.

## 6.8 LIBRARY::* – processamento de thesaurus e catálogos

Library::* é um conjunto de módulos construídos para processar thesaurus (Library::Thesaurus), catálogos (Library::Catalog e suas especializações) e bibliotecas/arquivos – thesaurus e conjuntos de catálogos (Library::Simple).

O thesaurus[38] constitui a estrutura classificativa que vai permitir *arrumar* um conjunto de documentos (cuja meta-informação é descrita nos catálogos), permitindo que se possa construir navegação e pesquisa conceptuais sobre o acervo, bem como algum tipo de inferência.

O thesaurus, ao estabelecer uma rede conceptual entre os termos classificativos que o constituem, é por si só uma ferramenta poderosa para a construção e manuseamento de léxicos estruturados, podendo ser usado como ferramenta lexicográfica/terminológica (independentemente de catálogos e de bibliotecas digitais).

Esta secção está incluída nesta dissertação porque:

1. Os thesauri são casos particulares de dicionários (e portanto estão no centro da área de interesse) – havendo necessidade de os estudar, modelar e processar.
2. Os módulos Library::* são ferramentas capazes de dicionarizar uma grande variedade de thesauri, catálogos e bibliotecas digitais.
3. A tecnologia associada a esta ferramenta foi usada para a construção gráfica da dissertação, nomeadamente do Apêndice A.

### 6.8.1 Bibliotecas digitais: introdução

Dum modo simplificado, uma biblioteca digital integra e anima um sistema classificativo, um conjunto de catálogos e um conjunto de documentos catalogados.

Incluídas nessa *animação* estão as operações de construção, enriquecimento, pesquisa, navegação conceptual, visão temporal e visão evolutiva.

Os documentos podem ser considerados internos ou externos ao sistema documental.

[38] Thesaurus neste contexto, está a ser tomado numa versão abrangente, abarcando uma parte significativa das redes-semânticas, das ontologias e dos topic-maps

$$
\begin{array}{lcrcl}
DigLib & = & t & : & thesaurus \quad \times \\
 & & cs & : & set(catalogo) \quad \times \\
 & & a & : & set(doc) \\
catalogo & = & f & : & file \quad \times \\
 & & ty & : & catalType \\
thesaurus & = & \multicolumn{3}{l}{\text{...estrutura classificativa}}
\end{array}
$$

### 6.8.2 Thesaurus: introdução

Dum modo simplificado, um thesaurus define um conjunto de termos e relações entre eles (um multigrafo pesado).

Normalmente, os thesauri são usados para definir conjuntos de termos a usar na classificação (ISO 2788, 1986; ISO 5964, 1985).

$$
\begin{array}{lcrcl}
thesaurus & = & rede & : & semNet \quad \times \\
 & & prop & : & Tprop \\
semNet & = & termos & : & set(termo) \quad \times \\
 & & ramos & : & set(ramo) \\
ramo & = & ori & : & termo \quad \times \\
 & & r & : & Rel \quad \times \\
 & & dest & : & termo
\end{array}
$$

Thesaurus é uma rede de termos (semNet) interligados por relações (Rel), sobre as quais conhecemos certas propriedades. Às relações estão associadas certas propriedades matemáticas (descrita em Tprop) que permitem inferência e validação de consistência.

Por uma questão de maior simplicidade, o modelo anterior está a considerar as relações homogéneas. No entanto, há vantagens em poder usar relações que não sejam homogéneas, como se verá abaixo.

Exemplo de algumas relações mais comuns (de acordo com o standard ISO):

- BT - broader term – termo genérico (antissimétrica, antireflexiva, transitiva) (inversa de NT). Ex: *primatas* BT *mamíferos* significa que mamíferos é uma classe mais geral que primatas.
- NT - narrower term – termo específico (antissimétrica, antireflexiva, transitiva) (inversa de BT)
- SN - scope note – nota explicativa. Ex: *canto* SN *No sentido desportivo – livre de canto.* Tipicamente as notas explicativas são frases usadas para clarificar o sentido e o âmbito do termo usado.
- USE – sinónimo preferencial, termo remissivo (inversa de USES). Ex: *equídeo* USE *cavalo.* Esta relação tem grande importância em Ciências

Documentais já que permite ligar termos não usados como classificadores a termos activos, levando a que um utilizador que consulte o sistema classificativo *entre* mais rapidamente na rede de termos activos.
– USES – sinónimo não preferencial (inverso de USE)
– RT - related term – termo associado (que neste texto se vai considerar simétrica).

À parte a relação SN, as outras relações referidas são homogéneas – i.e., são de domínio e contradomínio no conjunto dos `Termos`. O contradomínio de `SN` é textual (relação heterogénea).

Para facilitar as descrições e a visualização de cada conceito, quase todas as relações têm inversa.

Considere-se o seguinte modelo de Tprop que inclui apenas a capacidade de descrever relações inversas (i.e. permite associar uma relação à sua inversa).

$$Tprop \quad = \quad inversa \quad : \quad Rel \rightharpoonup Rel$$

A função de completação que seguidamente se apresenta, completa o thesaurus com os ramos novos (`n`) das relações inversas que ainda não estejam na rede.

$$
\begin{array}{l}
completacao : thesaurus \longrightarrow thesaurus \\
completacao(t) \stackrel{\text{def}}{=} \\
\quad \underline{let} \quad <rede, pr> = t \\
\qquad\quad <tes, ramos> = rede \\
\qquad\quad n = \{ramo(dest(x), pr(r(x)), ori(x)) \mid x \in ramos \wedge r(x) \in dom(pr)\} \\
\qquad\quad rn = ramos \cup n \\
\quad \underline{in} \quad thesaurus(semNet(tes, rn), pr)
\end{array}
$$

**Interface**

O módulo Library::Thesaurus oferece uma grande variedade de funções para manipular thesauri. Por exemplo:

– Funções ligadas ao carregamento e não volatilidade de um thesaurus:

$$
\begin{array}{l}
thesaurusLoad : ISOtheTxt \longrightarrow thesaurus \\
thesaurusLoad(file) \stackrel{\text{def}}{=} \\
\qquad \text{carrega um thesaurus a partir de um ficheiro ISO 5964}
\end{array}
$$

$$
\begin{array}{l}
save : thesaurus \times tabela \longrightarrow ISOtheTxt \\
save(t, detalhes) \stackrel{\text{def}}{=} \text{ constrói um texto ISO 5964 a partir do thesaurus}
\end{array}
$$

$storeOn : thesaurus \longrightarrow Storable$
$storeOn(t) \stackrel{\text{def}}{=}$ Guarda um thesaurus em ficheiro Storable

$retrive : Storable \longrightarrow thesaurus$
$retrive(t) \stackrel{\text{def}}{=}$
reconstitui um thesaurus previamente guardado com storeOn

– Funções ligadas à construção de vistas de vários formatos sobre o thesaurus:

$toXml : thesaurus \times tabela \longrightarrow XMLtxt$
$toXml(t, detalhes) \stackrel{\text{def}}{=}$ constrói um texto XML a partir do thesaurus

$toTex : thesaurus \times tabela \longrightarrow XMLtxt$
$toTex(t, detalhes) \stackrel{\text{def}}{=}$ constrói um texto LaTeX a partir do thesaurus

$navigate : thesaurus \longrightarrow HTML$
$navigate(t) \stackrel{\text{def}}{=}$ Comporta-se como um CGI de navegação no thesaurus

– funções para construção/modificação do thesaurus, orientadas ao termo,
– funções referente à gestão de thesaurus multilingue,
– funções para melhorar o feedback obtido nas várias vistas. Por omissão, há valores implícitos para gerar as várias vistas sobre a informação contida no thesaurus. No entanto, para conseguir os melhores resultados é útil poder (opcionalmente) definir/ter controlo sobre uma série de informação adicional. Exemplo:
  • definir a ordem pela qual as relações ligadas a um termo serão mostradas,
  • definir o texto de identificação ligado a cada relação (BT – Termo genérico),
– funções ligadas a processamento estrutural do thesaurus:
  • `downtr` (usada na escrita de várias outras, tal como `toTex, toXml`)
  • `depth-first` (dado um termo inicial, um conjunto de relações e uma profundidade, dá uma árvore de travessia)
  • `transitive-closure` (calcula o fecho transitivo, dado um termo e um conjunto de relações).

### 6.8.3 Exemplo: Thesaurus das ferramentas da dissertação

Depois de apresentados os conceitos básicos ligados a thesauri e de resumido o interface do módulo Library::Thesaurus, serão referidos como exemplo alguns detalhes ligados à criação/geração automática do Apêndice A – dicionário das ferramentas referidas nesta dissertação.

O Apêndice A, está a ser gerado automaticamente através de:

- dispor de um thesaurus de base (onde estão definidas as novas relações e suas propriedades, bem como alguns termos de ligação utilizados)
- incluir ao longo do texto entradas de thesaurus com marcação `addthe`, de modo a poderem ser distinguidas do restante texto.
- incluir na makefile uma operação `buildthesaurus` que extrai as entradas do texto da dissertação, concatenando-a com a base; seguidamente é feita a importação e completação; por último, é feita a geração de um texto LaTeX (que é incluído directamente no apêndice) e de outro HTML (para navegação pessoal).

$$\begin{array}{l} buildthesaurus : latex \times The \longrightarrow latex \\ buildthesaurus(diss, base) \stackrel{\text{def}}{=} \\ \quad \underline{let} \quad t = extraimarcacoes(diss) \\ \qquad\quad\; t2 = base \frown t \\ \quad \underline{in} \quad toTex(thesaurusLoad(t2)) \end{array}$$

Seguidamente serão apresentados exemplos contendo:

- a descrição do conjunto de relações específicas que se sentiu necessidade de criar para o Thesaurus das Ferramentas (ver exemplo 29),
- alguns extractos do thesaurus ligados à criação dessas novas relações (ver exemplo 30),
- o código Perl dum processador do thesaurus capaz de gerar HTML navegável por termos.

Apesar da óbvia utilidade das relações pré-definidas, referidas no standard ISO (apresentadas atrás), considerou-se crucial poder definir novas relações.

**Exemplo 29:** Descrição de novas relações

Além das relações descritas anteriormente, no Thesaurus das Ferramentas, estão a ser usadas as outras relações que seguidamente se resumem:

- IOF - instance of – instância de. Ex: *mktextrr* IOF *domain specific language*.
- INST - instances – instâncias. Relação inversa de IOF.
- ABOUT – acerca de.
- GENERATES – gera. Ex: *mktextrr* GENERATES *Perl*. Esta relação estabelece uma ligação entre um programa gerador e o tipo gerado.

- MADEBY – gerado por. Relação inversa de GENERATES.
- HAS – contém. Ex: *Library::** HAS *Library::Thesaurus*.
- POF - part of – parte constituinte de. Relação inversa de HAS.
- NEEDS – necessita de. Ex: *Lingua::PT::speaker* NEEDS *mbrola* significa que para poder usar o módulo Lingua::PT::speaker, precisamos de ter instalado o programa *mbrola*.
- USEDIN – usado em. Relação inversa de NEEDS.

A definição de novas relações pode ser feita de dois modos:

- usando primitivas especiais que estendem o standard ISO,
- através de relacionar as novas relações entre si e relacioná-las com alguns termos especiais.

**Exemplo 30:** Definição de novas relações

Com o presente exemplo pretende-se mostrar que:

- É possível definir novas relações de uma maneira simples (usando os mesmos mecanismos e a mesma sintaxe que o thesaurus).
- A definição de novas relações envolve:
  - Indicar que esse termo é uma relação (Ex: `BT _relation_`).
  - definição de informação ligada à sua apresentação (Ex: com SN).
  - definição de domínio e contradomínio (relacionando a relação com tipos, usando para isso as relações RANG e DOM) (por omissão o domínio e contradomínio são Termos).
  - definição de propriedades matemáticas (associando à relação essas propriedades. Ex: ser simétrica).
  - definição de relação com outras relações (Ex: a sua inversa).

```
_relation_
NT about, author, section, RT, POF, HAS, about, needs, usedIn,
  citedIn, IOF, INST, makes, by, USE, USES, SN, generates, MadeBy

_order_
NT SN, USE, section, ref, NT, IOF, POF, about, needs, usedIn,
  citedIn, HAS, INST, makes, by, USES, about, author, RT

RT
SN termo associado
BT _symmetric_
```

```
10 section
11 SN Secção
12 RANG  latex

13 generates
14 SN Gera
15 INV MadeBy

16 MadeBy
17 SN Gerado por

18 about
19 SN acerca de
20 INV citedIn
```

Notas:

**linha 1 a 3** - Indicar que um termo é específico (NT) de `_relation_`, serve para definir novas relações.

**linha 4 a 6** - `_order_` define uma ordem de processamento entre as diversas relações a aplicar em operações que processem os termos.

**linha 7 a 9** - Detalhes acerca da relação RT. Na linha 8, define-se uma frase a usar em operações de visualização da relação. Na linha 9, como a relação RT é descrita como sendo um caso particular de `_symmetric_`, essa propriedade é usada automaticamente para completação, quando o thesaurus é carregado.

**linha 10 a 12** - Definição da relação section.

**linha 12** - Por omissão as relações têm como domínio *Termos* e contradomínio *Termos*. Ao afirmar-se que o contradomínio de SECTION é LATEX, está-se a indicar que é uma relação não homogénea e externa. Deste modo, as várias secções não vão ser consideradas como termos activos do thesaurus.

**linha 13 a 15** - A definição de relações inversas (INV) permite que haja completação automática do thesaurus. A partir do momento em que se sabe que GENERATES tem MADEBY como relação inversa, qualquer ramo *A generates B* dá origem automaticamente a um outro *B MadeBy A*.

Por uma questão de simplificação, optou-se aqui por não abordar os detalhes ligados a thesauri multilingue.

Depois de definido um thesaurus, surge naturalmente a necessidade do seu processamento.

A função utilizada, downtr, tem uma razão de ser e um design muito semelhante à função `dt` descrita na Secção 6.6:

– definir processadores seguindo a estrutura formal do objecto em causa[39].

[39] Baseado em catamorfismo

Ou seja, de um modo simplificado, e considerando apenas a parte homogénea dos seus ramos, podemos ver um thesaurus como:

$$\begin{array}{lcl} thesaurus & = & termo \rightharpoonup Tinf \\ Tinf & = & Rel \rightharpoonup set(termo) \end{array}$$

A função `downtr` poderá fazer processamento estrutural sobre um thesaurus do seguinte modo[40]:

$$\begin{array}{l} downtr : thesaurus \times processador \longrightarrow any \\ downtr(x, p) \stackrel{\text{def}}{=} \\ \quad \underline{let} \quad pe = p(\texttt{"-end"}) \; or \; id \\ \qquad\quad\;\; pt = p(\texttt{"-eachTerm"}) \; or \; id \\ \qquad\quad\;\; tseq = <pt(proctermo(t, x(t), p)) \mid t \in dom(x)> \\ \quad \underline{in} \quad pe(strcat - red(\texttt{""}, tseq)) \end{array}$$

$$\begin{array}{l} proctermo : termo \times Tinf \times processador \longrightarrow any \\ proctermo(t, i, p) \stackrel{\text{def}}{=} \\ \quad strcat - red(\texttt{""}, <\underline{let} \quad pr = p(r) \; or \; p(\texttt{"-default"}) \; or \; id \quad \mid r \in dom(i)>) \\ \qquad\qquad\qquad\qquad\;\; \underline{in} \quad pr(t, r, i(r)) \end{array}$$

**Exemplo 31:** Processamento de um Thesaurus: geração de HTML

Neste exemplo, apresenta-se um processador de thesauri capaz de gerar uma página HTML com o conteúdo de um thesaurus, fazendo com que a definição de cada termo contenha uma âncora, de modo a permitir que cada termo existente num lado direito de uma relação fique ligado ao sítio onde o termo está definido.

Embora não seja muito relevante o código concreto do exemplo Perl que se segue, optou-se por o apresentar para mostrar que a sua descrição é feita com poucas linhas de código.

```
use Library::Thesaurus;
use CGI qw(:all);

my $t = thesaurusLoad("mythesaurus.the");

print $t->downtr({
  -end      => sub { h1("Thesaurus - all in one").dl($_)},
  -eachTerm => sub { dt(a({name=>"$term"},$term)). dd(dl($_))},
  -default  => sub {
     dt($t->describe($rel))."\n".
```

[40]A concatenação iterada que aqui se apresenta, pode naturalmente ser generalizada.

```
9       join("\n", (map {dd(a({href=>"#$_"},$_))} sort @terms))},

10    -order    => ["EN","FR","SP"],
11    URL       => sub { dt($t->describe($rel)).
12       join("\n", (map {dd(a({href=>"$_"},$_))} @terms))},
13  });
```

Notas:

**linha 2** - O módulo Perl CGI, facilita a construção de HTML fornecendo funções para construir as diversas etiquetas/atributos. Ex. as funções `h1, dl, dt, dd, a`, constroem o HTML correspondente.

**linha 3** - Importar o thesaurus.

**linha 5** - Gerar um título e incluir o resultado do processamento de todos os termos numa description list (dl).

**linha 6** - Para cada termo, criar uma entrada dt na dl anterior e criar uma nova description list para a sua informação interna.

**linha 7** - Processamento de cada relação em que o termo aparece: determinar qual a sua string de descrição (describe) e construir uma lista de ligações para cada termo que aparece no lado direito dessa relação.

**linha 10** - Definição da ordem de processamento das relações (Começar pelas relações ligadas às diferentes línguas, seguindo-se as outras (por qualquer ordem)).

**linha 11, 12** - Tratamento específico da relação heterogénea URL.

### 6.8.4 Library::Catalog

Como foi referido anteriormente, um catálogo agrupa meta-informação acerca de um conjunto de documentos. Esta meta-informação inclui normalmente todo um conjunto de dados de catalogação (Ex: o autor, título e dimensões de um livro), inclui informação classificativa – ligada ao conteúdo do documento (Ex: conjunto de palavras chave), podendo ainda conter descrições ou resumos do documento (Ex: o *abstract* de um artigo).

Existe uma enorme variedade de formatos ligados a catálogos. Dependente do objectivo do catálogo, há formatos que seguem normas ou directivas específicas (Ex. Regras Portuguesas de Catalogação, Dublin Core (Weibel, Godby, and Miller, 1995; Weibel, 2003)), outros que estão ligadas a ferramentas particulares (Ex: BiBTEX).

O módulo de gestão de catálogos, Library::Catalog, foi construído para criar uma vista comum sobre uma (potencialmente grande) variedade de formatos de catálogos.

O módulo Library::Catalog vê a definição de um tipo de catálogo através da definição das seguintes funções:

$$
\begin{array}{lcrcll}
catalType & = & asList & : & (file \longrightarrow entry^{\star}) & \times \\
 & & asHtml & : & (entry \longrightarrow \text{title * url * HTML }) & \times \\
 & & asText & : & (entry \longrightarrow text) & \times \\
 & & asRelations & : & (entry \longrightarrow set(rt)) & \times \\
 & & asIdentifier & : & (entry \longrightarrow ident) & \\
rt & = & Rel \times termo & & &
\end{array}
$$

em que:

- *asList* - é uma função que faz a extracção de entradas de um ficheiro
- *asHtml* - dada uma entrada do catálogo determina o seu título, url e converte para HTML
- *asText* - dada uma entrada do catálogo, calcula uma versão textual da informação associada a essa entrada.
- *asRelations* - dada uma entrada do catálogo, extrai os pares relação - termo nela contidos.
- *asIdentifier* - determina um identificador do documento de modo a permitir determinar alterações ao catálogo (i.e., determinar que novos documentos apareceram, que documentos foram alterados, etc.).

Associado a este módulo, há um conjunto módulos que lidam com alguns formatos de catálogo pré-definidos (sobre os quais se escreveu as funções de adaptação de acordo com o interface descrito). Exemplo:

- `Library::Catalog::bibtex` – que usa ficheiros `BibTeX` (aceitando como extensões os campos url – usado para estabelecer ligação com o documento, caso exista uma versão deste na rede, abstract e keywords – usadas para classificação)
- `Library::Catalog::Simple` – que usa catálogos que seguem um formato muito simples, desenhado especificamente para este módulo. Este módulo fornece algumas funções de interface WEB para construção/edição.
- `Library::Catalog::XML` – feito para funcionar como adaptador de um qualquer catálogo XML.

Para exemplos reais de catálogos múltiplos e usando diferentes formatos, ver a secção referente ao Alfarrábio (subsecção 8.4.4).

### 6.8.5 Library::Simple

Como foi referido tanto os catálogos, como os thesauri, como os próprios documentos são constituintes de uma entidade mais rica – a biblioteca digital.

O módulo `Library::Simple` (usando o Library::Thesaurus e o Library::Catalog) foi construído para processar bibliotecas digitais como um todo, permitindo:

– *construir* uma biblioteca digital, i.e., gerar um conjunto de estruturas auxiliares de modo a tornar mais eficiente um conjunto de operações de pesquisa e navegação

$$mkdiglib : thesaurus \times set(catalogo) \times config \longrightarrow DibLib$$
$$mkdiglib(t, cs, c) \stackrel{\text{def}}{=}$$
Cria vários índices para navegação e pesquisa sobre a bib. dig.

– gerar automaticamente navegação conceptual / pesquisa conceptual em HTML através da função `navigate` (ver Figura 8.2, na Secção 8.4).
– fazer vários tipos de pesquisas na biblioteca digital (dadas expressões sobre termos do thesaurus e padrões textuais)
– construir textos LaTeX ou HTML guiados por *templates* ou por thesaurus.
– extrair subthesaurus contidos em catálogos que estejam ligados a termos declarados como Meta no thesaurus.
– determinar quais os documentos novos, quais os documentos editados, ou seja, quais as novidades neste catálogo (eventualmente em formato `Library::catalog::News`)

Algumas das pesquisas têm comportamento idêntico ao de dicionários DDMF: ou seja, este módulo permite ver uma biblioteca digital como um dicionário.

O código deste módulo e mais detalhes acerca dele podem ser obtidos a partir da distribuição via CPAN (Comprehensive Perl Archive Network) (Koenig, 1995), ou da árvore CVS do projecto Natura (`http://natura.di.uminho.pt`).

Os módulos LIBRARY::* estão presentemente a ser usados numa grande variedade de projectos como seja:

– Projecto Natura
– Projecto Alfarrábio
– Arquivos sonoros de Ernesto Veiga de Oliveira
– Museu da Pessoa
– Alfa-Braga
– Arquivo de Etnomusicologia.

# Capítulo 7

# Funções: ferramentas ligadas a dicionários

Na continuação do capítulo anterior, apresenta-se mais um conjunto de ferramentas, desta vez directamente ligadas a dicionários.

Na Secção 7.1 descreve-se o DPL – uma linguagem de programação para criação de dicionários, e na Secção 7.2 apresenta-se um sistema – DictTEX para produção de dicionários em papel.

## 7.1 DPL– um compilador para programação/construção de dicionários

Em secções anteriores, falou-se de várias ferramentas ligadas à extracção, junção e transformação de recursos para criação de dicionários.

Nesta secção, é analisada uma ferramenta de criação e manutenção de dicionários que pode ser usada de modo independente e autónomo. DPL – Dictionary Programming Language – é o nome da linguagem de programação criada e do respectivo compilador desenvolvido.

Como foi referido, na definição (clássica) das entradas de dicionários há uma enorme quantidade de trabalho que é repetitivo. Esta repetição traduz-se num enorme esforço e numa grande dificuldade em se ser coerente. Muita dessa coerência consiste em representar de forma idêntica fenómenos comuns.

Com a ferramenta DPL, pretende-se dar suporte à definição de dicionários usando uma linguagem de definição (construída especialmente para o efeito) que possibilita:

- definição de entradas sem preocupação com o aspecto tipográfico;
- definição separada de formatos de saída (tipográficos ou não);
- definições orientadas ao conceito (em vez de orientado à palavra);
- definição de funções (que armazenem partes repetitivas a usar por várias entradas) ligadas a tornar explícito as propriedades metalinguísticas;
- definição de metadata rica ligada ao dicionário;
- definição de algumas propriedades de consistência;
- definição de expressões multi-termo com possibilidade de expressar variantes e alguns detalhes flexionais.

No desenho da linguagem DPL, foi seguida uma orientação geral baseada em estruturas (tipadas) de facetas (EF). A linguagem DPL inclui definição de funções em EF (para conceitos metalinguísticos, abreviaturas, etc.) e expressões em EF (para definição dos conceitos e dos termos do dicionário). O núcleo da DPL é uma calculadora de EF equipada com sintaxe concreta para alguns fenómenos habituais ligados a dicionários.

### 7.1.1 História do desenvolvimento de DPL

A necessidade de uma linguagem de programação de dicionários surgiu de um hobby: a colecção de expressões idiomáticas e de calão, que será descrita na Secção 8.2.

Após se ter disponibilizado na Internet a nossa colecção particular de termos idiomáticos e de calão, houve um elevado número de contribuições e surgiu uma grande dificuldade de controlar o processo de consulta e inserção de informação nova com um dispêndio mínimo de tempo.

Muito cedo, tornou-se evidente a utilidade de dispor de um compilador que aceitasse uma descrição compacta e eficaz das entradas.

Há várias maneiras de simplificar a inserção e definição de novos termos. Um dos meios mais eficazes de definir conceitos é por conjuntos de sinónimos (Miller et al., 1993; Fellbaum, 1998; Vossen, 1999, WordNet).

A necessidade de funções (com ou sem parâmetros) aparece naturalmente como mecanismo de abreviaturas de *pôr em evidência* estruturas e construções análogas.

A necessidade de uma estrutura formal foi já descrita, bem como o uso de modelos da família das estrutura de facetas.

### 7.1.2 Descrição sucinta da linguagem DPL

De um modo simplificado, a gramática da linguagem DPL é a seguinte:

**Gramática da DPL**

```
dpl        ->  (meta | entries | inlinecode)*
entries    ->  "<dict>" dict "</dict>"
inlinecode ->  "<perl>" perlcode "</perl>"
dict       ->  ( (concept | funCall | efDist ) ";" )*
concept    ->  ef ("|" ef)* ":" ef
funCall    -> "+" ID                      # invocação de função
            | "+" ID "(" ef (";" ef)* ")"

# operador distributivo de ef
efDist     -> "begin" ef dict "end"

ef         -> "{" pair-seq "}"       # extensão
            | ef "+" ef              # união
            | funCall                # invocação de função
            | STRING                 # exemplo
            | STRING "=" term        # ex com paráfrase
            | term                   # termo
            | ef "<" term            # relação isa
            | "[" ef (";" ef)* "]"  # sequência

pair-seq   -> ID "=" ef (";" ID "=" ef)*

meta       ->  "<meta>" metaDict "</meta>"
metaDict   -> types title authors introduction
                termSkell copyright sources ...
```

Essencialmente a linguagem DPL descreve três tipos de coisas:

– a meta informação acerca do dicionário;
– as funções para definição de conceitos metalinguísticos e abreviaturas;
– os conceitos (que darão origem a entrada no dicionário).

**Definição de termos**

Conforme já referido nesta dissertação, a informação acerca de um termo é descrita através de uma estrutura de facetas.

Por exemplo, a expressão *chover a cântaros* pode ser descrita pela seguinte estrutura de facetas, já descrita em DPL:

```
{ name = chover* a cântaros;
  isa = expressão idiomática;
  nivel = coloquial;
  traducao = [
    { lingua = Inglês;
      name = to rain cats and dogs};
    { lingua = Francês;
      name = ... } ];
  sem = chover abundantemente;
  syn = chover* canivetes
}
```

Como se disse, os programas DPL são compostos por definição de funções e por expressões do tipo sequência de estrutura de facetas. Para manusear estrutura de facetas, DPL implementa basicamente os seguintes operadores:

– união $(ef_1 + ef_2)$;
– extensão $(at = val; ...)$
– sequências $([ef_1; ef_2; ...])$

bem como um conjunto de abreviaturas ligadas a atributos habituais em dicionários:

– relação *isa* – para definir a classe à qual o elemento pertence (`ef1 < ef2` é uma abreviatura de `ef1 + { isa = ef2}`)
– relação (`"..."`) – usado para frases exemplo (`"exemplo"` é uma abreviatura de `{ ex = exemplo}`)
– frases exemplo com paráfrase explicativa (`"...."=...`) (`"exemplo"=parafrase do exemplo` é uma abreviatura de `{ ex = exemplo, paraf = parafrase do exemplo}`)
– definição semântica $(ef : semantica)$ (`ef:semantica` é uma abreviatura de `ef + {sem = semantica}`)
– lista de sinónimos $(ef_1|ef_2)$

Na Secção 8.2, serão apresentados vários exemplos de definição de termos e conceitos.

### Meta informação

É universalmente aceite que a definição de metadata é importante em dicionários tradicionais. Em DPL, essa definição é mais rica do que o caso tradicional e dela vão depender uma série de operações de processamento automático subsequentes.

O bloco de metadata vai incluir informação declarativa que será usada pelos diversos processadores quando for necessária/útil. O bloco metadata inclui vários capítulos, sendo quase todos opcionais:

1. informação tradicionalmente existente em dicionários:
   - título, autores, contactos
   - texto introdutório
   - agradecimentos e lista de contribuições
   - bibliografia
2. informação mais formal que pode modificar a semântica do dicionário e dos processadores associados:
   - definição de tipos associados aos vários atributos
   - estrutura abstracta de uma entrada típica
   - restrições de validade acerca de certos atributos

A informação contida nos atributos do tipo 1 é de utilidade óbvia, e é usada pelos vários processadores para criarem documentos finais completos. Por exemplo, uma CGI de pesquisa precisa de usar directamente os títulos, contactos, textos introdutórios, etc., para que a página HTML gerada seja completa.

A informação contida em atributos do tipo 2, necessita de uma explicação mais detalhada.

### Uso de tipos

Quando se associa um tipo a um atributo, essa informação pode ter várias utilidades. Por exemplo, um processador que gere HTML precisa de saber:

- quais os atributos do tipo URL para os converter em ligações;
- quais dos atributos são termos contidos no próprio dicionário (para estabelecer as ligações convenientes);
- quais os atributos que são repetitivos (de modo a criar formulários que permitam aos visitantes contribuir com novos sinónimos, etc.)

De modo semelhante, o facto de se indicar que um tipo é repetitivo altera o comportamento da função interna de união de estrutura de facetas.

Suponhamos que o atributo $a$ tem tipo declarado $t*$; então a união de duas estrutura de facetas contendo esse atributo passa a ser:

$$\{a = v_1\} + \{a = v_2\} = \{a = [v1, v2]\}$$

Por omissão (i.e., para atributos sem tipo declarado) a união está a ser a reescrita (plus), ou seja, quando há identificadores de atributo idênticos, o valor à direita reescreve o valor à esquerda.

Os tipos podem também ser usados pelas funções de validações. Na presente versão, os tipos *url*, *sentence*, *termInDict* e *sampa*[1] estão a ser usados em algumas ferramentas.

#### Uso de estrutura abstracta de termos

A estrutura abstracta de termo constitui uma espécie de esqueleto e indica a estrutura típica do termo. É utilizada para documentação e para geração de mecanismos de introdução de novos termos. Na Subsecção 8.2.6 é apresentado um exemplo de uso de esqueletos de termo.

### 7.1.3 Compilador DPL: descrição técnica sucinta

O processador desenvolvido para interpretar a linguagem DPL e criar um dicionário, é constituído por um analisador genérico DPL (um parser e analisador semântico) e um módulo pragmático contendo uma função *processa* que descreve a acção a realizar com cada definição de termo encontrada. Para cada compilador DPL que seja necessário construir, haverá que definir um módulo pragmático (ver Figura 7.1).

Internamente o analisador sintáctico DPL é gerado por uma variante do yacc para Perl chamado `Parse::Yapp` (Desarmenien, 2000). O `yapp` é um gerador de parsers LALR que partindo de uma gramática independente do contexto contendo acções semânticas em Perl associadas a cada produção, gera um módulo Perl (package).

Apesar da existência de geradores de parsers mais sofisticados (exemplo AntLR (Parr, 1998), LRC (Kuiper and Saraiva, 1998; Saraiva, 2000)), a utilização de `yapp` e Perl mostrou ser uma decisão acertada, devido a conciliar a eficiência do parser LALR com o poder expressivo que o Perl oferece, nomeadamente :

– no manuseamento de arrays associativos e listas,
– no interface fácil a módulos de bases de dados (DB_File) e a CGI
– no facto de ser uma linguagem reflexiva (isto é ter a capacidade de se mudar a si própria em *runtime*, por exemplo com a função `eval`)

[1]SAMPA - computer coding of the international phonetic alphabet (SAMPA, 1999).

permitindo obter código *inline* grátis e facilitando a definição de funções com enorme poder expressivo.

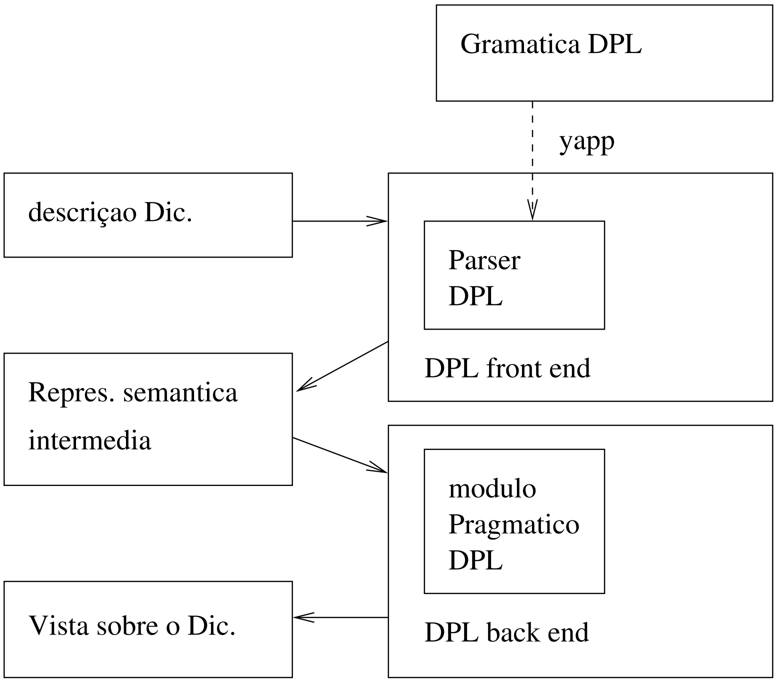



Figura 7.1: estrutura DPL

O Parser e analisador semântico de DPL interpretam a descrição de um dicionário e para cada estrutura (estrutura de facetas) calculada invocam uma função de reacção (semântica dinâmica ou pragmática) externa que faz o seu processamento, permitindo que possa haver várias ferramentas que partilhem o mesmo parser e analisador semântico.

As validações ao nível da análise semântica podem ser imediatas ou adiadas, permitindo que incompletudes ao nível da definição de certos campos de informação possam dar origem a mensagens de erro ou a reacções (pragmáticas) posteriores, como por exemplo, pedir o que falta quando a informação for consultada.

### Estado actual de desenvolvimento

Presentemente, existe disponível uma versão protótipo que tem sido usada para construir alguns dicionários, nomeadamente o dicionário aberto de calão e expressões idiomáticas (DAC) (ver Secção 8.2). Para além da construção de um dicionário interno (correspondente à semântica intermédia), várias outras saídas estão a ser produzidas automaticamente:

- ficheiros txt;
- ficheiros HTML;
- saídas LaTeX (e portanto PDF, PostScript) (ver Fig. 7.3);
- TEI-XML (versão inicial, em estado embrionário);
- esqueletos de CGIs que implementam consulta remota e inserção de novos termos.

Na Figura 7.2, apresenta-se o conjunto de termos multipalavra correspondentes à pesquisa de *puto* no DAC.

Saliente-se que cada resposta é a visualização da EF a ela ligada. Cada atributo do tipo termo contido no próprio dicionário, é uma ligação à própria CGI com a consulta respectiva.

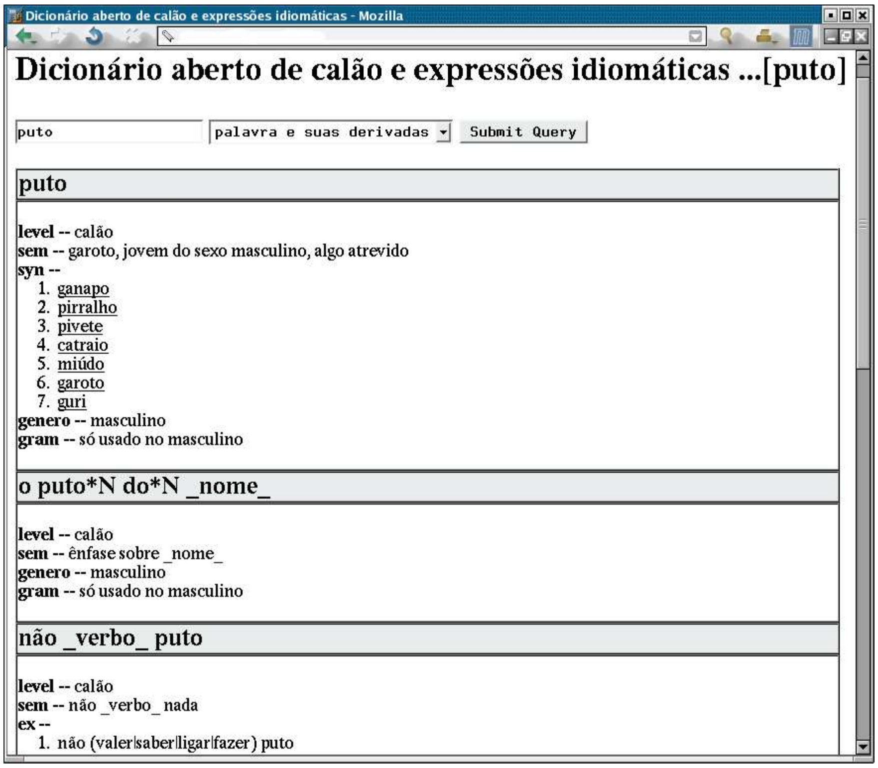



Figura 7.2: Página Web da CGI do dicionário aberto de calão

7 banana

**atilho**
*ex* tem um nagalho para atar este saco?
fio
*sin.:* nagalho; atilho

**atirar-lhe com**
bater em (física ou psicologicamente)
*sin.:* afinfar-lhe; atirar-lhe com

**atirar-se de cabeça**
*ex* O gajo atirou-se à garina de cabeça
Envolver-se numa nova situação sem hesitações

**atrapalhar**
*ex* fulano ficou entalado com a pergunta
*sin.:* entalar; atrapalhar

**autocolante**
*ex* és um autocolante do caralho
diz-se de um individuo incomodativo
*nivel* coloquial
*sin.:* chato; aborrecido; incomodativo; melga; gosma; cromo; postal; autocolante

**azeite**
aquele que põe à venda os préstimos sexuais de mulheres em proveito próprio
*nivel* calão
*sin.:* chulo; azeiteiro; azeite

**azeiteiro**
*nivel* calão
*sin.:* azeiteiro; parolo; chunga; chulo; pimba
*isa* insulto
-
aquele que põe à venda os préstimos sexuais de mulheres em proveito próprio
*nivel* calão
*sin.:* chulo; azeiteiro; azeite

**azelha**
diz-se de alguém com pouca destreza e pouco eficiência
*sin.:* trengo; pacóvio; azelha

# B

**babar*-se por**

**bacana**
*ex*
*frase* que idea bacana, meu
*equiv* que idea boa
bom, agradável
*nivel* coloquial
*sin.:* fixe; bestial; porreiro; bacana

**bacano**
usa-se como um modo, ligeiramente irreverente, de referir uma pessoa
*nivel* coloquial
*sin.:* gajo; tipo; meco; caramelo; indivíduo; bacano; melro

**bacoco**
*ex* aquele individuo é um bacoco
*isa* insulto
*sin.:* bacoco; tosco; idiota

**badalhoco**
pessoa ou actividade pouco limpa
*sin.:* badalhoco; porco

**baixar* a bola**

**balda**
situação em que reina o desinteresse e a irresponsabilidade
*nivel* calão
*sin.:* bandalheira; balda

**balelas**
*sin.:* conversa de chacha; conversa fiada; tretas; balelas; chuchadeira

**bamby**
*local* Brasil

**banana**
órgão sexual masculino
*nivel* calão
*sin.:* (pénis/pênis); pila; piça; caralho; cacete; pau; pinto; ponteiro; porra; drejo; bregalho; vergalho; piroca; pichota; banana; pirola; pissalho; piçalho; bitola

Figura 7.3: Uma página do dicionário aberto de calão

### Medidas

O protótipo do compilador DPL construído é muito pequeno: o parser ocupa cerca de 230 linhas Perl. O protótipo foi testado com exemplos pequenos/médios. Na tabela seguinte apresentam-se as medidas obtidas[2] com o processamento do dicionário aberto de calão.

| | |
|---|---|
| Tamanho do ficheiro de especificação do dicionário | 4649 linhas |
| Número de conceitos | 3660 |
| Número de definições de termo geradas | 4218 |
| Tamanho da saída em papel (A5, duas colunas) | 312 pag. |
| Tempo demorado na construção da semântica intermédia | 6.06s |
| Tempo necessário para a construção da saída LaTeX | 1.63s |

Para um exemplo mais completo do modo de fazer descrições DPL e dos respectivos dicionários obtidos, veja-se a Secção 8.2 – DAC.

[2] Medidas num Pentium III, 600MHz, 128Mbytes de RAM, correndo Linux, em carga

## 7.2 DictTEX – Dicionários usando LATEX

O DictTEX é uma ferramenta ligada ao LATEX desenhada para facilitar a impressão de dicionários em textos LATEX. Para além da ferramenta propriamente dita, o DictTEX inclui um estilo dicionário (dict.sty) e um conjunto de scripts que facilitam a conversão de certos tipos de formatos para LATEX/dict.sty.

Esta sub-secção aparece incluída na dissertação porque:

- Apesar de não ser o ponto central, a possibilidade de impressão em papel de dicionários é uma questão muito relevante.
- A existência de funções que produzam DictTEX a partir de outros formatos, torna o DictTEX e a família dos processadores de LATEX disponível para a *bancada de construção de dicionários*, ou dito de outro modo, para a álgebra de dicionários.
- Sendo o formato de base textual, é também possível gerá-lo a partir de outras ferramentas.
- A ferramenta, em si, é útil.

Como se referiu em 4.1.5, uma das operações associadas ao processo de construção de dicionários é a sua impressão em papel.

Associado à impressão em papel, há uma preocupação de:

- construir um formato compacto – permitir incluir o máximo de informação ao mínimo custo,
- manter ordenadas as entradas,
- facilitar a pesquisa através da inclusão de cabeçalhos que ajudem a visualizar sem grande esforço quais as entradas limites incluídas em cada página,
- construir marcas de início de letra que ajudem a mais facilmente situar o utilizador.

Algumas destas preocupações têm a ver com a eficiência da pesquisa: em línguas cuja escrita se baseia em alfabetos que dispõem de uma relação de ordem total conhecida universalmente, é crucial manter uma ordenação de fácil leitura de modo a permitir uma pesquisa proporcional, ou binária.

O DictTEX foi desenhado para facilitar a construção de dicionários em papel, usando LATEX e herdando toda a cadeia de conversores de LATEX para outros formatos como seja PostScript ou PDF.

Consiste em três partes:

- um estilo LATEX- `dict.sty` onde é definido um ambiente dicionário `dictionary`, um comando `term` e um comando `bigletter`

– um processador `dicttex`, que funciona de modo semelhante ao bibtex, e que analisa um índice (internamente gerado pelo comando LaTeX `term`) calculando os cabeçalhos do dicionário.
– algumas scripts de conversão de alguns formatos de dicionários para DictTeX:
  - `tab2dicttex` converte um dicionário em formato tabela textual para DictTeX
  - `teidict2dicttex` converte dicionários em formato XML/TEI printing dictionaries para DictTeX

Mostra-se seguidamente um exemplo de um texto LaTeX que usa o estilo `dict.sty`

```
1  \documentclass[twoside,twocolumn]{book}
2  \usepackage[portuges]{babel}
3  \usepackage[isolatin]{inputenc}
4  \usepackage{dict}

5  \begin{document}
6  \title{English -- Portuguese dictionary}
7  \author{The Internet Dictionary project}
8  \date{Abril 2001}
9  \maketitle
10 ...Introdução .......................
11 ...................gerado por DictTeX

12 \begin{dictionary}
13   \bigletter{A}
14       \term{a}{a [\emph{Article}]  }
15       \term{a}{uma [\emph{Article}]  }
16       \term{a}{um [\emph{Article}]  }
17       \term{aardvarks}{porco-da-terra, oricterope}
18       \term{aback}{às avessas [\emph{Adverb}]  }
19       \term{aback}{atrás, detrás [\emph{Adverb}]  }
20       \term{aback}{para trás [\emph{Adverb}]  }
21       ...
```

Notas:

**linha 4** - Importação do estilo dicionário, permitindo usar o ambiente `dictionary`, e os comandos `term` e `bigletter`.

**linha 6 a 11** - Texto normal – um dicionário, para além das entradas, tem sempre necessidade de incluir uma variedade de outras informações, como seja a definição da notação usada, introdução, etc.

**linha 12** - Início da zona de definição de entradas.

**linha 13** - Marcação de um início de letra – cria uma nova página e produz um letra muito grande.

**linha 14 a 20** - Definição de entradas. Cada entrada é construída com a marcação `term` que aceita dois argumentos: o termo e a sua definição, podendo esta ter a estrutura que desejarmos.

A Figura 7.4 mostra duas páginas de um dicionário Inglês-Português do *The Internet Dictionary project* (Chambers, 1999), geradas com o conversor `tab2dicttex`. Note-se que não houve qualquer intervenção manual na definição dos cabeçalhos que, como é habitual, contêm a primeira entrada das páginas pares e a última das páginas ímpares.

A Figura 7.5 mostra uma página do dicionário Inglês-Português do projecto FreeDict obtida usando o conversor `teidict2dicttex` que, apesar de estar em estado embrionário, já efectua as necessárias traduções para que a transcrição fonética seja convenientemente impressa.

Ao construir a totalidade das páginas da versão PostScript desse dicionário obtivemos as seguintes medidas[3]:

| | |
|---|---|
| número de páginas construídas | 124 |
| número de entradas do dicionário | 9295 |
| tempo gasto na conversão para DictTEX | 46 segundos |
| tempo total na conversão para PostScript | 1m24s |

### 7.2.1 Funcionamento interno

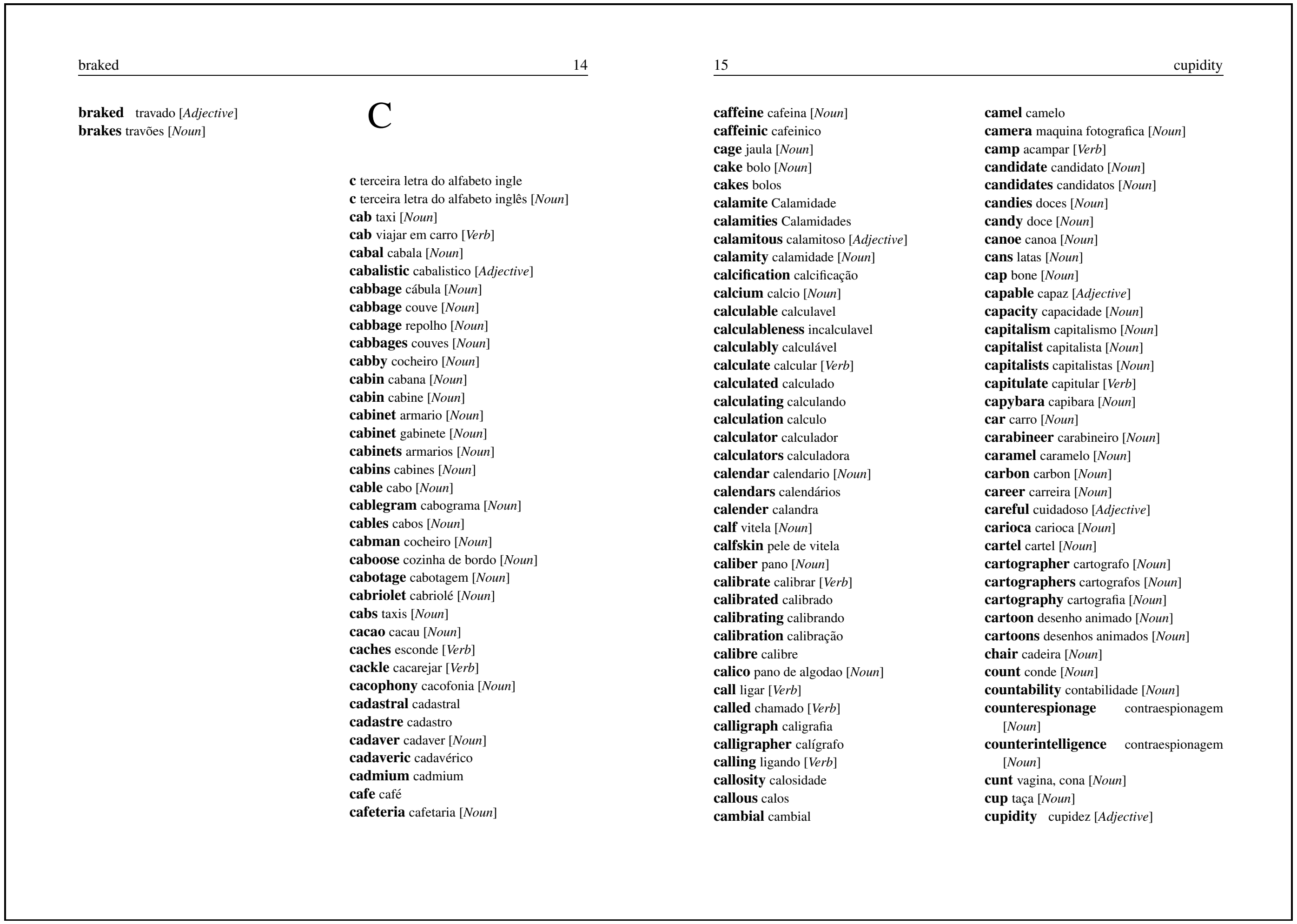
Internamente, o comando DictTEX e o estilo `dict.sty` funcionam do seguinte modo:

- o comando LATEX `term` criado, para além de escrever o termo e a informação associada, constrói um índice interno com os termos todos, cada qual associado ao número de página onde aparece. Se já existir uma tabela de cabeçalhos[4] (onde se guarda a lista dos termos que devem aparecer em cabeçalhos e detalhes associados) construída, e se o termo pertencer ao domínio dessa tabela, gera o cabeçalho respectivo. (Se essa tabela ainda não existir, não faz nada).
- o comando `dicttex` analisa esse índice gerado e cria uma tabela de cabeçalhos para ser usado por `term`

De um modo simplificado, mas usando uma notação formal:

[3]Num Celeron em carga, 700MHz, 64Mbytes de RAM, linux.

[4]Essa tabela poderá existir como resultado de execuções anteriores.


braked 14

**braked** travado [*Adjective*]
**brakes** travões [*Noun*]

# C

**c** terceira letra do alfabeto ingle
**c** terceira letra do alfabeto inglês [*Noun*]
**cab** taxi [*Noun*]
**cab** viajar em carro [*Verb*]
**cabal** cabala [*Noun*]
**cabalistic** cabalistico [*Adjective*]
**cabbage** cábula [*Noun*]
**cabbage** couve [*Noun*]
**cabbage** repolho [*Noun*]
**cabbages** couves [*Noun*]
**cabby** cocheiro [*Noun*]
**cabin** cabana [*Noun*]
**cabin** cabine [*Noun*]
**cabinet** armario [*Noun*]
**cabinet** gabinete [*Noun*]
**cabinets** armarios [*Noun*]
**cabins** cabines [*Noun*]
**cable** cabo [*Noun*]
**cablegram** cabograma [*Noun*]
**cables** cabos [*Noun*]
**cabman** cocheiro [*Noun*]
**caboose** cozinha de bordo [*Noun*]
**cabotage** cabotagem [*Noun*]
**cabriolet** cabriolé [*Noun*]
**cabs** taxis [*Noun*]
**cacao** cacau [*Noun*]
**caches** esconde [*Verb*]
**cackle** cacarejar [*Verb*]
**cacophony** cacofonia [*Noun*]
**cadastral** cadastral
**cadastre** cadastro
**cadaver** cadaver [*Noun*]
**cadaveric** cadavérico
**cadmium** cadmium
**cafe** café
**cafeteria** cafetaria [*Noun*]

15 cupidity

**caffeine** cafeina [*Noun*]
**caffeinic** cafeinico
**cage** jaula [*Noun*]
**cake** bolo [*Noun*]
**cakes** bolos
**calamite** Calamidade
**calamities** Calamidades
**calamitous** calamitoso [*Adjective*]
**calamity** calamidade [*Noun*]
**calcification** calcificação
**calcium** calcio [*Noun*]
**calculable** calculavel
**calculableness** incalculavel
**calculably** calculável
**calculate** calcular [*Verb*]
**calculated** calculado
**calculating** calculando
**calculation** calculo
**calculator** calculador
**calculators** calculadora
**calendar** calendario [*Noun*]
**calendars** calendários
**calender** calandra
**calf** vitela [*Noun*]
**calfskin** pele de vitela
**caliber** pano [*Noun*]
**calibrate** calibrar [*Verb*]
**calibrated** calibrado
**calibrating** calibrando
**calibration** calibração
**calibre** calibre
**calico** pano de algodao [*Noun*]
**call** ligar [*Verb*]
**called** chamado [*Verb*]
**calligraph** caligrafia
**calligrapher** calígrafo
**calling** ligando [*Verb*]
**callosity** calosidade
**callous** calos
**cambial** cambial
**camel** camelo
**camera** maquina fotografica [*Noun*]
**camp** acampar [*Verb*]
**candidate** candidato [*Noun*]
**candidates** candidatos [*Noun*]
**candies** doces [*Noun*]
**candy** doce [*Noun*]
**canoe** canoa [*Noun*]
**cans** latas [*Noun*]
**cap** bone [*Noun*]
**capable** capaz [*Adjective*]
**capacity** capacidade [*Noun*]
**capitalism** capitalismo [*Noun*]
**capitalist** capitalista [*Noun*]
**capitalists** capitalistas [*Noun*]
**capitulate** capitular [*Verb*]
**capybara** capibara [*Noun*]
**car** carro [*Noun*]
**carabineer** carabineiro [*Noun*]
**caramel** caramelo [*Noun*]
**carbon** carbon [*Noun*]
**career** carreira [*Noun*]
**careful** cuidadoso [*Adjective*]
**carioca** carioca [*Noun*]
**cartel** cartel [*Noun*]
**cartographer** cartografo [*Noun*]
**cartographers** cartografos [*Noun*]
**cartography** cartografia [*Noun*]
**cartoon** desenho animado [*Noun*]
**cartoons** desenhos animados [*Noun*]
**chair** cadeira [*Noun*]
**count** conde [*Noun*]
**countability** contabilidade [*Noun*]
**counterespionage** contraespionagem [*Noun*]
**counterintelligence** contraespionagem [*Noun*]
**cunt** vagina, cona [*Noun*]
**cup** taça [*Noun*]
**cupidity** cupidez [*Adjective*]

Figura 7.4: Redução de duas páginas geradas por DictTeX paradoxical 88

**paradoxical** [pærədɔksikl] paradoxal;
**paraffin-oil** petróleo;
**paragraph** [pærəgrɑːf] alínea; parágrafo; parágrafo;
**parallel** [pærəlel] paralelo;
**paralyse** [pærəlaiz] entorpecer; paralisar;
**paralysis** [pərælәsis] paralisia;
**parameter** [pəræmitər] parâmetro;
**paranoia** [pærənɔiə] paranóia;
**parasite** [pærəsait] parasita; parasita; parasita;
**parcel** [pɑːsl] pacote; lote; parcela;
**pardon** [pɑːdn] anistiar; desculpar; escusar; perdoar; perdão;
**parenthesis** [pərenθəsis] grampo; parêntese; parêntese;
**parents** [pɛərənts] pais; pai e mãe;
**parish** [pæriʃ] freguesia; paróquia;
**parishioner** [pəriʃənər] paroquiano; paroquiano;
**park** [pɑːk] estacionar; estacionamento; parque;
**parliament** [pɑːləmənt] parlamento;
**parliamentary** [pɑːləmentriː] parlamentar;
**parlour** [pɑːlər] sala de visitas; salão;
**parochial** [pəroukiəl] paroquial; paroquial;
**parrot** [pærət] papagaio; papaguear;
**parsley** [pɑːsliː] salsa;
**part** [pɑːt] parcela; parte; quinhão; parte; parte; papel;
**parterre** [pɑːtɛər] andar térreo; platéia;
**partial** [pɑːʃl] parcial; partitivo; parcial;
**partially** [pɑːʃəliː] em parte;
**participate** [pɑːtisipeit] participar; tomar parte;
**participle** [pɑːtispl] particípio;
**particle** [pɑːtikl] elemento; partícula;
**particular** [pətikjulər] avulso; especial; isolado; particular; peculiar; reservado; especial; extraordinário; particular; próprio;
**partly** [pɑːtliː] em parte;
**partner** [pɑːtnər] associado; sócio; associado; sócio;
**partridge** [pɑːtridʒ] perdiz;
**party** [pɑːtiː] grupo não organizado; celebração; festa; bandeira; facção; partida; partido;
**pass** [pɑːs] avaçar; licença; passar; fazer passar; alienar; transmitir;
**passage** [pæsidʒ] corredor; galeria; passagem; passagem; viela;
**passageway** [pæsidʒwei] passagem; viela;
**passenger** [pæsindʒər] passageiro;
**passenger train** [pæsindʒərtrein] trem de passageiros; mixto;
**passion** [pæʃən] ardor; brasa incêndio; ardor; paixão;
**passionate** [pæʃəneit] apaixonado; passional;
**passport** [pɑːspɔːt] passaporte;
**pass away** [pɑːsɔːwei] falecer; morrer;
**pass by** [pɑːsbiː] passar; exceder; passar de largo; ultrapassar;
**pass the night** [pɑːsðnait] passaranoiteemalgumlugar;
**past** [pɑːst] passado; passado; além; ao lado de; por diante de;
**paste** [peist] massa; pasta;
**pastor** [pɑːstər] cura; pároco; vigário; cura; pároco; vigário; pastor; clérigo; padre; sacerdote;
**pat** [pæt] bater de leve;
**patch** [pætʃ] remendo; consertar; remendar;
**patch up** [pætʃʌp] consertar; remendar;
**pate** [peit] cabeça; empada; pastel;
**path** [pɑːθ] senda; trilho; vereda; caminho;
**patience** [peiʃəns] paciência;
**patient** [peiʃənt] doente; paciente; paciente;
**patriarch** [peitriɑːk] patriarca;
**patriot** [pætriət] patriota;
**patriotic** [peitriɔtik] patriótico;
**patriotism** [pætriətizəm] patriotismo;
**patrol** [pətroul] patrulhar; patrulha; ronda;
**patrolman** [pətroulmən] gendarme;
**patron** [pætrən] patrono; patrono; protector; protetor;
**patronage** [pætrənidʒ] auspício; bom augúrio;
**pattern** [pætən] esquema; chapa; clichê; molde; padrão;
**pause** [pɔːz] fazer uma pausa; pausa; suspensão;
**pave** [peiv] calçar; pavimentar;
**pavement** [peivmənt] pavimento; calçada; passeio;
**pavilion** [pəviliən] pavilhão;
**paw** [pɔː] perna;
**pawn** [pɔːn] peão; praça; soldado;
**pay** [pei] custear; pagar;
**payment** [peimənt] pagamento;
**pay attention** [peiətenʃən] atender; fixar a atenção em; prestar a atenção;
**pay attention to** [peiətenʃəntou] atender; fixar a atenção em; prestar a atenção;
**pay homage to** [peihɔmidʒtou] homenagear;
**pay out** [peiaut] gastar;
**pay tribute** [peitribjuːt] tributar;
**pea** [piə] ervilha; ervilheira;
**peace** [piːs] paz;
**peaceful** [piːsfəl] pacífico; pacífico;
**peaceloving** [piːsilʌviŋ] pacífico;
**peach** [piːtʃ] pêssego;
**peacock** [piːkɔk] pavão;
**peak** [piːk] ápice; extremidade; pico; ponta;
**peal** [piːl] soar; vibrar;
**peanut** [piːnʌt] amendoim; aráquida;
**pear-tree** pereira;
**pear** [piər] pêra;
**pearl** [pəːl] pérola;
**peasant** [pesnt] camponês;
**peculiar** [pikjuːliər] estranho; esquisito; excêntrico;
**pedal** [pedəl] pedalar; pedal;
**peddle** [pedl] chatinar; mercadejar; traficar;

Figura 7.5: Redução de uma página gerada por DictTEX/teidict2dicttex

$$
\begin{array}{lcl}
indice & = & tp^{\star} \\
tp & = & \begin{array}{lcl} t & : & termo \quad \times \\ p & : & pagina \end{array} \\
headers & = & termo \rightharpoonup header \\
header & = & hesq + hdir
\end{array}
$$

**State** $ind : indice$

**State** $hea : headers$

$$
\begin{array}{l}
term : termo \times str \longrightarrow str \\
term(t,i) \stackrel{\text{def}}{=} \\
\quad \underline{post} \quad push(<t, thispage()>, ind) \\
\quad \underline{in} \quad \underline{let} \;\; printTermo = item(t) \frown i \\
\qquad\qquad\quad printHeader = hea(t) \; or \; \texttt{""} \\
\qquad\quad \underline{in} \;\; printTermo \frown printHeader
\end{array}
$$

$$
\begin{array}{l}
dicttex : indice \longrightarrow headers \\
dicttex(ind) \stackrel{\text{def}}{=} \\
\quad \underline{let} \;\; relev(x) = p(x) \neq p(x+1) \wedge impar(p(x)) \\
\qquad\quad pgbrk = \{<t(ind(x)), t(ind(x+1))> \mid x \in inds(ind) \wedge relev(x)\} \\
\qquad\quad h = \bigcup(\{\{<\pi_1(y), hesq(\pi_1(y))>, <\pi_2(y), hdir(\pi_2(y))>\} \mid y \in pgbrk\}) \\
\quad \underline{in} \;\; \begin{pmatrix} \pi_1(z) \\ \pi_2(z) \end{pmatrix}_{z \in h}
\end{array}
$$

Onde as funções `term` e `item` são LaTeX.

Ou seja, o comando `dicttex` vai criar uma tabela de cabeçalhos do seguinte modo: procura no índice dos termos situações em que o número de página muda de ímpar para par. Nessa situação, guarda-se os termos associados para cabeçalho direito (último termo de uma página ímpar) e cabeçalho esquerdo (primeiro termo da página par).

Esta tabela de cabeçalhos é gravada num ficheiro auxiliar que é carregado por `dict.sty` e usado em `term`, numa segunda passagem do LaTeX, de forma idêntica ao que se passa com as citações bibliográficas, como já se referiu.

# Capítulo 8

# Constantes: recursos relacionados com processamento de linguagem natural

Para que seja eficaz a proposda da dissertação (ver os dicionários como termos de uma álgebra), é necessário que existam os seus componentes básicos:

- funções: ferramentas e comandos – assunto que foi discutido nos capítulos 6 e 7 e no próximo capítulo,
- constantes: recursos ligados à linguagem – os dicionários estáticos, ficheiros, listas de palavras e todo um conjunto de elementos de alguma maneira ligados a linguagem natural, ou ao domínio do dicionário em causa.

Neste capítulo é feita a descrição de alguns recursos relacionados com processamento de linguagem natural cuja história seja de algum modo relevante para esta tese, quer pela sua utilidade enquanto recurso, quer pelo processo como foi construído.

No geral estes recursos estão ligados à classe *open source* (filosofia com a qual o autor se identifica, como foi referido). Nalguns casos descritos neste capítulo, houve:

- utilização de abordagens DDMF para a construção automática de um recurso que funcionou de ponto de partida e que depois seguiu um caminho próprio independente, muitas vezes completamente manual,

- trabalho cooperativo da equipa ligada ao projecto Natura,
- apoio a iniciativas culturais diversas que produziram os seus próprios recursos de variada natureza, por vezes de enorme importância,
- aplicação activa do princípio de que a disponibilização tem efeitos multiplicativos.

Naturalmente, tratando-se de um trabalho na área de informática, será dada ênfase ao processo de contrução dos recursos e aos algoritmos e estratégias associados.

Alguns dos recursos apresentados para além de disponíveis para utilização geral estão disponíveis para consulta através de DDMF.

Mais uma vez tentou-se que cada recurso construído servisse mais do que um fim.

## 8.1 Dicionário português para `ISpell` e `jspell`

Nesta secção, descreve-se o processo ligado à construção dos dicionários portugueses *open source* ligados às ferramentas `ISpell` e `jspell` que serviram de base a um conjunto de outros dicionários (destes directa ou indirectamente derivados).

### 8.1.1 História do desenvolvimento

À data da realização deste exercício, não havia qualquer dicionário *open source* para o Português, impedindo que algumas ferramentas *open source*, dependentes de dicionários, pudessem ser usadas com o Português.

A utilidade da existência de correctores ortográficos é universalmente conhecida. Em Unix, o corrector ortográfico de que dispunhamos (`ISpell`) não estava equipado de dicionário para o Português, pelo que se constatou a necessidade de construir um.

Devido à falta de recursos, foi de imediato posta de lado a solução de ir criando manualmente tal dicionário. Qualquer solução teria de se basear na automatização do que fosse possível.

### 8.1.2 Dicionário `ISpell`

Conforme descrito na Secção 6.1, um dicionário `ISpell` corresponde a um conjunto de regras e uma lista de lemas aos quais estão associados o conjunto de regras a eles aplicáveis.

$$\begin{array}{lcl} ispellDic & = & dic \times set(regras) \\ dic & = & palavra \rightharpoonup set(idRegra) \\ regra & = & \begin{array}{rcll} guarda & : & padrao & \times \\ suf1 & : & str & \times \\ suf2 & : & str & \times \\ id & : & idRegra & \end{array} \end{array}$$

A criação do primeiro dicionário para `ISpell` processou-se do seguinte modo:

- construiu-se um corpus de base: uma grande quantidade de textos (teses de mestrado, textos de enciclopédia, corpora jornalístico, etc.)
- construiu-se o primeiro dicionário, como sendo composto por um dicionário de lemas constituído pelo conjunto de palavras do corpus que

ocorriam um número de vezes maior do que um limiar, e um conjunto de regras morfológicas vazio.

$$\begin{array}{rll} dicio1' = & \underline{let} & oco = calculaOco(corpora) \\ & & uteis = \begin{pmatrix} d \\ \emptyset \end{pmatrix}_{d \in dom(oco) \wedge oco(d) \geq limiar} \\ & & regras = \emptyset \\ & \underline{in} & ispellDic(uteis, regras) \end{array}$$

Seguidamente, escreveu-se manualmente um conjunto de regras morfológias atómicas para os casos mais usuais de flexão verbal, plurais de nomes e adjectivos, derivações mais produtivas. Neste conjunto de regras incluiu-se sempre que possível regras que permitissem adivinhar propriedades acerca do lema.

$$\begin{array}{rll} dicio2' = & \underline{let} & d1 = ispell(d(dicio1), regrasAtPt) \\ & \underline{in} & munchlist(d1) \end{array}$$

Ou seja, juntou-se as regras ao dicionário anterior e usando o comando `munchlist` que vem com o `ISpell`, reduziu-se o dicionário ao mínimo. O comando `munchlist` retira toda a palavra $p_1$ derivável de outra palavra $p_2$ usando uma regra $r$ em $regrasAtPt$ e associa o respectivo identificador à palavra lema $p_2$.

### 8.1.3 Dicionário `jspell`

Usando uma tabela de relevância/confiança que associa a cada regra a classificação do lema e respectivo grau de confiança,

$$\begin{array}{lcl} tabRelev & = & idRegra \rightharpoonup infRelev \\ infRelev & = & classificacao \times int \end{array}$$

determinou-se um dicionário `jspell` contendo lemas classificados

$$\begin{array}{lcl} jspellDic & = & dic \times set(regra) \\ dic & = & palavra \rightharpoonup infPal \\ infPal & = & \begin{array}{ccl} cl & : & classificacao \quad \times \\ rs & : & set(idRegra) \end{array} \end{array}$$

$$\begin{array}{rcrcll} regra & = & guarda & : & padrao & \times \\ & & suf1 & : & str & \times \\ & & suf2 & : & str & \times \\ & & cl & : & classificacao & \times \\ & & id & : & idRegra & \end{array}$$

através de um processo de inferência de classificações baseada nas tabelas de relevância. Este processo faz também a separação de lemas aos quais estejam associados regras que impliquem classificações inconciliáveis.

Algumas das palavra de classe fechada foram classificadas manualmente e *retiradas* deste processo de inferência.

$dicio3' = adivClas(dicio2, tabelaDeRelevancia)$

$adivClas : ispell \times tabelaDeRelevancia \longrightarrow jspellDic$
$adivClas(d, tr) \stackrel{\text{def}}{=}$
determina classificação e grau de confiança de cada palavra

Com base em $dicio3$ substitui-se algumas regras atómicas por outras que as generalizam. Por exemplo, uma regra atómica que liga o verbo *( lavar – lavado )*, é generalizável por *( lavar – { lavado, lavada, lavadas, lavados } )*.

$dicio4' = extrapola(dicio3)$

Tem vindo a ser feita uma verificação sumária manual, tendo em conta o número de ocorrências.

$dicio5' = correccaoManual(dicio4)$

O dicionário tem sido ainda sujeito a enriquecimentos usando programação baseada nos vários interfaces descritos na sec. 6.1, tal como aqueles descritos no exemplo 5, página 98.

Por último, este dicionário `jspell`, serve de base a um conjunto de dicionários, para o que foi necessário construir um conjunto de comandos que partindo de dicionários `jspell` geram:

- dicionário de Português `ISpell`,
- dicionário de Português `aspell`,
- outros dicionários e listas de palavras em vários formatos.

Na versão actual, o dicionário Português do `jspell` está disponível em *open source* a partir da árvore CVS do projecto Natura, juntamente com um conjunto de comandos que a partir dele geram os outros. Várias pessoas têm enviado listas de palavras e sugestões de correcções referentes ao dicionário.

## 8.2 Dicionário de expressões idiomáticas e calão

Nesta secção, descreve-se uma experiência que tem vindo a decorrer no âmbito do projecto Natura, envolvendo a criação de um dicionário aberto de calão e de expressões idiomáticas (DAC), consultável e extensível via Internet. Trata-se de procurar um compromisso aceitável entre o gosto de coleccionar e partilhar, com uma vontade de obter qualidade, equilíbrio e coerência da informação.

### 8.2.1 Introdução

O desenho de um dicionário é uma tarefa reconhecidamente complexa, quer devido à enorme quantidade de informação que tem que ser processada, quer ao nível da delicada coerência e equilíbrio, tão difícil e trabalhosa de conseguir.

Um projecto de construção de um dicionário pode corresponder a situações e objectivos muito díspares.

No projecto DAC considerou-se importante o permitir que o dicionário estivesse *no ar* logo desde o início, de modo a prestar um serviço e funcionar como ponto de partilha e intercâmbio com contribuições externas. Ou seja, decidiu-se proceder à construção de um dicionário *em directo*, *à vista*.

Sabe-se que um dicionário não está (nem nunca vai estar!) acabado e defende-se que um dicionário que esteja em construção, com elevadas assimetrias e entradas de completude muito díspares, pode ainda ser um trabalho válido e constituir um recurso útil.

### 8.2.2 Motivação e características da área

A área particular das expressões idiomáticas (Simões, 1994) e do calão (Nobre, 2000) tem um interesse especial e envolve um rico património linguístico cujo registo é complexo.

Saliente-se no entanto que:

– ameaças infantis tão ricas como "*levas um biqueiro no céu da boca que ficas com a tosse nos calcanhares*" são tão contundentes que, por certo, vencem uma disputa antes de esta se iniciar;
– expressões de calão tão fortes como "*falhar por um pentelho seco de velha*"[1] ou "*confundir o olho do cu com a feira de Montemor*"[2], têm

[1] Usada no Minho, com o sentido de *falhar por muito pouco.*
[2] Centro-Sul de Portugal.

uma riqueza expressiva notável;
- o registo de expressões como estas, usando uma postura arquivística, é uma tarefa de inegável utilidade patrimonial.

Esta área constitui um domínio muito rico, um filão inesgotável mas ao mesmo tempo apresenta um conjunto de dificuldades e desafios ligados a:

- envolver termos multipalavra, com variantes flexionadas
- os seus elementos serem pouco normalizados, tanto pela existência dum grande número de variantes, como pelas dificuldades na grafia de termos que raramente aparecem escritos.
- haver um grande número de equivalências semânticas
- haver um grande dependência do contexto:
    - quem os usa
    - situação de uso, época
    - nível de linguagem
- envolver grandes dificuldades de catalogação.

### 8.2.3 Ambiente de desenvolvimento usado – DPL

Dadas as condicionantes atrás referidas, houve que optimizar o processo de construção do dicionário que passou a ser definido/mantido na linguagem de programação de dicionários – DPL descrita na Secção 7.1.

Esta linguagem é orientada ao conceito (definido através de conjuntos de sinónimos) e com possibilidade de usar notação específica para abreviar e evidenciar partes da informação e fenómenos tão variados como:

- origem etimológica
- origem geográfica
- exemplos e paráfrases
- classes a que pertence
- variantes (especialmente importantes em relação a termos multipalavra)
- adivinhas e provérbios
- trocadilhos fonéticos
- etc

Como referido atrás, o DPL permite: fazer definições de conceitos independentemente do seu uso; dispor de geração de vários tipos de saídasr; funcionar de um modo *declarativo*; manter informação *etiquetada* ou facetada; definir e usar funções.

A partir desta notação, e usando o compilador da linguagem de programação DPL, é gerado todo o sistema de consulta por rede que contém a informação necessária à inserção de termos desconhecidos ou à completação da informação associada.

O sistema de pesquisa – baseado no módulo DDMF (ver Sec. 9) contempla mecanismos para procurar termos e seus derivados morfológicos.

### 8.2.4 Abreviaturas e funções

Um dos aspectos importantes da linguagem DPL, é a capacidade de definição de funções, ou seja de poder definir os conceitos metalinguísticos a usar nas definições de termos. Esses conceitos vão desde a simples abreviatura, até construtores que criam mais que um termo.

As funções ajudam à coerência geral e a evidenciar certos fenómenos linguísticos e correspondem de um modo geral a construtores usados na composição de novos termos.

Alguns exemplos de definição de abreviaturas:

```
sub ptn { +{local  => 'norte de Portugal'}}
sub n2  { +{nivel  => 'coloquial'} }
sub n4  { +{nivel  => 'calão carroceiro'} }
sub sm  { +{genero => 'masculino',
            gram   => 'só usado no masculino'} }
sub diabo { +{sem => 'ordem de não aborrecer e de se ir embora',
              cat => 'interjeição'} }
```

As abreviaturas são, portanto, funções zero-árias que devolvem estrutura de facetas.

Uma função com parâmetros, pode corresponder a uma abreviatura mais complexa, ou pode já descrever conceitos metalinguísticos mais interessantes.

Considere-se o seguinte exemplo de definição de funções com parâmetro:

```
1 sub seems{ +{ isa    => 'eufemismo por semelhança sonora',
2               syn    => shift } }

3 <dict>
4   Júlio++seems(chulo)+"esse? é cá um júlio"++pt:;
5 </dict>
```

Notas:

**linha 1 2** - `Seems` é uma função que recebe um termo e devolve uma estrutura de facetas em que o parâmetro é posto como sinónimo.

**linha 4** - Na definição do termo *Júlio* é acrescentado a estrutura de facetas referida.

A função `seems` está a constituir uma representação explícita de um construtor linguístico muito usado em calão que corresponde a substituir um termo calão de nível elevado por um outro termo que tem semelhanças a nível fonético e que não é calão. Deste modo, está-se a constituir um novo termo calão de nível menos elevado (e portanto possível de utilizar em situações diferentes).

A escrita de funções como esta, ajuda a clarificar conceitos metalinguísticos e a tornar mais eficaz o processo de descrição dos termos.

Seguidamente, apresenta-se exemplos de entradas usando 3 funções habituais em expressões idiomáticas e calão, mostrando-se em seguida o resultado de consultar o dicionário especificado.

```
1 +trocfon(gay ; entreguei+{sem = estar cercado de homossexuais});
2 +adivinha([pi;piolho;olho];
3    qual é o animal que tem mais que três olhos e menos que quatro?;
4    piolho);
5 taveirada+"ela só pensa em taveiradas..."++n3++pt
6    ++afterevent(1986?; termo que apareceu após a circulação de um
7    vídeo clandestino documentando algumas orgias do arquitecto Tomás
8    Taveira)
9    : sexo em posições criativas;
```

Notas:

**linha 1** - Uso da função trocadilho fonético.

**linha 2** - Uso da função adivinha.

**linha 6** - Uso da função afterevent.

Exemplo: resultado de procurar *taveirada*

```
* taveirada
    sem -- sexo em posições criativas
    ex -- ela só pensa em taveiradas...
    local -- Portugal
    isa -- Termo nascido de evento
    nivel -- calão
    evento origem --
        termo que apareceu após a circulação de um vídeo clandestino
        documentando algumas orgias do arquitecto Tomás Taveira
    data -- aparecimento -- 1986 ?
```

Exemplo: procurar *gay*

```
* gay
  .  genero -- masculino
     sem -- homossexual masculino
            indivíduo afeminado
     syn --
         maricas; bicha; ......
     en -- homossexual mail
     nivel -- calão carroceiro

  .  trocadilho --
        name -- entreguei
        isa -- trocadilho fonético
               anedota
        sem -- estar cercado de homossexuais
```

Exemplo: resultado de procurar *entregar*

```
* entreguei
    sem -- estar cercado de homossexuais
    referente -- gay
    isa -- trocadilho fonético

* (encomendar*|entregar) a alma a Deus

* entregar-se de corpo e alma
    sem -- dedicar-se muito a uma tarefa

* (estar*|ficar*) entregue à bicharada

* estar* bem entregue*N
```

### 8.2.5 Definição orientada ao conceito

O facto de se fazer definições orientadas ao conceito permite uma grande economia de descrição. O simples facto de se juntar um conjunto de termos que têm uma acepção comum, é suficiente para definir essa mesma acepção. Considere-se os seguintes exemplos que incluem a palavra *cachimónia*:

```
mona++n2|
  cachimónia++n2|
  tola++fon(tóla)++n2|
```

```
  bestunto++n2+"puxar pelo bestunto"=pensar|
  bestunteira++n2|
  cornos++n3+"tens de meter nos cornos essa matéria"|
  cabeça|
  caixa dos pirolitos:;

dar*-lhe na (cachimónia|cabeça|mona|veneta)++n2|
  dar*-lhe na (real|) gana++n2
   :lembrar-se de;

faltar*-lhe*PN (um parafuso|parafusos)++n2|
  ter* um parafuso a menos++n2|
  ter* a rosca moída++n2|
  não bater* bem da (bola|mona|cachimónia|tola)++n2|
  não ser* bom da cabeça
   :ser ou parecer maluco;
```

A definição estruturada por conjuntos de sinónimos é especialmente útil em situações em que haja grande quantidade de representantes do mesmo conceito, como é o caso do dicionário aberto de calão.

Cada conceito dá origem a várias entradas no dicionário descrito.

Exemplo: resultado de procurar *cachimónia*

```
* cachimónia
    syn --
        mona
        tola
        bestunto
        bestunteira
        cornos
        cabeça
        caixa dos pirolitos
    nivel -- coloquial

* dar*-lhe na (cachimónia|cabeça|mona|veneta)
    syn -- dar*-lhe na (real|) gana
    sem -- lembrar-se de
    nivel -- coloquial

* não bater* bem da (bola|mona|cachimónia|tola)
    syn --
        faltar*-lhe*PN (um parafuso|parafusos)
        ter* um parafuso a menos
        ter* a rosca moída
```

```
       não ser* bom da cabeça
    sem -- ser ou parecer maluco
    nivel -- coloquial
```

### 8.2.6 Meta-informação

Como é sabido, um dicionário é algo mais que um conjunto de definições de termos. Para além da definição dos termos, qualquer dicionário contém um conjunto de informação, como seja o título, as notas prévias, os autores, etc.

No dicionário aberto de calão, a metadata associada ao dicionário aparece explícita e consultável (internamente é guardada como se de um termo normal se tratasse).

```
<meta>
 title:Dicionário aberto de calão e expressões idiomáticas;
 latex:{package="\usepackage{a4wide}"};
 author:{name = José João Almeida;
         email = jj@di.uminho.pt;
         project = Natura;
         org = Universidade do Minho, Departamento de Informática};
 introduction:
     /{ Acreditamos que as expressões idiomáticas e o calão são uma
     parte nobre e rica da Língua Portuguesa. Ao mesmo tempo que
     .............
     colaboração de vários +daci!informantes!agradecimentos! a quem
     muito agradecemos <p> } ;
 sources : [/{http://natura.di.uminho.pt/~jj/pln/calao.dic};
           /{http://natura.di.uminho.pt/~jj/pln/proverbio.dic}];
 descs:{ syn = sinónimos;
          isa = é um;
          ex  = exemplo } ;
 skell:{
       syn = sinónimos (separados por ,);
       isa = [termo; frase pitoresca; interjeição; provérbio;
              insulto; termo calão; idiomática; outra; adivinha];
       semantica = significado;
       level = [normal; calão; erudito; coloquial;
                     calão carroceiro ;
                     calão muito carroceiro ;
                     calão estupidamente carroceiro];
       local = [pt; pt norte ; pt centro ; pt sul ; Brasil ; Angola ;
                     Madeira; Açores; particular];
       outrolocal = origem geográfica;
```

```
31        from = {nome = nome;
32                    email = email (não fica público)} ;
33        ex = { frase = exemplo de uso;
34                   paraf = explicação}
35        } ;
36  copyright: Projecto Natura, J.Joao, Licença GNU;
37  keywords: slang, calão, expressões idiomática;
38 </meta>
```

Notas:

**linha 2** - O título é uma informação que provavelmente será usada por todos os processadores do dicionário.

**linha 3** - Esta informação vai ser usada apenas pelo processador LaTeX.

As ferramentas que processam o dicionário fazem uso parcial dessa meta--informação, ou seja tiram partido desta informação de acordo com as suas necessidades.
Exemplo: para geração da página Internet de consulta ao dicionário, o título, a nota introdutória e os ficheiros fonte da página inicial, são retirados dinamicamente da referida meta-informação.

Do mesmo modo, quando um termo é desconhecido, a ferramenta vai consultar o campo `skel` da metadata e com base nessa informação constrói-se uma *form* (ver Figura 8.1) cujo preenchimento alimenta um ficheiro que será posteriormente revisto manualmente e incorporado no dicionário oficial.

Ver também as Figuras 7.3 e 7.2 da Secção 7.1.

### 8.2.7 Criação duma CGI de consulta

Na versão actual, a criação do dicionário consultável, envolve compilar o dicionário, construindo uma `DB_File` perl. Esta base de dados permite rápido acesso a partir do termo origem, ao mesmo tempo que permite aceitar uma enorme quantidade de chaves (termos).

```
lpdc dac.dpl
```

Após esta operação está a ser feita a junção (composição paralela) com um outro dicionário constituído por pesquisa sobre uma lista de provérbios (cuja informação tem uma natural ligação às expressões idiomáticas).

Para realizar esta tarefa está a ser usado o módulo `ddmf.pm`, que será explicado no próximo capítulo, e que aceita expressões de construção de dicionários como esta:

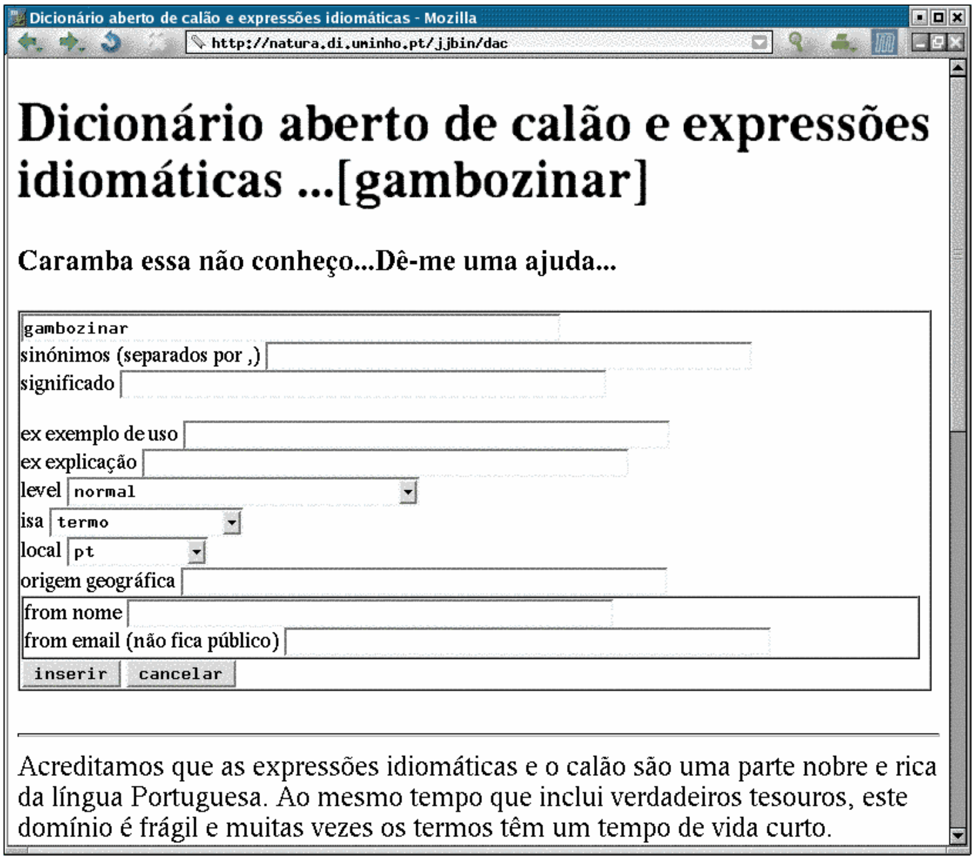



Figura 8.1: Resultado da pesquisa de *gambozinar*

```
1  $d1 = mkparcom(
2          mkdplddmf("dac.dpl"),
3          mkmttab("proverbios",
4                  { sep => ":", type => "ff", lang => "pt"},
5                  ["proverbio.txt","name","tipo","result"] ));
```

Notas:

**linha 1** - Fazer a composição paralela dos dois dicionários que se seguem.

**linha 2** - Construir um dicionário a partir dum texto DPL. Internamente, é determinado o conjunto dos lemas de modo a permitir pesquisa por lema – ou seja pesquisa a menos de normalização morfológica.

**linha 3** - Construir um dicionário a partir duma tabela textual (contendo os provérbios),

**linha 4** - ... em que o separador de campo é ":", a língua usada é o Português (activa a pesquisa com normalização morfológica),

**linha 5** - ... contidos no ficheiro `proverbios.txt`, sendo cada registo da tabela textual composto por um `nome`, um `tipo` e um `resultado` (utilizado quando o tipo é adivinha).

Alguns detalhes deste módulo serão descritos no Capítulo 9.

### 8.2.8 Conclusões

Com a construção / manutenção do dicionário aberto de calão, constatou-se que:

- o campo em causa é muito interessante.
- neste domínio, é indispensável tentar assumir uma postura arquivística, documentalista.
- como exercício de especificação que é, dispor de possibilidade de definir os operadores metalinguísticos a usar nas definições, é algo que ajuda a estudar e entender um domínio.
- o efeito de *disponibilizar tem efeito multiplicativo*, levou a uma grande quantidade de contribuições[3].

[3] Estando a situação presentemente ligeiramente fora de controle!

## 8.3 Corpora

Vários dos exercícios mencionados dependem directamente de corpora. Naturalmente, foi crucial coleccionar e dispor de uma variedade de recursos da família dos corpora, envolvendo diversas características.

Seguidamente, resume-se algumas actividades/recursos ligados directamente a corpora.

Como é sabido, as características ideais de um corpus, dependem do estudo que se pretenda fazer. Essas características têm a ver com:

- o tipo dos elementos constituintes,
- a dimensão,
- a homogeneidades/heterogeneidade ideal,
- o nível de etiquetação,
- etc.

Como princípio geral, procurou-se manter o mais possível a meta-informação e a estrutura de partida existente na informação e construir um conjunto de programas capazes de extrair a informação com as características necessárias/pretendidas – ferramentas de construção de subcorpus.

O trabalho realizado ligado à construção e disponibilização de corpora, consistiu essencialmente em:

- coleccionar recursos existentes e apoiar de diversos modos projectos e actividades que fomentem a sua criação;
- introduzir formatação textual com estrutura explícita;
- introduzir meta-informação;
- criar alguns programas de consulta, transformação de formatos, disponibilização como um todo.

A informação coleccionada envolve uma grande variedade de tipos, de dimensão e importância muito díspares:

1. Lista de provérbios – (cerca de 700 provérbios, incluindo *provérbios modificados*[4]). Esta lista foi construída por aplicação do princípio multiplicativo da disponibilização (através de oferecer uma colecção inicial e pedir colaboração/contribuições).
2. Arquivo de letras de música (cerca de 800 poesias) – constituindo um pequeno corpus de poesia variada, com capacidade de selecção por autor/época. Este arquivo foi também construído por aplicação do princípio multiplicativo.

[4]"*Penso, logo hesito*" e "*penso, logo exausto*" são versões modificadas de "*penso, logo existo*".

3. Arquivos de literatura – (apenas uma parte desta colecção está disponível por questões de direitos de autor).
4. Corpus jornalisticos:
   – Natura-Público
   – Natura-Diário do Minho.
5. Corpus Museu da Pessoa – contendo transcrição de entrevistas orais a pessoas de profissões e sectores muito variados com vista ao levantamento das suas histórias de vida.
6. Corpus Natura-Droga – (pequeno) corpus temático contendo entrevistas, recortes de imprensa, panfletos, etc., ligados directa ou indirectamente à droga. Este corpus foi construído por Zara Pinto Coelho.
7. Thesaurus e catálogos do projecto Alfarrábio, e uma série de colecções nele contidos e que se discutirão na Secção 8.4.
8. Corpora de legendas de filmes (alguns dos quais bilingues, alinhados).
9. Lista de nomes próprios, listas toponímicas.

Para além destes, usou-se ainda os corpora disponíveis, nomeadamente os vários corpora do projecto AC/DC[5] (Santos and Bick, 2000; Rocha and Santos, 2000) integrado no Projecto de Processamento Computacional do Português (Santos, 2000).

Naturalmente, para além dos corpora, foram sendo coleccionados derivados destes:

– listas de ocorrências de palavras
– ocorrências de bigramas, trigramas
– ...

Ligado às actividades acima citadas foi-se construindo e coleccionando um conjunto de pequenos programas tal como:

– ferramentas diversas de ajuda à depuração e verificação de integridade de textos para construção de corpus,
– CGI que implementam um sistema de pesquisa sobre corpora (várias variantes, umas baseadas do jspell – nlgrep[6] e o nlawk[7]; outras baseadas no CQP; outras ainda no htdig)
– um conjunto de pequenos programas de tratamento básico de corpora, como por exemplo:

[5] AC/DC – acesso a corpora, disponibilização de corpora.

[6] O comando nlgrep tem semlhanças com o `grep` só que os padrões de pesquisa aceites são expressões regulares extendidas com morfologia. Exemplo de padrão de pesquisa: `(o|do|no|pelo) (meu)?  {CAT=>"nc"} {}`.

[7] Semelhante ao anterior só que com a possibilidade de definir acções a relizar sobre os elementos encontrados.

    - contadores de ocorrências de palavras, bigramas, trigramas,
    - extracção/processamento de nomes próprios
    - funções para dividir um texto em frases, com marcação opcional em XML
    - tokenizadores sensíveis a *elementos não literários* (como endereços de emails e urls)

  Um parte destes programas estão juntos e disponíveis no módulo Perl `Lingua::PT::PLN`.
- ferramentas para extracção de corpora de legendas a partir de legendas divX e alinhadores para esse tipo de ficheiros.

## 8.4 Thesaurus e catálogo Alfarrábio

É reconhecido o enorme crescimento da quantidade de informação disponível na Internet. No entanto, o crescimento de conhecimento disponível não tem acompanhado esse ritmo. Há uma enorme falta de estruturação, de visão orientada aos conceitos e não ao design. Esse problema deu origem ao aparecimento (e proliferação) de *portais*, embora estes normalmente *arrumem conceptualmente* sites e não documentos.

Nesta secção, discute-se alguns recursos e ferramentas ligados ao projecto Alfarrábio, que visa a catalogação de documentos culturais relevantes contidos num conjunto heterogéneo de sites (os alfarrabistas).

Esta secção, e os recursos nela analisados, estão ligados à ferramenta LIBRARY::*, ver Secção 6.8.

### 8.4.1 Enunciado do problema: Projecto Alfarrábio

O Alfarrábio nasceu da necessidade de guardar uma série de iniciativas individuais em vias de extinção[8] ligadas de algum modo à cultura.

As iniciativas em causa envolvem pessoas (os alfarrabistas) com perfis, localização geográfica e formação muito variada. As áreas, as abordagens, as ferramentas, são também muito heterógeneas.

O Alfarrábio contém *sites* sobre assuntos tão variados como:

- música (poemas, partituras, karaokes)
- eventos culturais de Braga
- dicionário Chinês-Português
- arquivos de Literatura Portuguesa
- histórias infantis
- clubes científicos juvenis
- instituições e organizações diversas (Ex. a Cruz Vermelha de Amares)
- Histórias de Vida, Histórias de Família e tradição oral
- desporto
- lições de Português
- arquivos de Etnomusicologia, fonogramas
- comboios
- ...

[8] Refira-se a extrema volatilidade da informação da Internet. No caso presente, muitos dos desaparecimentos de páginas estavam a dever-se a mudança de estatuto (ex. alunos que acabavam o curso), mudanças geográficas, etc.

Para que essas partes constituíssem um projecto, era necessário estabelecer ligações entre os vários documentos produzidos pelos alfarrabistas; era necessário criar uma vista homogénea sobre a referida heterogeneidade.

O Alfarrábio nasceu com a intenção de ser uma espécie de cooperativa cultural, servida por um sistema Linux, usando software *open source*, e usando máquinas que não estivessem a ser necessárias[9].

### 8.4.2 Objectivos gerais

Um dos grandes objectivos deste projecto é garantir uma certa durabilidade da informação Alfarrabista.

Conforme referimos, os *alfarrabistas* constituem um conjunto extremamente heterogéneo de pessoas, com muito boa vontade, mas com as suas actividades profissionais e que apenas dispõem de uma curta fatia de tempo para desenvolvimento de informação cultural, como um hobby. Nesta situação, não é viável a imposição aos alfarrabistas de standards.

Para que se desenvolva um projecto a partir de um colecção de sites de Internet de estrutura tão díspar, decidiu-se construir um nível extra:

- que fosse baseado num tipo comum – a meta-informação, ou seja um catálogo rico que escondesse os detalhes específicos de cada documento mas que reflectisse os seus conceitos essenciais
- que constituísse uma vista comum orientada ao documento abstracto
- que estabelecesse o maior número possível de ligações entre os documentos.

Ou seja, para cada site, deve haver uma entrada de catálogo que o descreva como um todo, e um catálogo contendo descrições de cada sub-documento relevante.

Para facilitar o estabelecimento de relações, constatou-se a necessidade de construir e utilizar:

- um estrutura classificativa rica (uma ontologia)
- um conjunto grande de relações para classificar cada documento (relacionando-o com elementos da estrutura classificativa e com outros documentos)

e construir um conjunto de ferramentas que tirassem o máximo partido da estrutura criada e automatizassem, dentro do possível, o processo classificativo:

[9]Embora não seja muito relevante, o Alfarrábio nunca dispôs de máquinas em *primeira mão*...

– ferramentas para ajudar a construir catálogos
– ferramentas para construir navegação e pesquisa conceptuais (a principal ferramenta desta família é LIBRARY::*, ver Secção 6.8 e Figura 8.2)

Ou seja, aplicar uma abordagem de programação a um problema típico de Arquivística e Biblioteconomia.


Figura 8.2: Pesquisa de `brasileira!casa`

### 8.4.3 Thesaurus do Alfarrábio

Conforme referido, a existência de uma estrutura classificativa comum é um elemento importante neste projecto. Depois de uma análise geral em que se tentou usar estruturas classificativas baseadas no CDU (Classificação Decimal Universal) e no thesaurus da UNESCO, conclui-se da utilidade de desenvolver um thesaurus específico.

Actualmente, o thesaurus do Alfarrábio tem cerca de 700 termos classificativos e constitui por si próprio um recurso útil. De entre os muitos termos, alguns nasceram por necessidade geral (ex.: *pessoa, monumento, lenda* ou *poema*) outros tiveram origem em necessidades de alfarrabistas específicos (ex.: o *Museu da Pessoa* deu origem a uma enorme variedade de profissões).

Actualmente, está-se a construir um sistema para especificar um thesaurus com base numa expressão de construção que use uma variedade de sub-thesaurus. Os sub-thesaurus actuais incluem por exemplo:

- sub-thesaurus geográfico
- sub-thesaurus de profissões
- sub-thesaurus de música e instrumentos musicais
- sub-thesaurus de Etnografia

### 8.4.4 Os catálogos

A ferramenta LIBRARY::* aceita catálogos multi-formato desde que acompanhados com a necessária informação de extracção dos elementos indispensáveis. Isso permite que se possa facilmente aproveitar catálogos existentes ligados aos próprios sites individuais.

Sempre que houve necessidade de criar catálogos manualmente, foi usado para o efeito um formato XML que define que cada catálogo é composto por um conjunto de documentos contendo campos de acordo com os catálogos da Secção 6.8.4:

$$\begin{array}{lcrcll} catalog & = & entry^{\star} & & & \\ entry & = & t & : & title & \times \\ & & u & : & url & \times \\ & & d & : & description & \times \\ & & as & : & author^{\star} & \times \\ & & r & : & set(relation) & \end{array}$$

$$\begin{array}{lcl} author & = & mail \times name \times role \\ relation & = & Rel \times term \\ description & = & \text{...XML bem formado} \end{array}$$

Saliente-se que certos campos (Ex. **description**) podem ter uma estrutura interna qualquer (tipicamente um subconjunto do HTML) desde que seja XML bem formado.

No Alfarrábio há vários tipos de sites. Dum modo simplificativo, os sites construídos pelos alfarrabistas podem ser catalogados numa ou mais das seguintes classes:

1. Sites que são arquivos digitais – neste caso é possível a construção (dentro do arquivo) de funções de exportação do catálogo ou a construção de funções de travessia específicas
2. Sites contendo índices (em um ou mais níveis) – neste caso é possível fazer a extracção de catálogos explícitos por travessia dos mesmos. Para automatizar este tipo de extracção, foi construída uma ferramenta (WMBOI) que se descreve na próxima secção (Sec. 8.5).
3. Sites que apresentam um conjunto nítido de documentos – neste caso é possível tentar a extracção de informação a partir do próprio documento e da sua meta-informação, caso exista.
4. outros sites... – neste caso o catálogo poderá ser construído com recurso a ferramentas interactivas.

Para além destes tipos de sites, alargou-se posteriormente o projecto a *páginas cooperantes*, que embora não sediadas na máquina Alfarrábio, pretendiam/aceitavam aderir aos objectivos/princípios do Alfarrábio. No fundo, as páginas cooperantes são alfarrabistas remotos, havendo necessidade de que os respectivos catálogos sejam de algum modo enviados ou extraídos para o catálogo geral do Alfarrábio. Na próxima secção, é referido um dos mecanismos – WMBOI – de extracção automática de catálogos.

A heterogeneidade geral vai conduzir a que haja diferentes riquezas nos diferentes catálogos.

### Formato de catálogo baseado no DTD Alfarrábio

Um dos documentos de maior importância, e como tal verificado manualmente, é a lista dos *alfarrabistas*[10].

[10] Apesar do esqueleto das entradas desta lista poder ser gerada automaticamente, considerou-se importante fazer a sua edição manual para criar uma descrição e um conjunto de

Considere-se a seguinte descrição geral do projecto alfarrabista chamado **Cancioneiro**:

```
1  <doc>
2    <title lang="pt">Cancioneiro</title>
3    <resource lang="pt">
4            http://alfarrabio.um.geira.pt/cancioneiro
5    </resource>
6    <description><pre>
7      .  Introdução
8      .  Section 1: Canções infantis em português
9      .  Section 2: Canções em português</pre>
10    <p> Este conjunto de cerca 80 músicas...  </p>
11   </description>
12   <author email="dmorais@ip.pt"
13           role="Selecção, partituras, karaoke e notas">
14      Domingos Morais         </author>
15   <relations>
16     <rel type="iof">livro</rel>
17     <rel type="iof">arquivo</rel>
18     <rel type="pof">alfarrábio</rel>
19     <rel type="about">música</rel>
20     <rel type="about">Portuguese</rel>
21     <rel type="about">karaoke</rel>
22     <rel type="about">partitura</rel>
23     <rel type="about">poema</rel>
24   </relations>
25 </doc>
```

Notas:

**linha 2** - Cada documento é identificado com um título,

**linha 3 a 5** - tem um ou mais url de ligação ao documento propriamente dito,

**linha 6 a 15** - tem vária meta-informação.

**linha 16 a 23** - Os elementos `rel` estão a ser usados para classificar o documento, ligando-o a termos do thesaurus classificativo através de relações de determinado tipo.

Um documento pode ter informação menos óbvia, tal como o seguinte documento referente a uma fotografia contida no projecto alfarrabista **Alfabraga** que colecciona informação diversa acerca da cidade de Braga:

```
 <doc id="136">
  <resource>/alfabraga/photos/P0000146.JPG</resource>
  <icon>/alfabraga/photos/thumbs/P0000146.JPG</icon>
```

associações mais ricas.

```
4     <title>Arcada</title>
5     <author>Augustsson</author>
6     <relations>
7      <rel type="geo">Braga</rel>
8      <rel type="iof">Praça</rel>
9      <rel type="iof">fotografia</rel>
10     <rel type="pof">Alfabraga</rel>
11     ...
12    </relations>
13    <where mapa="braga1" x="0.4724324" y="0.6107611"/>
14   </doc>
```

Notas:

**linha 3** - Uma imagem miniatura (o que só faz sentido quando se trata de um recurso pictográfico).

**linha 13** - Uma localização num mapa (permitindo navegação geográfica (Rocha, Pedroso, and Almeida, 1999)).

Saliente-se que os catálogos incluem entradas de grande heterogeneidade, aparecendo lado a lado entradas referentes a documentos simples, referentes a colecções, a *sites* inteiros ou informação acerca de autores ou projectos.

### Sites com o seu próprio catálogo

Alguns sites alfarrabistas contêm o seu próprio catálogo explícito.

### Museu da Pessoa

O Museu da Pessoa (Almeida et al., 2001), é um arquivo digital na web, constituído por um conjunto variado de documentos, incluindo transcrições de histórias de vida (em XML, usando o DTD museu da pessoa), sub-histórias e episódios, fotografias, legendas, etc.

O Museu da Pessoa tem (entre outros modos) os documentos armazenados no sistema de ficheiros de um modo sistemático[11]; usando Directory Attribute Grammars (DAG (Simões, Almeida, and Henriques, 2002)) como formalismo de construção, produzem-se várias vistas sobre o seu acervo (ver Fig. 8.4), tal como:

[11] Isto é, de acordo com uma gramática em que os não terminais (como `Projectos` e `Pessoas`) correspondem a directorias e os terminais (como `fotografias` ou `entrevistas`) correspondem a ficheiros.

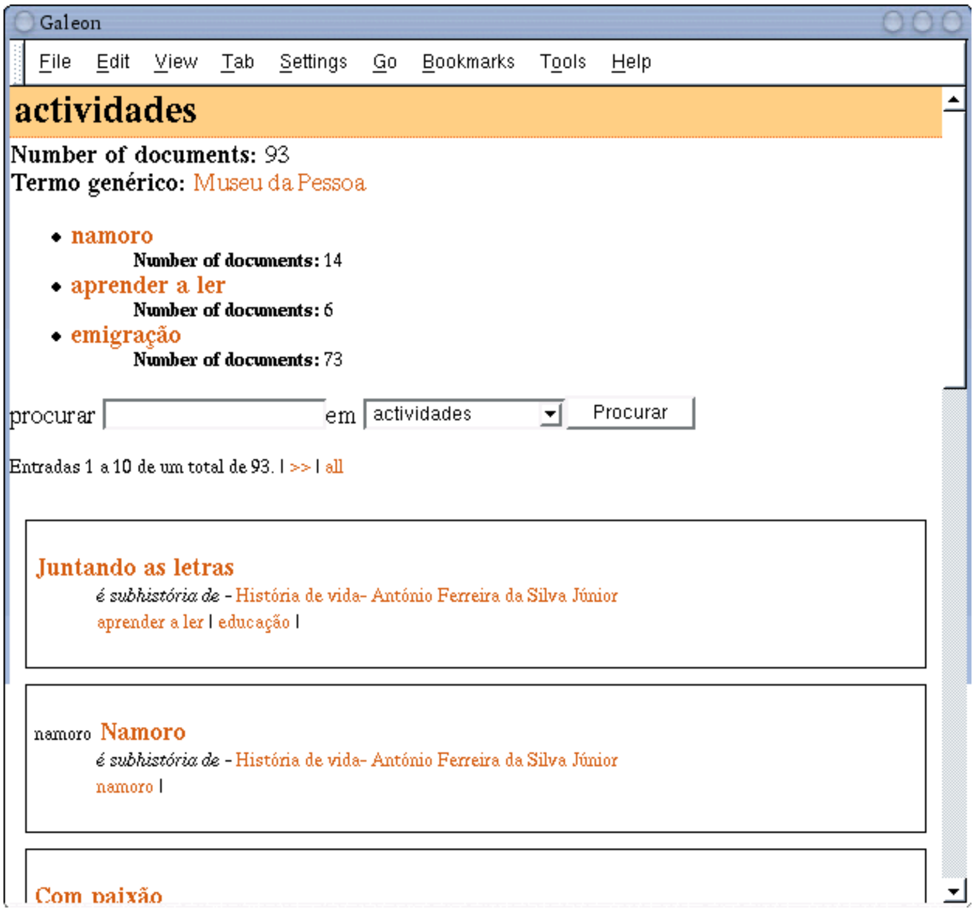



Figura 8.3: Navegação conceptual no Museu da Pessoa

- um conjunto de páginas Web,
- um conjunto de livros PDF,
- índices de projecto, de pessoas, índices temáticos,
- cronologias,
- dicionários,
- etc.

Uma outra vista que se acrescentou, foi a criação do seu catálogo completo, o qual permitiu navegação e pesquisa via LIBRARY::* (ver Fig. 8.3), e ao mesmo tempo integração na navegação e pesquisa Alfarrábio. Este catálogo tem bastante riqueza de catalogação, dado que essa informação existia explícita no arquivo digital do Museu da Pessoa. O número de documentos obtidos é superior a 1000.

#### Arquivos de Etnomusicologia

Os **Arquivos de etnomusicologia** incluem:

- os arquivos sonoros Ernesto Veiga de Oliveira,
- os arquivos sonoros de recolhas de Domingos Morais,
- a biblioteca de Etnomusicologia.

Estes arquivos são compostos por cerca 2Gbytes de fonogramas em MP3 (cerca de 900 registos de música etnográfica), juntamente com alguns artigos, textos introdutórios, fotografias. Para a definição do site foi construído um conjunto de ficheiros XML. O catálogo alfarrábio está a ser obtido por transformação desses ficheiros para o formato alfarrabio.

### 8.4.5 Conclusões

Adicionar uma camada conceptual a um conjunto de documentos, aumenta a sua riqueza, ou seja, o conhecimento contido num arquivo digital, excede o conhecimento contido nos seu documentos:

$$conhecimento(arquivodigital) > \sum conhecimento(partes)$$

Algumas características que se obtiveram com a abordagem:

- os alfarrabistas não são forçados a seguir um formato standard;
- os alfarrabistas podem contribuir/guiar o processo classificativo e de catalogação;
- os elementos do sistema são escaláveis, sendo possível usá-los em projectos alfarrabistas grandes e pequenos – levando obviamente a que os catálogos produzidos tenham tamanhos diferentes;
- todo o sistema está baseado em ferramentas e formatos *open source*.

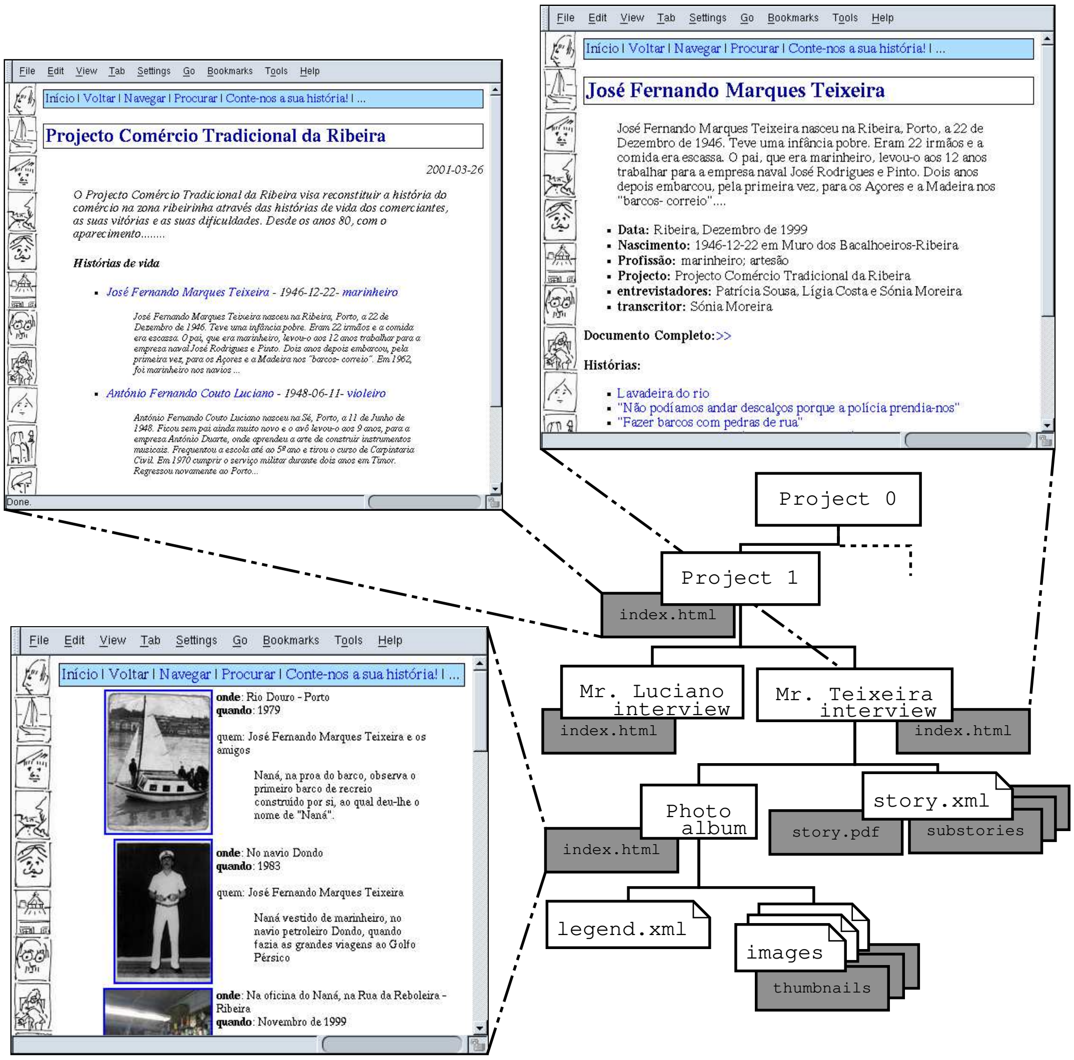



Figura 8.4: Museu da Pessoa

## 8.5 Extracção de catálogos com a ferramenta WMBOI

Nesta secção, é sumariamente descrita uma ferramenta de extracção automática de catálogos a partir de sites que dispõem de índices (minimamente) organizados. Para melhor descrever o seu funcionamento, será usado o exemplo dum site alfarrabista denominado (anarco-)enciclopédia do Alfarrábio.

A motivação para a criação da ferramenta WMBOI, nasceu da necessidade concreta de criar catálogos ligados a certo tipo de sites do Alfarrábio – as páginas cooperantes remotas e os sites alfarrabistas que não têm capacidade de facilmente manter os seus próprios catálogos.

No entanto, constatou-se que o seu uso pode ser alargado a um conjunto mais vasto de sites externos (desde que disponham de um índice estruturado).

Nesta secção é descrito esse processo de incorporação de informação extraída de sites na Web, por análise de páginas índice, usando a ferramenta WMBOI – Web Mining Based On Indexes.

Esta ferramenta baseia-se num conjunto de constatações muito simples:

- quase todos os arquivos na Internet têm uma página índice de escrita razoavelmente sistemática,
- esses índices podem ter um ou mais níveis,
- normalmente há uma classe comum aos elementos de cada índice,
- normalmente os índices estabelecem ligações aos elementos (documentos) através de links HTML em que o texto da ligação descreve de alguma maneira o documento ligado (ou a classe do subíndice).

O WMBOI toma como ponto de partida uma configuração onde se define pontos de entrada, estratégias de extracção, informação implícita, etc.

### 8.5.1 (anarco-)enciclopédia do Alfarrábio

O projecto (anarco-)enciclopédia do Alfarrábio – AEA, visa criar um conjunto de catálogos por extracção de conhecimento de um conjunto de sites culturais. Dado que cada catálogo deste projecto vai relacionar um título com informação cultural a ele associado, está a comportar-se de modo semelhante a uma enciclopédia:

- cada título corresponde a uma entrada;
- a informação associada ao título corresponde à matéria associada à entrada;

– cada site está normalmente ligado a um domínio.

Para o estudo presente, a AEA, é composta por um conjunto de páginas cooperantes.

$$aea \;=\; set(pagCoop) \times ...$$

### 8.5.2 Breve descrição do WMBOI

A ferramenta WMBOI é um módulo Perl que contém funções que extraem catálogos de arquivos na Internet, partindo de uma página índice e de uma especificação declarativa de características desse índice.

De entre as várias funções que compõem o módulo saliente-se:

– a função de extracção baseada em página com índice
  $wm : configuracaoPag \longrightarrow ef$
  $wm(c) \stackrel{\text{def}}{=}$ extrai um catálogo a partir da configuração de um índice

– e a função de extracção baseada em definição abstracta de enciclopédia
  $encwm : configuracaoEnc \longrightarrow ef$
  $encwm(c) \stackrel{\text{def}}{=}$
  extrai o catálogo de uma enciclopédia a partir de uma especificação

### 8.5.3 Breve descrição do algoritmo `wm`

$$\begin{array}{lll}
pagCoop & = & \begin{array}{rcll} name & : & str & \times \\ meta & : & ef & \times \\ calalogFileName & : & str & \times \\ catal & : & catalog & \times \\ validfor & : & int & \times \\ make & : & \text{scriptExt} + \text{wmboi} & \end{array} \\
catalog & = & term \rightharpoonup ef \\
scriptExt & = & url \longrightarrow catalog \\
wmboi & = & \begin{array}{rcll} u & : & url & \times \\ tobeadded & : & ef & \times \\ follow & : & \text{wmboi} + \text{NIL} & \end{array}
\end{array}$$

O processo de extracção de catálogos pode ser descrito através duma script específica de extracção ou através de uma configuração WMBOI. No segundo caso, a função de extracção usa a função *wm* que extrai os links dum ficheiro, seguindo recursivamente caso haja $follow$[12].

Dum modo simplificado, $wm$ corresponde a:

$$
\begin{array}{l}
wm : wmboi \longrightarrow set(ef) \\
wm(w) \stackrel{\text{def}}{=} \\
\quad \underline{let} \;\; a = getFile(u(w)) \\
\qquad\quad lls1 = extractLinks(a) \\
\qquad\quad lls2 = \{l \dagger tobeadded(u) \mid l \in lls1\} \\
\quad \underline{in} \;\; \begin{cases} \text{is-}wmboi(follow(w)) & \Rightarrow \;\; \left\{l \dagger \left(\left(\begin{array}{c} has \\ wm(next(w,l)) \end{array}\right)\right) \mid l \in lls2\right\} \\ \underline{otherwise} & \Rightarrow \;\; lls2 \end{cases}
\end{array}
$$

$$
\begin{array}{l}
next : wmboi \times ef \longrightarrow wmboi \\
next(w, entrada) \stackrel{\text{def}}{=} follow(w) \dagger \left(\left(\begin{array}{c} u \\ url(entrada) \end{array}\right)\right)
\end{array}
$$

$getFile : url \longrightarrow TXT$
$getFile(u) \stackrel{\text{def}}{=}$ dado um url u, vai buscar o texto associado

$extractLinks : TXT \longrightarrow set(url\_string)$
$extractLinks(t) \stackrel{\text{def}}{=}$ dado um texto extrai os pares url - nome nele contidos

As configurações referidas incluem informação acerca de:

- **url** url de origem
- **name** nome do arquivo em causa
- **file** nome do ficheiro catálogo a criar
- **isa** tipo dos elementos que nele podemos encontrar
- **tobeadded** informação a adicionar a cada ficha de catálogo encontrada neste índice
- **follow** informação acerca dos filhos
- **tobepropagated** informação a propagar de modo a ser usada no processamento dos filhos
- **validfor** informação acerca da periodicidade com que os catálogos devem ser recalculados.

[12]Estamos em presença de mais um processamento da família dos hilomorfismos (um hilomorfismo monádico) em que a estrutura de entrada é um site Web, a estrutura interna é a árvore de índice e a estrutura de saída o catálogo

### 8.5.4 Análise de uma configuração particular

Considere-se o seguinte exemplo extraído da AEA:

```
1  [
2    { name        => ZecaAfonso,
3      isa         => [arquivo web],
4      url         => http://alfarrabio.di.uminho.pt/zeca,
5      rel         => { pof   => [Alfarrabio],
6                       about => [música,poema]},
7      autor       => [Arménia Rua],
8      description => .... o compositor, poeta e cantor Zeca Afonso,
9      wmboi       => {
10       file => ZecaAfonso.enc,
11       validfor => 30,
12       conf => {
13         url => http://alfarrabio.di.uminho.pt/zeca/disc.html,
14         tobeadded=>{
15            author=> Zeca Afonso,
16            isa   => [disco] },
17         follow=>{
18            tobepropagated=>{ name   => in,
19                              author => author},
20            tobeadded     =>{ isa    =>[poema]}}}}},
21  ..........
```

Notas:

**linha 1 a 20** - Descrição da página cooperante (arquivo digital) `ZecaAfonso`.

**linha 9 a 20** - Descrição do modo a ser seguido para extracção do catálogo acerca deste arquivo digital.

**linha 10** - Definição do ficheiro onde o catálogo vai ser guardado

**linha 11** - Definição acerca da actualização da página: só é feita nova análise do índice e do site, quando o ficheiro `ZecaAfonso.enc` tiver mais que 30 dias de idade.

**linha 13** - Definição do url inicial

**linha 14 a 16** - A informação definida com `tobeadded` é adicionada a todos os elementos extraídos do índice. Neste caso, todos os elementos do primeiro nível são discos da autoria de Zeca Afonso.

**linha 17 a 20** - A existência de `follow` está a indicar que o índice tem mais que um nível, ou seja, que cada link encontrado deve ser seguido e que se deve proceder a nova extracção de documentos desta vez do tipo poema.

**linha 18, 19** - Definição do modo de propagar informação para o subíndice: o campo `name` do presente nível (disco) será o campo `in` do poema; o campo `autor` do presente nível é copiado para o nível seguinte.

Como a entrada apresentada contém um atributo `wmboi`, o programa que periódicamente actualiza a AEA, invoca a extracção WMBOI com a respectiva configuração.

A informação de `from` é propagada para todos os subíndices a menos que haja alteração explícita desse campo.

Um índice terá tantos níveis quantos os `follow` aninhados que existirem na especificação da configuração.

Analogamente, a próxima entrada descreve o arquivo dos versos de segunda:

```
1  { name=> Versos de Segunda,
2    isa => [arquivo web],
3    url => http://www.isr.ist.utl.pt/~cfb/VdS,
4    rel => { about => [poema]},
5    autor => [Carlos Bispo],
6    description => uma colecção de poemas de diversos autores
7       seleccionados por Carlos Bispo e enviados semanalmente para
8       a rede às segundas-feiras -- daí a razão do nome). ,
9    wmboi => {
10     file => "vds.enc",
11     validfor => 7,
12     conf =>{
13       url        => "http://www.isr.ist.utl.pt/~cfb/VdS/zlista.html",
14       beginafter => quotemeta( poemas presentemente disponíveis.),
15       endwith    =>  "<H3>",
16       tobeadded  => { isa      => [poeta] },
17       follow     => {
18           tobepropagated=>{ name => author},
19           tag => "obra",
20           tobeadded  => { isa => [poema] }}}}},
21  ... ]
```

Notas:

**linha 14** - `beginafter` permite especificar uma expressão regular que defina a zona da página a partir da qual a extracção deve ser iniciada.

**linha 15** - Analogamente `endwith` é usado para especificar a zona onde a extracção deve ser abandonada.

**linha 20** - `tag` permite definir o identificador da etiqueta que agrupa os documentos dum subíndice (por omissão – `has`).

### 8.5.5 Resultado produzido pelo extractor

Após executar a função de `wm`, os ficheiros catálogos (`ZecaAfonso.enc`, `vds.enc`, etc.) são automaticamente criados/actualizados (a menos que eles já existam e estejam dentro da validade especificada).

```
[
  { url => http://alfarrabio.di.uminho.pt/zeca/primeiras.html,
    isa => [ disco ],
    author => Zeca Afonso,
    from => ZecaAfonso,
    name => Primeiras Canções
    has => [
      { url => http://alfarrabio.di.uminho.pt/zeca/cancoes/15.html,
        isa => [ poema ],
        author => Zeca Afonso,
        from => ZecaAfonso,
        in => Primeiras Canções,
        name => Menino D'oiro },

      { url => http://alfarrabio.di.uminho.pt/zeca/cancoes/17.html,
        isa => [ poema ],
        author => Zeca Afonso,
        from => ZecaAfonso,
        in => Primeiras Canções,
        name => Menino do Bairro Negro },
        .............
     ],
   },
   { url => http://alfarrabio.di.uminho.pt/zeca/cantares.html,
     name => Cantares do Andarilho
     isa => [ disco ],
     from => ZecaAfonso,
     author => Zeca Afonso,
     has => [
       { url => http://alfarrabio.di.uminho.pt/zeca/traz.html,
         isa => [ poema ],
         author => Zeca Afonso,
         from => ZecaAfonso,
         in => Cantares do Andarilho,
         name => Traz outro amigo também },
       { url => http://alfarrabio.di.uminho.pt/zeca/cancoes/41.html,
         isa => [ poema ],
         author => Zeca Afonso,
         from => ZecaAfonso,
         in => Cantares do Andarilho,
```

```
         name => Natal dos simples },
                .....
                     ],
   },
   ....  ]
```

```
[ { url => http://www.isr.ist.utl.pt/~cfb/VdS/alegre.html,
    from => Versos de Segunda,
    name => Manuel Alegre
    isa => [ poeta ],
    obra => [{ url => http://www.isr.ist.utl.pt/~cfb/VdS/v008.txt,
               isa => [ poema ],
               author => Manuel Alegre,
               from => Versos de Segunda,
               name => Trova do vento que passa },
             { url => http://www.isr.ist.utl.pt/~cfb/VdS/v023.txt,
               isa => [ poema ],
               author => Manuel Alegre,
               from => Versos de Segunda,
               name => Lisboa perto e longe },
            ................  ],
    },
    .................  ]
```

Só estas duas *páginas cooperantes* estão a produzir 645 entradas (poetas, discos, poemas).

## 8.6 Recursos ligados à ferramenta TEXT::TRANSLATE

Uma ferramenta de tradução industrial é complexa e cara porque necessita de uma enorme quantidade de informação. Uma tradução correcta necessita de compreensão dos textos a nível morfológico, léxico, sintáctico, semântico e pragmático (Allegranza et al., 1991).

No entanto, o problema da tradução pode variar muito a nível de complexidade e de exigência. Por um lado, a tradução de poesia ou de anedotas exige muita criatividade e talento (em alguns casos a tradução de certos textos pode ser mesmo impossível). No extremo oposto, encontramos problemas ligados a um domínio muito específico e técnico, com frases não ambíguas e vocabulário restrito (Filgueiras, 1994). São exemplos desta classe a tradução de mensagens de programas, manuais ou de curriculum vitae. Para estes casos, é possível construir ferramentas simples, pouco inteligentes, mas ainda assim úteis.

Nesta secção defende-se que é útil construir ferramentas que lidem com dicionários controlados pelo utilizador e que, por vezes, um conjunto de pequenas ferramentas (ou funções) programáveis, pode produzir resultados positivos em problemas tão complexos como o da tradução.

TEXT::TRANSLATE é uma biblioteca Perl que implementa um conjunto de pequenas funções ligadas a tradução.

Esta biblioteca não pretende servir directamente o utilizador final. A sua finalidade é ajudar o perito a construir pequenas ferramentas a ser usadas pelos utilizadores finais.

TEXT::TRANSLATE não conhece directamente a noção de língua. Os dicionários a usar podem referir-se a um qualquer par de línguas ou até a substituições dentro de uma mesma língua.

Esta secção está incluída nesta dissertação com a finalidade de:

- mostrar que se houver as precaucões necessárias na etapa do projecto, a construção de ferramentas constitui também uma forma de criar e enriquecer recursos de linguagem natural;
- mostrar que a subdivisão de um tradutor (naïf) em funções elementares permite atacar uma série de outros problemas para além dos inicialmente previstos.

Neste caso a ferramenta TEXT::TRANSLATE vai simultaneamente:

- usar um DDMF para tratar de palavras desconhecidas

– construir/enriquecer os dicionários bilingues de tradução.

Na subsecção 8.6.1, é feita uma descrição geral do TEXT::TRANSLATE. Em 8.6.2, é mostra-se que é natural o uso de DDMF para tratamento de palavras desconhecidas. Em 8.6.3, estudam-se alguns problemas exemplo, dando especial ênfase à questão da adaptabilidade e programação; em 8.6.4 estuda-se a função de pós-processamento ligada à reescrita final e revisão de concordâncias. Ao longo dos vários exemplos, será analisada a questão do tratamento das palavras desconhecidas e aprendizagem assistida.

### 8.6.1 Descrição sumária da biblioteca TEXT::TRANSLATE

Conforme se disse, em vez de tentar resolver o problema geral da tradução, optou-se por construir uma biblioteca de funções (bastante pequena e ligeiramente naïf) para ajuda à tradução, a ser usada para programar ferramentas para resolução de problemas envolvendo situações, formatos e convenções específicas.

TEXT::TRANSLATE contém funções para as seguintes tarefas:

– substituir sequências de palavras por sequências de palavras (baseado em dicionários completamente textuais), escolhendo as maiores sequências em primeiro lugar,
– usar hierarquias de dicionários do utilizador (permitindo que se configure uma estratégia de procura contemplando consultar primeiro os dicionários mais específicos da área em causa, e posteriormente os dicionários mais genéricos),
– definir o tratamento a dar às palavras desconhecidas (Ex. mantê-las inalteradas; acrescentar-lhes um prefixo de aviso; perguntar ao utilizador;), e construir a lista dessas palavras,
– definir zonas do documento que não devem ser tocadas (através de uma lista de expressões regulares que delimitem essas zonas),
– suportar funções de pré e pós-processamento – para alterar detalhes textuais (formatos, correcções morfo-sintácticas, etc) baseadas em sistemas de reescrita.

Para descrição da biblioteca TEXT::TRANSLATE, começar-se-á pela apresentação de um programa exemplo, simples e anotado, para tradução de texto informático Português – Inglês.

**Exemplo 1:** Programa de tradução

Considere-se o seguinte programa de tradução:

```
1 use text::translate;
2 trans_dic("pt2en.dic","inf_pt2en.dic","pessoa.dic");
3 trans_prefix("@");
4 trans_und_function("JUSTPREFIX");
5 trans_und("en2pt.und");
6 trans_dont_touch('<.*?>');

7 while(<>) {print trans($_);}
```

Notas:

**linha 1** - Importar a biblioteca `text::translate`.

**linha 2** - Definição da lista de dicionário a usar (a ordem é relevante: últimos dicionários têm prioridade sobre os primeiros).

**linha 3, 4 e 5** - Tratamento de palavras desconhecidas: são assinaladas com o prefixo '@' e guardadas em `en2pt.und`; não há interacção com o utilizador.

**linha 6** - Ignora a marcação HTML.

**linha 7** - Traduzir as linhas do texto.

### Dicionários tradutores

O TEXT::TRANSLATE pode usar uma sequência de dicionários. As definições contidas nos últimos dicionários têm prioridade sobre os primeiros. Deste modo o utilizador pode ter um dicionário com as definições mais gerais e eventualmente reescrevê-las num dicionário de domínio mais específico que seja posto no final da lista de dicionários.

**Exemplo 2:** Tradução de termos específicos (*man*)

Em textos gerais, a palavra inglesa *man* pode ser traduzida por *homem*. No entanto, se os textos a traduzir forem acerca do sistema operativo Unix, é mais provável que seja a abreviatura de *manual pages*, sendo neste caso aconselhável a tradução *manual*. Supondo que a primeira tradução tenha sido guardada em `general.dic` e a segunda em `unix.dic`, a seguinte sequência de dicionários seria correcta:

```
trans_dic("general.dic","unix.dic");
```

**Exemplo 3:** Formato dos dicionários

Considere-se o seguinte extracto de um dicionário de tradução Português - Inglês

```
1  # Marcações especiais
2  #  #1 - usado para ajectivos (alteração de ordem)
3  #  #2 - nomes próprios       (o João => João)
4  #  #3 - "a"                  (an elephant, a table)
5  actividades de investigação=research activities
6  alguma=some
7  algumas=some
8  caso a caso=each case
9  caso=case
10 ciências da computação=computer science
11 corrector ortográfico=spell checker
12 experimental=experimental#1
13 foi=has been
14 habilitações académicas=educational qualifications
15 identificação=personal information
16 já não é=is no longer
17 o melhor=the best
18 melhor=better
19 o=the
20 à=to the
21 Francisco=Francisco#2
22 um=a#3
23 num=in a#3
```

Notas:

**linha 5 a 23** - Cada linha consiste num par de sequências de palavras.

**linha 6, 7** - As regras 1-1 são usadas para o sentido mais comum da palavra.

**linha 8-9 e 17-18** - Em vários casos uma palavra pode ter várias traduções, mas o contexto por vezes permite resolver essas ambiguidades.

**linha 12, 21, 22 e 23** - Em vários casos usa-se o símbolo #... para marcar necessidade de pós-processamento adicional – referente a certos fenómenos como sejam questões ligadas a concordâncias, ou ligadas a alteração de ordem.

**linha 12** - #1 usado para marcar adjectivos (tipicamente a ordem de colocação do adjectivo difere entre Portugês e Inglês).

**linha 21** - #2 usado para marcar nomes próprios (frequentemente os nomes próprios levam artigo definido no Português e no Inglês não).

**linha 22 e 23** - #3 usado para marcar o artigo indefinido *a* (que é alterado para *an* quando o palavra seguinte começa por vogal).

**Função `trans_dont_touch`**

Como foi referido anteriormente, a função `trans_dont_touch` permite definir (através de expressões regulares) as partes do texto que não devem ser traduzidas.

Esta função é necessária para proteger os elementos não palavra (previamente designados por ENL) que aparecem em todos os texto reais (Ex. Marcações HTML, texto matemático, programas, etc).

### 8.6.2 Tratamento de palavras desconhecidas com DDMF

Quando uma palavra não tem tradução definida nos dicionários (palavra desconhecida), várias estratégias estão previstas no módulo TEXT::TRANSLATE.

Referiremos apenas uma das mais complexas, por se tratar de um dicionário dinâmico multi-fonte:

- as formas de procedimento são:
  - f1 – usar o tradutor automático `babelfish` (Babelfish, 2000), ou seja, enviar uma expressão de pesquisa ao tradutor, receber o ficheiro html com a resposta, extrair a resposta e, se ela for diferente do termo cuja tradução foi pedido, devolvê-la.
  - f2 – se associado ao termo existir um manual em Unix (man), considerar que o termo não deve ser alterado.
  - f3 – perguntar interactivamente como se deve traduzir o termo, guardando a respectiva tradução no dicionário *dict*
- a estratégia a usar é a de perguntar por ordem até que uma fonte responda (definida na Secção 5.2.1)

$$f1(t) \stackrel{\text{def}}{=} trInBabelfish(t)$$

$$f2(t) \stackrel{\text{def}}{=} \begin{cases} man(t) \neq empty & \Rightarrow \quad t \\ \underline{otherwise} & \Rightarrow \quad () \end{cases}$$

$$f3 : termo \longrightarrow inf$$
$$f3(t) \stackrel{\text{def}}{=} \begin{array}{l} \underline{let} \quad a = lerstr(t \frown \texttt{"?"}) \\ \underline{in} \quad \underline{post} \quad dict' = dict \dagger \left(\begin{pmatrix} t \\ a \end{pmatrix}\right) \\ \qquad \underline{in} \quad a \end{array}$$

$$estrategia(t, ef) \stackrel{\text{def}}{=} pergPorOrdem(<'man,' babel,' interact>, t, ef)$$

$$desconhecidas' = ddmf(\left(\begin{pmatrix}'babel\\ f1\end{pmatrix}\begin{pmatrix}'man\\ f2\end{pmatrix}\begin{pmatrix}'interact\\ f3\end{pmatrix}\right), estrategia)$$

Uma versão mais sofisticada com aprendizagem supervisionada (dicionário com cache), é descrito no exemplo 5, página 273.

Refira-se que a utilização, em regime experimental de alguns tradutores usando esta abordagem, levaram a que os utilizadores criassem alguns dicionários bilingues sem se aperceberem.

### 8.6.3 Exemplos

Nesta subsecção, apresenta-se dois exemplos de utilização de TEXT::TRANSLATE com características muito distintas.

No primeiro exemplo, traduz-se um programa C – traduzindo apenas as mensagens. Este exemplo pretende mostrar a utilidade de poder ter controle sobre o programa de tradução.

No segundo exemplo, usa-se o TEXT::TRANSLATE para realizar uma falsa tradução: a acentuação de um texto não acentuado. O objectivo deste exemplo é mostrar que uma biblioteca de tradução pode ser útil num leque mais vasto de situações.

**Exemplo 4:** Tradução das mensagens de um programa C

A tradução de um programa C não pode ser conseguida através da tradução da totalidade das palavras nele contido. Somente as *strings* deverão ser traduzidas.

O programa Perl que se segue, baseado em TEXT::TRANSLATE, poderia ser usado para implementar essa tradução[13]:

```
#!/usr/bin/perl
use text::translate;

trans_dic("en2pt.dic");     # Dicionário
trans_prefix("@");
trans_und("en2pt.und");

trans_dont_touch('\\%(\w+)');

while(<>){
```

[13]Na realidade, haveria que ter em conta uma série de outros detalhes C...

```
8      s/".*?"/trans($&)/ge;
9      print; }
```

Notas:

**linha 4 e 5** - As palavras desconhecidas são prefixadas com `@` e guardadas em `en2pt.und`.

**linha 6** - Não traduzir as especificações de formato (usadas, por exemplo, no `printf`).

**linha 8** - Sempre que uma string aparece, substituí-la pela sua tradução.

Quando aplicado ao seguinte ficheiro C:

```
1  main(){ char a[]="the main goal is",
2               b[]="make miracles",
3               c[]="use this tool";
4    if (...) printf("if %s %s then %s",a,b,c) ...  }
```

Produz o seguinte resultado:

```
1  main(){ char a[]="o objectivo principal é",
2               b[]="construir milagres",
3               c[]="usar esta ferramenta";
4    if (...) printf("se %s %s então %s",a,b,c) ...  }
```

Notas:

**linha 1, 4** - `main`, `char`, `if`, não foram tocadas, embora `if` dentro da string tivesse sido traduzida.

**linha 4** - `%s`, dentro da string de formato do `printf`, também não foi alterado devido à invocação `trans_dont_touch` anterior.

Naturalmente, não seria prático usar um tradutor convencional num problema como o acabado de descrever.

Este exemplo demonstra que há utilidade em dispor de um conjunto de funções, com as quais seja fácil construir ferramentas adaptadas ao problema em causa.

Devido a várias razões, é frequente aparecer textos sem acentuação[14]. Para que se possa tirar total partido da informação neles contida, esses textos precisam de sofrer um processo de reciclagem, o que pode envolver um enorme esforço manual.

No próximo exemplo, esquematiza-se uma ferramenta de ajuda neste processo de acentuação.

[14]Há alguns anos atrás, o uso de acentos trazia problemas informáticos em certos ambientes.

**Exemplo 5:** TEXT::TRANSLATE para reciclagem de textos não acentuados

Na primeira fase, vai ser construído um dicionário de acentuação. A estratégia seguida é: partindo dum corpus de dimensões razoáveis, para cada palavra acentuada, guardar num dicionário uma correspondência entre a sua versão desacentuada e a versão correcta.

$desac : palavra \longrightarrow palavra$

$desac(p) \stackrel{\text{def}}{=}$ ...p após desacentuada

$$
\begin{array}{l}
makedict : corpus \longrightarrow dict \\
makedict(c) \stackrel{\text{def}}{=} \\
\quad \underline{let} \quad pals = palavras(c) \\
\quad \underline{in} \quad \left( \begin{array}{c} desac(p) \\ p \end{array} \right)_{p \in pals \wedge desac(p) \neq p}
\end{array}
$$

$dicAcent \stackrel{\text{def}}{=} makedict($ "Corpus"$)$

O dicionário de acentuação criado, inclui, por exemplo, as seguinte entradas:

```
Varias=Várias
estrategias=estratégias
sao=são
possiveis=possíveis
```

Várias estratégias são possíveis para acentuar[15]. Seguidamente, mostra-se um programa Perl que acentua texto não acentuado, por simples tradução usando o dicionário de acentuação atrás descrito.

```
1 use text::translate;
2 trans_dic("dicAcent.dic");
3 trans_und_function("ID");
4 while(<>){ print trans($_); }
```

Notas:

**linha 2** - Usar o dicionário `dicAcent` acima calculado.

**linha 3** - Usar a função identidade como estratégia para tratar as palavras desconhecidas (i.e., para as palavras não existentes no dicionário de acentuação).

**linha 4** - Traduzir as linhas (ou seja, acentuar).

Quando a acentuador é aplicado ao seguinte texto,

*... Varias estrategias sao possiveis para acentuar...*

[15] Por exemplo acentuar todas as palavras desconhecidas (não existentes num dicionário como o `ispell` ou `jspell`) que existam no dicionário de acentuação.

produz o seguinte resultado,

*... Várias estratégias são possíveis para acentuar...*

Este exemplo demonstra que há utilidade em dispor de formatos de armazenamento de dicionários que sejam textuais e simples. Deste modo, foi possível pôr outras ferramentas a gerá-los. Mostra, também, que funções de tradução (ou de substituição textual...) têm cabimento numa bancada de tratamento de texto, mesmo que seja para actividades diferentes da tradução convencional.

### 8.6.4 Sistema de reescrita de pós-processamento

Em tradução, ou em qualquer ferramenta que tenha geração de língua natural, é habitual haver um bloco de pós-processamento que se encarregue dos fenómenos de superfície do texto gerado (exemplo: concordâncias e movimentos de superfície).

No caso particular do TEXT::TRANSLATE, a não existência dum bloco de pós-processamento obrigaria a escrever um maior número de regras de tradução.

Considere-se o seguinte dicionário de tradução:

```
a dog=um cão
the dog=o cão
the dogs=os cães
some dogs=alguns cães
no dog=nenhum cão
a big dog=um cão grande
a small dog=um cão pequeno
the big dog=o cão grande
...

a table=uma mesa
the table=a mesa
the tables=as mesas
no table=nenhuma mesa
some tables=alguns mesas
a big table=uma mesa grande
a small table=uma mesa pequena
the big table=a mesa grande
...
```

Como se observa, o número de regras envolvidas na implementação de concordância e mudanças de ordem, tende a ser elevado.

Uma solução melhor seria:

```
a=um#9
the=o#9
some=algum#9
no=nenhum#9
big=grande#1
small=pequeno#1
dog=cão
dogs=cães
table=mesa
tables=mesas
...
```

Onde as regras descrevem a tradução básica e fazem a marcação de fenómenos vários para facilitar a escrita do bloco de pós-processamento.

TEXT::TRANSLATE tem um nível de pós-processamento que é uma função gerada por MKTEXTRR com base num conjunto de regras de reescrita.

Para activar o pós-processamento, um programa tradutor invoca a função `trans_ppp("en2pt.rr")`, sendo `en2pt.rr` o nome do ficheiro onde estão definidas as regras de reescrita.

**Exemplo 6:** Extractos dum pós-processador

Neste exemplo, apresenta-se um extracto de um pós-processador dum tradutor Inglês - Português escrito em notação MKTEXTRR e que trata da concordância *artigo-nome.*

```
RULES posproc

em ([ao]s?)==>n$1

o#9\s+($work)==>o $1!!       gn_is({G=>m,N=s},$1)
o#9\s+($work)==>os $1!!      gn_is({G=>m,N=p},$1)
o#9\s+($work)==>a $1!!       gn_is({G=>f,N=s},$1)
o#9\s+($work)==>as $1!!      gn_is({G=>f,N=p},$1)
um#9\s+($work)==>um $1!!     gn_is({G=>m,N=s},$1)
um#9\s+($work)==>uns $1!!    gn_is({G=>m,N=p},$1)
um#9\s+($work)==>uma $1!!    gn_is({G=>f,N=s},$1)
um#9\s+($work)==>umas $1!!   gn_is({G=>f,N=p},$1)
```

```
11 ENDRULES

12 sub gn_is{my ($constraint,$word)=@_;
13   onethat($constraint,fea($word));
14 }
```

Notas:

**linha 2** - Exemplo de uma regra para realizar contracções – converte `em os` em `nos`.

**linha 3 a 10** - Regras de reescrita condicionais para cálculo da concordância artigo – nome.

**linha 4** - Exemplo de uma regra para calcular concordâncias de artigos com nomes masculino, plural[16] – converte `o#9 cães` em `os cães`.

**linha 12 a 14** - A função `gn_is` verifica se há alguma análise da palavra `word` que concorde com as restrições `constraint` impostas.

permitindo que

*... the dicionary on the table ...*

seja traduzido por

*... o dicionário na mesa ...*

[16]Na realidade funciona com qualquer outra palavra masculino, plural (exemplo pronomes possessivos, adjectivos, etc).

# Capítulo 9

# DDMF - uma implementação

*A programming language is a tool that has a profound influence on our thinking habits.*

*Dijkstra*

Na sequência dos capítulos anteriores, apresenta-se neste capítulo um módulo que ajuda à construção e manuseamento de DDMF.

De acordo com a filosofia proposta, este conjunto de funções pretende apenas ser um suplemento à funcionalidade existente nos sistemas operativos e constituir genericamente peças que ajudem a compor os vários recursos e funções existentes.

Nesta implementação, pretendeu-se estudar a complexidade da construção de DDMF reais, com especial preocupação em:

- obter rapidez de desenvolvimento (tentar escolher soluções simples, ainda que por vezes de menor eficiência na execução),
- fornecer algum tipo de sofisticação,
- prever funcionamento com escalas reais,
- conseguir uma versão testável inicial o mais cedo possível, e funcionar em regime open-source `cvs`.

Este módulo está escrito em Perl devido à necessidade de boas capacidades de ligação a outras ferramentas.

Este capítulo surge na dissertação devido a considerar-se que:

- o módulo DDMF constitui em si um recurso útil
- este módulo permite experimentar e avaliar a abordagem proposta

- alguns dos exemplos/experiências que se apresentam, constituem uma história que por si só parece ter uma moralidade própria

Neste capítulo, começa-se por uma secção onde se apresenta a interface sumária do módulo, para seguidamente se incluir uma secção por cada grupo de operações, onde se mostram exemplos de uso das funções do grupo.

## 9.1 Interface sumária

Das várias espécies envolvidas neste módulo a principal é o dicionário – DDMF, que contém um estado interno, meta-informação e algumas funções – a função estratégia, a função domínio (só existente em alguns casos) e outras que não referiremos por questões de simplicidade.

Este modelo tem pequenas diferenças em relação ao apresentado na Secção 5.1, resultante de:

- ser preciso concretizar alguns elementos ligados à meta-informação (questão referida em 4.2 e 5.2.2) – a escolha da metadata adequada, tem implicitamente associado uma intenção de utilização; para além duma componente tradicional (como por exemplo o esquema descrito no Dublin Core (Weibel, 2003)) que define a informação de catalogação, há necessidade de dispor de informação adicional que é imprescindível para certo tipo de processamento.
- ser útil prever (possibilidade de) definir/consultar o domínio do dicionário.
- ser necessário definir um estado onde se guarde informação variada que seja necessária à implementação de algumas funções.

$$\begin{array}{lcrcll}
ddmf & = & estrat & : & termo \times estado \to ef & \times \\
 & & meta & : & metaFontInf & \times \\
 & & estado & : & any & \times \\
 & & dominio & : & \to set(term) &
\end{array}$$

| | | |
|---|---|---|
| $termf$ | = | ...Texto com um termo por linha |
| $dbdic$ | = | ...Dicionário em formato MLDBM file |
| $dpl$ | = | ...Dicionário em formato DPL |
| $txt$ | = | ...Texto |
| $tabtxt$ | = | ...Tabela – texto com registos e campos |
| $filert$ | = | ...Texto em que se acrescentou a cada linha as formas lematizadas |
| $thesaurus$ | = | ...Thesaurus |
| $diglib$ | = | ...Biblioteca Digital (library::*) |
| $ef$ | = | ...estrutura de facetas |

$$
\begin{aligned}
&consulta : termo \times ddmf \longrightarrow ef\\
&consulta(t,d) \stackrel{\text{def}}{=}\\
&\quad estrat(d)(d,t)
\end{aligned}
$$

A função consulta, difere ligeiramente da apresentada em 5.1, já que se pretende que a estratégia de consulta possa usar a meta-informação, o estado, e todas as outras componentes de DDMF.

O nome de algumas das estratégias e respectivos construtores, diferem dos referidos no Capítulo 5, no sentido de usar a língua Inglesa de modo a facilitar o seu uso por uma comunidade mais vasta.

Há muitas maneiras de armazenar um dicionário. Nesta abordagem, houve a preocupação natural de:

- nos concentrarmos num conjunto de hipóteses ricas que funcionem como formas canónicas
- e de construir (ou esperar que outros construam) ferramentas que traduzam outros formatos para os canónicos.

Alguns dos detalhes que se refere, dizem respeito apenas à versão actual, podendo vir a ser alterados futuramente e são referidos por constituírem um dado a ter em conta nas questões de escala.

Quando surgiu a necessidade de armazenar dicionários em extensão e com a preocupação de obter um dicionário que possa ser facilmente consultável e que (quase) não tenha limitações de escala, optou-se por uma representação baseada em árvores-B.

Dado que se pretende que a um termo possa estar associada informação de estrutura complexa (EF), escolheu-se o sistema de arquivo em ficheiro `MLDBM` (Sarathy, 1998) que internamente armazena em árvores-$B^+$ (Berkeley DB_File (Marquess, 1995)), tendo como chaves termos simples e como informação uma string que é uma expressão Perl cujo `eval` dá uma estrutura generalizada, capaz de cobrir as EF desejadas.

O `MLDBM.pm` é um módulo Perl que permite encapsular os detalhes de implementação constituindo um razoável ambiente de base. É usado através do mecanismo de ligação (`tie`) a um array associativo[1], permitindo que o código seja mais legível.

Várias outras hipóteses poderiam ter sido consideradas como forma canónica para armazenar de modo eficiente dicionários de dimensão potencial-

[1] Também chamado HASH na documentação Perl, funções finitas na documentação CAMILA.

mente grande.

Nas funções que se apresenta nas secções seguintes, quando forem construídos DDMF, apresentaremos apenas a parte mais relevante: a função de estratégia de consulta associada ao DDMF resultado. É, no entanto, relativamente fácil *adivinhar* quais as outras componentes do DDMF.

Sempre que nos exemplos se apresentar o código associado, em vez de Perl mostra-se uma versão aligeirada, em que se retiraram os marcadores de variável, de lista e de mapping (\$,@,%,&), separadores, aspas, de modo a tornar a leitura mais leve mas tentando deixar o necessário a permitir sentir a sintaxe concreta e a dimensão real do programa.

**Construção de DDMF a partir de elementos diversos (dicionarização):**

À semelhança do que se fez em 5.2.1, sempre que possível, vai-se definir construtores de DDMF para facilitar a composição funcional.

As funções que se referem seguidamente, são construtores de DDMF a partir de informação de vários tipos. Cada uma delas cria um DDMF, registando na meta-informação ou no estado as suas capacidades implícitas (ex: ser ou não capaz de calcular o domínio), a sua meta-informação e criando pelo menos uma função de estratégia de consulta. A generalidade das funções recebe um parâmetro opcional (`opt`), onde se agrupa um conjunto de dados: a meta-informação e os parâmetros adicionais que forem necessários (ligados à configuração).

$mkddmf : term \longrightarrow ef \times opt \longrightarrow ddmf$
$mkddmf(f, o) \stackrel{\text{def}}{=}$ ... Constrói um ddmf a partir de uma função

$mknlgrepdd : txt \times opt \longrightarrow ddmf$
$mknlgrepdd(t, o) \stackrel{\text{def}}{=}$
... Constrói um ddmf a partir de um texto, usando nlgrep

$mkmttab : tabtxt \times opt \longrightarrow ddmf$
$mkmttab(t, o) \stackrel{\text{def}}{=}$ ... Constrói um ddmf a partir de uma tabela textual

$mkdplddmf : dpl \times opt \longrightarrow ddmf$
$mkdplddmf(d, o) \stackrel{\text{def}}{=}$ ... Constrói um ddmf a partir de uma texto dpl

$mkdbddmf : dbdic \times opt \longrightarrow ddmf$
$mkdbddmf(d, o) \stackrel{\text{def}}{=}$
... Constrói um ddmf a partir de um dicionário em formato MLDBM

$mktheddmf : thesaurus \times opt \longrightarrow ddmf$
$mktheddmf(d, o) \stackrel{\text{def}}{=}$ ... Constrói um ddmf a partir de thesaurus

$mkdlddmf : diglib \times opt \longrightarrow ddmf$
$mkdlddmf(d, o) \stackrel{\text{def}}{=}$ ... Constrói um ddmf a partir de uma biblioteca digital

**Funções relacionadas com o domínio:**

As funções relacionadas com o domínio têm grande importância prática, devido a permitirem converter dicionários dinâmicos em estáticos. São também o caminho necessário para converter dicionários com domínios potencialmente infinitos em formatos necessariamente limitados, como sejam os formatos impressos.

$domrestrict : ddmf \times set(term) \times opt \longrightarrow ddmf$
$domrestrict(d, s, o) \stackrel{\text{def}}{=}$
... Constrói um ddmf por restrição ao domínio de um ddmf

$txt2terms : txt \times opt \longrightarrow set(term)$
$txt2terms(t, o) \stackrel{\text{def}}{=}$ ... extrai um conjunto de palavras a partir de um texto

$dbdic2terms : dbdic \times opt \longrightarrow set(term)$
$dbdic2terms(t, o) \stackrel{\text{def}}{=}$
... extrai um conjunto de palavras a partir de uma base de dados

**Funções relacionadas com a composição de DDMF**

A capacidade de formar um dicionário por composição de dicionários é um dos elementos centrais da tese.

$mkparcom : set(ddmf) \times opt \longrightarrow ddmf$
$mkparcom(ds, o) \stackrel{\text{def}}{=}$ ... Composição paralela de ddmf

$mktagparcom : set(ddmf) \times opt \longrightarrow ddmf$
$mktagparcom(ds, o) \stackrel{\text{def}}{=}$ ... Composição paralela etiquetada de ddmf

$mksercom : ddmf^{\star} \times opt \longrightarrow ddmf$
$mksercom(ds, o) \stackrel{\text{def}}{=}$
... Composição série de ddmf: pergunta até que um ddmf responda

$mktagsercom : ddmf^{\star} \times opt \longrightarrow ddmf$
$mktagsercom(ds, o) \stackrel{\text{def}}{=}$
... Composição série etiquetada de ddmf:
pergunta até que um ddmf responda

$mkddmfrr : ddmf \times rewriting \times opt \longrightarrow ddmf$
$mkddmfrr(td, r, o) \stackrel{\text{def}}{=}$
... extrai um conjunto de termos a partir de um texto

$mkenritch : ddmf \times set(ddmf) \times opt \longrightarrow ddmf$
$mkenritch(d, ds, o) \stackrel{\text{def}}{=}$
... para todos os termos do domínio de d, enriquece a sua
... informação com consultas a ds

$mkcachedddmf : ddmf \times str \times opt \longrightarrow ddmf$
$mkcachedddmf(d, cachename, o) \stackrel{\text{def}}{=}$
... dado um ddmf, cria um dicionário cache para optimização

### Funções relacionadas com a animação de DDMF

As funções que aqui se enunciam têm a ver com o uso dos DDMF. Para além de funções ligadas à consulta, existe um conjunto de funções que traduzem para outros formatos e que permitirão usar directa ou indirectamente os dicionários.

$intrepret : ddmf \longrightarrow$
$intrepret(d) \stackrel{\text{def}}{=}$
...comando de consulta a dicionário (ler; pesquisar; mostrar;)*

$mklatex : ddmf \times set(term) \longrightarrow$
$mklatex(d, ts) \stackrel{\text{def}}{=}$ ...Constrói um dicionário LateX usando dicttex

$mkdictprotocol : ddmf \times set(term) \longrightarrow$
$mkdictprotocol(d, ts) \stackrel{\text{def}}{=}$
...Constrói um dicionário de acordo com o formato DICT

$mktei : ddmf \times set(term) \longrightarrow$
$mktei(d, ts) \stackrel{\text{def}}{=}$
...Constrói um dicionário de acordo com o formato TEI/XML

## 9.2 Funções de dicionarização – $any \to ddmf$

O conjunto de funções de *dicionarização* destina-se a criar um vista idêntica para um conjunto heterogéneo de objectos.

Apresenta-se de seguida alguns exemplos que permitem entender melhor o modo de usar algumas das funções do módulo. Em certos casos, esses exemplos apresentam experiências práticas interessantes.

### 9.2.1 Dicionários a partir de funções

#### `mkddmf` - make dicionário dinâmico

**Exemplo 1:** Construção de um dicionário com base na função `jspell` fea

Para construir uma ferramenta que, usando o `jspell`, calcule as análises morfológicas de todas as palavras que forem introduzidas, podemos escrever o seguinte comando[2]:

```
1  use ddmf
2  use jspell
3  jspell_dict("port")

4  dic = mkddmf(\*fea,{id=>"jspell-fea"})
```

Notas:
**linha 3** - Usar o dicionário Português[3].
**linha 4** - Construir um DDMF, tendo como base a função `fea` do módulo `jspell`. O DDMF construído está a ter um identificador `jspell-fea` (meta-informação, que neste caso não está a ser utilizada).

[2] Um exemplo semelhante foi apresentado na Secção 5.4 mas sem explicações.
[3] Em próximas versões será implementado um parâmetro opcional com uma função de inicialização onde esta linha seria *arrumada*.

O dicionário construído pode ser usado de diferentes modos. No programa exemplo seguinte, está a ser criado um comando que permite fazer consultar interactivamente o dicionário, usando para isso a função **interpret** – que, dado um DDMF, aceita termos e apresenta a informação associada.

```
use ddmf
use jspell
jspell_dict("port")

dic = mkddmf(\*fea,{id=>"jspell-fea"})
interpret(dic)
```

A sua execução produz um diálogo como o que se segue (cujos detalhes não comentaremos):

```
1  > rio
2    [ { N   => 's',
3        G   => 'm',
4        CAT => 'nc',
5        rad => 'rio' },
6      { N   => 's',
7        P   => '1',
8        CAT => 'v',
9        rad => 'rir',
10       T   => 'p',
11       TR  => '_' } ]

12 > rebentamento
13   [ { N    => 's',
14       G    => 'm',
15       FSEM => 'mento',
16       CAT  => 'nc',
17       rad  => 'rebentar',
18       T    => 'inf',
19       TR   => '_' } ]
```

Notas:

**linha 1 e 12** - Termos (perguntas) introduzidos pelo utilizador.

**linha 2 a 11 e 13 a 19** - Respostas do dicionário.

### 9.2.2 Dicionários a partir de textos

`mknlgrepdd` – **make nlgrep** DDMF

Há mais do que uma maneira de ver um texto sob o prisma de dicionário. Um dos modos mais usuais corresponde ao comando unix `grep` ( e suas variantes exemplo `agrep` (Wu and Manber, 1992)): ver a consulta como uma função que dada um termo, devolva um conjunto de elementos (frases, linhas, registos, etc.) em que ela ocorra. As várias variantes correspondem a diferentes tipos *elementos* e a diferentes tipos de expressões de pesquisa (palavra, expressão regular, expressão regular estendida com morfologia, etc).

A função `mknlgrepdd` corresponde a uma abreviatura de criar um DDMF com a função nlgrep referida em 6.1:

$$mknlgrepdd : txt \times opt \longrightarrow ddmf$$
$$mknlgrepdd(f, o) \stackrel{\text{def}}{=}$$
$$mkddmf(\lambda\, a\, .\, nlgrep(a, f, o))$$

Podendo ser invocada com opções que indiquem:

- Língua a usar na morfologia,
- número máximo de respostas pretendidas,
- separador de registo a considerar no texto `t`,
- etiqueta a usar,
- etc.

```
use ddmf
interpret(mknlgrepdd("f",{lang=> "pt",
                          id  => "Frases exemplo",
                          tag => "Fra",
                          max => 20}))
```

A sua execução produz:

```
> rebentar
 Fra => [
      Na Primavera rebentam os botões das flores.
      A corda rebentou com o excesso de peso.
      Quando o corredor entrou no estádio, rebentou uma...
      A câmara fez cortar as ... a rebentar o pavimento.
      Rebentou uma conduta da rede de distribuição de água à cidade.
      O prédio ficou todo inundado, quando a canalização rebentou.]
```

### 9.2.3 Dicionários a partir de tabelas textuais

#### `mkmttab` – tabela multi-termo

Chamaremos tabelas textuais a ficheiros de texto contendo registos que por sua vez são compostos por campos.

Considere-se, a título de exemplo, a seguinte tabela textual, constituída por linhas (registos) com campos (separados por ':'): termo em Português, termo em Inglês e proveniência.

```
jornal:news paper:freedict project
boas notícias:good news:freedict project
ciência:science:unesco
ciências da computação:computer science:
```

A criação de dicionários a partir de tabelas é aqui definida como sendo um caso especial do construtor de dicionários a partir de texto, em que cada *registo* encontrado tem uma estrutura mais rica, dando origem a uma estrutura de facetas mais complexa.

$$
\begin{array}{l}
mkmttab : txt \times opt \times str^{\star} \longrightarrow ddmf \\
mkmttab(f, op, campos) \stackrel{\mathrm{def}}{=} \\
\quad \lambda\, t\, .\, < li2fs(l, sep(op), campos) \mid l \in nlgrep(t, f) >
\end{array}
$$

onde a função `li2fs` quebra a linha pelo separador de campo `sep(op)`, juntando ao resultado se o valor correspondente aos campos não vazios.

$$
\begin{array}{l}
li2fs : str \times str \times str^{\star} \longrightarrow FS \\
li2fs(line, sep, campos) \stackrel{\mathrm{def}}{=} \\
\quad \underline{let} \;\; a = split(sep, line) \\
\quad \underline{in} \;\; \begin{pmatrix} campos(i) \\ a(i) \end{pmatrix}_{i \in inds(a) \wedge a(i) \neq \texttt{""}}
\end{array}
$$

Seguidamente, apresenta-se um programa que cria um interpretador de expressões de consulta, correspondente à tabela acima mostrada:

```
interpret(
   mkmttab(
       "f1",
       { sep  => ":",
         type => "ff",
```

```
6            id   => "pt-en",
7            lang => "pt"
8          },
9          ["Português", "English", "Origem"] );
```

Notas:

**linha 3** - Indica que o ficheiro `f1` vai ser o ficheiro tabela textual analisado.

**linha 5** - Define que se pretende ver o conjunto de respostas agrupado segundo a variante `ff` – correspondente a associar ao primeiro campo (ou ao campo especificado como `key`, se definido) a informação total de cada registo.

**linha 9** - Indica a lista dos identificadores de campo que compõe cada registo.

A execução do respectivo interpretador têm o seguinte aspecto:

```
> ciência
 { ciências da computação =>  {
       Português => ciências da computação
       English => computer science}
   ciência =>  {
       Português => ciência
       English => science
       Origem => unesco}
 }
```

Repare-se que o primeiro registo tem o campo proveniência vazio porque o correspondente elemento da tabela também era vazio.

Repare-se também, que como foi dada a opção de língua, a pesquisa está a ser feita de modo a incluir variante morfológicas da palavra pesquisada.

### 9.2.4 Pesquisa por padrões a partir árvores-B

`mkdbddmf` – **make database** DDMF

Como foi atrás referido, os ficheiros `MLDBM`s oferecem acesso directo à informação a partir do termo (consulta por termo).

Para se obter consulta por padrões é necessário algum trabalho adicional. A função `mkdbddmf` enriquece um dicionário `MLDBM d` do seguinte modo:

– extrai o domínio da árvore-$B^+$ correspondente, armazenando-o num ficheiro de texto `d.wrdlst`, um termo por linha.

– se for pretendida pesquisa morfológica, é calculado um ficheiro do tipo filert `d.wrdlstnl` que acrescenta a cada linha o conjunto de palavras normalizadas (lematizadas) correspondentes a cada palavra existente no termo.

$$
\begin{aligned}
&mkrt : term^{\star} \longrightarrow termrt^{\star}\\
&mkrt(f) \stackrel{\text{def}}{=}\\
&\quad <t \frown \texttt{"@"} \frown join(\texttt{""}, <rad(p) \mid p \in t>) \mid t \in f>
\end{aligned}
$$

A consulta com padrões morfológicos é realizada em 3 passos:

1. pesquisa do padrão no ficheiro `d.wrdlstnl`;
2. extracção dos termos a partir da resposta do primeiro passo;
3. consulta do dicionário para cada um desses termos extraídos

$$
\begin{aligned}
&consulta : padrao \times dbddmf \longrightarrow ef^{\star}\\
&consulta(p, d) \stackrel{\text{def}}{=}\\
&\quad \underline{let}\quad t1 = grep(p, tLstRt(d))\\
&\quad \phantom{\underline{let}}\quad t2 = <extraiTermo(t) \mid t \in t1>\\
&\quad \underline{in}\quad \begin{pmatrix} x \\ db(d)(x) \end{pmatrix}_{x \in t2}
\end{aligned}
$$

ou de um modo mais compacto

$$
\begin{aligned}
&consulta : padrao \times dbddmf \longrightarrow ef^{\star}\\
&consulta(p, d) \stackrel{\text{def}}{=}\\
&\quad db(d) \mid <extraiTermo(t) \mid t \in grep(p, tLstRt(d))>
\end{aligned}
$$

Assim, tomando como exemplo o dicionário de calão e expressões idiomáticas atrás descrito (ver Secção 8.2), a construção seguinte cria um interpretador com pesquisa morfológica de padrões:

```
 use ddmf
 interpret(mkdbddmf("dac",{lang=>"pt"}))
```

Resultando então a seguinte funcionalidade:

```
 > batata
{ ser* um (bananas|merdas|batata|...) =>[
     { genero => masculino
       sem => [ alguém que não tem vontade própria
                incapaz de ter uma postura masculina]
```

```
      gram => só usado no masculino}]

  batata =>  [
     { isa => alimento}
     { sem => nariz
       nivel => coloquial
       syn => [ penca]}]

  passar* a batata quente => []

  querias! batatinhas com enguias! =>  [
     { sem => exclamação de recusa}]}
```

Repare-se que *querias! batatinhas com enguias* foi encontrado devido a *batatinhas* ser um termo derivado de *batata* na língua em causa (pt – Português).

## 9.3 Funções relacionadas com o domínio

Vários dos construtores de dicionários apresentados não oferecem capacidade de calcular o domínio (i.e. o conjunto de termos que podem ser consultados).

Alguns dicionários têm um domínio potencialmente infinito. Isto surge, por exemplo, quando há funções envolvidas.

A possibilidade de controlar (restringir) o domínio é quase obrigatória quando queremos:

- fazer caching de dicionário dinâmicos
- pretendemos traduzir para outros formatos dicionários que sejam dinâmicos ou que de alguma maneira tenham domínios demasiado grandes para o fim em vista.

**Exemplo 2:** Conversão de dicionários Babylon para formatos abertos

Neste exemplo, vai ser feito um exercício de migração de formato de dados usando a seguinte estratégia:

- para ver os dados como um DDMF, vai-se criar uma função que consulte os dados através de uma pipe bidireccional sobre um comando externo.
- seguidamente, vai-se criar um dicionário como resultado de aplicar uma restrição ao domínio, usando um conjunto de restrição tão grande quanto possível.

Babylon (Babylon.com Ltd., 2000) é uma ferramenta ligada à tradução de termos e que disponibiliza um conjunto de dicionários em formato proprietário, bem como um conjunto de ferramentas para construção de glossários.

Existe disponível um pequeno programa C (`babylon-c`) que permite consultar um termo num dicionário Babylon, programa este que foi construído para permitir o uso dos dicionários Babylon dentro do editor Emacs.

Dado que o `babylon-c` é um recurso *open-source*, começou-se por estender a utilidade deste recurso através de ir buscar os ficheiros fonte e alterar uma parte mínima dos mesmos (9 linhas), de modo a ficar um comando dicionário que (usando a biblioteca readline) permitisse consultar todos os termos que se pretendesse, e com inclusão das comodidades habituais em interpretadores de comandos (edição de linha e história).

Seguidamente, construi-se uma vista DDMF sobre este comando, à custa de o consultar através de uma pipe bidireccional (usando o módulo (Wall, 2000, IPC::Open3)).

Por último, calculou-se a restrição do domínio a uma lista de palavras inglesa muito grande (/usr/share/dict/words – 45424 palavras) salvando o resultado num dos formatos canónicos (para ser posteriormente traduzido para os formatos que se desejar).

```
1  use IPC::open3;
2  open3( W,R,E,"babylon ... EngtoPor.dic");

3  babylon(x)={ ... escrever (W,x);
4               ... return (ler(R))}

5  d1 = mkddfm(babylon)

6  domrestrict( d1, {dictname=> "f1"}, "/usr/share/dict/words" );
```

Notas:

**linha 2** - Executar o comando `babylon` criando canais de comunicação com a `stdin`, `stdout` e `stderr` do processo respectivo.

**linha 3,4** - Definir uma função `babylon` que calcule a informação associada a um termo através de o enviar ao comando e ler a respectiva resposta. Estão a ser omitidos alguns detalhes desta função correspondentes à separação em campos.

**linha 5** - Criar um DDMF com e função `babylon`.

**linha 6** - Calcular o DDMF correspondente a determinar a imagem de todas as palavras inglesas de que se dispunha.

O dicionário, depois de convertido para TEI, contém entradas com o seguinte aspecto:

```
<entry>
  <term>
     <orth>abbess</orth>
     <pos>n.</pos>
  </term>
  <translation>madre superiora,
               a principal freira de um convento</translation>
</entry>
<entry>
  <term>
     <orth>abbey</orth>
     <pos>n.</pos>
  </term>
  <translation>abadia, mosteiro</translation>
</entry>
```

A título de mostrar o comportamento do módulo em exemplos reais, procedeu-se à medida da execução desse comando e respectiva tradução para formato de *printing dictionaries* XML/TEI[4], tendo-se obtido os seguintes resultados [5]:

| | |
|---|---|
| Número de palavras do conjunto de restrição | 81697 |
| Número de entradas extraída do dicionário (eng-por) | 44030 |
| Tempo demorado | 98 segundos |

## 9.4 Funções relacionadas com a composição de DDMF

A composição de DDMF, como foi atrás referido, pode ser feita de muitos modos, tendo em conta:

- a estratégia de consulta (escolhida em função da qualidade das respostas individuais, do custo da consulta, do excesso ou falta de informação, etc.)
- a maneira de conciliar as potencialmente várias respostas obtidas.

A maneira de conciliar as várias respostas depende do seu tipo. No caso da composição paralela, funções como:

- reescrita (plus) de sub-dicionários

[4]O conversor para dictionary XML/TEI, está ainda pouco desenvolvido e vai ser melhorado em próximas versões.

[5]Num Pentium III, em carga, 600MHz, 128Mbytes de RAM, linux.

– unificação de estruturas de facetas
– reunião de conjuntos de respostas (*mkparcom*)
– composição com etiquetação de proveniência (*mktagparcom*)

constituem exemplos de funções de conciliação naturais.

$$
\begin{array}{l}
mkparcom : set(ddmf) \times opt \longrightarrow ddmf \\
mkparcom(ds, o) \stackrel{\text{def}}{=} \\
\quad \underline{let} \quad f(t) = \{procura(i)(t) \mid i \in ds \wedge procura(i)(t) \neq ()\} \\
\quad \underline{in} \quad mkddmf(f, o)
\end{array}
$$

$$
\begin{array}{l}
mktagparcom : set(ddmf) \times opt \longrightarrow ddmf \\
mktagparcom(ds, o) \stackrel{\text{def}}{=} \\
\quad \underline{let} \quad f(t) = \begin{pmatrix} meta(i)(id) \\ procura(i)(t) \end{pmatrix}_{i \in ds \wedge procura(i)(t) \neq ()} \\
\quad \underline{in} \quad mkddmf(f, o)
\end{array}
$$

Os outros modos de composição de DDMF referidos no interface sumário, são idênticos. Antes de passar aos exemplos, descreve-se com mais detalhe a função de criação de DDMF com cache.

### 9.4.1 DDMF com cache

#### `mkcachedddmf` – make cached ddmf

A estratégia de este tipo de construtor consiste em:

– associar a uma cache a um DDMF
– a cache é também um DDMF
– perante uma consulta de um termo `t`:
    - primeiro procura-se na cache
    - e se não houver resposta, consulta-se `t` no dicionário, inserindo-se a resposta obtida na cache

ou seja: é a composição série:

– da cache
– com uma consulta ao dicionário com registo na cache:

$$
\begin{array}{l}
mkcachedddmf : ddmf \times str \longrightarrow ddmf \\
mkcachedddmf(d, cachename) \stackrel{\text{def}}{=} \\
\quad \begin{array}{ll}
\underline{let} & cache = mkdbddmf(cachename, \left(\begin{pmatrix} add \\ 1 \end{pmatrix}\right)) \\
& f(t) = \begin{array}[t]{ll} \underline{let} & r = consult(d)(t) \\ & i = add(cache)(t, r) \\ \underline{in} & r \end{array} \\
& daux = mkddmf(f) \\
\underline{in} & mksercom(cache, daux)
\end{array}
\end{array}
$$

### 9.4.2 Exemplos

**Exemplo 3:** Base para edição manual de um dicionário convencional

Neste exemplo, construiremos um esqueleto para definição de um dicionário usando a seguinte estratégia:

- baseando-nos num corpus de base, determinaremos as frequências associadas às formas lematizadas (para ajudar a escolher / propor as 60000 palavras a incluir no dicionário)
- baseando-nos no corpus de base, será construída uma base de frases exemplo (abonações) à custa de partir o corpus em frases e escolher as que têm dimensão própria para exemplificação (ex. de comprimento entre 30 e 80 caracteres)
- as palavras que pertencem ao conjunto de palavras do Português Fundamental (as cerca de 2000 palavras mais frequentes, incluindo as palavra estruturais e as palavras mais habituais) serão definidas manualmente, calculando neste caso apenas a sua fonética.
- as outras 60000-2000 palavras serão analisadas por várias fontes de informação sendo calculado:
  - a sua classe gramatical (usando o analisador morfológico `jspell`)
  - a sua fonética (usando Lingua::PT::speaker)
  - a sua frequência/confiança (informação extraída por `freqnormpt`)
  - um conjunto de 50 frases exemplo (usando nlgrep) existente num corpus de frases exemplo, para que dessas 50 frases se escolha algumas que sejam incluídas como frase exemplo, e para que ajudem a lembrar algumas acepções que possam ficar esquecidas.

```
1  use Lingua::PT::speaker;

2  sub fon1{ pal = shift;
3            {freq=>"alta", phon => toPhon(pal)}}

4  dptFun = domrestrict("ptfundamental", mkddmf( fon1 ))

5  system("freqnormpt *.xml | head -60000> l1");

6  system("divideFrases *.xml|awk 'length($0)<75' > l2");

7  dfon  = mkddfm(sub{ pal = shift;
8                      {phon => toPhon($pal)}});
9  dcat  = mkddfm(sub{ pal = shift;
10                     {cat => .....cat de fea($pal)...}});
11 dex   = mknlgrepdd( "l2",
12                     {lang=>"pt",max=>50} );
13 dfreq = mkmttab("l1",
14                 { sep  => "\t", type => "ff", },
15                 "term","freq","conf");

16 d3=mkcomser( dptFun, mkcompar(dfon,dcat,dex,dfreq))

17 .....Salvar d3 restricto a l1...
```

Notas:

**linha 1** - Importação do módulo `Lingua::PT::speaker` para poder seguidamente usar a função `toPhon` que faz a transcrição fonética de uma palavra.

**linha 2,3,4** - Criação de um dicionário *dptFun* que calcula a fonética em SAMPA, dizendo sempre que a frequência associada é "alta"; restringir este dicionário ao conjunto de termos contido no ficheiro das palavras do *Português Fundamental*.

**linha 5** - Cálculo das frequências lematizadas, ordenadas por termo e contendo número de ocorrências e grau de confiança.

**linha 6** - Divisão em frases seleccionando apenas as de tamanho conveniente.

**linha 7** - Criação de um DDMF que faz transcrição fonética.

**linha 9** - Criação de um DDMF que determina a categoria gramatical provável.

**linha 11** - Criação de um DDMF que dá frases exemplo para uma palavra.

**linha 13** - Criação de um DDMF que forneça as frequências de confiança associadas a uma palavra.

**linha 16** - Criação de um DDMF que se a palavra existe em d2 (Português Fundamental), dá essa informação; senão faz a composição paralela dos outros dicionários.

**linha 17** - Gravação em formato TEI/SGML ou em qualquer outro formato para se constituir um ponto de partida inicial a partir do qual se vai sucessivamente enriquecer a informação coleccionada.

Segue-se um pequeno extracto do dicionário produzido[6] (retirou-se a quase totalidade dos exemplos):

```
term => aldeia
freq => alta
phon => ald6:j6

term => amizade
freq => 110
cat  => nc
conf => (conf=100%)
phon => 6miza:d@
ex   => Há uma enorme amizade entre nós.
ex   => É um dever de amizade mostrar-lhe que está enganado.
ex   => As Nações Unidas procuram fomentar as relações de amizade
          entre os povos.

term => regular
freq => 110
cat  => adj
conf => (conf=82%)
phon => Regula:r
ex   => Os dois amigos escreviam-se regularmente.
ex   => A História também obedece a leis regulares.
ex   => É um autor com uma produção muito irregular.

term => impacientar
freq => 110
cat  => v
conf => (conf=30%)
phon => i~pasie~ta:r
ex   => A Rosa esperava impaciente a chegada do comboio, no cais.
ex   => Somos tão novos, diz o homem /
        e agora é a vez de a mulher se impacientar
ex   => Quando um barulho de cama / a voltar-se d'impaciente / ...
ex   => Mãos frementes e impacientes / Mãos desoladas e sombrias /
        Desgraçadas, doentias ...
ex   => Nota-se que está impaciente.
```

A informação aqui gerada pretende ser apenas um ponto de partida.

[6]Neste exemplo, foram usados alguns livros do Júlio Dinis, Miguel Torga, 200Klinhas do corpus CETEMPublico e 800 poemas do Arquivo de Letras de Música.

Apesar desta informação conter incorrecções [7] [8], acreditamos que ela permite poupar muitas horas de trabalho a quem está a construir dicionários.

Os casos de estudo que se seguem pretendem apenas dar uma ideia mais clara de algumas potencialidade do módulo.

**Exemplo 4:** Contribuição man para o foldoc

Como já se referiu anteriormente, o Free Online Dictionary on Computing (Howe, 1993) constitui uma muito interessante iniciativa de Denis Howe, que vem mantendo (com contribuições de pessoas de todo mundo) um dicionário de Computação.

Neste exemplo, vai-se construir uma contribuição para o foldoc: um dicionário com a linha de resumo do `man`. Este dicionário pode ser incorporado automaticamente ou poderá ser feita uma conciliação manual com o Foldoc actual.

A linha resumo do man tem características de dicionário. Com a opção k (`man -k`), o comando `man` fornece as linha de resumo correspondentes a um padrão recebido como parâmetro.

No caso do padrão ser a expressão regular ".", obtemos a totalidade das linhas de resumo. Da saída respectiva, podemos extrair os campos correspondentes e formar um dicionário explícito:

```
open(F,"man -k . |") or die;

d1=mkdbddmf("f",{add => 1, id => "man-k"});

while(<F>){
  chomp;
  if(/^(.+?)\s*\[(.*?)\]\s*\((.*?)\)\s*-\s*(.*)$/){
    d1->add($1,
           { section => $3,
             seealso => $2
             desc    => $4,
```

[7] Alguns dos exemplos (principalmente em palavras com menor grau de confiança, e portanto ligadas a maior ambiguidade) não estão correctamente colocados. Ex: a frase exemplo *A Rosa esperava impaciente a...* não corresponde a uma forma do verbo impacientar.

[8] A transcrição fonética calculada, nem sempre está correcta: Ex: em vez de `ĩpasiĕta:r` deveria ser `ĩp6siĕta:r`.

```
10                url     => "man://localhost/$1($3)" }) }
11    }
```

Notas:

**linha 1** - Cada linha de saída de `man -k` tem a seguinte estrutura:

$$
\begin{array}{rcrcll}
mank & = & nome & : & str & \times \\
 & & seealso & : & str & \times \\
 & & section & : & \text{...número de secção do manual} & \times \\
 & & desc & : & str &
\end{array}
$$

e segue a seguinte sintaxe concreta:
`sprintf [printf]      (3)  - formatted output conversion`

**linha 2** - A opção `add` indica que o dicionário pode receber novos termos.

**linha 6** - Esta expressão regular é usada para extrair os vários componentes da linha de resumo: nome do comando, secção a que o comando pertence, resumo.

O dicionário *calculado* contém entradas como:

```
sprintf => {
  seealso => 'printf',
  url     => 'man://localhost/sprintf(3)',
  section => '3',
  desc    => 'formatted output conversion'
}
```

Quando se juntar (pelos seus responsáveis) esta informação ao Foldoc, cada entrada pode ser usada como entrada independente, ou como informação complementar (enriquecimento de entradas existentes).

Ao executar o programa acima obtiveram-se os seguintes resultados [9]:

| | |
|---|---|
| número de entradas extraídas | 15477 |
| tempo demorado | 74 segundos |
| tamanho da dbfile criada | 1.2 Mbytes |

**Exemplo 5:** Dicionário como cache dum processo mais complexo

Para a ferramenta de tradução atrás referida (8.6), surgiu a necessidade de arranjar um modo versátil de tratar as palavras desconhecidas.

Se uma palavra é desconhecida, i.e., se não existe nos dicionários oficiais, vai ser feita uma pesquisa sucessivamente em:

- cópia local do freeDict eng-por,
- cópia remota do babelfish (Babelfish, 2000)
- cópia remota do dicionário Inglês-Português da Porto Editora

[9]Num pentium III, em carga, 600MHz, 128Mbytes de RAM, linux.

Seguidamente, a informação obtida vai ser sujeita a um processo de:

- validação interactiva dessa informação (i.e., decidir se se aceita ou não essa informação, alterando-a se necessário).
- armazenamento em dicionário temporário (cache) para evitar duplicação de pesquisas remotas e permitir posterior oficialização.

Dado que há dicionários envolvidos cujo acesso é lento, há necessidade de escolher um modo de composição que reduza as consultas a esses dicionários quando tal for possível. No caso que seguidamente se apresenta, isso é conseguido fazendo o *caching* das respostas a perguntas anteriores e a composição série dos dicionários remotos: vai sendo feita a consulta até que o primeiro responda.

Dum modo simplificado, esse tratamento corresponde a:

```
1 babelfish      = ...dicionário lento que pergunta via rede...
2 peEn2Pt        = ...dicionário lento que pergunta via rede...
3 freeDictEn2Pt = ...

4 d = mkcachedddmf(
5         mkddvalinteractiv(
6             mkcompar(freeDictEn2Pt,
7                      mkcomser(babelfish,peEn2Pt,...))),
8         en2pt)
```

Notas:

**linha 1,2,3** - Cria DDMF para aceder às várias informações.

**linha 7** - Pergunta sucessivamente aos dicionários `babelfish`, `peEn2Pt` , ... até que um deles dê uma resposta não vazia.

**linha 6** - Junta essa resposta com a resposta dada pelo `freeDictEn2Pt`.

**linha 5** - Valida interactivamente as respostas referidas.

**linha 4** - Faz caching da informação conseguida por questões de eficiência e para permitir posterior edição/incorporação da informação.

## 9.5 Dicionários com pós-processamento através de reescrita

### `mkddmfrr` - dynamic dictionary with rewriting-system

Por vezes, é necessário fazer pós-processamento na saída produzida pela consulta a um dicionário. Esse pós-processamento poderá:

– alterar nomes de campos ou de valores particulares (por exemplo, para conseguir coerência com outra informação a ser adicionada)
– apagar alguma informação que possa ser irrelevante na situação presente
– acrescentar informação nova, funcionanto como processo de inferência
– etc.

Seguidamente, vai-se apresentar um modelo para o sistema de reescrita, e uma descrição (superficial) do processo de reescrita.

$$\begin{array}{lcl} RS & = & regra^{\star} \\ regra & = & condicao \times alteracoes \\ condicao & = & ef \longrightarrow Bool \\ alteracoes & = & ef \longrightarrow ef \end{array}$$

O sistema de reescrita (RS) é definido através duma sequência de regras; cada regra é composta por uma `condição` (uma função booleana sobre a EF informação) e uma função `alteracoes` que transforma EF em EF.

O processo de reescrita, aplica as regras de reescrita a todos as EF na árvore de EF da informação. Dum modo simplificado:

$rewrite : RS \times EF \longrightarrow EF$
$rewrite(rs, f) \stackrel{\text{def}}{=}$
AplyR(r,fe) a todos os fe na árvore f, para todas as regras r em rs

$$\begin{array}{l} AplyR : regra \times EF \longrightarrow EF \\ AplyR(r, f) \stackrel{\text{def}}{=} \\ \quad \underline{let} \;\; <con, alte> = r \\ \quad \underline{in} \;\; \left\{ \begin{array}{rcl} con(f) & \Rightarrow & alte(f) \\ \underline{otherwise} & \Rightarrow & f \end{array} \right. \end{array}$$

No exemplo que se segue, vai ser usado um sistema de reescrita que funciona como processo de inferência para juntar a semântica implícita à derivação pelo sufixo `mento`.

**Exemplo 6:** Dicionário com inferência

Uma versão mais completa deste caso foi apresentada no exemplo da Secção 5.4.

Retomando o exemplo 1, pg.259, em que um dicionário `d` baseado no analisador morfológico `jspell` dava a seguinte informação acerca da palavra *rebentamento*:

```
> rebentamento
  [ { N    => 's',
```

```
       G    => 'm',
       FSEM => 'mento',
       CAT  => 'nc',
       rad  => 'rebentar',
       T    => 'inf',
       TR   => '_' } ]
```

Vamos agora acrescentar uma regra de inferência que deduza que se o atributo `FSEM` tem o valor "mento", então existe alguma informação implícita que podemos facilmente acrescentar de acordo com a seguinte regra de reescrita:

```
1  rr= [
2   [sub{my ef = shift;
3        ef{FSEM} == "mento" },

4    sub{my ef = shift;
5        delete(ef{FSEM});
6        ef{sem}  = "O acto de '$ef{rad}'";
7        ef{gram} = "Pode reger a preposição 'de' seguido do elemento
8                      que se vai $ef{rad}"
9           unless (ef{T} == "i");
10       ef} ]
11   ];

12 interpret(mkddmfrr(mkddfm(\*fea) , rr))
```

Notas:

**linha 2,3** - Definição da função *condição de aplicabilidade*: ter FSEM = "mento".

**linha 4 a 10** - Definição da função de alteração:

**linha 5** - remoção do atributo FSEM,

**linha 6** - junção do atributo *semântica* implícito ao uso do sufixo "mento",

**linha 7 a 9** - Junção de informação gramatical implícita ao uso de "mento", excepto se o verbo for *intransitivo*.

Após a inclusão deste processo de reescrita, o resultado passa a ser:

```
> rebentamento
  [ { N    => 's',
      G    => 'm',
      CAT  => 'nc',
      sem  => 'acto de rebentar',
      gram => "pode reger a preposição 'de' seguido do elemento
               que se vai rebentar",
      rad  => 'rebentar',
      T    => 'inf',
      TR   => '_' } ]
```

Também existe disponível um conjunto de funções que produzem algumas das regras mais habituais, como sejam mudar o nome de um atributo (**`rren`**). A função **`rren`**`(n1,n2)` gera a seguinte regra de reescrita:

```
[ sub{my ef = shift;
      defined ef{n1}}

  sub{my ef = shift;
      ef{n2} = ef{n1};
      delete(ef{n1});
      ef}
]
```

Em exemplos com sistemas de reescrita mais complexos, constatou-se que este processo pode ser pesado[10].

Este mecanismo pode também ser usado para implementar estratégias guiadas por sistemas de produções (ver Secção 5.2.3).

Outros exemplos mais complexos (envolvendo modos de escolher dinamicamente a estratégia de composição) podem ser encontrados em (Almeida and Henriques, 1998).

## 9.6 Funções relacionadas com a animação de DDMF

Por funções de animação de DDMF entenda-se o conjunto de funções que ajudam a **permitir que um DDMF seja utilizado**.

Para animar um dicionário, há várias hipóteses. Para além de gerar um interpretador de consultas (função `interpret` atrás exemplificada), podemos animar um dicionário através de gerar formatos que tenham *vida própria* como sejam:

– o formato DICT (Faith and Martin, 1998) - permitindo criar consultas locais, em intranet ou sobre internet (existem clientes para muitos sistemas operativos)

[10]Embora este valor não signifique nada sem a descrição completa das condições de teste, chegamos a medir valores da ordem dos 0.05 segundos por entrada; estes valores embora não constituam problema em consultas interactivas, afectam as operações que envolvem processamento maciço de entradas.

- gerar LaTeX(usando DictTeX ver 7.2) – permitindo que se possa fazer impressão de qualidade e seguir o ciclo de vida normal de dicionários em papel
- gerar formato TEI para facilitar difusão e durabilidade
- etc.

A animação via Web, baseada na função de consulta, é igualmente fácil (ver Secção 8.2.7).

Algumas destas operações mostraram ser lentas, o que não nos pareceu constituir um problema grave dado que podem ser feitas uma única vez. De qualquer forma, comparam muito positivamente (mostram-se de grande valor) com idênticas tarefas realizadas no tradicional processo manual.

# Capítulo 10

# Conclusões e trabalho futuro

*Ainda não é o fim nem o princípio do mundo calma é apenas um pouco tarde*

*Manuel António Pina*

Ao longo da dissertação, percorreu-se um estudo de especificação e tratamento de dicionários envolvendo a utilização cooperativa de várias ferramentas capazes de contribuir com informação associada a termos.

Esta álgebra de dicionários foi usada em vários contextos:

- **exemplos de demonstração**: dicionário aberto de calão, diversos dicionários criados em vários capítulos por composição de fontes e recursos dicionarizados, etc.
- **criação de recursos**: uso do sistema para criar dicionários com base nos quais se extraem sub-dicionários (exemplo: conversão de dicionário Babylon para formatos textuais – ver Secção 9.3); uso de dicionários com caches para aprendizagem assistida; geração de derivados de dicionários.
- **internamente, em partes localizadas de ferramentas**: por exemplo, na gestão de palavras desconhecidas na ferramenta EMS ou na ferramenta TEXT::TRANSLATE.

Percorreu-se um conjunto de experiências de desenvolvimento de programas e de recursos que:

- ajudassem à minha formação pessoal na área de PLN,
- fossem úteis à comunidade,
- funcionassem como funções e constantes na álgebra de dicionários proposta.

Das experiências realizadas concluiu-se que:

– Há uma série de fenómenos que podem ser descritos com naturalidade usando os construtores associados a DDMF.

Voltando ao escrito na subsecção 1.4, mostrou-se que:

– De um ponto de vista "filosófico":
    - a visão de DDMF constitui um meio de especificar e construir soluções para um conjunto de problemas envolvendo dicionários
    - é útil e eficaz a seguinte estratégia de resolução de problemas: (1) dado um problema $p$ (2) procurar um problema $p'$ relacionado, para o qual haja uma solução (um programa ou módulo) *open source* (3) construir o seu modelo abstracto $m = modelo(p')$ (4) analisar as alterações que é necessário efectuar no modelo $m$ para... (5) determinar como é que essas alterações se vão reflectir no código fonte que resolve $p'$.

– Do ponto de vista de ferramentas e recursos construídos:
    - o CAMILA, como plataforma de suporte à especificação, ajuda no processo de reflexão e funciona como meio de diálogo e de produção de texto matemático.
    - O conjunto de ferramentas e recursos apresentados nos capítulos 6, 7, 8 e 9, permite animar o processo em discussão, e é por si só uma contribuição.

## 10.1 Trabalho futuro

### Avaliação geral de recursos e ferramentas

Sempre que possível, utilizou-se as ferramentas e recursos contruídos em mais que uma situação. Essa reutilização mostrou ser muito importante para a sua maturação e, de um modo geral, para melhorar a qualidade e versatilidade dos elementos em causa.

Embora haja sistemas/aplicações com diferentes níveis de maturidade, a generalidade dos recursos e ferramentas apresentados necessitam de ser sujeitos a um processo rigoroso de avaliação (e posterior tratamento das fraquezas detectadas).

Esse processo, no entanto, envolve um esforço de trabalho apreciável, quer em quantidade quer em complexidade.

### Outras recursos previstos

Iniciaram-se já estudos numa nova linha, visando construção de *funções* ligadas a recursos multilingue baseados em corpora:

- extracção de textos bilingue a partir da Web – construir mecanismos automáticos de encontrar bitextos (textos que sejam a tradução um do outro) na Web (Almeida, Simões, and de Castro, 2002).
- construção de corpora paralelo a partir de textos bilingues – segmentar os textos e fazer o seu alinhamento à frase.
- extracção de terminologia e dicionários bilingues a partir de corpora paralelo – alinhar textos à palavra, extracção de terminologia multipalavra, alinhamento de segmentos (Simões and Almeida, 2003).

## Bibliografia


Abiteboul, Serge. 1997. Querying semi-structured data. *Proc. of Int. Conf. on Database Theory (ICDT), Delphi, Greece.*

Abramson, Harvey and Veronica Dahl. 1989. *Logic Grammars.* Symbolic Computation - Artificial Intelligence. Springer-Verlag.

Abrial, Nicolas Quint. 1999. *Dicionário Caboverdeano Português.* coordenação de Mafalda Mendes (verbalis), `http://www.priberam.com/dcvpo`.

Aguilar, Ana and Amat Castillo. 1996. Los problemas linguísticos de la traducción automática en el campo del léxico. In Javier Guinovart and Anxo Suárez, editors, *Linguística e informática.* Tórculo Edicións.

Ahmed, Kal. 2000. Topic maps for repositories. In *XMLEurope'2000.*

Aho, A. V., R. Sethi, and J. D. Ullman. 1986. *Compilers Principles, Techniques and Tools.* Addison-Wesley.

Ait-Kaci, Hassan. 1990. Is there a meaning to life? Technical report, Digital Equipment Corporation.

Ait-Kaci, Hassan and Patrick Lincoln. 1988. Life: a natural language for natural language. Technical Report ACA-ST-074-88, MCC Technical Report.

Allegranza, Valerio, Jacques Durand, Frank van Eynde, Lee Humphreys, Paul Schemidt, and Erich Steiner. 1991. Linguistics for machine translation: The Eurotra linguistic specification. *Studies in Machine Translation and Natural Language Processing*, 1:15–124.

Almeida, J.J. 1994. GPC – a tool for higher-order grammar specification. In Carlos Martin Vide, editor, *Actas del X Congreso de Lenguajes Naturales e Leanguajes Formales, Sevilla.*

Almeida, J.J. 1995. YaLG – extending DCG for natural language processing. In Carlos Martin Vide, editor, *Actas del XI Congreso de Lenguajes Naturales e Leanguajes Formales, Tortosa*, pages 621–628.

Almeida, J.J. 1996. NLlex – a tool to generate lexical analysers for natural language. *SEPLN*, 19:81–90, Sep.

Almeida, J.J. 1997. Natura - natural language processing. (páginas do projecto), Universidade do Minho, Departamento de Informática. `http://natura.di.uminho.pt/`.

Almeida, J.J. 1998. Programação de dicionários. In *Actas do XIII Encontro da Associação Portuguesa de Linguística*, volume 1, pages 21–28, Lisboa 1997.

Almeida, J.J., L.S. Barbosa, J.B. Barros, and L.F. Neves. 1998. On the development of CAMILA. In L.S. Barbosa and J.A. Saraiva, editors, *Workshop on Research Themes on Functional Programming.* Proc. 3rd Summer School on Advan. Funct. Prog., Braga, 18 Sep.

Almeida, J.J., L.S. Barbosa, F.L. Neves, and J.N. Oliveira. 1997a. CAMILA: Formal software engineering supported by functional programming. In A. De Giusti, J. Diaz, and P. Pesado, editors, *Proc. II Conf. Latino Americana de Programación Funcional (CLaPF97)*, pages 1343–1358, La Plata, Argentina, October.

Almeida, J.J., L.S. Barbosa, F.L. Neves, and J.N. Oliveira. 1997b. CAMILA: Prototyping and refinement of constructive specifications. In M. Johnson, editor, *6th International Conference on Algebraic Methods and Software Technology (AMAST'97)*, pages 554–559, Sydney, Australia, December. Springer Lect. Notes Comp. Sci. (1349).

Almeida, J.J., P. Rangel Henriques, J. Gustavo Rocha, and Alberto Simões. 2001. Alfarrábio: Adding value to an heterogeneous site collection. In *Congresso Nacional de Bibliotecários, Arquivistas e Documentalistas*, Porto, Maio.

Almeida, J.J. and P.R. Henriques. 1998. Dynamic dictionary = cooperative information sources. In *Proc. II Conference on Knowledge-based Intelligent Electronic Systems (Kes98)*, Australia, April.

Almeida, J.J. and Ulisses Pinto. 1995. Jspell – um módulo para análise léxica genérica de linguagem natural. In *Actas do X Encontro da Associação Portuguesa de Linguística*, pages 1–15, Évora 1994.

Almeida, J.J. and José Carlos Ramalho. 1999. XML::DT a perl down-translation module. In *XML-Europe'99, Granada - Espanha*, May.

Almeida, J.J., J. Gustavo Rocha, P. Rangel Henriques, Sónia Moreira, and Alberto Simões. 2001. Museu da Pessoa – arquitectura. In *Congresso Nacional de Bibliotecários, Arquivistas e Documentalistas*, Porto, Maio.

Almeida, J.J. and A. M. Simões. 2001. Text to speech – a rewriting system approach. *SEPLN*, 27:247–255, Sep.

Almeida, J.J. and Alberto M. Simões. 2002. Jspell.pm – um módulo de análise morfológica para uso em processamento de linguagem natural. In *Actas do XVII Encontro da Associação Portuguesa de Linguística*, Lisboa 2001.

Almeida, J.J., Alberto M. Simões, and J. Alves de Castro. 2002. Grabbing parallel corpora from the web. *SEPLN*, 29:13–20, Sep.

Almeida, J.J., Alberto M. Simões, and Paulo A. Rocha. 2002. Cálculo de frequências de palavras para entradas de dicionários através do uso conjunto de analisadores morfológicos, taggers e corpora. In *Actas do XVII Encontro da Associação Portuguesa de Linguística*, Lisboa 2001.

Almeida, J.J. and Alberto Manuel Simões. 2003. Engenharia reversa de HTML usando tecnologia XML. In José Carlos Ramalho, editor, *XATA – XML Aplicações e Tecnologias Associadas, Braga.*

Amsler, R. A. and T. Tompa. 1988. An sgml-based standard for english monolingual dictionaries. In *Proc. of 4th Conf. of UW Center for the New OED*, pages 61–79. Waterloo.

Atkinson, Kevin. 2000. Aspell spell checker. (Home page) Aspell, GNU. `http://aspell.sourceforge.net/`.

Augusteijn, Lex. 1999. Sorting morphisms. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, Setembro.

Babelfish. 2000. Altavista babelfish translation. (home page). `http://world.altavista.com/`, powered by Systran.

Babylon.com Ltd. 2000. Babylon at a click. (home page). `http://www.babylon.com/`.

Bach, Maurice J. 1986. *The design of the Unix operating system*. Prentice Hall.

Backhouse, R. 1979. *Syntax of Programming Languages: Theory and Practice*. Prentice Hall.

Balabanovic, M. and Y. Shoham. 1997. Content-based, collaborative recommendation. *Communications of the ACM 4 n. 3 66-72*.

Baptista, Jorge M. E. 1994. Estabelecimento e formalização de classes de nomes compostos. Tese de mestrado, Fac. de Letras, Universidade de Lisboa.

Barbosa, L.S. and J.J. Almeida. 1995. CAMILA: A reference manual. Technical Report DI-CAM-95:11:2, University of Minho.

Barbosa, L.S. and J.J. Almeida. 1998. Systems prototyping in CAMILA. Lecture Notes for the Bristol Course (1st ed. 1995) DI-CAM-95:11:1:v98, DI (U. Minho).

Barbosa, L.S., J.B. Barros, and J.J. Almeida. 2000. Polytypic recursion patterns. In *SBLP'2000 (to appear as a ENTCS volume)*, UFP, Recife, Brasil, May.

Barwise, Jon and Robin Cooper. 1991. Simple situation theory and its graphical representation. Working version.

Barwise, Jon and Larry Moss. 1991. Lectures on situation theory and its foundations. reader of 3.lli.

Bentley, J.L. 1986. Programming pearls: little languages. *Communications of the ACM*, 29(1):711–721, August.

Bird, R. and O. Moor. 1997. *The Algebra of Programming*. Series in Computer Science. Prentice Hall.

Brian, Dan. 2001. Lingua::Wordnet - Perl extension for accessing and manipulating wordnet databases. (perl module), CPAN - Comprehensive Perl Archive Network.

Brill, E. 1992. A simple rule-based part of speech tagger. In *Third Conference on Applied Natural Language Processing, ACL*, Trento, Italy.

Brill, E. 1993. *A Corpus-Based Approach to Language Learning*. Ph.D. thesis, University of Pennsylvania.

Brill, E. 1994. Some advances in rule-based part of speech tagging. In *Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle.

Briscoe, Ted. 1991. Lexical issues in natural language processing. In E. Klein and F. Veltman, editors, *Natural Language and speech*, pages 39 – 68. Springer-Verlag.

Briscoe, Ted and John Carroll. 1993. Generalised probabilistic lr parsing of natural language (corpora) with unification-based grammars. *Computational Linguistics*, 19(1):25–59.

Buitelarr, Paul. 1998. *CoreLex: systenatic polysemy and underspecification*. Ph.D. thesis, Computer Science, Brandeis University, February.

Cabral, Alexandre. 1988. *Dicionário de Camilo Castelo Branco*. Caminho.

Calzolari, Nicoletta. 1993. Computational lexicons. reader of 5.LLI, Faculdade de Letras, Universidade de Lisboa.

Calzolari, Nicoletta and Ted Briscoe. 1994. ACQUILEX i and ii. In N. Varile and N. Zampolli, editors, *European Projects*.

Calzolari, Nicoletta, Johan Hagman, Elisabetta Marinai, Simonetta Montemagni, Antonieta Spanu, and Antonio Zampolli. 1993. Encoding lexicographic definitions as typed feature structures. In Herausgegeben von Frank Beckmann and Gerhard Heyer, editors, *Theorie unt Praxis des Lexikons*, pages 274 – 315. Walter de Gruyter, Berlin, New York.

Carlsoon, M., J. Widen, J. Andersson, A. Andersson, K. Boortz, H. Nilsson, and T. Sjoland. 1991. Sicstus Prolog user'a manual. Technical Report T91:11B, SICS.

Chambers, Tyler, 1999. *Internet Dictionary Project*. `http://www.june29.com/IDP/`.

Copestake, Ann, Antonio Sanfilippo, Ted Briscoe, and Valerio de Paiva. 1992. The ACQUILEX LKB: an introdution. In *Default Inheritance in Unification Based Approaches to the Lexicon*, pages 182 – 202.

Correia, Margarita. 1994. Bases digitais lexicais na União Europeia. In *Simpósio de Lexicologia, Lexicografia e Terminologia*. Araraquara.

Crystal, David. 1997. *The Cambridge Encyclopedia of the Language*. Cambridge University Press.

Cunha, Celso and Lindley Cintra. 1987. *Nova Gramática do Português Contemporâneo*. João Sá da Costa, Lisboa. 4. Edição.

Desarmenien, Francois. 2000. Parse::Yapp - Perl extension for generating and using LALR parsers. (Perl module) Parse::Yapp, CPAN - Comprehensive Perl Archive Network.

Dodd, Chris and Vadim Maslov. 2002. BtYacc – Backtracking Yacc. home page. `http://siber.org/btyacc/`.

Eyermenn, Horst. 1999. Freedict - for free bilingual dictionaries. `http://www.FreeDict.de`.

Faith, Rik and Bret Martin. 1998. Dict protocol. Technical Report RFC 2229. `http://www.dict.org`.

Fellbaum, Christiane. 1998. *WordNet an electronic lexical database*. The MIT Press.

Fielding, Roy and et al. 1998. Web-based development of complex information products. *Communications of the ACM, 41-8*, Aug.

Filgueiras, Miguel. 1994. A successful case of computer aided translation. In Morgan Kauffman, editor, *Fourth Conference on Applied Natural Language Processing*, pages 91–94.

Florescu, D., A. Levy, and A. Mendelzon. 1998. Database techniques forthe www: A survey. *SIGMOD Record 27-3*, Sep.

Fogel, Karl. 1999. *Open source development with CVS*. Coriolis Group Books.

Gazdar, G. and R. Evans. 1989. The semantics of DATR. pages 66–71.

Gazdar, G., E. Klein, G. Pullum, and I. Sag. 1985. *Generalized Phrase Structure Grammar*. Blacbwell, Oxford.

GENELEX, Eureka project. 1993. Report on the syntactic layer. Technical report, Genelex Consortium, December.

GENELEX, Eureka project. 1994a. Report on the morphological layer. Technical report, Genelex Consortium, November.

GENELEX, Eureka project. 1994b. Report on the semantic layer. Technical report, Genelex Consortium, September.

Gühring, Philipp. 2001. *Quick – Open translation system. home page. `http://www3.futureware.at/equick.htm`.

Goossens, Michel, Frank Mittelbach, and Alexander Samarin. 1999. *The LaTeX Companion*. Addison-Wesley.

Goossens, Michel and Sebastian Rahtz. 1999. *The LaTeX Web Companion: integrating TeX, HTML and XML*. Addison-Wesley.

Grefenstette, Gregory and Pasi Tapanainen. 1994. What is a word, what is a sentence? problems of tokenization. Technical Report MLTT-004, Xerox Research Centre Europe, MLTT.

Gudivada, V., V. Raghavan, W. Grosky, and R. Kasanagottu. 1997. Information retrieval on the www. *IEEE Intelligent Computing*.

Hendersen, P. 1984. Me too: A Language for Software Specification and Model Building — Preliminary Report. Tecnhical Report, University of Stirling.

Holzer-Klupfel, Matthias and Christian Gebauer. 1998. Kdict client – a graphical interface for the DICT protocol. (home page). `http://www.rhrk.uni-kl.de/~gebauerc/kdict/`.

Horowitz, Ellis and Sartaj Sahni. 1977. *Fundamentals of data structures.* Pitman Press.

Hovinen, Bradford, Spiros Papadimitious, and Mike Hughes. 1999. Gdict client for the mit dictionary server. (home page). `http://gdict.dhs.org/`.

Howe, Denis. 1993. FOLDOC – the free on-line dictionary of computing. home page. `http://wombat.doc.ic.ac.uk`.

Ide, Nancy and Jean Véronis. 1995. Encoding dictionaries. In *Text encoding iniciative*, pages 167–179. Kluwer Academic publisher.

ISO 2788. 1986. Guidelines for establishment and development of monolingual thesauri. Technical Report ISO 2788:1986, International Organization for Standardization, Geneva.

ISO 5964. 1985. Guidelines for establishment and development of multilingual thesauri. Technical Report ISO 5964:1985, International Organization for Standardization, Geneva.

Johnson, S. C. 1975. YACC yet another compiler compiler. Technical Report CSTR32, Bell Laboratories, Murray Hill.

Johnson, S. C. and M. E. Lesk. 1978. Language development tools. *Bell System Technical Journal*, 57(6).

Jones, Cliff B. 1986. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice Hall.

Karttunen, Lauri. 1991. Finite-state constraints. In *Proceedings International Conference on Current Issues in Computational Linguistics*, Universiti Sains Malaysia, Penang.

Kay, Martin. 1979. Functional grammar. In *5th Annual Meeting of the Berkeley Linguistic Society.* Berkeley.

Knoblock, Naveen Ashish Craig. 1997. Wrapper generation for semi-structured internet sources. *SIGMOD Record, 26-4*, Dec.

Koenig, Andreas, 1995. *CPAN – Comprehensive Perl Archive Network.* `http://www.perl.org`.

Kominek, Jay. 1999. Jiben – the perl Dict server. (home page). `http://ucsub.colorado.edu/~kominek/jiten`.

Koskenniemi, 1983. *Two-level Morphology: A General Computational Model for Word-Form Recognition and Production.* University of Helsinki, Department of General Linguistics.

Krieger, Hans-Ulrich. 1995. Typed feature formalisms as a common basis for linguistic specification. Research Report RR-94-39, Deutsches Forschungszentrum für Künstliche Intelligenz, Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany. To appear in Machine Translation and the Lexicon, Springer, Berlin, 1995.

Krieger, Hans-Ulrich and Ulrich Schäfer. 1994. $\mathcal{TDL}$—a type description language for HPSG. part 1: Overview. Research Report RR-94-37, Deutsches Forschungszentrum für Künstliche Intelligenz, Saarbrücken, Germany.

Kuenning, Geoff, 1993. *ISPELL – Interactive spelling checking*. Home page `http://fmg-www.cs.ucla.edu/geoff/ispell.html`.

Kuiper, Matthijs and João Saraiva. 1998. Lrc - a generator for incremental language-oriented tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April.

Lamport, Leslie. 1986. *LaTeX User's Guide and Reference Manual*. Addison-Wesley Publishing Company.

Lamport, Leslie and Lawrence C. Paulson. 1999. Should your specification language be typed. *ACM Transactions on Programming Languages and Systems*, 21(3):502–526.

Lano, K. 1996. *The B Language and Method: A Guide to Practical Formal Development*. FACIT. Springer-Verlag.

Lesk, M. E. and E. Schmidt. 1975. LEX - Lexical Analyzer Generator. CS Technical Report 39, Bell Laboratories.

Marquess, Paul. 1995. DB_File – Perl access to berkeley db version 1.x. (Perl module) DB_File, CPAN - Comprehensive Perl Archive Network.

Marrafa, Palmira, António Branco, and Paula Guerreiro. 2001. Wordnet--PT: Rede Semântica-Lexical da Língua Portuguesa. (páginas do projecto), Centro de Linguística Universidade de Lisboa. `http://www.clul.ful.pt/sectores/projecto_wordnet.html`.

Masayuki Ishii, Kazuhisa Ohta, Hiroaki Saito. 1994. An efficient parser generator for natural language. *COLING*.

Medeiros, José Carlos. 1995. Análise morfológica e correcção ortográfica do Português. Tese de mestrado, Instituto Superior Técnico, Universidade Técnica de Lisboa.

Medeiros, José Carlos, Diana Santos, and Rui Marques. 1993. Português quantitativo. In *Actas do 1.º Encontro de Processamento de Língua Portuguesa – EPLP'93*, pages 33–38, Lisboa, Fevereiro.

Mel'chuk, Igor A. 1984. Un nouveau type de dictionnaire: Le dictionaire explicatif et combinatoire du français contemporain. In Igor A. Mel'chuk, Arbatchewsky-Jumarie, L. Elnitsky, L. Iordanskaja, and A. Lessard, editors, *Recherches lexico-sémantiques I*, pages 3–16. Les Presses de L'Université de Montreál.

Mel'chuk, Igor A. 1988. Semantic description of lexical units in an explanatory combinatorial dictionary: basic principles and heuristic criteria. *International Journal of Lexicography*, 1(1):1–13.

Miller, George A. 1990. An on-line lexical database. *International journal of lexicography*, 3(4). `http://www.cogsci.princeton.edu/~wn`.

Miller, George A., Richard Beckwith, Christiane Fellbaum, Derk Gross, and Katherine Miller. 1993. Introdution to wordnet: An on-line lexical database. Research report.

Nestorov, S., S. Abiteboul, and R. Motwani. 1997. Inferring structure in semistructured data. *SIGMOD Record 26-4*, Dec.

Nobre, Eduardo. 2000. *Dicionário de Calão.* Publicações Dom Quixote. 2.ª Edição.

O'Keefe, R.A. 1990. *The Craft of Prolog.* Logic Programming. The MIT Press.

Oliveira, J. N. 1995a. Formal Specification and Prototyping of a Building Description Language. In *Proc. CIVIL-COMP'95, Cambridge*, August.

Oliveira, J. N. 1995b. Fuzzy Object Comparasion and Its Application to a Self-Adaptable Query Mechanism. In *Proc. IFSA'95, S. Paulo*, July.

Oliveira, José Nuno. 1991. *Especificação & Semântica.* Departamento de Informática, Univ. do Minho, October. 2.ª Edição.

Ousterhout, J. K. 1994. *Tcl and the Tk Toolkit.* Addison-Wesley.

Ousterhout, J. K. 1998. *Scripting: higher level programming for the 21st century.* IEEE Computer, March.

Parr, Terence. 1998. Antlr. home page. `http://www.antlr.org/`.

Parr, Terence John. 1996. *Language Translation Using PCCTS and C++.* ISBN: 0-9627488-5-4.

Patrotte, Wolf and Frank Schumacher. 1993. MULTILEX - final report wp 9: Mlexd. Technical Report MWP 8 - MS, Westfälische Wilhelms – Universität Münster, December.

Paxson, Vern, 1995. *Flex a fast Scanner Generator.* Free Software Foundation, version 2.5 edition, January.

Pennings, Maarten. 1994. *Generating Incremental Evaluators.* Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands, November. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Pennings/`.

Pepper, Steve. 2000. The tao of the topic maps. In *XMLEurope'2000*, pages 229–235.

Pereira, F. C. N. and D. H. D. Warren. 1980. Definite clause grammars for language analysis: A survey of the formalism and comparison with augmented transistion networks. *Artificial Intelligence*, 13(3):231–278.

Pinto, Ulisses and J.J. Almeida. 1996. Tratamento automático de termos compostos. In *Actas do XI Encontro da Associação Portuguesa de Linguística*, volume 2, Lisboa 1995.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago.

Popowich, Fred. 1993. Lexical characterization of local dependencies with tree unification grammar. Technical Report CMPT TR 93-13, School of Computing Science,Simon Fraser University.

Quass, D., A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. 1995. Querying semistructured heterogeneous information. *International Conf. on Deductive and Object-Oriented Databases*.

Ramalho, J.C., J.J. Almeida, and P.R. Henriques. 1998. Algebraic specification of documents. *Theoretical Computer Science*, (199):231–247.

Ramalho, José Carlos Leite. 2000. *Anotação Estrutural de Documentos e sua Semântica*. Tese de doutoramento, Universidade do Minho.

Reis, Ricardo and J.J. Almeida. 1998. Etiquetador morfo-sintáctico para o português. In *Actas do XIII Encontro da Associação Portuguesa de Linguística*, Lisboa 1997.

Ritchie, Graeme D., Graham J. Russel, Alan W. Black, and Stephen G. Pulman. 1992. *Computational Morphology: Practical Mechanisms for the English Lexicon*. Series in Natural Language Processing. The MIT Press.

Rocha, Jorge, Tiago Pedroso, and J.J. Almeida. 1999. MAPit - a tool set for automatically generation of HTML maps. In *Conferência da Association of Geographic Information Laboratories for Europe (AGILE)*, Roma.

Rocha, Paulo Alexandre and Diana Santos. 2000. CETEMPúblico: Um corpus de grandes dimensões de linguagem jornalística portuguesa. In *Actas do V Encontro para o processamento computacional da língua portuguesa escrita e falada (PROPOR'2000)*, pages 131–140. Atibaia, São Paulo, Brasil, 19 a 22 de Novembro.

Rubinstein, Dmitry and Neil Bowers. 2000. Net::Dict – client API for accessing dictionary servers (RFC 2229). (Perl module) Net::Dict, CPAN - Comprehensive Perl Archive Network.

SAMPA. 1999. SAMPA: computer readable phonetic alphabet. (home page). `http://www.phon.ucl.ac.uk/home/sampa/home.htm`.

Sanfilippo, Antonio, Nicoletta Calzolari, Sophia Ananiadou, Rob Gaiazauskas, Patrick Saint-Dizier, Piek Vossen, Antonietta Alonge, Nuria Bel, Kalina Bontcheva, and Pierrette Bouillon. 1999. Preliminary recommendations on lexical semantic encoding, final report. Technical Report LE3-4244, EAGLES.

Sanromán, Álvaro. 2000. *A unidade lexicográfica, palavras, colocações, frasemas pragmatemas*. Tese de doutoramento, Universidade do Minho.

Santos, Diana. 2000. O projecto Processamento Computacional do Português: balanço e perspectivas. In Maria das Graças Nunes, editor, *Actas do V Encontro para o Processamento Computacional da Língua Portuguesa Escrita e Falada (PROPOR'2000)*, pages 105–113. Atibaia, São Paulo, Brasil, 19 a 22 de Novembro.

Santos, Diana and Eckhard Bick. 2000. Providing internet access to portuguese corpora: the ac/dc project. In Maria Gavrilidou et al., editor, *Proceedings of the Second International Conference on Language Resources and Evaluation, LREC 2000*, pages 205–210.

Santos, Diana and Elisabete Ranchhod. 1999. Ambientes de processamento de corpora em português: Comparação entre dois sistemas. In *Actas do PROPOR'99*, pages 257–268. Évora, 21-22 Setembro.

Saraiva, João. 1999. *Purely Functional Implementation of Attribute Grammars*. Ph.D. thesis, Department of Computer Science, Utrecht University, The Netherlands, December. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/phdtheses/Saraiva/`.

Saraiva, João. 2000. LRC – A Purely Functional, Higher-Order Attribute Grammar based System. (home page). `http://www.di.uminho.pt/~jas/Research/LRC/lrc.html`.

Sarathy, Gurusamy. 1998. Mldbm – store multi-level hash structure in single level tied hash. (Perl module) MLDBM, CPAN - Comprehensive Perl Archive Network.

Schaps, Gary L. 1999. Compiler construction with antlr and java. *Dr. Dobb's*, March.

Schwartz, Randal and Tom Christiansen. 1997. *Learning Perl*. O'Reilly Associates, second edition.

Sergeant, Matt and Christian Glahn. 2001. XML::LibXML – interface to the gnome libxml2 library. (Perl module) XML::LibXML, CPAN - Comprehensive Perl Archive Network.

Silvestre, João Paulo. 2001. O dicionário de Bluteau. Cursos da Arrábida. Comunicação integrado em Dicionarística da Língua Portuguesa: património e renovação.

Simões, Alberto M. and J.J. Almeida. 2002. Library::* – a toolkit for digital libraries. In *Elpub 2002 – International Conference on Electronic Publishing*, pages 203–211, Karlov Vary, República Checa, Nov.

Simões, Alberto M. and J.J. Almeida. 2003. NATools – a statistical word aligner workbench. *SEPLN*, Sep. forthcomming.

Simões, Alberto M., J.J. Almeida, and Pedro R. Henriques. 2002. Directory attribute grammars. In *VI Simpósio Brasileiro de Linguagens de Programação*, pages 297–308, Rio de Janeiro, Brasil.

Simões, Alberto Manuel and J.J. Almeida. 2003. Histórias de vida + processamento estrutural = Museu da Pessoa. In José Carlos Ramalho, editor, *XATA – XML Aplicações e Tecnologias Associadas, Braga*.

Simões, Guilherme. 1994. *Dicionário de expressões populares portuguesas*. Publicações D. Quixote.

Sinclair, John, editor. 1987. *Looking Up: an account of the COBUILD project in lexical computing*. London, Collins. ten essays about background details of the *Collins Cobuild English Language Dictionary*.

Sperberg-McQueen, C.M. and Lou Burnard. 1994. *Guidelines for Electronic Text Encoding and Interchange (TEI P 3)*. Chicago: ACH/ACL/ALLC.

Spivey, J. M. 1989. *The Z Notation: A Reference Manual.* Series in Computer Science. Prentice Hall.

Swierstra, S.D. and H.H. Vogt. 1991. Higher order attribute grammars, lecture notes of the Int. Summer School on Attribute Grammars, Applications and Systems. Technical Report RUU-CS-91-14, Dep. of Computer Science / Utrecht Univ., Junho.

TEI, 1995. *Print Dictionaries*, chapter 12. Guidelines for Electronic Text Encoding and Interchange (TEI P3). Chicago: ACH/ACL/ALLC. `http://etext.virginia.edu/tei-tocs3.html`.

Teitelbaum, T. and T. Reps. 1981. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9), September.

Teixeira, José. 1996. Branco é, galinha o põe. In Faria and Correia, editors, *Actas do XI Congresso da Associação Portuguesa de Linguística, Lisboa 1995*, pages 229–235.

Tomita, M. 1987. An efficient context-free parsing algorithm. *Computational Linguistics*, 13:31–46.

Tomita, M., editor. 1991. *Current Issues in Parsing Technology.* Kluwer Academic Publishers, Norwell, MA.

van den Brand, Mark, Paul Klint, and Pieter Olivier. 1999. Compilation and Memory Management for ASF+SDF. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction*, volume 1575 of *LNCS*, pages 198–213. Springer-Verlag, Março.

van Deursen, Arie, P. Klint, and J.M.W. Visser. 2000. Domain-specific languages. Technical Report SEN-R0032, ISSN 1386-369X, CWI. an annotated bibligraphy.

Verdelho, Telmo. 1995. *As origens da gramaticografia e da lexicografia latino-Portuguesas.* Aveiro.

Vilela, Mário. 1991. *Dicionário do Português Básico.* Edições Asa.

Vilela, Mário. 1994. *Estudos de Lexicologia do Português.* Almedina, Coimbra.

Vogt, H.H., S.D. Swierstra, and M.F. Kuiper. 1989. Higher order attribute grammars. *Communications of the ACM*, pages 131–145.

Vossen, Piek. 1999. EuroWordNet general document. Research report, University of Amsterdam, July.

Waite, W. and G. Goos. 1984. *Compiler Construction.* Springer-Verlag.

Wall, Larry. 2000. IPC::Open3 – open a process for reading, writing, and error handling. (Perl module) IPC::Open3, CPAN - Comprehensive Perl Archive Network.

Wall, Larry and Clark Cooper. 2000. XML::Parser – Perl module for parsing XML documents. (Perl module) XML::Parser, CPAN - Comprehensive Perl Archive Network.

Wall, Larry and Randal Schwartz. 1996. *Programming Perl*. O'Reilly Associates, second edition.

Weibel, S. 2003. Dublin core metadata iniciative. (home page), Oclc. `http://dublincore.org/`.

Weibel, S., J. Godby, and E. Miller. 1995. Oclc/ncsa metadata workshop report. Technical report. Dubin Core - `http://www.ocl.org:5046/conferences/metadata/dublin_core_report.html`.

Wirth, Niklaus. 1976. *Algorithms + data structures = programs*. Prentice Hall, July.

Wrightson, Ann. 2001. Topic maps and knowledge representation. Ontopia. `http://www.ontopia.net/topicmaps/materials/kr-tm.html`.

Wu, Sun and Udi Manber. 1992. Fast text searching allowing errors. *Communications of the Association for Computing Machinery*, 35(10):83–91, Outubro. agrep - `file://cs.arizona.edu/agrep`.

Xavier, M. F. and M. H. Mateus. 1992. *Dicionário de Termos Linguísticos*, volume II. Cosmos, Lisboa.

XML. 1998. eXtended Markup Language (XML) version 1.0 recommendation. Technical report, World Wide Web Consortium. `http://www.w3.org/TR/1998/REC-xml-19980210.html/`.

XPATH. 1999. XML Path Language (XPath) Version 1.0 recommendation. Technical report, World Wide Web Consortium. `http://www.w3.org/TR/1999/REC-xpath-19991116.html`.

# Apêndice A

# Dicionário das ferramentas e recursos usados

Este dicionário é gerado automáticamente com base em anotações contidas ao longo deste texto. Na Secção 6.8, página 181, é descrito sucintamente o processo de geração.

Alfabraga
  ***parte de*** – Alfarrábio
  ***página*** – 187 229
Alfarrábio
  ***partes*** – Arquivos sonoros Ernesto Veiga de Oliveira ◇ Cancioneiro ◇ Alfabraga ◇ Museu da Pessoa ◇ anarco-thesaurus Alfarrábio
  ***instância de*** – biblioteca digital
  ***necessita de*** – Library::*
  ***Ref*** – (Almeida et al., 2001);
  ***secção*** – 8.4
  ***Url*** – http://alfarrabio.di.uminho.pt
analisador morfológico
  ***instâncias*** – jspell
anarco-thesaurus Alfarrábio
  ***parte de*** – Alfarrábio
  ***instância de*** – thesaurus
  ***autor*** – J.João Almeida
  ***necessita de*** – XML::DT ◇ Library::Thesaurus
  ***secção*** – 8.4
arquivo de Etnomusicologia
  ***instância de*** – biblioteca digital
arquivos sonoros Ernesto Veiga de Oliveira

***parte de*** – Alfarrábio
***instância de*** – biblioteca digital

aspell
***página*** – 209

biblioteca C
***instâncias*** – PTC ♢ jspell
***genéricos*** – C

biblioteca digital
***partes*** – catálogo ♢ thesaurus
***instâncias*** – arquivos sonoros Ernesto Veiga de Oliveira ♢ arquivo de Etnomusicologia ♢ Alfarrábio ♢ Museu da Pessoa ♢ Natura
***referido em*** – DAG ♢ library::*
***página*** – 177

biblioteca Perl
***instâncias*** – Lingua::PT::speaker ♢ XML::DT ♢ Lingua::PT::PLN ♢ MLDBM ♢ text::translate ♢ DDMF ♢ library::* ♢ jspell ♢ wmboi
***parte de*** – Perl
***pode ser gerado por*** – mktextrr

biblioteca Prolog
***instâncias*** – YaLG ♢ DCG ♢ jspell

Bibtex
***instância de*** – ferramenta LaTeX
***descrição*** – Ferramenta LaTeX para gestão de referências bibliográficas
***página*** – 28 30 200

Brill tagger
***instância de*** – tagger
***autor*** – Eric Brill
***referido em*** – EMS
***página*** – 144 150
***Ref*** – (Brill, 1992; Brill, 1993; Brill, 1994);

C
***específicos*** – biblioteca C
***instância de*** – linguagem de programação
***pode ser gerado por*** – flex

camila
***instância de*** – linguagem de especificação
***termo associado*** – camila prototyping environment
***acerca de*** – especificação formal ♢ refinamento ♢ SETs
***autor*** – José Nuno Oliveira, Luís Barbosa, J.João Almeida, Luís Neves
***secção*** – 2.1

camila bibioteca C
***parte de*** – camila prototyping environment

camila interpreter
***parte de*** – camila prototyping environment

camila prototyping environment

***partes*** – camila-tlk ◊ campretty ◊ camtex ◊ camila bibioteca C ◊ camila interpreter
***instância de*** – tool ◊ programming language ◊ compilador
***termo associado*** – camila
***autor*** – J.João Almeida, José Nuno Oliveira, Luís Barbosa, Luís Neves

camila-tlk
***parte de*** – camila prototyping environment

campretty
***parte de*** – camila prototyping environment
***instância de*** – tool ◊ compilador
***descrição*** – campretty - camila to LaTeX translation
***acerca de*** – LaTeX
***autor*** – Luís Neves, J.João Almeida
***secção*** – 2.2.2
***usado em*** – camtex

camtex
***parte de*** – camila prototyping environment
***instância de*** – tool ◊ compilador ◊ ferramenta LaTeX
***descrição*** – CAMTEX - embeding camila in LaTeX
***acerca de*** – document processing ◊ LaTeX
***autor*** – J.João Almeida José Nuno Oliveira
***necessita de*** – campretty
***secção*** – 2.2.2

Cancioneiro
***parte de*** – Alfarrábio
***página*** – 229

catálogo
***página*** – 185 227

catálogo
***parte de*** – biblioteca digital
***descrição*** – documento contendo meta-informação acerca de um conjunto de documentos
***termo associado*** – Library::Catalog ◊ thesaurus
***referido em*** – library::*

compilador
***instâncias*** – mktextrr ◊ campretty ◊ camtex ◊ camila prototyping environment ◊ NLlex ◊ jbuild ◊ DPL
***específicos*** – domain specific language ◊ conversor
***genéricos*** – processamento de linguagens
***Ref*** – (Aho, Sethi, and Ullman, 1986; Waite and Goos, 1984; Backhouse, 1979; Teitelbaum and Reps, 1981);

conceito
***instâncias*** – provérbio modificado

conceito geral
***instâncias*** – ENL ◊ termos compostos

conversor

***instâncias*** – teidict2dicttex ◊ tab2dicttex
***genéricos*** – compilador

corpora
***referido em*** – YaLG ◊ freqnormpt ◊ divisor de sílabas
***página*** – 42
***secção*** – 8.3

corrector ortográfico
***instâncias*** – ispell ◊ jspell
***referido em*** – dicionário português do ispell

Dac
***instância de*** – dicionário DPL
***descrição*** – Dicionário aberto de expressões idiomáticas e calão
***termo associado*** – DPL
***acerca de*** – TFS ◊ dicionário
***autor*** – J.João Almeida
***Ref*** – (Almeida, 1998);
***secção*** – 8.2
***Url*** – http://natura.di.uminho.pt/jjbin/dac

DAG
***instância de*** – domain specific language ◊ tool
***descrição*** – Directory attribute grammars
***acerca de*** – gramáticas atributos ◊ biblioteca digital
***Ref*** – (Simões, Almeida, and Henriques, 2002);
***usado em*** – Museu da Pessoa

data mining
***referido em*** – wmboi

DCG
***instância de*** – logic grammar ◊ biblioteca Prolog
***descrição*** – Definite clause grammars
***autor*** – F. Pereira, D. Warren
***genéricos*** – logic grammar
***Ref*** – (Pereira and Warren, 1980);

DDMF
***instância de*** – tool ◊ biblioteca Perl
***descrição*** – Biblioteca de ajuda à construção e manipulação de dicionário dinâmico multi-fonte
***autor*** – J.João Almeida
***genéricos*** – dicionário
***secção*** – 5, 9

dicionário
***específicos*** – dicionário jspell ◊ dicionário TC ◊ dicionário ispell ◊ dicionário DPL ◊ DDMF
***genéricos*** – PLN
***referido em*** – dicttex ◊ Dac ◊ DPL ◊ tab2dicttex

dicionário DPL
***instâncias*** – Dac

  ***termo associado*** – DPL
  ***genéricos*** – dicionário
dicionário ispell
  ***instâncias*** – dicionário português do ispell
  ***genéricos*** – dicionário
dicionário jspell
  ***instâncias*** – dicionário português do jspell
  ***genéricos*** – dicionário
dicionário português do ispell
  ***instância de*** – dicionário ispell
  ***descrição*** – dicionário português para o corrector ortográfico `ISpell`
  ***acerca de*** – corrector ortográfico
  ***autor*** – J.João Almeida, Ulisses Pinto, Paulo Rocha
  ***necessita de*** – dicionário português do jspell ◇ ispell
  ***secção*** – 8.1
dicionário português do jspell
  ***instância de*** – dicionário jspell
  ***descrição*** – dicionário português para o analisador morfológico jspell
  ***acerca de*** – morfologia
  ***autor*** – J.João Almeida, Ulisses Pinto, Paulo Rocha
  ***necessita de*** – jspell
  ***secção*** – 8.1
  ***usado em*** – dicionário português do ispell
dicionário TC
  ***descrição*** – dicionário de termos compostos
  ***termo associado*** – PTC
  ***acerca de*** – termos compostos ◇ morfologia
  ***genéricos*** – dicionário
  ***necessita de*** – jspell ◇ PTC
  ***Ref*** – (Pinto and Almeida, 1996);
  ***secção*** – 6.4
dicttex
  ***partes*** – tab2dicttex ◇ teidict2dicttex
  ***instância de*** – estilo LaTeX ◇ ferramenta LaTeX
  ***descrição*** – impressão de dicionários usando LaTeX
  ***acerca de*** – dicionário ◇ LaTeX
  ***autor*** – J.João Almeida , José Luís Santos
  ***secção*** – 7.2
divisor de sílabas
  ***instância de*** – tool
  ***descrição*** – Divide em sílabas de acordo com as regras do Português
  ***acerca de*** – corpora ◇ PLN
  ***necessita de*** – mktextrr
  ***secção*** – exemplo 25, pg.171
document processing
  ***genéricos*** – PLN

***referido em*** – XML::DT ◊ camtex ◊ text::translate ◊ DPL

domain specific language

***instâncias*** – NLlex ◊ DAG ◊ mktextrr ◊ DPL

***genéricos*** – linguagem ◊ compilador

DPL

***instância de*** – domain specific language ◊ linguagem de programação ◊ compilador

***descrição*** – Dictionary programming language - Linguagem de programação/construção de dicionários

***termo associado*** – Dac ◊ dicionário DPL

***acerca de*** – TFS ◊ document processing ◊ dicionário

***autor*** – J.João Almeida, Alberto Simões

***gera*** – TXT ◊ MLDBM ◊ LaTeX

***secção*** – 7.1

EMS

***instância de*** – tagger

***descrição*** – Etiquetador morfo-sintáctico

***acerca de*** – Brill tagger ◊ morfologia ◊ sintaxe

***autor*** – Ricardo Reis, José João Almeida

***necessita de*** – jspell

***Ref*** – (Reis and Almeida, 1998);

***secção*** – 6.5

ENL

***instância de*** – conceito geral

***descrição*** – *elementos não literários* – elementos contidos nos textos que não são palavras no sentido convencional. Exemplo: etiquetas HTML, XML, comandos LaTeX, datas, medidas, ângulos, URLs, endereços de email, etc.

***página*** – 104 106

***secção*** – 6.2, pg.104

especificação formal

***específicos*** – linguagem de especificação ◊ reescrita

***descrição*** – Formal specification

***referido em*** – camila

***Ref*** – (Oliveira, 1991);

estilo LaTeX

***instâncias*** – dicttex

***genéricos*** – LaTeX

ferramenta LaTeX

***instâncias*** – dicttex ◊ Bibtex ◊ camtex

***genéricos*** – LaTeX

flex

***instância de*** – tool ◊ gerador de analisador léxico

***autor*** – Vern Paxson

***gera*** – C

***pode ser gerado por*** – NLlex

***página*** – 105 110 147

***Ref*** – (Paxson, 1995);
***usado em*** – NLlex

fonética
***referido em*** – Lingua::PT::speaker

freqnormpt
***instância de*** – tool
***descrição*** – calcula frequências de palavras normalizadas (formas lematizadas) e respectivo grau de confiança.
***acerca de*** – corpora
***necessita de*** – jspell
***Ref*** – (Almeida, Simões, and Rocha, 2002);
***secção*** – exemplo 4, pg.96, exemplo 3, pg.269

gerador de analisador léxico
***instâncias*** – flex

gerador de analisador morfológico
***instâncias*** – NLlex

gramática de ordem superior
***referido em*** – YaLG

gramáticas atributos
***referido em*** – DAG

HTML
***instância de*** – linguagem etiquetada
***pode ser gerado por*** – library::*

ispell
***instância de*** – corrector ortográfico
***termo associado*** – jspell
***página*** – 90 91 207 207 209
***Ref*** – (Kuenning, 1993);
***usado em*** – dicionário português do ispell

jbuild
***parte de*** – jspell
***instância de*** – compilador
***descrição*** – compilador para dicionários jspell

jspell
***partes*** – jbuild
***instância de*** – biblioteca C ◊ corrector ortográfico ◊ analisador morfológico ◊ biblioteca Perl ◊ stemmer ◊ biblioteca Prolog
***descrição*** – analisador morfológico
***termo associado*** – kspell ◊ ispell
***acerca de*** – morfologia
***autor*** – J.João Almeida, Ulisses Pinto
***página*** – 52 76 108 118 120 137 141 144 151 175 207 248 259 261 269 275
***Ref*** – (Almeida and Pinto, 1995; Almeida and Simões, 2002);
***secção*** – 6.1
***usado em*** – freqnormpt ◊ dicionário português do jspell ◊ dicionário TC ◊ PTC ◊ NLlex ◊ EMS

kspell
  ***descrição*** – Sistema de especificação de morfologia, baseada em esquemas, semelhante a uma gramática invertida. Permite expandir para um conjunto de regras atómicas jspell.
  ***termo associado*** – jspell
  ***acerca de*** – morfologia
  ***autor*** – J.João Almeida
  ***secção*** – 6.1.5

LaTeX
  ***específicos*** – estilo LaTeX ◇ ferramenta LaTeX
  ***instância de*** – linguagem etiquetada
  ***descrição*** – Sistema tipográfico de base completamente textual
  ***referido em*** – dicttex ◇ camtex ◇ campretty ◇ teidict2dicttex ◇ tab2dicttex
  ***pode ser gerado por*** – DPL ◇ library::*
  ***página*** – 27 104 180 187 204 218
  ***Ref*** – (Goossens and Rahtz, 1999; Goossens, Mittelbach, and Samarin, 1999; Lamport, 1986);

library::*
  ***página*** – 84 224 226 231
  ***usado em*** – Alfarrábio ◇ Museu da Pessoa ◇ Natura

Library::*
  ***partes*** – Library::Catalog ◇ Library::Simple ◇ Library::Thesaurus
  ***instância de*** – tool ◇ biblioteca Perl
  ***descrição*** – Conjunto de módulos para tratamento de bibliotecas digitais
  ***acerca de*** – catálogo ◇ biblioteca digital ◇ thesaurus
  ***autor*** – J.João Almeida, Alberto Simões
  ***gera*** – XML ◇ HTML ◇ LaTeX
  ***Ref*** – (Simões and Almeida, 2002);
  ***secção*** – 6.8; 8.4

Library::Catalog
  ***parte de*** – library::*
  ***termo associado*** – catálogo

Library::Simple
  ***parte de*** – library::*

Library::Thesaurus
  ***parte de*** – library::*
  ***termo associado*** – thesaurus
  ***usado em*** – anarco-thesaurus Alfarrábio

Lingua::PT::PLN
  ***instância de*** – biblioteca Perl
  ***descrição*** – Conjunto de funções ligadas a processamento de textos em português incluindo contadores de ocorrências de palavras, tokenizadores, divisão em sílabas, processamento de nomes próprios
  ***acerca de*** – PLN
  ***página*** – 223
  ***usado em*** – Lingua::PT::speaker

Lingua::PT::speaker
  ***instância de*** – biblioteca Perl
  ***descrição*** – sintetizador de voz para Língua Portuguesa
  ***acerca de*** – fonética
  ***autor*** – José João Almeida, Alberto Simões
  ***necessita de*** – mktextrr ◊ Lingua::PT::pln ◊ mbrola
  ***página*** – 269
  ***Ref*** – (Almeida and Simões, 2001);
linguagem
  ***específicos*** – domain specific language ◊ linguagem de programação ◊ linguagem de especificação ◊ linguagem etiquetada
linguagem de especificação
  ***instâncias*** – camila
  ***genéricos*** – especificação formal ◊ linguagem
linguagem de programação
  ***instâncias*** – C ◊ Perl ◊ DPL
  ***genéricos*** – linguagem
linguagem de scripting
  ***termo associado*** – Perl
linguagem etiquetada
  ***instâncias*** – XML ◊ HTML ◊ LaTeX
  ***genéricos*** – linguagem
logic grammar
  ***instâncias*** – YaLG ◊ DCG
  ***específicos*** – DCG
  ***descrição*** – gramáticas lógicas
mbrola
  ***usado em*** – Lingua::PT::speaker
mktextrr
  ***instância de*** – domain specific language ◊ compilador
  ***descrição*** – Gerador de sistemas de reescrita textual
  ***acerca de*** – reescrita
  ***autor*** – J.João Almeida
  ***gera*** – Perl ◊ biblioteca Perl
  ***página*** – 250
  ***secção*** – 6.7
  ***usado em*** – Lingua::PT::speaker ◊ text::translate ◊ divisor de sílabas
MLDBM
  ***instância de*** – biblioteca Perl
  ***descrição*** – Módulo perl que permite tornar não voláteis árvores generalizadas, arrays associativos.
  ***autor*** – Gurusamy Sarathy
  ***pode ser gerado por*** – DPL
  ***Ref*** – CPAN MLDBM;
morfologia
  ***genéricos*** – PLN

***referido em*** – dicionário português do jspell ♢ dicionário TC ♢ PTC ♢ EMS ♢ kspell ♢ NLlex ♢ jspell

Museu da Pessoa
- ***parte de*** – Alfarrábio
- ***instância de*** – biblioteca digital
- ***descrição*** – Museu virtual de histórias de vida e história oral
- ***necessita de*** – Library::* ♢ DAG
- ***página*** – 222 227 230
- ***Ref*** – (Almeida et al., 2001; Simões and Almeida, 2003);
- ***secção*** – 8.4
- ***Url*** – http://alfarrabio.di.uminho.pt/mp

Natura
- ***instância de*** – biblioteca digital ♢ projecto
- ***descrição*** – Projecto Natura ferramentas e recursos associados à linguagem natural
- ***acerca de*** – PLN
- ***necessita de*** – Library::*
- ***página*** – 98 103 187 209 211 222
- ***secção*** – 3.9.1
- ***Url*** – http://natura.di.uminho.pt

NLlex
- ***instância de*** – domain specific language ♢ gerador de analisador morfológico ♢ compilador
- ***descrição*** – gerador de analisadores léxicos para linguagem natural; $nllex = lex + morfologia$
- ***acerca de*** – morfologia
- ***autor*** – J.João Almeida
- ***gera*** – flex
- ***necessita de*** – flex ♢ jspell
- ***Ref*** – (Almeida, 1996);
- ***secção*** – 6.2
- ***usado em*** – YaLG

Perl
- ***partes*** – biblioteca Perl
- ***instância de*** – linguagem de programação
- ***termo associado*** – linguagem de scripting
- ***pode ser gerado por*** – mktextrr
- ***Ref*** – (Schwartz and Christiansen, 1997; Wall and Schwartz, 1996);

PLN
- ***específicos*** – document processing ♢ dicionário ♢ pragmática ♢ morfologia ♢ sintaxe ♢ semântica
- ***descrição*** – Processamento de linguagem natural
- ***genéricos*** – processamento de linguagens
- ***referido em*** – Lingua::PT::PLN ♢ PTC ♢ Natura ♢ divisor de sílabas

pragmática
- ***genéricos*** – PLN

***referido em*** – YaLG
processamento de documentos
***genéricos*** – processamento de linguagens
processamento de linguagens
***específicos*** – processamento de documentos ♢ compilador ♢ PLN
programming language
***instâncias*** – camila prototyping environment
projecto
***instâncias*** – Natura
provérbio modificado
***página*** – 221
provérbio modificado
***instância de*** – conceito
PTC
***instância de*** – biblioteca C ♢ tagger
***descrição*** – Processador de Termos Compostos
***termo associado*** – dicionário TC
***acerca de*** – termos compostos ♢ PLN ♢ morfologia
***autor*** – Ulisses Pinto, José João Almeida
***necessita de*** – jspell
***Ref*** – (Pinto and Almeida, 1996);
***secção*** – 6.4
***usado em*** – dicionário TC
reescrita
***genéricos*** – especificação formal
***referido em*** – YaLG ♢ mktextrr ♢ text::translate
refinamento
***referido em*** – camila
semântica
***genéricos*** – PLN
***referido em*** – YaLG
SETs
***referido em*** – camila
SGML
***termo associado*** – TEI
sintaxe
***genéricos*** – PLN
***referido em*** – YaLG ♢ EMS
stemmer
***instâncias*** – jspell
tab2dicttex
***parte de*** – dicttex
***instância de*** – conversor
***descrição*** – converte dicionários em formato de tabela textual para LaTeX/DictTeX
***acerca de*** – dicionário ♢ LaTeX
***autor*** – J.João Almeida

tagger
  ***instâncias*** – PTC ◇ EMS ◇ Brill tagger
TEI
  ***descrição*** – TEI - Text Encoding Initiative
  ***termo associado*** – XML ◇ SGML
  ***referido em*** – teidict2dicttex
  ***página*** – 38 42 50 61 259 266 278
  ***Ref*** – (Sperberg-McQueen and Burnard, 1994);
teidict2dicttex
  ***parte de*** – dicttex
  ***instância de*** – conversor
  ***descrição*** – conversor de dicionários em formato de TEI/XML printing dictionaries para LaTeX/DictTeX
  ***acerca de*** – XML ◇ TEI ◇ LaTeX
  ***autor*** – J.João Almeida
  ***necessita de*** – XML::DT
termos compostos
  ***instância de*** – conceito geral
  ***descrição*** – termo multipalavra; termo composto por várias palavras em que a semântica do termo não é inferível a partir da semântica das partes constituintes.
  ***referido em*** – dicionário TC ◇ PTC
text::translate
  ***instância de*** – tool ◇ biblioteca Perl
  ***descrição*** – naïf text translation
  ***acerca de*** – document processing ◇ reescrita ◇ tradução automática
  ***autor*** – J.João Almeida
  ***necessita de*** – mktextrr
  ***secção*** – 8.6
TFS
  ***referido em*** – Dac ◇ DPL
thesaurus
  ***instâncias*** – anarco-thesaurus Alfarrábio
  ***parte de*** – biblioteca digital
  ***descrição*** – Ontologia classificativa
  ***termo associado*** – catálogo ◇ Library::Thesaurus
  ***referido em*** – library::* ◇ wmboi
  ***página*** – 51 178 227
tool
  ***instâncias*** – flex ◇ camtex ◇ camila prototyping environment ◇ DAG ◇ freqnormpt ◇ campretty ◇ text::translate ◇ divisor de sílabas ◇ DDMF ◇ library::*
tradução automática
  ***referido em*** – text::translate
TXT
  ***pode ser gerado por*** – DPL

wmboi
  ***instância de*** – biblioteca Perl
  ***descrição*** – Web Mining Based on Indexes – pequeno robot de extracção de árvores catálogo
  ***acerca de*** – data mining ♢ thesaurus
  ***secção*** – 8.5

XML
  ***instância de*** – linguagem etiquetada
  ***termo associado*** – TEI
  ***referido em*** – XML::DT ♢ teidict2dicttex
  ***pode ser gerado por*** – library::*

XML::DT
  ***instância de*** – biblioteca Perl
  ***descrição*** – XML::DT a Perl down translate module;
  ***acerca de*** – XML ♢ document processing
  ***autor*** – J.João Almeida
  ***necessita de*** – XML::Parser
  ***Ref*** – (Almeida and Simões, 2003; Almeida and Ramalho, 1999);
  ***secção*** – 6.6
  ***usado em*** – anarco-thesaurus Alfarrábio ♢ teidict2dicttex

XML::Parser
  ***usado em*** – XML::DT

YaLG
  ***instância de*** – logic grammar ♢ biblioteca Prolog
  ***descrição*** – yet another logic grammar – DCG + acesso a dicionário externo (usando NLlex e jspell) + reescrita léxica, sintáctica, semântica e pragmática
  ***acerca de*** – corpora ♢ pragmática ♢ reescrita ♢ sintaxe ♢ gramática de ordem superior ♢ semântica
  ***autor*** – J.João Almeida
  ***necessita de*** – NLlex
  ***Ref*** – (Almeida, 1995);
  ***secção*** – 6.3