INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

RuDriCo2 - Um Conversor Baseado em Regras de Transformação Declarativas

Cláudio Filipe Paiva Diniz

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Professor Doutor António Rito Silva
Orientador:	Professor Doutor Nuno João Neves Mamede
Co-Orientador:	Professor Doutor João Dias Pereira
Vogal:	Professor Doutor António Paulo Teles de Menezes Correia Leitão

Outubro 2010

Agradecimentos

Gostaria de agradecer a dedicação, a disponibilidade e a mestria do meu orientador - Professor Nuno Mamede. Gostaria de agradecer também ao meu co-orientador - Professor João Dias Pereira pela ajuda preciosa que prestou.

Lisboa, 26 de Novembro de 2010
Cláudio Diniz

Resumo

No processamento de texto existem palavras ambíguas na medida em que uma palavra tanto pode ter a categoria de verbo como de nome, por exemplo. De forma a resolver estas ambiguidades, existe um módulo na cadeia de Processamento de Língua Natural (PLN) do L^2F, o RuDriCo. O RuDriCo é assim um desambiguador morfossintáctico baseado em regras que, além de desambiguar, também permite alterar a segmentação do texto.

Na comparação deste módulo com os restantes módulos da cadeia, verifica-se que este é substancialmente mais lento, sendo o desempenho do sistema um dos problemas abordados neste trabalho.

O sistema RuDriCo é baseado em regras e são estas que permitem a realização da desambiguação ou a alteração da segmentação. Por sua vez, essas regras são escritas pelo utilizador, sendo a sua sintaxe uma das características mais importantes do sistema.

Tendo em conta estes aspectos, o presente documento aborda a desambiguação morfossintáctica a partir de um estudo entre os principais sistemas da área. Aqui, é ainda analisada a sintaxe das regras do RuDriCo, sendo comparada com outra sintaxe. Em consequência desta comparação, são descritas as várias alterações ao sistema RuDriCo que o tornam no sistema RuDriCo2, nomeadamente as alterações à sintaxe das regras e as optimizações ao algoritmo principal do sistema.

Abstract

There are ambiguous words in text processing, as a word can have the name category and the verb category, for instance. In order to resolve these ambiguities, there is a module in the Natural Language Processing (NLP) chain of L^2F, RuDriCo. The RuDriCo is a morphological disambiguator based on rules with the possibility to change the segmentation of the text.

When comparing this module with others in the chain, it appears that the RuDriCo is the bottleneck of the chain. RuDriCo's performance is one of the problems addressed in this work.

The system RuDriCo is based on rules that enable the realization of disambiguation or the segmentation changes. The rules are written by the user and the rules' syntax is one of the most important features of the system.

Given these aspects, this paper addresses the morphological disambiguation, conducting a study among the major systems of the area. The syntax of the rules is analyzed and compared with another syntax. Consequently, several modifications are described in RuDriCo to implement RuDriCo2, including changes in rule syntax and optimizations in the system's main algorithm.

Palavras Chave
Keywords

Palavras Chave

Processamento de Língua Natural
Desambiguação Morfossintáctica
Regras
Sintaxe
Desempenho

Keywords

Natural Language Processing
Morphological Disambiguation
Rules
Syntax
Performance

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O Processamento de Língua Natural (PLN) é um dos tópicos mais importantes da área de Inteligência Artificial. Muitos dos sistemas desenvolvidos nesta área, como sistemas de diálogo ou sistemas de correcção ortográfica, usam um conjunto de módulos responsáveis pelo processamento de texto.

A cadeia de Processamento de Língua Natural do L^2F (Laboratório de Sistemas de Língua Falada), representada na figura 1.1, é composta por 6 módulos: o Segmentador, o Palavroso [Medeiros, 1995], o Sentences, o RuDriCo [Pardal, 2007], o Marv [Ribeiro et al., 2003] e o XIP [Xerox, 2003].

O Segmentador recebe o texto de entrada e divide o mesmo em segmentos, correspondendo a cada segmento, por exemplo, uma palavra ou um caractere de pontuação.

O Palavroso é um analisador morfológico que é responsável pela anotação morfossintáctica. Este recebe os dados do Segmentador e, com o auxílio de um dicionário, atribui todas as possíveis anotações a cada segmento. Neste processo, há palavras que ficam com mais do que uma anotação, uma vez que são palavras ambíguas.

Depois do texto estar anotado, o módulo Sentences agrupa os segmentos em frases de acordo com a pontuação.

O módulo que sucede o analisador morfológico é o desambiguador morfossintáctico RuDriCo (Rule Driven Converter), cujo objectivo principal é resolver as ambiguidades introduzidas pelo Palavroso. Este módulo também é usado para alterar a segmentação adaptando-a às necessidades dos módulos seguintes. O RuDriCo usa um conjunto de regras declarativas de transformação para realizar a desambiguação de palavras e a alteração da segmentação. Note-se que essas regras se baseiam no conceito de emparelhamento de padrões.

O Marv, por sua vez, é uma ferramenta de desambiguação probabilística que faz a desambiguação morfossintáctica estatística, pelo que é esta ferramenta que soluciona os problemas de ambiguidade deixados pelos módulos anteriores. O Marv baseia-se em modelos de Markov e usa o algoritmo de Viterbi [Viterbi, 1967] para encontrar a anotação mais provável para cada palavra. A probabilidade de uma anotação existir num determinado contexto é estimada usando um *corpus* de treino.

O último módulo da cadeia de processamento é o XIP, um analisador sintáctico que, com o auxílio

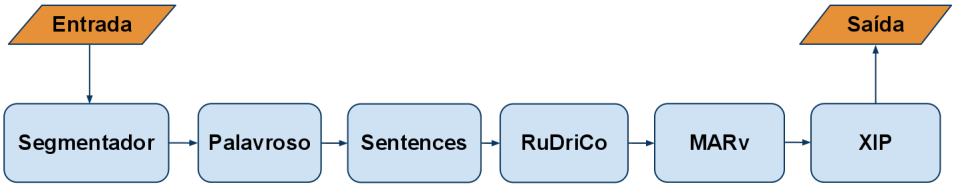

Figura 1.1: Cadeia de Processamento de Língua Natural do L^2F

de gramáticas, obtém a estrutura gramatical do texto de entrada. Este analisador tem um módulo que permite fazer desambiguação por regras e tratar de algumas ambiguidades que não são tratadas nem pelo RuDriCo, nem pelo Marv.

O presente trabalho centra-se em apenas um dos módulos da cadeia de PLN do L^2F acima descritos, a ferramenta de desambiguação morfossintáctica RuDriCo. Tomando como exemplo a palavra ambígua "comer", verifica-se que o analisador morfológico (Palavroso) lhe atribui as categorias de verbo e nome. Em seguida, o RuDriCo utiliza um conjunto de regras declarativas para escolher a anotação correcta, tendo em conta o contexto em que a palavra ocorre. Assim sendo, na frase "Vou comer uma cenoura", o desambiguador morfossintáctico deve ter uma regra que opte pela anotação de verbo para a palavra "comer" em detrimento da anotação nome.

A ferramenta de desambiguação morfossintáctica é, deste modo, importante na cadeia de Processamento de Língua Natural, pois se não tiver sucesso ao desambiguar as palavras, o analisador sintáctico (XIP) ficará bastante mais sobrecarregado [Hagège et al., 1998].

Nesta tese, após o estudo de ferramentas de desambiguação morfossintáctica e após a análise do RuDriCo original, são descritas todas as alterações que possibilitaram a transformação do RuDriCo no RuDriCo2. Por último, é realizada uma avaliação do sistema RuDriCo2, na sua comparação com o sistema RuDriCo.

1.1 Objectivos do Trabalho

Este trabalho tem como ponto de partida o sistema de desambiguação RuDriCo [Pardal, 2007]. Este é um sistema que, além de desambiguar, também altera a segmentação do texto, no entanto, apresenta ainda os seguintes problemas:

- baixo desempenho;
- regras pouco expressivas;
- limite no tamanho do ficheiro de entrada;
- sequências de regras que podem gerar ciclos infinitos.

O trabalho aqui apresentado visa solucionar alguns dos problemas do sistema RuDriCo e, como tal, tem dois objectivos principais: aumentar a eficiência do RuDriCo e tornar a sua sintaxe mais expressiva e mais compacta.

Em relação à eficiência do RuDriCo, pode observar-se na tabela 1.1, onde se apresenta um estudo do tempo de processamento de cada um dos módulos da cadeia de processamento de texto do L^2F, que o RuDriCo é o módulo que demora mais tempo, tendo, como tal, um baixo desempenho.

Módulo	Tempo de CPU (ms/palavra)
Segmentador	0.11
RuDriCo	4.24
Marv	0.20
XIP	1.67
Conversões entre módulos	0.50
Total	6,73

Tabela 1.1: Média do tempo de CPU gasto por palavra na cadeia de processamento de texto do L^2F

Quanto às regras, o RuDriCo é baseado em regras declarativas e a expressividade das mesmas caracteriza-se pela sintaxe e pelos operadores disponíveis. No entanto, há regras que não podem ser

escritas no RuDriCo, porque a sintaxe deste não é suficientemente expressiva, estas regras são escritas no módulo de desambiguação por regras do sistema XIP.

Adicionalmente, pretende-se resolver o problema das regras que provocam recursão e o limite no tamanho do ficheiro de entrada.

Capítulo 2

Trabalho Relacionado

Este capítulo descreve os métodos normalmente usados para fazer desambiguação morfossintáctica e apresenta o sistema RuDriCo [Pardal, 2007] e o sistema XIP [Xerox, 2003]. Este último inclui módulos para realizar a desambiguação morfossintáctica e a análise sintáctica, no entanto, aqui, será ignorado o módulo de análise sintáctica, pois esse não cabe no âmbito deste trabalho.

Na secção 2.1, são descritos os vários métodos de desambiguação morfossintáctica, sendo que é analisado em maior detalhe o método baseado em regras, pois é aquele que está a ser usado pelo RuDriCo. Na secção 2.2, é feita uma descrição do sistema RuDriCo e são apresentadas as suas funcionalidades. Já na secção 2.3, descrevem-se as funcionalidades do sistema XIP. Em último lugar, na secção 2.4, é feita uma comparação entre os dois sistemas .

2.1 Desambiguação Morfossintáctica

Na cadeia de PLN do L^2F, co-existem dois desambiguadores morfossintácticos, o RuDriCo e o Marv [Ribeiro et al., 2003], e, ainda assim, são resolvidas algumas ambiguidades no XIP. Os desambiguadores morfossintácticos podem ser classificados consoante a metodologia usada para resolver o problema. [Cole et al., 1995] dividem estes sistemas em dois tipos:

- desambiguadores baseados em regras;
- desambiguadores probabilísticos (estocásticos).

Há ainda autores que classificam estes sistemas em outros tipos, como, por exemplo, [Schmid, 1994b], [Schmid, 1994a] e [Schulze et al., 1994] que classificam estes sistemas em mais um tipo distinto: desambiguadores baseados em redes neuronais. Contudo, neste trabalho, vão ser consideradas apenas as classificações de [Cole et al., 1995].

Os sistemas de desambiguação baseados em regras, também conhecidos como sistemas com conhecimento linguístico [Márquez and Padró, 1997], são os sistemas alvo deste trabalho. As regras usadas nestes sistemas são escritas por linguistas, o que requer um esforço adicional. As regras verificam o contexto em que uma palavra está inserida e depois, consoante o mesmo, fazem a respectiva desambiguação. Os desambiguadores deste tipo deixam algumas ambiguidades por resolver, mas, mesmo assim, é comum os sistemas actuais terem uma taxa de acerto próxima de 99%[1]. Os principais trabalhos realizados nesta área são:

- *Computational Grammar Coder* (CGC) [Klein and Simmons, 1963];

[1] A taxa de acerto não tem em conta as palavras que não são desambiguadas.

- TAGGIT [Greene and Rubin, 1962];
- EngCG [Voutilainen, 1995a] [Voutilainen, 1995b];
- XIP [Xerox, 2003];
- RuDriCo [Pardal, 2007];
- Brill Tagger [Brill, 1992].

Na comparação destes trabalhos, observa-se que o CGC é um analisador e desambiguador morfológico que, com o auxílio de um léxico de 1500 palavras, começa por tratar de algumas excepções de que o analisador morfológico não consegue dar conta. Depois, é executado o seu analisador morfológico e, por último, o seu sistema de desambiguação que é baseado em regras (tem cerca de 500 regras). Já o TAGGIT baseia-se no CGC, mas utiliza um léxico maior. O EngCG também não é só um desambiguador, pois, para além de anotar e desambiguar, realiza ainda algumas tarefas extra, como a segmentação do texto anterior à anotação. A sequência de tarefas do EngCG é a seguinte:

- segmentação;
- análise morfológica;
- desambiguação morfológica;
- procura de anotações sintácticas alternativas;
- desambiguação sintáctica de estado finito.

O desambiguador morfológico pode ser visto como um conjunto de regras, sendo que cada regra especifica um ou mais contextos onde uma anotação está errada. Uma anotação é retirada se o padrão se verificar e, caso a palavra fique só com uma anotação, esta deixa de ser ambígua. Este sistema deixa entre 3% e 7% de palavras ambíguas, mas a sua taxa de acerto é de 99,7%.

Os sistemas XIP e RuDriCo serão retomados em grande detalhe nas secções seguintes.

Por último, o sistema Brill Tagger descrito em [Brill, 1992] é um analisador morfológico que tem em atenção o contexto das palavras quando lhes atribui as anotações. Este sistema é baseado em regras aprendidas automaticamente, ou seja, na atribuição da anotação, o sistema realiza a desambiguação baseada em regras. Uma das desvantagens dos sistemas baseados em regras é a sua escrita, mas [Brill, 1992] mostra que, com a aprendizagem automática das regras, a escrita pode ser evitada. Este sistema começa por atribuir a anotação mais provável a cada palavra, ignorando o contexto, e, depois, realiza a parte de aprendizagem que considera oito tipos de regras pré-definidas. O sistema instancia-as e escolhe as regras que têm uma taxa de erro menor. Após esta selecção, as regras são aplicadas ao texto. O autor refere também que este sistema pode ser expandido para obter melhores resultados, se forem adicionadas algumas regras escritas manualmente.

Os sistemas de desambiguação probabilísticos usam um *corpus* anotado de onde extraem a probabilidade de uma palavra ter uma anotação num determinado contexto. Têm vindo a ser utilizados vários métodos para a desambiguação probabilística, como, por exemplo, o método baseado em probabilidades condicionadas, por [Stolz et al., 1965] no sistema WISSYN; os métodos baseados em bigramas, por [Garside et al., 1997] no CLAWS e por [Church, 1988] no Parts; o método com modelos de Markov não observáveis, por [Cutting et al., 1992] num sistema da XEROX; o método baseado em n-gramas, por [Schmid, 1994b] e o método baseado em entropia máxima [Ratnaparkhi, 1996] [Ratnaparkhi, 1998]. O desambiguador probabilístico da cadeia de processamento do L^2F é o Marv [Ribeiro et al., 2003] que tem como base modelos de Markov.

Os sistemas de desambiguação probabilísticos são os mais utilizados, mas na cadeia de processamento do L^2F usa-se uma estratégia híbrida: o RuDriCo seguido do Marv, que tem uma taxa de acerto global de 94,23%. Para melhorar esta taxa, antes do texto passar no Marv, executa-se o sistema de desambiguação morfossintáctica RuDriCo. Este, por sua vez, tem um conjunto de regras de desambiguação cujo objectivo é a resolução dos casos em que o Marv falha. Com esta estratégia, consegue-se aumentar a taxa de desambiguação.

2.2 RuDriCo

O sistema RuDriCo é baseado em regras e tem duas funcionalidades: a desambiguação morfossintáctica e a alteração da segmentação. Na secção 2.2.1, são descritos os dados de entrada e as funcionalidades do sistema. Na secção 2.2.2, são apresentadas as regras que permitem fazer a desambiguação e, na secção 2.2.3, são descritas as regras que permitem fazer a alteração da segmentação.

2.2.1 Descrição do sistema RuDriCo

O sistema RuDriCo recebe como entrada um ficheiro em formato XML constituído por um conjunto de frases, sendo que cada frase tem um ou mais segmentos (word), como se pode observar na figura 2.1. Cada segmento tem uma forma superficial e pode ter uma ou mais anotações (class), que consistem no lema, que está sempre presente, e no conjunto de pares propriedade e respectivo valor. Uma das propriedades mais frequente é a categoria da forma superficial.

```
<l2f_annotation>
  <sentence>
    <word name="Forma superficial">
      <class root="Lema">
        <id atrib="propriedade 1" value="valor 1"/>
        ...
      </class>
      ...
    </word>
    ...
  </sentence>
  ...
</l2f_annotation>
```

Figura 2.1: Estrutura do ficheiro XML que é dado como entrada no RuDriCo

No caso de o analisador morfológico categorizar uma forma superficial com mais de uma anotação, há ambiguidade. Antes de se observar um caso de ambiguidade, importa ter em conta que, nas anotações que o analisador morfológico atribui às palavras, o nome e o valor das propriedades são abreviaturas, pelo que a propriedade CAT com o valor "nou" significa que a propriedade categoria tem o valor nome. Assim, como se pode observar na figura 2.2, a forma superficial "pesquisa" tem três anotações. No primeiro caso, o lema coincide com a forma superficial "pesquisa" e a categoria atribuída é nome singular do género feminino. No segundo caso, em que o lema é "pesquisar", a categoria atribuída é verbo e as propriedades correspondem ao tempo verbal, modo, número e pessoa. No terceiro caso, o lema também é "pesquisar", mas as propriedades do verbo são outras.

O RuDriCo permite resolver ambiguidades deixadas pelo analisador morfológico, com o auxílio de regras de desambiguação, e também permite alterar a segmentação do texto, fazendo contracção e descontracção de segmentos, com as regras de recomposição.

```
<word name="pesquisa">
  <class root="pesquisa">
    <id atrib="CAT" value="nou"/>
    <id atrib="NUM" value="s"/>
    <id atrib="GEN" value="f"/>
  </class>
  <class root="pesquisar">
    <id atrib="CAT" value="ver"/>
    <id atrib="MOD" value="ind"/>
    <id atrib="TEN" value="prs"/>
    <id atrib="PER" value="3"/>
    <id atrib="NUM" value="s"/>
  </class>
  <class root="pesquisar">
    <id atrib="CAT" value="ver"/>
    <id atrib="MOD" value="imp"/>
    <id atrib="PER" value="2"/>
    <id atrib="NUM" value="s"/>
  </class>
</word>
```

Figura 2.2: Exemplo das várias anotações que o analisador morfológico atribui à palavra "pesquisa"

A contracção ocorre quando se juntam dois ou mais segmentos num só. Na figura 2.3 apresenta-se um exemplo de como a segmentação da frase "A Coreia do Sul é na Ásia" é transformada, na medida em que "Coreia do Sul" passa a corresponder a um só segmento. A descontracção ocorre quando se

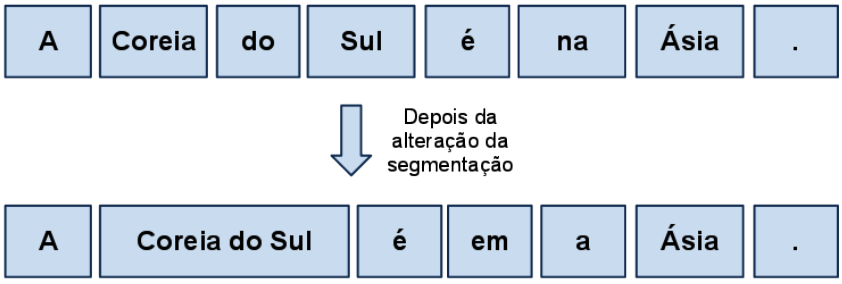

Figura 2.3: Alteração de segmentação

tem um segmento com a respectiva forma superficial e há necessidade de transformá-lo em dois ou mais segmentos. No exemplo da figura 2.3, a descontracção acontece no segmento de forma superficial "na" que se transforma em dois segmentos: "em" e "a".

2.2.2 Regras de desambiguação

Neste primeiro momento, impõe-se descrever a sintaxe das regras no sistema RuDriCo:

$$\text{antecedente} --> \text{consequente} .$$

No antecedente, definem-se as condições de emparelhamento da regra que, por sua vez, define um padrão. Quando o antecedente emparelhar com uma sequência de segmentos, a respectiva sequência é substituída pelo consequente. O antecedente é constituído por um conjunto de um ou mais itens e cada um tem a seguinte constituição:

$$\text{forma\_superficial ['lema' , 'prop 1'/'valor 1' , 'prop 2'/'valor 2' ...]}$$

A forma superficial é uma palavra e o que está entre parênteses rectos é um bloco, constituído pelo lema e por uma ou mais propriedades com o respectivo valor. Os itens emparelham com segmentos e os blocos emparelham com anotações, podendo haver mais do que um bloco em cada item, como se verifica no exemplo seguinte:

'escrita'['escrita','CAT'/'nou']['escrever','CAT'/'ver']

Neste caso, emparelha com o item uma forma superficial "escrita" que tenha as seguintes anotações: o lema é "escrita" e a categoria é nome; o lema é "escrever" e a categoria é verbo. Note-se que o consequente tem a mesma sintaxe do antecedente.

Para que o sistema possa suportar regras mais complexas e mais expressivas, é usado um conjunto de operadores, havendo uns que se usam só no antecedente e outros que se usam só no consequente das regras. Os operadores para usar no antecedente estão descritos na tabela 2.1 e podem ser usados com qualquer item no antecedente.

Operador	Descrição
[...]+	Deve ocorrer um ou mais segmentos que emparelhem com o item.
[...]*	Deve ocorrer zero ou mais segmentos que emparelhem com o item.
[...]?	Deve ocorrer zero ou um segmento que emparelhem com o item.
[...]n+	Deve ocorrer "n"ou mais segmentos que emparelhem com o item.
[...]n1,n2	Deve ocorrer entre "n1"e"n2"segmentos que emparelhem com o item.
[...][...]	Representa que o segmento deve emparelhar com ambas as anotações.
[...]!	Representa que todas as anotações do segmento devem emparelhar com o item.
[...]!!	Representa que o segmento só pode ter uma anotação igual à do item.

Tabela 2.1: Operadores do sistema RuDriCo para os itens do antecedente

A forma superficial, o lema e os valores das propriedades podem ser substituídos por variáveis, sendo que o nome da variável tem de ter a primeira letra capitalizada. Tal faz com que uma regra não seja específica para uma determinada forma superficial, lema ou valor de uma propriedade. Considere-se o item seguinte:

S1[L1 , 'CAT'/'pre']

As variáveis S1 e L1 permitem que qualquer segmento emparelhe com este item, desde que a sua categoria seja preposição. O valor de uma variável no consequente é o valor da propriedade que emparelhou com o respectivo item. Se no antecedente uma variável emparelhar com mais do que um valor, o que pode acontecer com o auxílio de alguns dos operadores apresentados na tabela 2.1, a variável guarda todos os valores. Na tabela 2.2, descrevem-se os operadores que se podem usar no consequente. Nesta tabela, a utilização de [...] na descrição dos operadores significa que um bloco é reescrito no consequente.

O RuDriCo adiciona um segmento no início e um segmento no fim de cada frase antes de a processar. O segmento adicionado no início é o segmento Head e o segmento adicionado no fim é o segmento TAIL. Estes segmentos podem ser testados no antecedente de uma regra e são usados como itens, por exemplo:

Head, S1[L1 , 'CAT'/'pre']

este antecedente emparelha com os primeiros segmentos de cada frase se estes tiverem uma anotação de preposição. O segmento TAIL é usado da mesma forma, mas testa se uma regra emparelha apenas no final da frase, por exemplo:

Operador	Descrição
S1 + S2[...]	Concatenação sem espaço dos valores das duas variáveis, S1 e S2.
S1 @+ S2[...]	Concatenação com espaço dos valores das duas variáveis, S1 e S2.
"_"+ s1 + "_"	Concatenação de "_"com o valor da variável S1, seguido de "_".
S1*[...]	No caso de, no antecedente, a variável S1 emparelhar com mais que um segmento, faz a concatenação sem espaços de todas as formas superficiais.
S1@*[...]	No caso de, no antecedente, a variável S1 emparelhar com mais que um segmento, faz a concatenação com espaços de todas as formas superficiais.
S1 [...]	No caso de, no antecedente, a variável S1 emparelhar com mais que um segmento, a variável fica com o primeiro valor que obteve.
S1$[...]	No caso de, no antecedente, a variável S1 emparelhar com mais que um segmento, a variável fica com o último valor que obteve.
S1*	Representa o valor do segmento que emparelhe com a variável S1, sem nenhuma mudança ao segmento.
S1[L1,...]*	Representa a sequência de segmentos que emparelham com a variável S1 no antecedente, com um novo bloco.
S1[L1,...]-	Representa o segmento que emparelhe com a variável S1 no antecedente, excluindo a anotação presente no bloco.
S1[L1,...]+	Representa o segmento que emparelhe com a variável S1 no antecedente, mantendo a anotação presente no bloco, e excluindo as restantes.

Tabela 2.2: Operadores do sistema RuDriCo para os itens do consequente

S1[L1 , 'CAT'/'pre'],TAIL

este antecedente emparelha com os últimos segmentos de cada frase se estes tiverem uma anotação de preposição. Note-se que o segmento Head é usado como primeiro item de uma regra e o segmento TAIL é usado como último item.

Para exemplificar as regras de desambiguação, considere-se a forma superficial "poder". Esta forma superficial pode ser um nome ou um verbo, mas se o contexto em que está inserida for analisado, consegue-se desambiguar e atribuir a categoria correcta. A regra para desambiguar esta forma superficial no caso de ser um verbo é:

S1 [L1,'CAT'/'pre']
S2 ['poder','CAT'/'nou'] ['poder','CAT'/'ver','MOD'/'inf']
S3 [L3,'CAT'/'ver','MOD'/'inf'] $--> $
S1*
S2 ['poder','CAT'/'nou']-
S3* .

O antecedente da regra significa que, para esta regra emparelhar, é necessário existir: um segmento com a categoria preposição, seguido de um segmento com duas anotações com o lema "poder", onde, na primeira, a categoria é nome e, na segunda, a categoria é verbo no modo infinitivo ('CAT'/'ver','MOD'/'inf'), seguido de um segmento anotado de verbo no modo infinitivo. Quando o antecedente é satisfeito, a forma superficial "poder" é desambiguada. Note-se que é usado o operador " - " no consequente, para retirar uma das anotações da forma superficial com o lema "poder". Esta regra emparelha, por exemplo, com a frase "Para poder andar".

2.2.3 Regras de Recomposição

As regras de recomposição dividem-se em dois subtipos: regras de contracção e regras de descontracção. O RuDriCo tem a mesma sintaxe para todos os tipos de regras, portanto, a sintaxe destas regras é igual à sintaxe das regras de desambiguação, apresentada na secção 2.2.2. Os operadores usados nas regras também são comuns em todos os tipos de regras no RuDriCo. Como exemplo, apresenta-se uma regra de contracção:

'coreia' [L1,'CAT'/C1]
'do' [L2,'CAT'/C2]
'sul' [L3,'CAT'/C3]
$-->$
'Coreia do Sul' ['Coreia do Sul','CAT'/'nou','GEN'/'f','NUM'/'s'] .

e uma regra de descontracção:

'na' [L1,'CAT'/'pre']
$-->$
'em' ['em','CAT'/'pre']
'a' ['o','CAT'/'art','SCT'/'def','NUM'/'s','GEN'/'f'] .

Estas regras realizam as transformações representadas na figura 2.3. A primeira regra faz a contracção de "Coreia do Sul" num só segmento e a segunda faz a descontracção do segmento "na" nos segmentos "em" e "a".

2.3 XIP

O sistema XIP tem um conjunto de funcionalidades mais extenso que o RuDriCo. Na secção 2.3.1, são apresentadas as funcionalidades do sistema. A única funcionalidade que o XIP tem em comum com o RuDriCo é a desambiguação. Em 2.3.2, são apresentadas as regras que a permitem realizar.

2.3.1 XIP

A representação dos dados de entrada no sistema XIP é diferente do sistema RuDriCo. No RuDriCo, os dados de entrada são representados por uma estrutura sequencial de segmentos. No sistema XIP, os dados de entrada são um conjunto de nós representados numa estrutura hierárquica, como mostra a figura 2.4. Os nós folha representam os segmentos do texto, enquanto os nós intermédios contêm propriedades sobre os nós folha.

No XIP, existem regras de desambiguação, mas não existem regras de contracção nem de descontracção, existe sim um outro tipo de regras que são as regras de *chunking*. Este tipo de regras é constituído por dois subtipos: regras de sequência e regras ID/LP. As regras de sequência adicionam nós à hierarquia, por exemplo, uma regra que diz que um determinante seguido de um nome constituem um sintagma nominal. Quando se aplica a regra à árvore da figura 2.4, o resultado é a árvore da figura 2.5. A diferença das regras ID/LP para as de sequência é que nas regras ID/LP a ordem dos nós é ignorada.

O XIP tem mais dois tipos de regras: regras de dependência e regras para modificar a árvore de *chunks*. As regras de dependência servem para modificar as dependências entre os nós da estrutura hierárquica. As regras para modificar a árvore de *chunks* servem para modificar a estrutura hierárquica dos nós depois desta ser aumentada pelas regras de *chunking*. As regras de dependência e as regras para

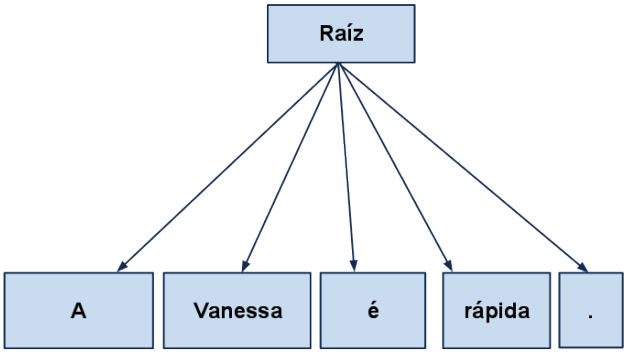

Figura 2.4: Representação dos segmentos da frase "A Vanessa é rápida" no sistema XIP

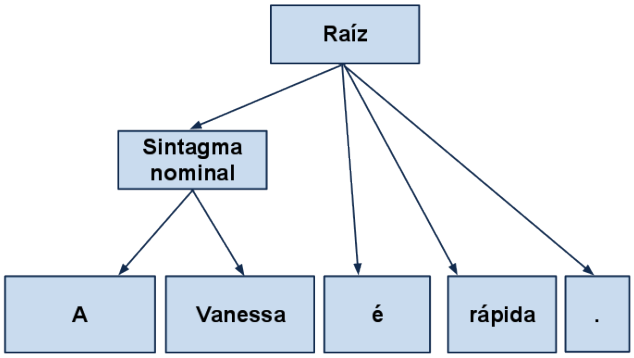

Figura 2.5: Hierarquia resultante da aplicação de uma regra de sequência

modificar a árvore de *chunks* não são abordadas em detalhe porque saem fora do âmbito deste trabalho, dado que a estrutura dos dados de entrada no RuDriCo não é hierárquica.

2.3.2 Regras de desambiguação

No sistema XIP, a sintaxe das regras é a seguinte:

$$\text{camada} > \text{antecedente} = |\text{contexto à esquerda}|\ \text{consequente}\ |\text{contexto à direita}|$$

As regras estão organizadas por camadas, sendo que cada camada é representada por um número e constituída por zero ou mais regras. As regras são aplicadas de acordo com as camadas a que pertencem, começando por serem aplicadas as regras da camada com o menor número. As regras que não têm camada são colocadas na camada de maior prioridade, a camada número zero.

O antecedente é um nó e representa-se por uma ou mais categorias separadas por vírgulas. Exemplo:

$$\text{noun,verb}$$

Este antecedente emparelha quando há um segmento que tenha duas anotações, uma anotação com categoria verbo e outra com categoria nome. Adicionalmente à categoria, podem-se representar outras propriedades. Para representar as propriedades dos nós, existe o conceito de propriedades locais e de propriedades globais. As propriedades globais são o conjunto das propriedades de todas as anotações de um nó. As propriedades locais referem as propriedades de cada anotação individualmente. Para representar propriedades globais, usa-se o operador parênteses rectos ([ ]) e, para representar propriedades locais, usa-se o operador menor e maior (< >). Por exemplo, o antecedente:

$$\text{noun} <\text{sing}>$$

emparelha com um segmento que tenha a categoria nome e a propriedade singular na mesma anotação. Neste exemplo, testa-se uma propriedade local, mas, se fosse usado o operador ([ ]), o antecedente emparelharia com um segmento que tivesse a categoria nome e a propriedade singular, mesmo que estas pertençam a anotações diferentes. Há propriedades que o XIP coloca automaticamente ao receber uma frase como entrada: no primeiro nó de uma frase, adiciona a propriedade first; no último nó de uma frase, adiciona a propriedade last. Em todas as formas superficiais que comecem por uma letra capitalizada, adiciona a propriedade Uppercase e, às formas superficiais que tenham todas as letras capitalizadas, adiciona a propriedade AllUppercase. O lema e a forma superficial também são propriedades presentes nos nós.

O contexto à direita e o contexto à esquerda são representados por sequências de nós. Para o consequente de uma regra ser aplicado, o contexto à esquerda tem de emparelhar antes do antecedente e o contexto à direita tem de emparelhar depois do antecedente. Nos contextos, os nós são representados por uma categoria e pelas propriedades. Existem dois operadores que permitem representar vários nós nos contextos das regras: o operador "," representa a conjunção de nós e o operador ";" representa a disjunção de nós.

O consequente é representado por um nó, tal como o antecedente. Se o antecedente e os contextos emparelharem, o segmento que emparelhar no antecedente mantém as propriedades representadas no consequente e descarta as restantes. Exemplo:

$$\text{noun,verb} = |\text{noun}|\ \text{verb}\ |\text{noun}|$$

Neste caso, a regra indica que se houver um segmento que foi classificado morfologicamente como nome e verbo, que tenha um segmento classificado como nome antes e outro depois, a anotação de categoria nome é descartada. Como a anotação de categoria nome é descartada, o segmento deixa de ser ambíguo porque fica apenas com a categoria verbo.

Existem dois operadores que são usados na construção de regras: "*" e "?". O operador "*" significa zero ou mais, o operador "?" significa zero ou um. Estes dois operadores usam-se nas regras como nós, com excepção do operador "?" que, adicionalmente, pode ser usado como propriedade de um nó. Por exemplo, se o operador "?" for usado como categoria, representa que se pode ter um nó com uma categoria qualquer ou não ter nenhum a emparelhar com o operador. Existe também um operador que permite que se mude a forma superficial ou o lema, é o operador "+=".

Existe também o conceito de variável, em que uma variável fica associada a um nó. Esta é declarada nas regras quando um nó é precedido por "#número", sendo que o número tem de ser maior ou igual a um. As variáveis servem para testar os valores das propriedades dos nós. Por exemplo, a regra

$$\text{noun,verb} = |\text{noun\#1}|\ \text{verb}\ |\text{noun}|$$

associa a variável #1 ao nó que emparelhar com o contexto à esquerda. São descritos na tabela 2.3 um conjunto de operadores a ter em conta quando se usam variáveis. No entanto, há um operador que não está descrito na tabela, o "Where". Este operador usa-se no fim das regras e funciona como mais um teste para a regra ser aceite. Segue-se um exemplo explicativo da utilização deste operador:

$$\text{nó} = \text{nó1} \mid \text{nó2\#1} \mid \text{Where(\#1[propriedade:valor])}$$

Neste caso, o operador testa o valor de uma propriedade do nó nó2. Se existir mais do que uma variável, podem-se comparar valores das propriedades de variáveis diferentes.

Para testar um valor de uma propriedade de um nó, existe o conjunto de operadores descritos na tabela 2.4. Refira-se que existem mais operadores deste tipo, mas foram omitidos porque saem do âmbito desta tese. Existe também mais um operador que não está presente na tabela, o "%". Este operador permite que sejam atribuídas propriedades no consequente que não estejam representadas no nó antecedente. Por exemplo:

Operador	Descrição
variável[prop1:valor,prop2:valor]	Testa um conjunto de propriedades numa variável.
variável1::variável2	Verifica se duas variáveis representam o mesmo nó. Os 2 nós têm de ter todas as propriedades com o mesmo valor.
variável1:variável2	Compara as propriedades de um nó, com outro, e vê se todas as propriedades de um dos nós estão presentes no outro.
variável1∼:variável2	Verifica se duas variáveis são nós diferentes.
variável1 < variável2	Verifica se a variável1 está antes da variável2 na estrutura hierárquica.
variável1 > variável2	Verifica se a variável1 está depois da variável2 na estrutura hierárquica.
∼	É usado em conjunto com outros operadores, representando a negação.
variável1[propriedade]=nó1,nó2...	Transfere propriedades de um nó para outros.
teste1 & teste2	Verifica se dois testes são verdade.
teste1 | teste2	Verifica se um de dois testes é verdade.

Tabela 2.3: Operadores do sistema XIP para as variáveis

Operador	Descrição
[propriedade:valor]	Se a propriedade não tiver o respectivo valor, o teste falha.
[propriedade=valor]	A propriedade fica com este valor, a não ser que já tenha outro e neste caso o teste falha.
[propriedade:?]	Se a propriedade não tiver um valor, o teste falha.
[propriedade]	Se a propriedade não tiver um valor, o teste falha.
[propriedade:∼]	Se a propriedade tiver um valor, o teste falha.
[propriedade:∼valor]	Se a propriedade tiver o valor referido, o teste falha.
[propriedade∼]	No caso da propriedade ter um ou mais valores, fica sem nenhum.

Tabela 2.4: Operadores do sistema XIP para as propriedades

pron<lemma:"nada"> %= |verb|adv|prep|

Neste caso, para esta regra emparelhar, o antecedente significa que é necessário que exista um segmento com uma anotação de categoria pronome (pron) e lema "nada". Se o antecedente emparelhar, verificam-se os contextos. Neste caso, o contexto à esquerda indica que tem de existir um segmento com uma anotação de categoria verbo antes do segmento em questão e o contexto à direita indica que a seguir ao segmento tem de existir um segmento com a categoria preposição. Se o antecedente e os contextos se verificarem, o segmento passa a ter apenas uma anotação de categoria advérbio. Note-se que a categoria advérbio não está presente no antecedente. Considerando a frase "Não vou fazer nada ao shopping", a regra vai emparelhar com a parte da frase "fazer nada ao".

Para concluir esta secção, apresenta-se mais um exemplo de uma regra de desambiguação:

num,adj,noun = | num,adj;noun,?[lemma:e],(art) | num

O objectivo desta regra é desambiguar um segmento que tenha três anotações correspondentes a três categorias diferentes: número, adjectivo e nome. Neste exemplo, o antecedente especifica que um segmento tem de ter as três anotações referidas para emparelhar com esta regra. O contexto à esquerda indica que tem de ocorrer antes do segmento em questão um segmento anotado de número (num), seguido de um anotado de adjectivo (adj) ou de nome (noun), sendo que, opcionalmente, poderá haver um segmento que tenha o lema "e" depois destes e, finalmente, poderá ter um anotado de artigo. Se o contexto e o antecedente existirem nos dados, o segmento é desambiguado, ficando anotado com a categoria número. A seguinte frase é um exemplo que emparelha com esta regra: "duas canecas e um quarto", onde a palavra "quarto" que o analisador morfológico classifica com três categorias é desambiguada para a categoria número.

2.4 Principais diferenças entre os 2 sistemas

A tabela 2.5 apresenta um resumo das funcionalidades de cada sistema, sendo possível, assim, compará-los. Como se pode observar, as três funcionalidades não contempladas no RuDriCo são específicas

Funcionalidade	Sistema RuDriCo	Sistema XIP
Regras de desambiguação	x	x
Regras de contracção	x	
Regras de descontracção	x	
Regras de *chunking*		x
Regras para modificar a árvore de *chunks*		x
Regras de dependência		x

Tabela 2.5: Funcionalidades dos sistemas RuDriCo e XIP

para a estrutura hierárquica de nós do XIP. Note-se que o XIP não permite realizar mudanças à segmentação original do texto, ou seja, não tem regras de contracção ou de segmentação.

Ao analisar as sintaxes das regras dos sistemas, nota-se que há diferenças cruciais. No RuDriCo, a forma superficial e o lema estão sempre presentes quando se representa um item. No XIP, a forma superficial e o lema são propriedades e podem ser omitidas. Há regras que não necessitam de usar o lema nem a forma superficial. No RuDriCo, ignora-se a forma superficial e o lema usando variáveis, o que é uma desvantagem em relação ao XIP, porque o uso de variáveis requer mais computação.

Ainda no que diz respeito à análise da sintaxe, é de salientar que no RuDriCo não existe o conceito de camada. Se houver uma regra cujo efeito emparelhe com as condições de outra regra e se esta segunda repuser as condições da primeira, vai haver recursão entre as duas. O algoritmo que aplica as

regras vai aplicar a primeira e, de seguida, aplica a segunda, depois, volta a aplicar a primeira e assim sucessivamente. Para evitar estes casos de recursão, a solução proposta pelo RuDriCo é limitar o número de passos no algoritmo. No XIP, utilizando as camadas, consegue-se evitar este problema, uma vez que basta colocar as regras que provocam recursão em camadas diferentes, porque o algoritmo que aplica as regras, em cada passo, só aplica as regras de uma camada. É de notar que o RuDriCo testa todas as regras em cada passo do algoritmo, enquanto o XIP testa apenas um subconjunto que corresponde às regras de uma camada.

O conceito de contexto à esquerda e à direita existe no XIP mas não no RuDriCo. Contudo, é possível simular os contextos com a ajuda de operadores e variáveis. Tome-se como exemplo a seguinte regra no sistema XIP:

noun,verb = |det| noun |verb|

Agora, observe-se a mesma regra no RuDriCo:

```
S0[L0,'CAT'/'det']
S1[L11,'CAT'/'noun'][L22,'CAT'/'verb']
S2[L2,'CAT'/'verb']
-->
S0*
S1[L11,'CAT'/'noun']+
S2* .
```

Como se pode observar, no RuDriCo é necessário usar variáveis e a regra é mais extensa.

Quanto às variáveis, estas têm diferenças entre os dois sistemas, sendo aplicadas a conceitos diferentes devido à representação dos dados em cada um desses sistemas. No RuDriCo, uma variável pode ser associada ao lema, à forma superficial ou a uma propriedade. No XIP, uma variável é associada a um nó que contém uma propriedade que representa o lema, uma propriedade que representa a forma superficial e as restantes propriedades. No XIP, se o operador "?" for usado no lugar de um nó, o operador representa uma variável anónima, conceito este que não existe no RuDriCo. No XIP, podem-se comparar duas variáveis, para saber se estas são o mesmo nó, enquanto no RuDriCo não existe o conceito de comparação entre variáveis que representem um nó.

No que diz respeito aos operadores, estes são elementos que permitem dar expressividade às regras. Em primeiro lugar, refira-se um dos operadores do XIP que não está presente no RuDriCo, o operador negação "∼". Este operador pode ser usado para testar se uma propriedade não tem um determinado valor ou se duas variáveis são nós diferentes. Como exemplo, considere-se a seguinte regra no sistema XIP:

art<lemma:o,gen:m>,pron<lemma:o,gen:m>=| [verb:∼] | art | noun<gen:m> |

Nesta regra, o operador negação é aplicado à categoria verbo e significa que o contexto à esquerda emparelha com qualquer categoria excepto verbo. Para traduzir esta regra para o sistema RuDriCo, é necessário recorrer a várias regras, nomeadamente uma regra por categoria, excepto a categoria negada, que é verbo. Segue-se um excerto do grupo de regras que traduzem a regra do XIP:

```
S1 [L1,'CAT'/'adv']
S2 ['o','CAT'/'art','GEN'/'m'] ['o','CAT'/'pro','GEN'/'m']
S3 [L3,'CAT'/'nou','GEN'/'m']
-->
```

S1*
S2 ['o','CAT'/'pro']-
S3* .

S1 [L1,'CAT'/'adj']
S2 ['o','CAT'/'art','GEN'/'m'] ['o','CAT'/'pro','GEN'/'m']
S3 [L3,'CAT'/'nou','GEN'/'m']
−− >
S1*
S2 ['o','CAT'/'pro']-
S3* .

S1 [L1,'CAT'/'nou']
S2 ['o','CAT'/'art','GEN'/'m'] ['o','CAT'/'pro','GEN'/'m']
S3 [L3,'CAT'/'nou','GEN'/'m']
−− >
S1*
S2 ['o','CAT'/'pro']-
S3* .

. . .

Como se pode comprovar com este exemplo, a solução no RuDriCo para simular a negação de uma categoria tem como grande desvantagem a necessidade de nomear todas as categorias excepto a que se nega. Esta solução não é escalável porque quanto mais categorias existem, mais regras têm de ser feitas. Uma outra desvantagem reside na adição de propriedades, na medida em que se for adicionada uma categoria, por exemplo, todas as regras que usaram esta solução para simular a negação vão ter de ser revistas.

No XIP, testa-se se uma propriedade tem algum valor, independentemente do valor em questão, usando o operador"?". Também é possível testar se um nó está antes ou depois de outro nó, utilizando os operadores "<" e ">". Ainda no XIP, para testar se uma propriedade tem um respectivo valor, usa-se o operador ":", embora este ignore as restantes propriedades que não são testadas no nó, assim como quando se testam propriedades nos itens do antecedente no RuDriCo. No entanto, com o operador "::", pode-se testar se um nó só tem um certo conjunto de propriedades e mais nenhuma. No RuDriCo, também é possível fazer este tipo de teste, usando o operador "!!".

Um dos operadores que é apresentado juntamente com as regras de desambiguação do XIP é o operador disjunção ";", conceito esse que não existe no RuDriCo. Como exemplo, considere-se a seguinte regra no sistema XIP:

noun,verb = |det;prep| noun |verb|

Como não existe disjunção no RuDriCo, para escrever uma regra equivalente a esta é necessário recorrer a duas regras, uma para cada elemento da disjunção:

S0[L0,'CAT'/'det']

S1[L11,'CAT'/'noun'][L22,'CAT'/'verb']
S2[L2,'CAT'/'verb']
−− >
S0*
S1[L11,'CAT'/'noun']+
S2* .

S0[L0,'CAT'/'prep']
S1[L11,'CAT'/'noun'][L22,'CAT'/'verb']
S2[L2,'CAT'/'verb']
−− >
S0*
S1[L11,'CAT'/'noun']+
S2* .

Com base neste exemplo, é possível perceber como a inexistência do operador disjunção no RuDriCo constitui uma desvantagem. A solução encontrada para simular a disjunção não é, contudo, uma boa solução, pois, para realizar a disjunção entre cinco elementos, é necessário escrever cinco regras.

Ainda assim, analisando a comparação dos sistemas, verifica-se, por um lado, que o RuDriCo tem a vantagem de poder alterar a segmentação e, por outro lado, que o XIP tem a sintaxe das regras mais expressiva e compacta.

Capítulo 3

Arquitectura original do RuDriCo

Neste capítulo, são descritas as principais classes do sistema RuDriCo e são descritos alguns dos algoritmos envolvidos no mesmo sistema.

As principais entidades do RuDriCo são as frases do texto de entrada e as regras usadas pelo sistema. Na secção 3.1, analisa-se a representação interna dessas entidades, a partir da enumeração dos principais constituintes de cada uma.

Na secção 3.2, são apresentadas as principais classes responsáveis pelo processamento; são analisadas as responsabilidades de cada classe, assim como alguns dos seus atributos e são também apresentados os principais algoritmos do sistema: o algoritmo de processamento de frases resultado e o algoritmo de emparelhamento de regras.

3.1 Representação interna das principais entidades

O RuDriCo recebe como entrada um ficheiro em formato XML com texto a processar, descrito na secção 2.2.1, e um ficheiro com as regras. O RuDriCo utiliza um *xml dom parser*[1] para ler o ficheiro de entrada, transformando cada frase do texto de entrada numa instância da classe Sentence. Utiliza-se um *parser* implementado em *lex* e *yacc*[2] para processar o ficheiro com as regras, e é este *parser* que transforma as regras em instâncias da classe RRule. Após a geração das entidades, o RuDriCo aplica as regras à entrada e gera um ficheiro em formato XML com o resultado, utilizando um *xml dom parser*.

A classe RRule é usada para representar as regras de desambiguação e de recomposição. Na figura 3.1, onde se apresenta o diagrama de classes de uma RRule, verifica-se que cada RRule tem uma instância da classe Antecedent e uma instância da classe Consequent. Cada um destes objectos contém uma lista de instâncias da classe RRuleItem que representa itens. A classe RRuleItem é constituída por um atributo _word, que representa a forma superficial do item, um conjunto de atributos boleanos, que representam os operadores associados a cada item, e uma lista de objectos da classe RRuleDesc. A classe RRuleDesc representa um bloco e é constituída pelo atributo _root, que representa o lema, e por uma lista de instâncias da classe Tag, que representam pares atributo-valor.

A classe Sentence é usada para representar internamente as frases de entrada descritas na secção 2.2.1. A figura 3.2 apresenta um diagrama de classes da classe Sentence. Cada Sentence tem um conjunto de objectos da classe Segment, sendo esta classe constituída por um atributo _word, que representa a forma superficial de um segmento, e por uma lista de instâncias da classe Description. A classe Description representa uma anotação e é constituída pelo atributo _root, que representa o lema, e por uma lista de instâncias da classe Tag. Cada objecto da classe Tag representa um par atributo-valor.

[1] http://www.w3.org/DOM/
[2] http://dinosaur.compilertools.net/

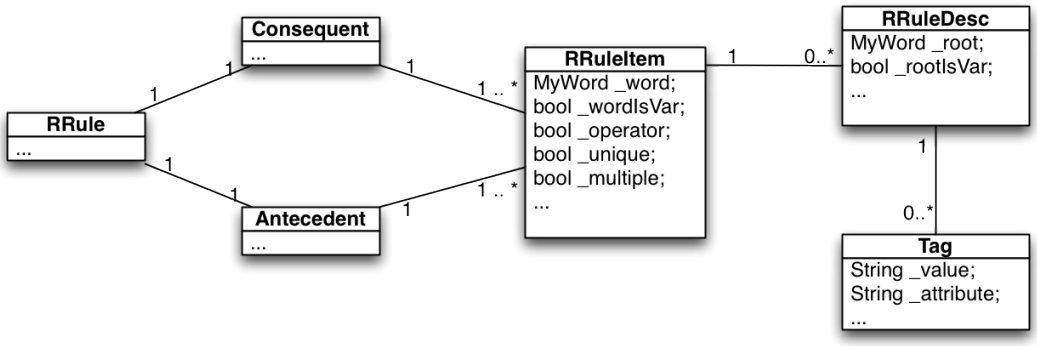

Figura 3.1: Diagrama UML simplificado da classe RRule

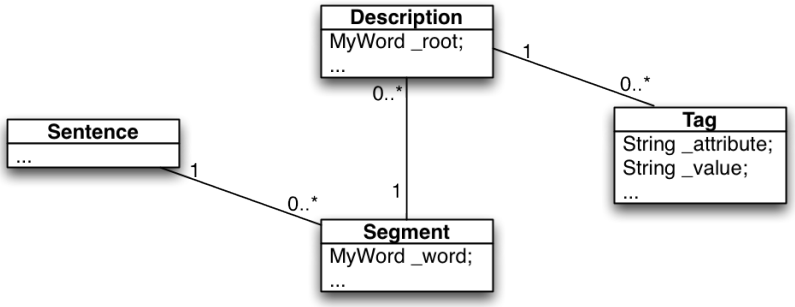

Figura 3.2: Diagrama UML simplificado da classe Sentence

Para cada Sentence, o algoritmo de regras permanece em execução até não haver mais regras para aplicar.

Para determinar as RRules que podem ser aplicadas, definiu-se a classe Arule e a classe AruleItem. A classe Arule representa uma regra que está efectivamente a emparelhar com um ou mais segmentos, ou seja, a classe Arule representa regras activas. A classe AruleItem representa itens das Arules. Na figura 3.3 e na figura 3.4 apresentam-se os diagramas de classes destes objectos.

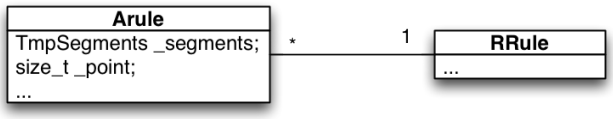

Figura 3.3: Diagrama UML simplificado da classe Arule

No que diz respeito à Arule, existem dois conceitos importantes: aumentar e aplicar. Aumentar uma Arule significa adicionar-lhe um segmento, desde que o segmento emparelhe no ponto onde o antecedente está a ser satisfeito. Aplicar uma Arule consiste em aplicar o consequente aos segmentos que emparelharam com o antecedente da regra. Uma Arule pode ser aplicada quando todos os itens do antecedente são satisfeitos. A classe Arule contém um objecto RRule associado, sendo que este que corresponde à regra que está activa. A Arule contém adicionalmente dois campos: o campo _segments e o campo _point. O campo _segments guarda os segmentos que emparelham com a RRule correspondente e estes são ordenados pela ordem de emparelhamento. O campo _point identifica os itens da RRule que já foram satisfeitos. As Arules têm um método importante para o algoritmo principal do RuDriCo, o método Add, que é o método que permite aumentar as Arules. Para verificar se um segmento emparelha com um item, o item

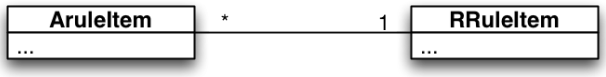

Figura 3.4: Diagrama UML simplificado da classe AruleItem

é transformado numa instância da classe AruleItem. Uma classe AruleItem tem o método Add e é este que verifica efectivamente se um segmento emparelha com o respectivo item, comparando a forma superficial e todas as outras propriedades.

3.2 Classes de processamento

Em seguida, apresentam-se as classes de processamento. O processamento do RuDriCo está então dividido em 3 classes: Rudrico, Analisador e Agenda. A primeira, descrita na secção 3.2.1, é a classe que processa o comando de entrada quando o RuDriCo é invocado. A segunda, descrita em 3.2.2, é a classe responsável pela entrada e saída de dados e pelo controlo da aplicação das regras. Esta classe lê as regras e frases e converte-as para a estrutura interna correspondente. Note-se que a classe Analisador também é responsável por fazer o pré-processamento das regras. A terceira, a classe Agenda, descrita em 3.2.3, contém a lógica de emparelhamento de regras com segmentos.

3.2.1 Classe Rudrico

As principais funcionalidades da classe Rudrico são ler o comando de entrada e controlar as tarefas a realizar. No comando de entrada, passa-se o comando a executar e os seus argumentos, por exemplo, a localização do ficheiro de regras e a localização do ficheiro de entrada. Consoante a informação passada no comando, o sistema vai ter um de dois comportamentos:

- pré-processar as regras para o formato optimizado;
- aplicar as regras ao texto de entrada.

Independentemente da funcionalidade, a responsabilidade desta classe é passar esta informação para a classe Analisador, de modo a que esta se comporte de acordo com a informação do comando de entrada. Outra responsabilidade da classe consiste em inicializar algumas das restantes classes do sistema, por exemplo, a classe Analisador.

3.2.2 Classe Analisador

As principais tarefas da classe *Analisador* são: (i) pré-processar as regras; (ii) ler as regras pré-processadas ou normais; e (iii) aplicar as regras ao texto de entrada.

As RRules são carregadas a partir de um ficheiro com regras no formato descrito nas secções 2.2.2 e 2.2.3. Este formato é fácil quanto à escrita e interpretação realizadas por humanos, mas, no RuDriCo, ler as regras e converter para a estrutura interna é um processo que pode ser agilizado. O pré-processamento das regras consiste na sua leitura para a representação interna (RRules), sendo as mesmas, consequentemente, guardadas num ficheiro. Este processo é uma serialização manual dos objectos que representam as RRules. O ficheiro de regras pré-processadas tem a seguinte constituição:

<Número de RRules>
<RRule 1>
...
<RRule N>

Esta optimização foi introduzida no RuDriCo por [Marques, 2008] e reduziu o tempo despendido ao carregar as regras para 57% face ao tempo inicial.

Na leitura das regras, estejam estas no formato normal ou pré-processado, é feita a sua indexação para facilitar o processo de escolha das regras candidatas a emparelhar com um segmento. A alocação de regras aos índices é realizada de acordo com o primeiro item de cada regra e tem como base as propriedades seguintes:

- a forma superficial e o lema são constantes;
- a forma superficial é constante e o lema é variável;
- a forma superficial é variável e o lema é constante;
- a forma superficial e o lema são variáveis.

Consoante a propriedade do primeiro item de cada regra, a regra é alocada no respectivo índice de regras que partilham essa mesma propriedade.

A aplicação das regras ao texto de entrada, implementada no método Analise, é realizada depois da leitura e da indexação das regras. Este método gere os segmentos que constituem as frases e recorre à classe Agenda para a aplicação de regras aos respectivos segmentos. Adicionalmente, o método também escreve as frases resultantes na saída do sistema. O método Analise executa o algoritmo apresentado na figura 3.5. Observe-se que o processamento é feito frase a frase, sendo que, em cada frase, o processamento é feito de segmento a segmento (passo 4), dado que a Agenda processa um segmento de cada vez e aplica as regras a conjuntos de segmentos. Quando o algoritmo está a processar uma frase (PreProcessedSentence) e descobre que a Agenda aplicou uma regra a um conjunto de segmentos da frase, este gera a frase resultante (PosProcSentence) com base na PreProcessedSentence e nos segmentos que a Agenda modificou (ChangedSegments), utilizando o método ConstructNewSentence. A frase resultante substitui a frase original e é processada para verificar se há alguma regra que se aplique. Se existir, este processo é repetido (passo 8). No caso de não existir nenhuma regra que se aplique à frase, dá-se por concluído o processamento dessa frase e a mesma é escrita na saída (passo 15).

3.2.3 Classe Agenda

A classe Agenda tem como tarefa principal a aplicação de regras a segmentos. Esta classe tem acesso às RRules e aos respectivos índices do Analisador e, com base nestes, tenta emparelhar os segmentos com as regras. Assim que alguma regra possa ser aplicada, a classe Agenda avisa o Analisador e a mesma é reinicializada. Os principais constituintes desta classe são uma lista de segmentos e uma lista de Arules (AruleList). A lista de segmentos inclui os segmentos que emparelham com as regras e a AruleList contém as regras parcialmente emparelhadas até ao momento, as Arules.

O método Add é o método principal da classe Agenda e é este que recebe os segmentos que o Analisador processa. Dependendo dos segmentos que este recebe, o método Add identifica se há alguma regra que emparelhe com um subconjunto desses segmentos. Este método tem o comportamento do algoritmo apresentado na figura 3.6.

```
Method Analise(Text T)
1:  FOR EACH sentence S in T DO
2:    PreProcessedSentence=S
3:    PosProcessedSentence={}
4:    I = first segment of PreProcessedSentence
5:    WHILE (I  is a segment)
6:      applied? = agenda(I)
7:      IF (applied?) THEN
8:        ConstructNewSentence(PosProcessedSentence,
                               PreProcessedSentence,ChangedSegments)
9:        PreProcessedSentence=PosProcessedSentence
10:       I = first segment of PreProcessedSentence
11:     ELSE
12:       Add(PosProcessedSentence,I)
13:       I = next segment of PreProcessedSentence
14:   ENDWHILE
15:   Write(PosProcessedSentence)
16: ENDFOR
```

Figura 3.5: Algoritmo de processamento de frases

```
Method add(Segment S)
1:  ForwardArules(S)
2:  SearchNewRules(S) /* this method adds new Arules to AruleList */
3:  Sort(AruleList)
4:  IF (Done?(First(AruleList))) THEN
5:    ApplyFirstRule(AruleList)
6:    Erase(AruleList)
7:    Return true
8:  ELSE
9:    Return false
```

Figura 3.6: Algoritmo de aplicação de regras a segmentos

O método ForwardArules realiza duas tarefas: (i) adiciona o segmento S a todas as Arules na AruleList e (ii) retira da AruleList todas as Arules que não podem ser aumentadas com o segmento.

O método SearchNewRules procura regras que emparelham com o segmento S, consulta todos os índices apresentados na secção anterior (3.2.2) e escolhe as regras que são candidatas a emparelhar com esse segmento. O método cria uma Arule para cada uma das regras seleccionadas e tenta adicionar o segmento S a estas Arules. As Arules que não podem ser aumentadas com o segmento são descartadas e as restantes são adicionadas à AruleList.

O método Sort ordena a AruleList de modo a que as regras fiquem por ordem de emparelhamento, ou seja, se a primeira regra da lista puder ser aplicada, é aplicada independentemente das restantes. A ordem da lista de Arules é mantida através de algumas propriedades destas, sendo a ordem em que as regras estão nos ficheiros o primeiro factor de ordenamento. Adicionalmente a este factor, quando existe na lista a mesma regra replicada com diferente número de segmentos emparelhados, a regra que aparece primeiro é a que tem mais segmentos. Para finalizar, quando existe a mesma regra replicada e ambas têm o mesmo número de segmentos emparelhados, o critério de desempate é o momento em que foram adicionadas à lista, pelo que a regra que tiver sido adicionada primeiro é a regra que aparece primeiro.

O método Done? verifica se uma Arule pode ser aplicada, ou seja, se a regra representada na Arule tem todos os itens do antecedente satisfeitos.

O método Apply1stRule é o método responsável por fazer as transformações aos segmentos que em-

parelhem com as Arules. A Arule a aplicar é a primeira da AruleList, pois, deste modo, garante-se que a regra que é aplicada é a regra com maior prioridade.

Para exemplificar o comportamento da Agenda, considere-se a frase "A Coreia de o Sul" e um ficheiro com as seguintes regras:

'coreia' [L1,'CAT'/C1]
'do' [L2,'CAT'/C2]
'sul' [L3,'CAT'/C3]
$--> $
'Coreia do Sul' ['Coreia do Sul','CAT'/'nou','GEN'/'f','NUM'/'s'] .

'de' ['de','CAT'/'prep']
'o' ['o','CAT'/'art']
$--> $
'do' ['do','CAT'/'pre'] .

O processamento da frase "A Coreia de o Sul" é apresentado nas figuras 3.7, 3.8 e 3.9. Relativamente a este exemplo, note-se que:

- para todas as frases, o sistema coloca o segmento Head no início da frase e o segmento Tail no fim;
- uma Arule tem um ponto depois de um item do antecedente que indica os itens que estão satisfeitos (à esquerda do ponto) e os itens que falta satisfazer (à direita do ponto);
- quando um arco acaba com uma cruz, a Arule é eliminada da AruleList;
- quando um arco acaba com um OK, a Arule é aplicada.

Tendo como ponto de partida a figura 3.7, observa-se que os dois primeiros segmentos da frase não adicionam nenhuma Arule à AruleList, uma vez que não há regras que emparelhem com estes. Já para o terceiro segmento, a Agenda encontra a regra que junta "Coreia do Sul" e coloca-a na AruleList, com o primeiro item satisfeito. Em seguida, e uma vez que a regra que junta "Coreia do Sul" não pode ser aumentada, esta é excluída da lista. Contudo, neste passo, ou seja, quando o segmento é adicionado à Agenda, o que acontece é que a regra que junta os segmentos "de" e "o" é colocada na AruleList. Quando o Analisador adiciona o segmento "o", nenhuma regra é encontrada, mas a Arule presente na AruleList é aumentada e fica no estado em que pode ser aplicada. Assim, como a primeira regra da lista pode ser aplicada à frase, os segmentos que faltavam processar são ignorados e a regra é aplicada. Depois do Analisador aplicar a regra à frase original, processa a nova frase, como se pode verificar na figura 3.8 onde está representado este passo. Aqui, a regra que junta os segmentos "Coreia do Sul" é aplicada assim que o segmento "Sul" é adicionado à Agenda. Consequentemente, a frase resultante é processada novamente (figura 3.9). Neste último passo, nenhuma regra foi encontrada, todos os segmentos foram processados e o Analisador dá por concluído o processamento.

Para exemplificar os casos em que a Agenda tem Arules que podem ser aplicadas apenas no final da frase, considere-se a mesma frase. Note-se que o ficheiro de regras apresenta as seguintes regras:

'S1[L1,'CAT'/C1]
S2['poder','CAT'/C2]
$--> $

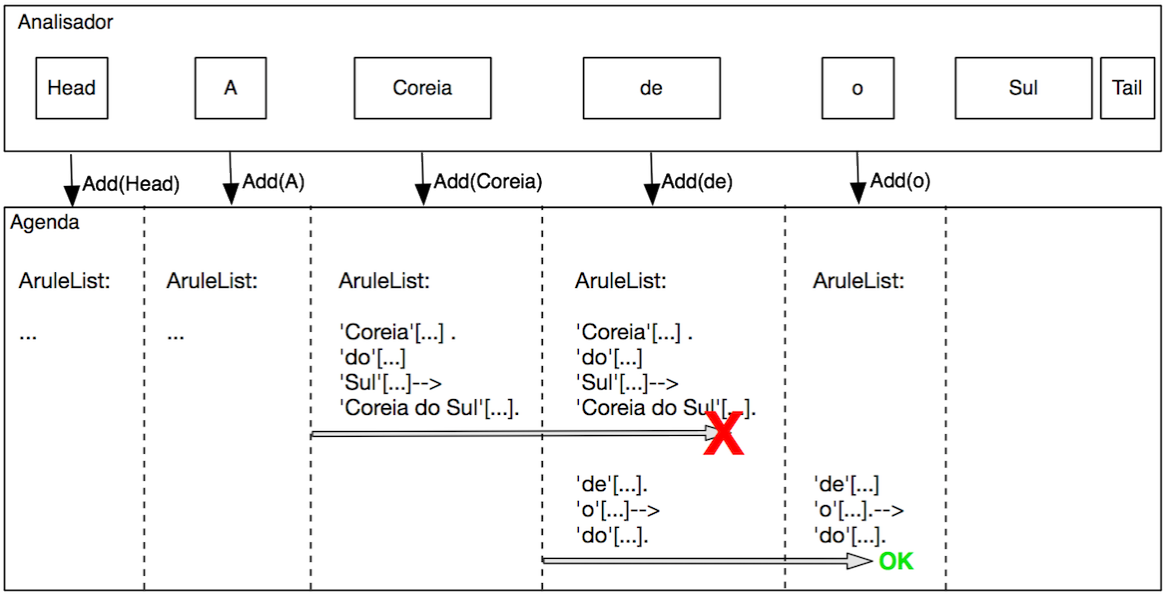

Figura 3.7: Primeiro passo do processamento da frase “A Coreia de o Sul”

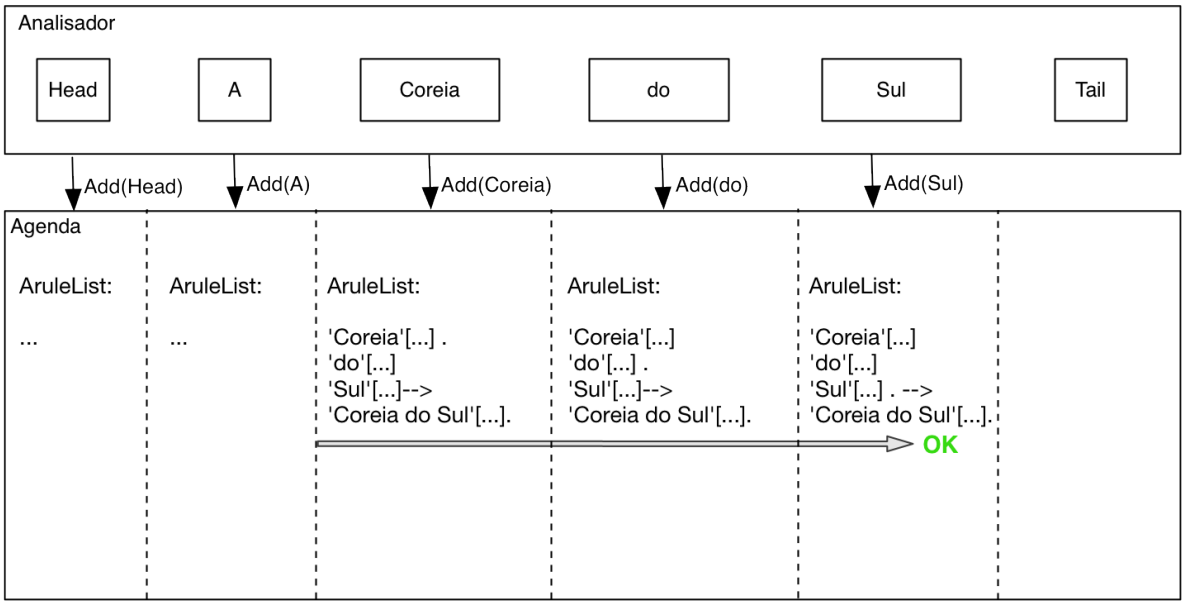

Figura 3.8: Segundo passo do processamento da frase “A Coreia do Sul”

S1*
S2['poder','CAT'/'ver''] .

'de' ['de','CAT'/'prep']
'o' ['o','CAT'/'art']
−− >
'do' ['do','CAT'/'pre'] .

A primeira regra é um exemplo de uma regra que emparelha o seu primeiro item com qualquer segmento, sendo esta a que tem mais prioridade, pois é a primeira do ficheiro de regras. Com estas regras, a frase é processada em dois passos que se encontram representados nas figuras: 3.10 e 3.11. No primeiro passo (figura 3.10), observa-se que, quando o segmento "o" é adicionado à Agenda, a AruleList fica com duas regras: a regra que emparelha com todos os segmentos e a regra que junta o segmento "de" ao segmento "o". De acordo com o algoritmo de aplicação de regras, é feita uma ordenação à AruleList indicando que a primeira regra do ficheiro tem mais prioridade do que a segunda, logo, a regra que junta os segmentos "de" com "o" não pode ser aplicada neste passo, uma vez que não é a primeira da lista. Esta regra é guardada na lista até poder ser aplicada ou até que haja uma regra mais prioritária que possa ser aplicada. No caso deste exemplo, a regra só pode ser aplicada quando o segmento Tail é adicionado à Agenda. No segundo passo (figura 3.11), deste modo, não há nenhuma regra que seja aplicada.

O sistema garante que as regras são aplicadas pela ordem em que aparecem nos ficheiros, na medida em que só se aplica uma regra quando esta é a primeira da AruleList.

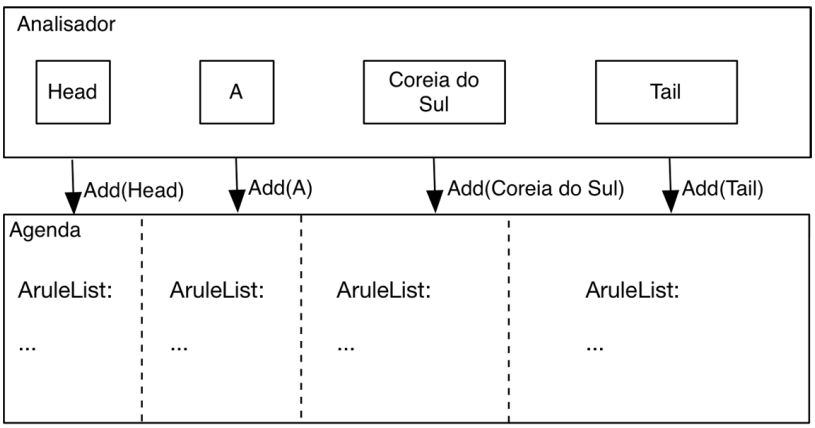

Figura 3.9: Terceiro passo do processamento da frase “A Coreia do Sul”

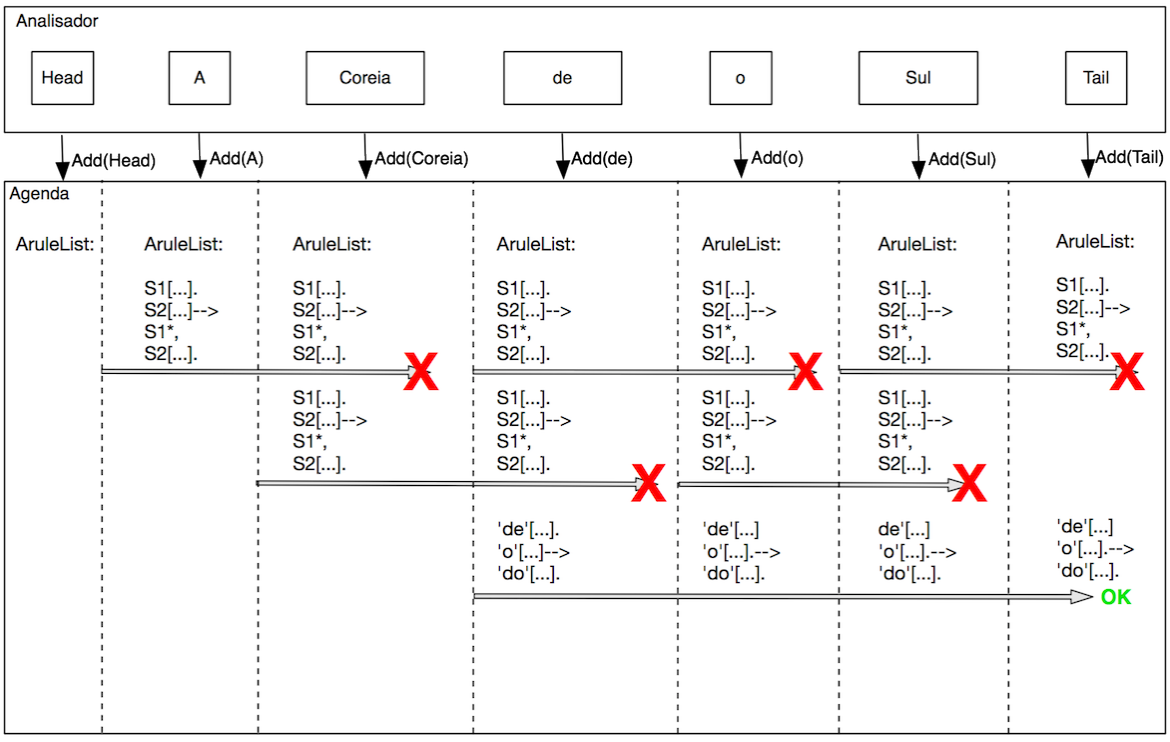

Figura 3.10: Primeiro passo do processamento da frase “A Coreia do Sul”, com um novo conjunto de regras

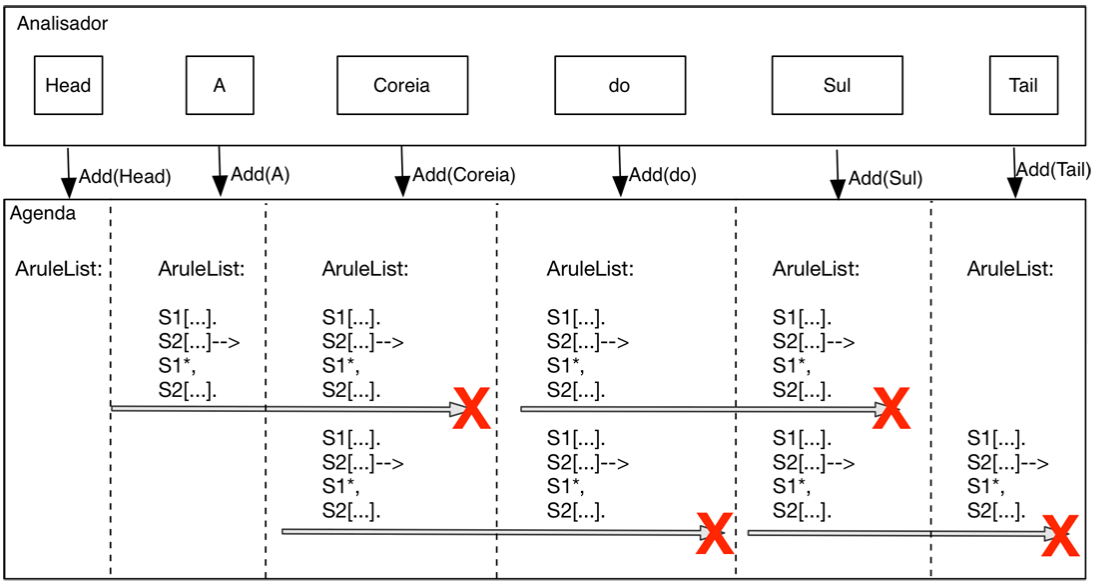

Figura 3.11: Segundo passo do processamento da frase “A Coreia do Sul”, com um novo conjunto de regras

Capítulo 4

Alterações ao sistema RuDriCo

À medida que as alterações são implementadas, a sintaxe do RuDriCo vai sendo modificada gradualmente, transformando-se na sintaxe do RuDriCo2.

A primeira alteração a ser realizada ao sistema original é a mudança do processo usado para ler e escrever os ficheiros, como se descreve na secção 4.1, sendo que, depois desta alteração, é introduzido o conceito de camada, explicitado na secção 4.2. Consequentemente, são introduzidos contextos e é alterada a representação dos itens, de modo a que a forma superficial e o lema possam não ser especificados, como se verifica em 4.3. Depois, na secção 4.4, é descrito um conjunto de propriedades e o modo de as calcular internamente.

Ao longo das alterações realizadas, verificou-se que cada tipo de regra tem um conjunto de características que não são comuns aos restantes tipos. Uma consequência desta situação é a necessidade de se alterar a sintaxe de cada tipo de regra para que cada uma possa ter uma sintaxe diferente, tal como descrito na secção 4.5. Na secção 4.6, são introduzidos novos operadores na sintaxe.

Os segmentos que representam o início e fim de frase são excluídos do processamento, como se explica na secção 4.7. Em 4.8, são apresentados novos métodos de validação de regras, por exemplo, para evitar que existam regras que têm variáveis inconsistentes. Para finalizar, na secção 4.9, são apresentadas as optimizações realizadas ao sistema.

4.1 Metodologia usada para escrita e leitura de ficheiros

O formato de dados de entrada e saída do RuDriCo é XML e e esses dados são processados com um *xml dom parser*. O *parser* constrói uma árvore em memória com todos os elementos do ficheiro de entrada e o algoritmo de aplicação de regras acede a esta árvore frase a frase. O problema desta solução, ou seja, de colocar toda a entrada em memória, é que não é possível executar o RuDriCo com ficheiros de entrada que excedam a memória. Como o RuDriCo processa os dados frase a frase, não há necessidade de ter todos os dados de entrada carregados em memória. A solução para este problema foi utilizar um *xml sax parser*[1], dado que este usa um *handler* que passa o controlo ao RuDriCo sempre que existir uma frase ainda não processada.

O RuDriCo usa o *xml dom parser* para escrever a saída do sistema, seja a saída direccionada para um ficheiro ou não. Esta tecnologia foi alterada porque o *parser* constrói uma árvore com todos os dados de saída, o que faz também com que a saída não possa exceder a memória. Assim, a solução encontrada consiste em escrever os dados de saída frase a frase.

Na arquitectura original do RuDriCo, a classe Analisador controla o fluxo dos dados de entrada. A classe Rudrico, após realizar as suas tarefas, invoca a classe Analisador e esta controla o fluxo de todo o

[1] http://www.w3.org/SAX

algoritmo. Para introduzir o *xml sax parser* nesta arquitectura, o controlo do fluxo de informação foi transferido para o *handler* do novo *parser*. Este *handler* lê uma frase de cada vez e invoca o Analisador para processar cada uma das frases. O Analisador, ao realizar a tarefa de processamento de dados de entrada, tem também a tarefa de ler os dados, mas, com esta alteração, a tarefa passou a ser apenas processar frases. A única mudança no algoritmo de processamento de frases do Analisador é não ter o ciclo sobre as frases, dado que o fluxo é controlado pelo *handler*. O algoritmo é apresentado em 4.1.

```
Method Analise(Sentence S)
 1:  PreProcessedSentence=S
 2:  PosProcessedSentence={}
 3:  I = first segment of PreProcessedSentence
 4:  WHILE (I  is a segment)
 5:    applied? = agenda(I)
 6:    IF (applied?) THEN
 7:      ConstructNewSentence(PosProcessedSentence,
                              PreProcessedSentence,ChangedSegments)
 8:      PreProcessedSentence=PosProcessedSentence
 9:      I = first segment of PreProcessedSentence
10:    ELSE
11:      Add(PosProcessedSentence,I)
12:      I = next segment of PreProcessedSentence
13:  ENDWHILE
14:  Write(PosProcessedSentence)
```

Figura 4.1: Algoritmo de processamento de frases resultados alterado

Com estas alterações, o algoritmo RuDriCo2 processa ficheiros de entrada que o RuDriCo não processava.

4.2 Camadas

No XIP, as camadas correspondem a subconjuntos de um conjunto de regras num ficheiro. A sintaxe do RuDriCo2 é igual à do XIP, ou seja, coloca-se o número da camada antes da regra:

$$\text{camada} > \text{antecedente} --> \text{consequente} .$$

As regras que não têm número da camada ficam na camada de maior prioridade, a camada número zero.

No RuDriCo, todas as regras são testadas no algoritmo de aplicação de regras, sendo testadas pela ordem em que estão no ficheiro de entrada. A entrada do RuDriCo é um ficheiro e este, por sua vez, é o resultado de uma junção das regras de vários ficheiros. Como exemplo, considere-se que as regras estão organizadas em três ficheiros, sendo que o primeiro tem regras de descontracção, o segundo tem regras de contracção e o terceiro tem regras de desambiguação. O conteúdo destes ficheiros é colocado no ficheiro final pela ordem em que foram referidos anteriormente. Deste modo, sabe-se que as regras de descontracção têm mais prioridade do que qualquer outro tipo de regras, porque são colocadas no início do ficheiro de entrada. O problema reside na adaptação do conceito de camada a múltiplos ficheiros, uma vez que cada um assume prioridade sobre os outros quando se juntam as regras no ficheiro de entrada.

A solução para este problema consiste em mudar o ficheiro de entrada para um ficheiro que indique os ficheiros que devem ser considerados. Nesta solução, as camadas são relativas ao ficheiro a que pertencem. Todas as camadas do primeiro ficheiro têm prioridade perante as camadas dos ficheiros seguintes, independentemente dos seus números. O número que representa a camada só é utilizado para ordenar camadas do mesmo ficheiro. Como exemplo, considerem-se três ficheiros ilustrados na figura 4.2, cuja ordem

de apresentação corresponde à ordem pela qual são colocados no ficheiro de entrada. A correspondência

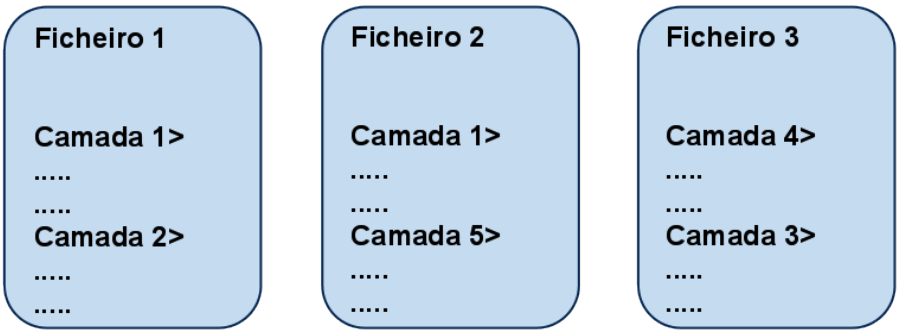

Figura 4.2: Exemplo de camadas em ficheiros de entrada

entre as camadas na representação interna e as camadas nos ficheiros está descrita na tabela 4.1.

Representação interna	Camadas nos ficheiros de entrada
Camada 0	Camada 1 do ficheiro 1
Camada 1	Camada 2 do ficheiro 1
Camada 2	Camada 1 do ficheiro 2
Camada 3	Camada 5 do ficheiro 2
Camada 4	Camada 3 do ficheiro 3
Camada 5	Camada 4 do ficheiro 3

Tabela 4.1: Correspondência entre camadas

A entrada do RuDriCo2 é um ficheiro constituído pelas localizações dos ficheiros de regras. A estrutura interna das regras do RuDriCo é uma lista de regras representada por um objecto da classe RRuleList. Para suportar as camadas, o RuDriCo2 guarda internamente um vector de objectos da classe RRuleList, em que cada posição corresponde a uma camada com a respectiva ordem. A composição das RRules não foi alterada.

A optimização ao RuDriCo feita por [Marques, 2008] sobre o facto de as regras serem pré-processadas teve de ser revista, porque agora existe um vector de listas de regras. Assim, foi adicionado um campo com o número de camadas ao ficheiro das regras pré-processadas. No RuDriCo2, este ficheiro tem a seguinte composição:

<Número de camadas>
<Número de RRules>
<RRule 1>
...
<RRule N>
<Número de RRules>
<RRule 1>
...
<RRule N>
...

Para introduzir as camadas na arquitectura original do RuDriCo, a classe Agenda foi modificada. Esta classe tem acesso a todas as regras que estão no sistema, de modo a escolher que regras emparelham com

os segmentos que recebe, no entanto, as regras passaram a estar organizadas em camadas, pelo que a Agenda precisa de saber que camada de regras vai usar em cada momento. Para resolver este problema, foi adicionado um parâmetro à Agenda, a camada. Assim, a Agenda só usa regras da camada indicada. A classe Analisador fica responsável por gerir a camada que a Agenda vai usar. Houve alterações no algoritmo apresentado na figura 4.1 para este suportar as camadas e gerir a camada que a Agenda vai utilizar. O novo algoritmo é apresentado na figura 4.3. Note-se que, se o sistema for iniciado apenas com uma camada, o algoritmo é idêntico ao algoritmo sem camadas. Um dos objectivos ao adicionar este conceito é resolver o problema da recursão entre regras, colocando as regras que provocam recursão em camadas diferentes.

```
Method Analise(Sentence S, LayerList layersList)
 1:  FOR EACH layer L in layersList DO
 2:    Agenda.layer(L) /*tells agenda to use rules from layer L */
 3:    IF (L==0) THEN
 4:      PreProcessedSentence=S
 5:    ELSE
 6:      PreProcessedSentence= PosProcessedSentence
 7:    PosProcessedSentence={}
 8:    I = first segment of PreProcessedSentence
 9:    WHILE (I  is a segment)
10:      applied? = agenda(I)
11:      IF (applied?) THEN
12:        ConstructNewSentence(PosProcessedSentence,
                                PreProcessedSentence,ChangedSegments)
13:        PreProcessedSentence=PosProcessedSentence
14:        I = first segment of PreProcessedSentence
15:      ELSE
16:        Add(PosProcessedSentence,I)
17:        I = next segment of PreProcessedSentence
18:    ENDWHILE
19:  ENDFOR
20:  Write(PosProcessedSentence)
```

Figura 4.3: Algoritmo de processamento de frases com camadas

4.3 Alteração da representação dos itens e introdução de contextos

Na sintaxe do RuDriCo, sempre que se escreve um item, é obrigatório colocar a forma superficial e o lema. Quando não se quer testar o lema ou a forma superficial, usam-se variáveis para o item poder emparelhar com qualquer lema ou forma superficial. Por exemplo:

S1 [L1,'CAT'/'pre']
S2 ['poder','CAT'/'nou'] ['poder','CAT'/'ver','MOD'/'inf']
S3 [L3,'CAT'/'ver','MOD'/'inf']
$--> $
S1*
S2 ['poder','CAT'/'nou']-
S3* .

nesta regra, usam-se as variáveis L1 e L3 no antecedente, mas não no consequente. O uso de variáveis para esta situação pode ser evitado se o lema e a forma superficial forem, como no XIP, pares propriedade-valor, pois deixa de ser obrigatório testá-los em todos os itens.

No RuDriCo2, passa a existir a propriedade lemma e a propriedade surface. A sintaxe é:

antecedente −− > consequente .

onde o antecedente e o consequente são constituídos por itens com a seguinte sintaxe:

[prop 1='valor 1' , prop 2='valor 2' ...][...]

Os itens são separados por uma vírgula e constituídos por um ou mais blocos (um bloco é o que está entre parênteses rectos):

[prop 1='valor 1' , prop 2='valor 2' ...][...],[prop 1='valor 1' , prop 2='valor 2' ...][...]

Nesta sintaxe, a propriedade surface só pode ocorrer uma vez em cada item e a propriedade lemma só pode ocorrer uma vez em cada bloco. Esta sintaxe permite escrever a regra anterior da seguinte forma:

```
[surface=S1,CAT='pre'],
[lemma='poder',CAT='nou'] [lemma='poder',CAT='ver',MOD='inf'],
[surface=S3,CAT='ver',MOD='inf']
−− >
S1*,
[lemma='poder',CAT='nou']-,
S3* .
```

Neste caso, a sintaxe permite usar menos três variáveis e a descrição dos nós é uniforme.

Os contextos são uma das vantagens identificadas no XIP, pois permitem escrever as regras mais simples e compactas, usando menos variáveis. Pelos motivos referidos, os contextos são introduzidos no antecedente das regras no RuDriCo2 e têm a seguinte sintaxe:

| contexto à esquerda | Item1 Item2 ... ItemN | contexto à direita |

Os contextos são compostos por itens e é possível usar todo o tipo de operadores permitidos nos itens do antecedente. Na secção 2.4, foi apresentada a seguinte regra XIP com contexto:

noun,verb = |det| noun |verb|

Para escrever esta regra no RuDriCo, é necessário usar variáveis para emparelhar com os itens que simulam os contextos e escrevê-los no consequente. Com a sintaxe do RuDriCo2, esta regra escreve-se da seguinte maneira:

```
|[CAT='det']|
[CAT='noun'][CAT='ver']
|[CAT='verb']|
−− >
[CAT='noun']+ .
```

Acerca do uso de contextos, importa referir que não é necessário reescrever no consequente os segmentos relativos ao contexto. Com a adição dos contextos e das alterações anteriores é possível tornar as regras mais simples, compactas e expressivas.

Tendo sido realizado um programa para converter as regras usadas pelo RuDriCo para a sintaxe do RuDriCo2, as regras passaram a usar contextos e a beneficiar da possibilidade de ocultar o lema e a forma superficial. Por exemplo, a seguinte regra:

[surface=S1,lemma=L1, CAT='art'],
[surface=S2,lemma=L2, CAT='num']*,
[surface=S3,lemma=L3, CAT='adj', NUM='s'],
[surface=S4,lemma=L4, CAT='nou'][lemma=L42, CAT='ver', PER='1']
−− >
S1*,
S2*,
S3*,
[surface=S4,lemma=L42, CAT='ver']-.

é convertida para:

|[CAT='art'],[CAT='num']*,[CAT='adj', NUM='s']|
[surface=S4,lemma=L4, CAT='nou'][lemma=L42, CAT='ver', PER='1']
−− >
[surface=S4,lemma=L42, CAT='ver']- .

Note-se que a propriedade surface do item do antecedente não é necessária, no entanto, não foi possível tratar este caso automaticamente. Com as novas funcionalidades, as regras são mais compactas e têm menos variáveis.

Para adicionar os contextos na arquitectura do RuDriCo, alterou-se a estrutura interna das regras (RRules) e a estrutura das Arules. Foram adicionados dois campos às RRules: um para guardar o contexto à esquerda e um para guardar o contexto à direita. O diagrama com estas alterações apresenta-se na figura 4.4. Nas Arules, foram adicionados quatro campos: um campo para guardar o contexto à esquerda, um campo para o contexto à direita e um campo _point para cada contexto. O campo _point identifica que segmentos do contexto estão satisfeitos na Arule. O novo diagrama das Arules é apresentado na figura 4.5.

Para suportar esta nova funcionalidade, as alterações mais relevantes foram realizadas no método ForwardRule e no método Add das Arules, referidos na secção 3.1, de modo a que os segmentos dos contextos sejam tratados como tal.

4.4 Propriedades automáticas

No RuDriCo, existe um conjunto de propriedades que são comuns a todas as regras. A propriedade que diz se uma forma superficial está capitalizada é um exemplo disso mesmo. Na secção 4.4.1, a propriedade de capitalização é apresentada e é descrita uma solução para que essa propriedade seja automática. As restantes propriedades são apresentadas em 4.4.2.

4.4.1 Capitalização

No RuDriCo, a capitalização dos segmentos é indicada por uma propriedade, a propriedade UPC. A propriedade UPC com o valor "true" representa o facto de uma palavra estar capitalizada. Esta

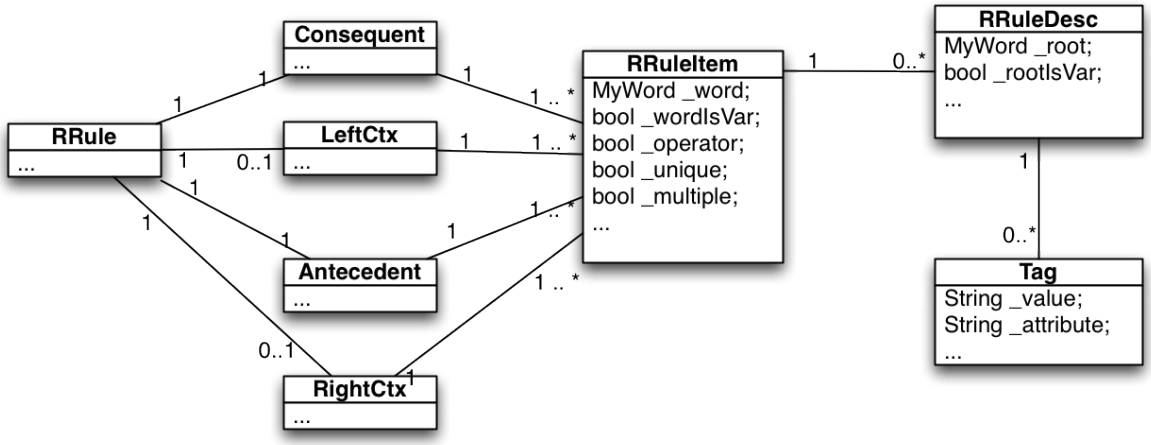

Figura 4.4: Diagrama da entidade RRule

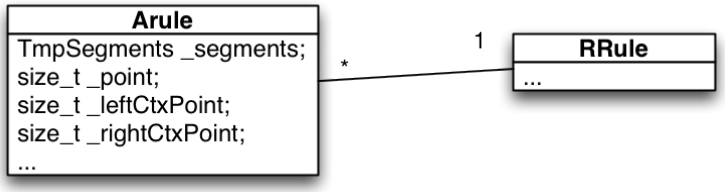

Figura 4.5: Diagrama da entidade Arule

propriedade existe porque o RuDriCo não identifica se uma palavra está capitalizada verificando o estado da forma superficial. Esta propriedade é irrelevante para os restantes módulos do sistema, mas tem de estar presente nos segmentos do RuDriCo. Considere-se o segmento representado na figura 4.6, onde se verifica que a propriedade UPC tem o valor "true". Aqui também se pode concluir que a palavra está capitalizada observando a forma superficial.

```
<word name="Posteriormente">
  <class root="posteriormente">
    <id atrib="CAT" value="adv"/>
    <id atrib="DEG" value="pst"/>
    <id atrib="UPC" value="true"/>
  </class>
</word>
```

Figura 4.6: Segmento da forma superficial "Posteriormente"

As regras que fazem novos segmentos têm de escrever a propriedade UPC, processo este que implica o uso de variáveis para emparelhar com a propriedade, de modo a calcular o valor da propriedade para os segmentos criados no consequente das regras. Observe-se o exemplo:

[surface='taco-a-taco',CAT='nou', UPC=U1]
−− >
[surface='taco-a-taco',lemma='taco-a-taco', CAT='adv', UPC=U1].

Para evitar o uso destas variáveis, inclui-se no RuDriCo2 um modo de distinguir as palavras capita-

lizadas por observação da forma superficial, para assim, ser possível remover a propriedade UPC do ficheiro de entrada.

Nas regras do RuDriCo2, quando existe um item no antecedente que testa a forma superficial, a capitalização desta é ignorada no emparelhamento com os segmentos. Para obrigar um item a emparelhar com segmentos que tenham a forma superficial capitalizada, usa-se a propriedade UPC com o valor "true". Por exemplo:

[surface='taco-a-taco', CAT='nou', UPC="true"]
−− >
[surface='taco-a-taco',lemma='taco-a-taco', CAT='adv'].

Esta regra só emparelha quando a forma superficial do item do antecedente estiver grafada com maiúscula.

Nos itens do consequente, a capitalização das formas superficiais é calculada automaticamente com base na capitalização dos segmentos que emparelham no antecedente e este cálculo depende do tipo de regra. Assim, para as regras de descontracção, a determinação da capitalização é realizada da seguinte forma:

- se a forma superficial do segmento que emparelha com o antecedente tem todos os caracteres capitalizados, as formas superficiais dos segmentos resultantes vão ter todos os caracteres capitalizados;
- se a forma superficial do segmento que emparelha com o antecedente é capitalizada, o primeiro segmento resultante do consequente fica com a forma superficial capitalizada e os restantes segmentos não ficam com a forma capitalizada;
- se a forma superficial do segmento que emparelha com o antecedente não for capitalizada, todas as formas superficiais dos segmentos resultantes não são capitalizadas.

Para as regras de contracção, a capitalização calcula-se usando a seguinte estratégia:

- se a forma superficial do segmento do consequente for construída com base em variáveis, usa-se a capitalização das formas superficiais emparelhadas;
- se a forma superficial do segmento do consequente for uma constante, a forma superficial do segmento resultante fica com a capitalização do primeiro segmento que emparelha com o antecedente.

Para as regras de desambiguação, como a segmentação não é alterada, os segmentos resultantes mantêm a capitalização dos segmentos que emparelham com o antecedente. Se o utilizador desejar um comportamento diferente deste, pode usar a propriedade UPC com o valor "true" nos segmentos do consequente para que estes fiquem com a forma superficial capitalizada. Pode-se usar também a propriedade UPC com o valor "false", para garantir que os segmentos não ficam capitalizados.

Existe uma alternativa ao cálculo automático da capitalização das formas superficiais, de modo a que o utilizador tenha mais liberdade sobre a capitalização, a propriedade CSE. Se a propriedade CSE com o valor "true" for usada no consequente, a forma superficial passa a ser *Case Sensitive*, ou seja, a capitalização da forma superficial final vai ser a capitalização da forma superficial que está na regra.

Nas regras do RuDriCo2, por omissão, os testes aos lemas são *Case Insensitive*, mas nos itens do consequente o lema é *Case Sensitive*: a capitalização usada é a capitalização dos lemas nos segmentos resultantes.

4.4.2 Propriedades extra

No RuDriCo, todos os segmentos têm uma propriedade que indica a posição do primeiro caractere relativamente à frase em que estão inseridos e outra que indica a posição do último caractere, respec-

tivamente designadas como propriedades LOW e HIGH. Estas propriedades foram ignoradas até agora porque não são relevantes para explicar as funcionalidades do RuDriCo. Na figura 4.7, apresenta-se um segmento com as respectivas propriedades. A propriedade LOW tem o valor "0" e significa que o primeiro

```
<word name="Posteriormente">
  <class root="posteriormente">
    <id atrib="CAT" value="adv"/>
    <id atrib="DEG" value="pst"/>
    <id atrib="LOW" value="0"/>
    <id atrib="HIG" value="13"/>
    <id atrib="UPC" value="true"/>
  </class>
</word>
```

Figura 4.7: Segmento da forma superficial "Posteriormente"

caractere da palavra está na posição zero da frase e que esta palavra é a primeira da frase a que pertence. A propriedade HIG tem o valor "13" porque a forma superficial tem 14 caracteres. O próximo segmento da frase a que este segmento pertence irá começar na posição 14.

A cadeia de processamento necessita que os segmentos resultantes mantenham as propriedades e as regras do RuDriCo têm de ter isso em conta. As regras que fazem novos segmentos têm de escrever estas propriedades e, sendo assim, este processo implica o uso de variáveis para emparelhar com as propriedades, de modo a passar o valor destas para os segmentos no consequente das regras. Observe-se o exemplo:

[surface='taco-a-taco', CAT='nou', LOW=W1, HIG=I2]
$--> $
[surface='taco-a-taco',lemma='taco-a-taco', CAT='adv', LOW=W1, HIG=I2].

As propriedades LOW e HIG são variáveis no antecedente, pelo que são escritas com o mesmo valor no consequente. Neste exemplo, onde o segmento do antecedente dá origem a um segmento do consequente, os valores das propriedades mantêm-se, mas o mesmo já não acontece nos casos de contracção e descontracção de segmentos. Começando por definir uma estratégia base para as regras de contracção, observe-se esta regra:

[surface=S1,lemma='sexta', NUM=N1, LOW=W1],
[surface=S2,lemma='feira', HIG=I2]
$--> $
[surface=S1 @+ S2,lemma='sexta-feira', CAT='nou', SCT='com', GEN='f', NUM=N1, LOW=W1, HIG=I2].

Os valores que estas propriedades vão ter no consequente são calculados tendo como base as seguintes regras:

- a propriedade LOW fica com o valor da propriedade LOW do primeiro segmento a emparelhar com a contracção;
- a propriedade HIG fica com o valor da propriedade HIG do último segmento a emparelhar com a contracção;

Para as regras de descontracção, a estratégia é diferente. Observe-se a seguinte regra:

[surface='nos', CAT='pre', LOW=W1, HIG=I1]

−− >
[surface='em',lemma='em', CAT='pre', LOW=W1, HIG=I1],
[surface='os',lemma='o',CAT='art',SCT='def',NUM='p',GEN='m',LOW=W1,HIG=I1].

A estratégia utilizada é a seguinte:

- a propriedade LOW dos segmentos resultantes fica com o valor da propriedade LOW do segmento que foi descontraído;
- a propriedade HIG dos segmentos resultantes fica com o valor da propriedade HIG do segmento que foi descontraído;

Note-se que o cálculo do valor das propriedades no consequente depende do tipo de regra que é realizada. No RuDriCo2, estas propriedades são calculadas automaticamente e, assim, deixam de existir estas duas propriedades nas regras. Se for desejado outro comportamento no cálculo das propriedades, é possível usar variáveis para calcular os valores das propriedades, como no RuDriCo original.

No RuDriCo original, todas as regras de descontracção têm duas propriedades ainda não referidas que marcam o primeiro e o último segmento de uma descontracção: a propriedade TOKENS (token start) e TOKENE (token end), respectivamente. Estas propriedades servem para identificar nos dados de saída do sistema que conjuntos de segmentos fazem parte de uma descontracção. Por exemplo, a seguinte regra:

[surface='àquele', CAT='pre']
−− >
[surface='a',lemma='a', CAT='pre', FOR='sim', TOKENS='true'],
[surface='aquele',lemma='aquele', CAT='pro', SCT='dem', NUM='s', GEN='m', TOKENE='true'].

marca os segmentos resultantes da descontracção com as propriedades TOKENS e TOKENE. Verifica-se que estas propriedades estão presentes em todas as regras de descontracção, sendo possível realizá-las automaticamente. O RuDriCo2 calcula estas duas propriedades automaticamente.

Ao realizar a conversão das regras para serem retiradas estas propriedades, verifica-se que o tamanho total dos ficheiros das regras diminuiu, pois as regras ficaram mais compactas. Ao calcular estas propriedades automaticamente, diminui-se também a probabilidade de haver erros nas regras. Ainda assim, foram encontradas cinco regras com erros no cálculo destas propriedades.

4.5 Sintaxes diferentes

No RuDriCo, existem três tipos de regras: regras de descontracção, regras de contracção e regras de desambiguação. No RuDriCo2, a sintaxe de ambas as regras começa por se diferenciar no símbolo que separa o antecedente do consequente, tal como indicado na tabela 4.2. Com esta distinção, é possível fazer uma verificação mais eficaz à construção das regras. Cada ficheiro de regras pode ter regras de vários

Tipo de regra	Símbolo
Desambiguação	:=
Descontracção	:<
Contracção	:>

Tabela 4.2: Correspondência entre tipos de regras e símbolos que separam o antecedente do consequente

tipos, mas o tipo de regra tem de ser indicado antes das regras com uma das directivas apresentadas na tabela 4.3. Para facilitar e não ser necessário escrever o tipo de regra em todas as regras, as regras

do mesmo tipo que sejam consecutivas necessitam da directiva apenas na primeira regra. Por exemplo, se um ficheiro tiver apenas regras de um tipo, basta ter a directiva correspondente a esse tipo antes da primeira regra.

Observa-se que cada tipo de regra tem uma relação entre o número de itens do antecedente e o número de itens do consequente. As regras de contracção têm obrigatoriamente mais itens no antecedente, contrariamente às regras de descontracção que têm mais itens no consequente. As regras de desambiguação têm exactamente o mesmo número de itens, pois estas não alteram a segmentação. Como no RuDriCo2 cada regra tem um tipo associado, é possível verificar se as regras de um respectivo tipo estão bem classificadas e alertar o utilizador em caso de erro, por exemplo, se uma regra for de descontracção e se o número de itens no consequente não for superior ao número de itens no antecedente, a regra ou está errada ou mal classificada. Esta verificação é feita para os três tipos de regras. É também verificado se o símbolo que separa o antecedente do consequente está correcto em relação ao tipo da regra.

Tipo de regra	Directiva
Desambiguação	disamb:
Descontracção	expand:
Contracçao	join:

Tabela 4.3: Correspondência entre tipos de regras e directivas

4.6 Introdução de novos operadores

A introdução de novos operadores à sintaxe permite que esta fique mais expressiva. Ainda assim, tal como foi referido na secção 2.4, o RuDriCo não tem o operador negação, sendo este uma das vantagens do XIP. O operador negação está descrito na secção 4.6.1. Na secção 2.4, mostra-se que a disjunção pode ser simulada com replicação de regras, mas, uma vez que esta solução não é escalável, introduz-se o operador disjunção, como se observa em 4.6.2. Na secção 4.6.3, são introduzidos dois novos operadores específicos para as regras de contracção e, na secção 4.6.4, é introduzido o operador item opcional. A introdução dos novos operadores não implicou nenhuma mudança substancial no modelo de dados do sistema nem nos algoritmos de processamento.

4.6.1 Operador Negação

Na sintaxe do RuDriCo, não existe o operador negação, mas, como referido na secção 2.4, a negação permite escrever regras mais simples e expressivas, o que levou à introdução do operador negação (∼) na sintaxe do RuDriCo2. O operador negação é utilizado para negar o valor de uma propriedade. Dentro de cada bloco, o operador pode ser utilizado em todos os pares propriedade-valor ou apenas em alguns. Observe-se o exemplo:

[prop 1=∼'valor 1' , prop 2='valor 2' ...][...]

Neste caso, a propriedade 1 do item não pode ocorrer com o valor 1. Com este operador resolve-se o problema apresentado na secção 2.4, porque é possível escrever a seguinte regra do XIP:

art<lemma:o,gen:m>,pron<lemma:o,gen:m>=| [verb:∼] | art | noun<gen:m> |

numa regra do RuDriCo2:

disamb:

```
|[CAT=~'verb']|,
[lemma='o',CAT='art',GEN='m'] [lemma='o',CAT='pro',GEN='m'],
|[CAT='nou',GEN='m']|
:=
[CAT='art']+.
```

4.6.2 Operador Disjunção

No RuDriCo, apesar de não existir o operador disjunção, existem algumas regras que necessitam desse conceito para serem escritas sem replicação de regras, como, por exemplo, a seguinte regra do XIP:

noun,verb = |det;prep| noun |verb|

Para escrever esta regra no RuDriCo é necessário escrever duas regras, como é descrito na secção 2.4. O problema é que quanto mais elementos fizerem parte da disjunção, mais regras são escritas no RuDriCo. Para resolver esta questão, introduziu-se o operador disjunção ";". Este é utilizado para fazer disjunção entre itens no antecedente. A sintaxe é a seguinte:

[prop 1='valor 1' , prop 2='valor 2' ...];[prop 1='valor 3' , prop 2='valor 4' ...];[...]....

Segue-se o exemplo da regra do XIP com o operador disjunção na sintaxe do RuDriCo2:

```
disamb:
|[CAT='det'];[CAT='prep']|
[CAT='noun'][CAT='verb']
|[CAT='verb']|
:=
[CAT='noun']+.
```

Neste exemplo, para um segmento emparelhar com o contexto à esquerda, basta emparelhar com algum dos itens da disjunção.

4.6.3 Operadores @@ e @@+

As regra de contracção realizam a tarefa de agrupar segmentos num único segmento. Ao observar todas as regras de contracção, nota-se que na maioria das regras existe um padrão: a forma superficial do segmento resultante é composta pela concatenação com ou sem espaços das formas superficiais dos segmentos que são contraídos. Por exemplo, na regra:

```
join:
[surface=S1,lemma='sexta',NUM=N1],
[surface=S2,lemma='feira']
:>
[surface=S1 @+ S2,lemma='sexta-feira',CAT='nou',SCT='com',GEN='f',NUM=N1].
```

a forma superficial do segmento resultante é composta pela concatenação com espaços das formas superficiais dos segmentos originais. No RuDriCo2, foram introduzidos dois operadores que só podem ser usados nas regras de contracção: o @@ e o @@+. O operador @@ significa concatenação de todas as formas superficiais que emparelham no antecedente e o operador @@+ significa concatenação com espaços de

todas as formas superficiais que emparelham no antecedente. Estes operadores são usados como valores das propriedades. Deste modo, a regra anterior pode ser escrita assim:

```
join:
[lemma='sexta',NUM=N1],
[lemma='feira']
:>
[surface=@@+,lemma='sexta-feira',CAT='nou',SCT='com',GEN='f',NUM=N1].
```

Note-se que não foi necessário usar variáveis para guardar as formas superficiais do antecedente, o que reduz ainda mais o tamanho dos ficheiros de regras e o número de variáveis usadas.

4.6.4 Operador item opcional [?]

No RuDriCo existem alguns casos em que há necessidade de representar um item que emparelhe com qualquer segmento, por exemplo, o item

```
S1[L1,CAT/C1]
```

emparelha com qualquer segmento, porque tem a forma superficial variável, o lema variável e a categoria variável. No RuDriCo2, introduziu-se um operador que tem estas características, o operador:

```
[?]
```

Este operador é usado como item e só pode ser utilizado no antecedente ou nos contextos. Todos os operadores que podem ser usados com itens podem ser usados com este operador.

4.7 Head e Tail como pares propriedade-valor

O RuDriCo tem dois segmentos especiais que são adicionados a cada frase que processa: o segmento Head é adicionado no início de cada frase e o segmento Tail é adicionado no fim de cada frase. No RuDriCo2, estes segmentos foram removidos e são substituídos por duas propriedades: FST (first) e LST (last). A propriedade FST é colocada com o valor "true" em todos os primeiros segmentos de cada frase e a propriedade LST é colocada com o valor "true" em todos os segmentos finais de cada frase. Estas propriedades são removidas antes do RuDriCo2 escrever as frases na saída.

No RuDriCo, para testar se um segmento é o primeiro de uma frase, usa-se o item Head antes do segmento que se quer testar:

```
Head, S1[L1 , 'CAT'/'pre' ]
```

No RuDriCo2, usa-se a propriedade FST:

```
[CAT='pre',FST='true' ]
```

Estes exemplos só emparelham com segmentos que apareçam no início de uma frase e que tenham a categoria preposição. Também é possível testar se um item é o último de uma frase usando a propriedade LST:

```
[CAT='pre',LST='true' ]
```

Este item emparelha com segmentos que sejam os últimos de uma frase e que tenham uma anotação de preposição.

Ao retirar o Head e o Tail das frases, foi necessário alterar o algoritmo de aplicação de regras, dado que o Tail desempenha um papel fundamental para o funcionamento do algoritmo no RuDriCo. Retomando o exemplo apresentado anteriormente, em que o sistema tem as seguintes regras:

```
disamb:
|[?]|
[lemma='poder']
:=
[lemma='poder',CAT='ver'] .

join:
[surface='de',lemma='de',CAT='prep'],
[surface='o',lemma='o',CAT='art']
:>
[surface='do',lemma='do',CAT='pre'] .
```

Observe-se a figura 4.8, onde está ilustrado o processamento da frase "A Coreia de o Sul" sem os segmentos adicionais no algoritmo do RuDriCo. Como o sistema só aplica uma Arule, se esta puder ser

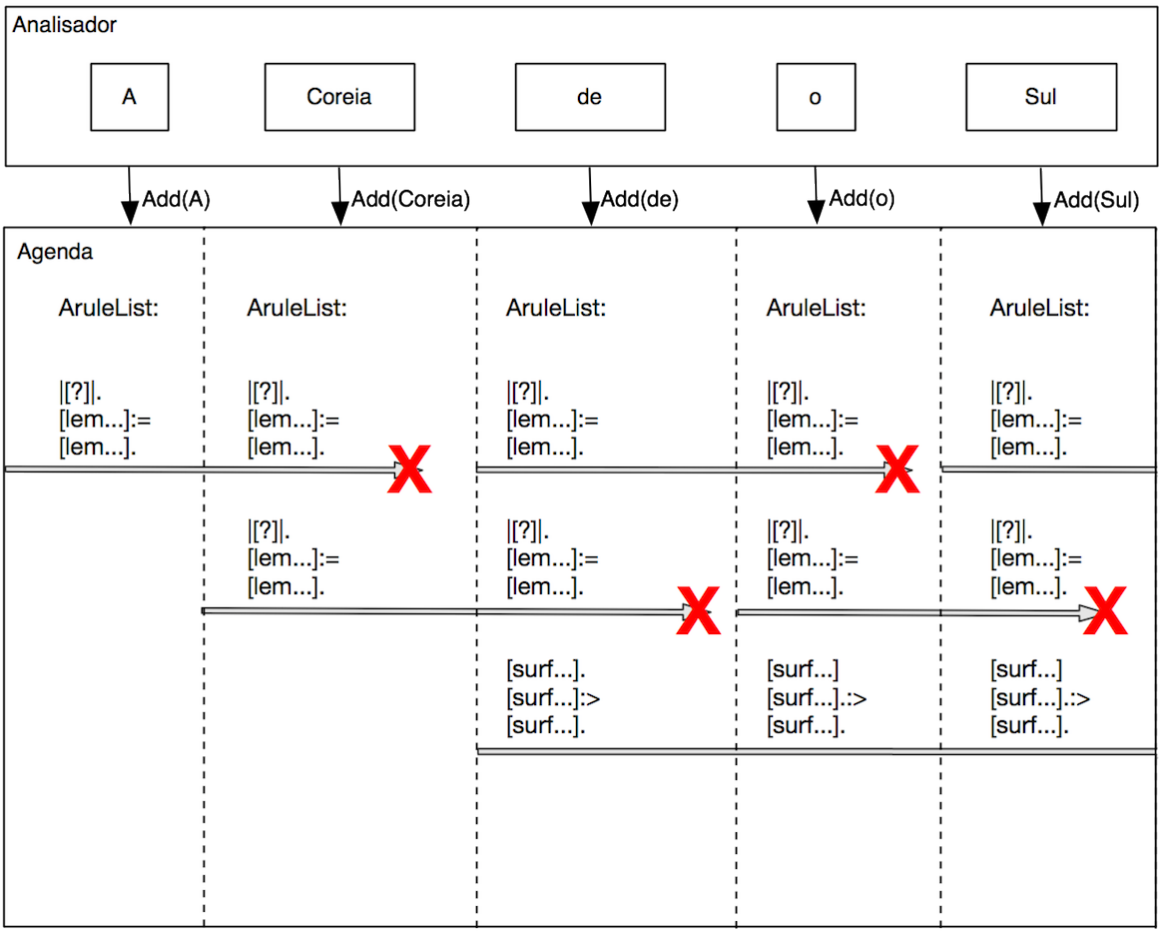

Figura 4.8: Primeiro passo do processamento da frase "A Coreia de o Sul", sem os segmentos Head e Tail

aplicada e for a primeira da lista, de modo a manter a prioridade das regras, a segunda regra nunca é aplicada se não existir o segmento Tail, porque a primeira é candidata em todos os segmentos. Para o RuDriCo2 manter o mesmo comportamento que o RuDriCo, o algoritmo de aplicação de regras foi alterado e, consequentemente, o algoritmo de processamento de frases também. A principal diferença entre o processamento do RuDriCo2 e do RuDriCoconsiste no facto de o RuDriCo2 processar frases em vez de processar segmentos individuais. O algoritmo de processamento de frases apresentado na figura 4.3 foi alterado e está descrito na figura 4.9. O novo algoritmo de aplicação de regras é apresentado na figura 4.10.

```
Method Analise(Sentence S, LayerList layersList)
 1:  FOR EACH layer L in layersList DO
 2:    Agenda.layer(L) /*tells agenda to use rules from layer L */
 3:    IF (L==0) THEN
 4:      PreProcessedSentence=S
 5:    ELSE
 6:      PreProcessedSentence= PosProcessedSentence
 7:    PosProcessedSentence={}
 8:    applied? = agenda(PreProcessedSentence)
 9:    WHILE (applied?)
10:      ConstructNewSentence(PosProcessedSentence,
                              PreProcessedSentence,ChangedSegments)
11:      PreProcessedSentence=PosProcessedSentence
12:      applied? = agenda(PreProcessedSentence)
13:    ENDWHILE
14:  ENDFOR
15:  Write(PosProcessedSentence)
```

Figura 4.9: Algoritmo de processamento de frases com a nova Agenda

```
Method add(Sentence S)
 1:  FOR EACH Segment s in S DO
 2:    ForwardArules(s)
 3:    SearchNewRules(s) /* this method adds the new Arules to AruleList */
 4:    Sort(AruleList)
 5:    IF (Done?(First(AruleList))) THEN
 6:      ApplyFirstRule(AruleList)
 7:      Erase(AruleList)
 8:      Return true
 9:  ENDFOR
10:  Return ApplyFirstReduced(AruleList)
```

Figura 4.10: Algoritmo de aplicação de regras da nova Agenda

O algoritmo de aplicação de regras passou a ter um ciclo para processar os segmentos de cada frase e, quando processa toda a frase sem aplicar nenhuma regra, aplica o método ApplyFirstReduced. O método ApplyFirstReduced verifica se existe alguma regra que possa ser aplicada na AruleList. Se existir mais do que uma regra que possa ser aplicada na AruleList, o método escolhe a primeira regra que encontra, dado que a lista está ordenada por ordem de aplicação das regras e, assim, é aplicada a regra que tem mais prioridade. O método remove todas as Arules da AruleList e, se tiver aplicado uma regra, retorna *true*, senão, retorna *false*. Com a Agenda alterada, o processamento do algoritmo demonstrado na figura 4.10 é realizado como mostra a figura 4.11 e a figura 4.12.

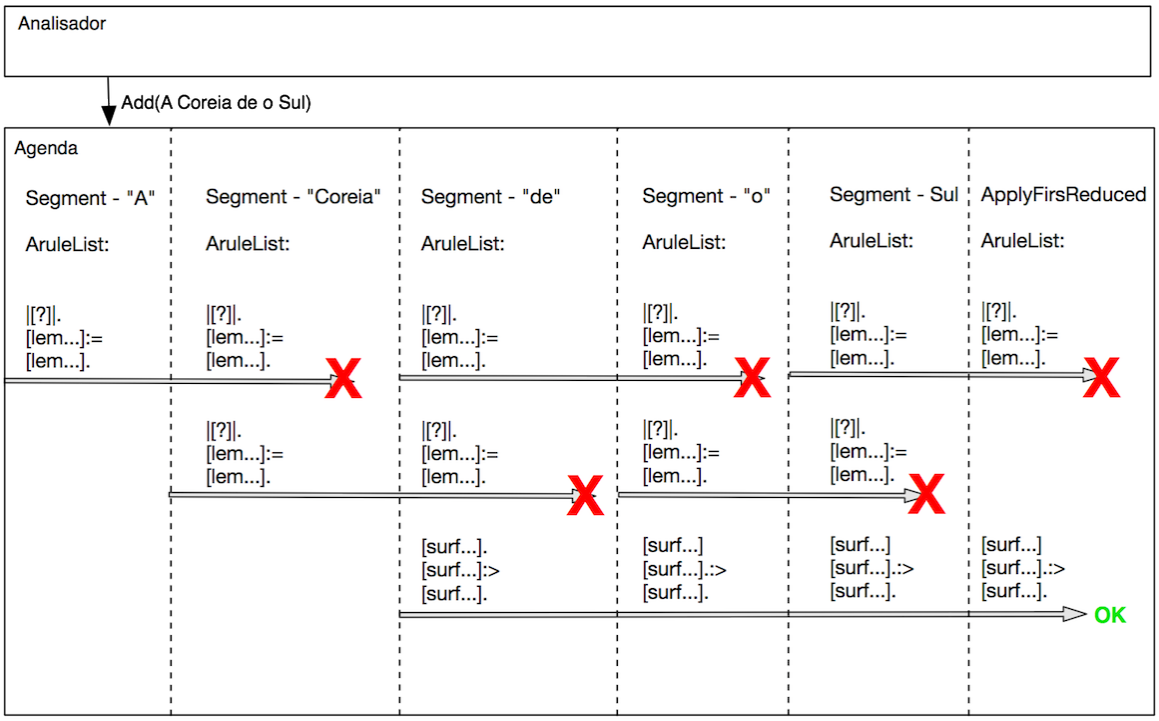

Figura 4.11: Primeiro passo do processamento da frase “A Coreia de o Sul”, na nova Agenda

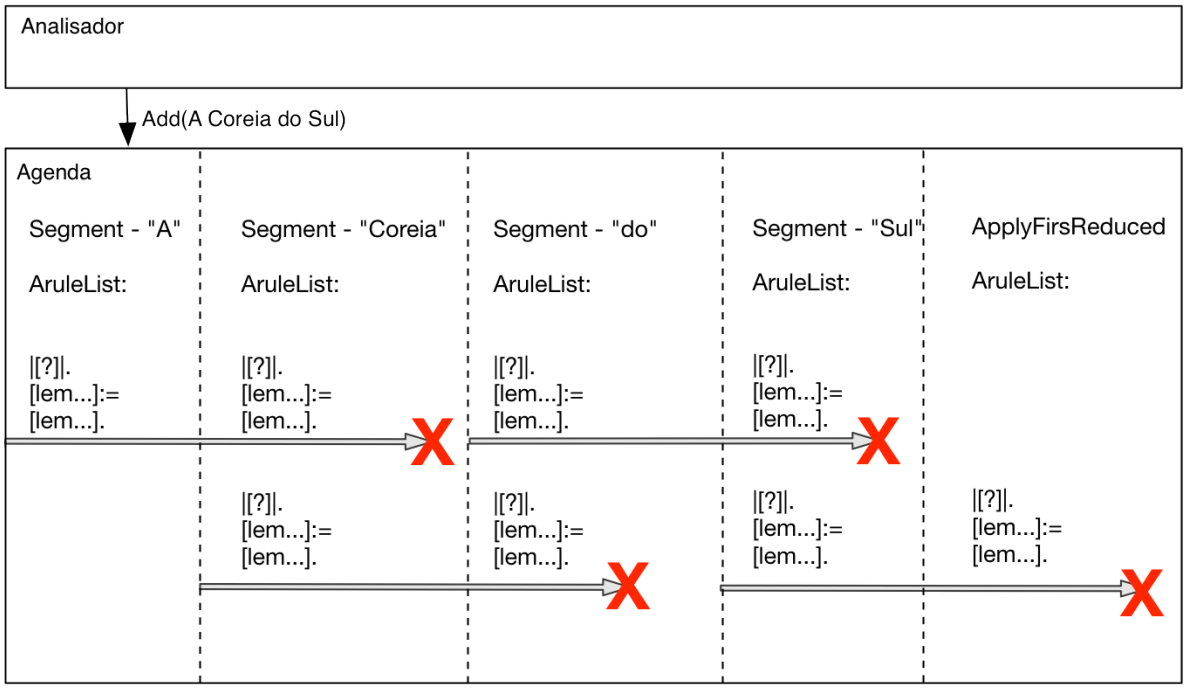

Figura 4.12: Segundo passo do processamento da frase: “A Coreia de o Sul”, na nova Agenda

4.8 Validação de regras

No RuDriCo, qualquer regra que seja bem constituída sintacticamente é aceite no sistema. Contudo, ao longo da utilização do sistema, foram detectados alguns casos em que o sistema devia alertar o utilizador da existência de um erro, por exemplo, quando há um engano no nome de uma variável.

Para combater esse problema, no RuDriCo2, foram introduzidas as seguintes verificações: verificação do nome das propriedades e dos valores, como se observa na secção 4.8.1, e verificação da consistência das variáveis, como se verifica em 4.8.2. Foi também introduzido um alerta para quando há regras de contracção que se sobrepõem, como descrito em 4.8.3.

4.8.1 Ficheiro de verificação de propriedades e valores

O conjunto de propriedades presente nos segmentos de entrada é conhecido antes da execução do sistema e os seus valores também, com excepção do lema e da forma superficial. O valor do lema e da forma superficial não são conhecidos, porque não têm um domínio limitado. As restantes propriedades têm um domínio de valores limitado, por exemplo, o género de uma palavra tem dois valores possíveis: masculino ou feminino. Os nomes das propriedades e dos valores que tenham um domínio limitado podem ser verificados na leitura das regras. Como esta verificação não é feita no RuDriCo original, pode haver erros nas regras que dificilmente são detectados. Considere-se a seguinte regra:

```
disamb:
|[CAT='prp']|
[lemma='poder', CAT='nou'] [lemma='poder', CAT='ver' ,MOD='inf']
|[CAT='ver', MOD='inf']|
:=
[lemma='poder', CAT='nou']-.
```

Imagine-se que é dado um erro numa letra de um dos valores das propriedades, por exemplo, na propriedade CAT do item no contexto à esquerda, em vez de estar o valor "prp", que representa preposição, está o valor "pro". O valor "pro"não faz parte do domínio da propriedade CAT e o que acontece é que esta regra fica no sistema mas nunca vai emparelhar com nenhum dos segmentos. Identificar estes erros é uma tarefa difícil e, para evitar este tipo de erros, é adicionado um passo à leitura das regras que é a verificação das propriedades e dos valores. Esta verificação é feita com base num ficheiro que contém as propriedades que o sistema vai aceitar e os seus valores possíveis. No caso de existir uma regra com um par propriedade-valor que não esteja presente no ficheiro, a regra é considerada errada e é reportado um erro. A sintaxe do ficheiro que contém as propriedades e os valores é a seguinte:

```
propriedade1 :{ valor1 valor2 valor3 }
propriedade2 :{ valor 4 valor 5 valor 6 }
...
propriedadeX :{ valor x ..... }
```

Quando esta alteração foi adicionada ao sistema, verificou-se que 12 das 3096 regras tinham erros.

4.8.2 Verificação de variáveis

No RuDriCo, quando há variáveis no consequente e estas não têm o respectivo par no antecedente, o valor destas é uma *string* vazia. No RuDriCo2, quando não existe o par de uma variável no antecedente,

a regra é considerada inválida. Quando foi adicionada esta verificação, detectaram-se 13 regras com erros nas 3096 regras utilizadas para teste.

No RuDriCo2, quando há variáveis no antecedente usadas uma única vez e estas não têm o respectivo par no consequente, é gerado um *warning* que alerta o utilizador para a existência de variáveis desnecessárias.

4.8.3 Regras de contracção

No RuDriCo, e consequentemente no RuDriCo2, se duas regras forem da mesma camada, a regra que tem mais prioridade é a que aparece primeiro no ficheiro de regras. Nas regras de contracção, existem regras que podem nunca ser aplicadas, dado que têm menos prioridade do que outras. Considerem-se as seguintes regras de contracção:

```
join:
[lemma='de'],[lemma='uma'],[lemma='vez']
:>
[surface=@@+,surface='de uma vez'].

[lemma='de'],[lemma='uma'],[lemma='vez'],[lemma='por'],[lemma='todas']
:>
[surface=@@+,surface='de uma vez por todas'].
```

Pela ordem em que as regras estão declaradas, a segunda regra nunca vai poder ser aplicada independentemente das frases de entrada, porque a primeira regra é aplicada primeiro e muda a segmentação do texto. Nestes casos, o sistema gera um aviso para alertar o utilizador. Note-se que, se as regras forem declaradas pela ordem inversa, ambas as regras podem ser aplicadas.

4.9 Optimizações

Na análise do comportamento do algoritmo de aplicação de regras com a nova Agenda, apresentado em 4.7, constata-se que o algoritmo pode ser melhorado, uma vez que realiza alguns passos redundantes. Em 4.9.1, apresentam-se duas optimizações que reduzem o número de iterações do algoritmo. Já em 4.9.2, é apresentado um problema presente nos índices de regras contidos no Analisador e, em seguida, é proposta uma solução.

4.9.1 Optimizações ao algoritmo de aplicação de regras

As optimizações realizadas ao algoritmo são as seguintes:

- em alguns casos especiais, não esperar pelo final da frase para poder aplicar uma regra, mesmo que existam outras mais prioritárias;
- quando é aplicada uma regra, não voltar a processar a frase desde início, mas sim desde o ponto em que é realmente necessário começar a processar.

A primeira optimização é conseguida através da ordenação das Arules. A ordem das regras no ficheiro é o primeiro factor na prioridade das Arules, enquanto no RuDriCo2 o primeiro factor é o seguinte: se uma regra puder ser aplicada e se o último segmento da mesma aparecer na frase antes do primeiro segmento de todas as outras regras, esta regra é a primeira da AruleList. Quando este factor não se

verificar, a ordem das regras é igual à ordenação original. Na figura 4.13, observa-se como o primeiro passo do exemplo de execução do algoritmo de aplicação de regras, apresentado na secção 4.7, é realizado com a nova ordenação de regras. O segundo passo é igual ao exemplo da secção 4.7.

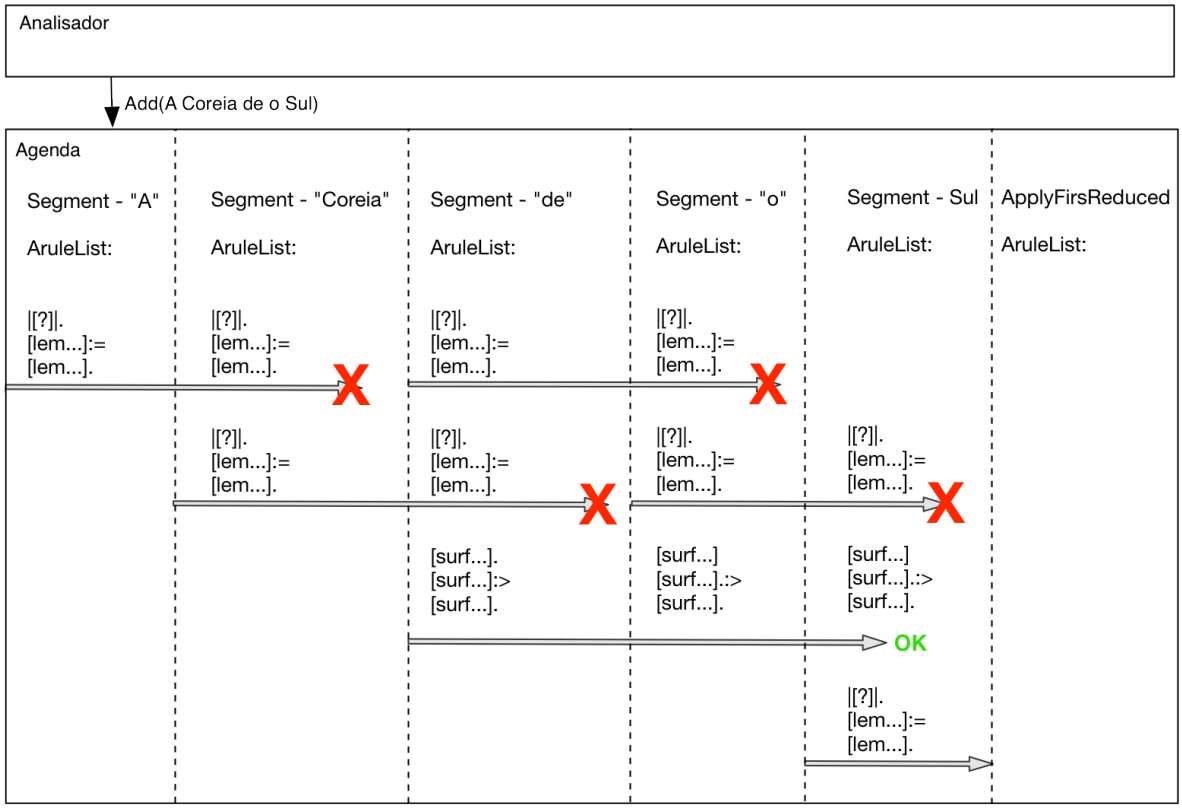

Figura 4.13: Primeiro passo do processamento da frase "A Coreia de o Sul", com nova ordenação de regras

Observando a figura 4.13, verifica-se que a regra de contracção não é aplicada assim que satisfeita, dado que, nesta fase do processamento, ainda existe uma regra na AruleList que tem mais prioridade. A regra de contracção é aplicada no segmento "Sul", porque, devido à nova ordenação, a Arule correspondente à regra de contracção passa a ser a primeira da lista. Importa referir que, neste exemplo, não há uma grande melhoria, mas na hipótese de a frase a processar ser "A Coreia de o Sul é na Ásia ", a regra seria aplicada sem processar os segmentos: "é","na" e "Ásia".

Na segunda optimização, os segmentos de cada frase estão associados a um índice numérico, sendo calculado para cada segmento um valor de recomeço, o que implica que, se for aplicada uma regra que comece neste segmento, o sistema tem de voltar a processar a frase começando no segmento de índice x. O valor de recomeço de um segmento é calculado quando o respectivo segmento é processado e o valor corresponde ao índice mais pequeno dos primeiros segmentos das Arules presentes na AruleList. Para exemplificar o cálculo do valor de recomeço, apresenta-se novamente o exemplo anterior com esta optimização. Na figura 4.14, encontra-se o primeiro passo e, na figura 4.15, é apresentado o segundo.

Ao se observar a figura 4.14, verifica-se que a regra que foi aplicada começa no segmento de índice 2 e o valor de recomeço deste segmento é 1, ou seja, a frase vai ser processada começando no segmento de índice 1, como ilustra a figura 4.15. Note-se que o valor de recomeço do segmento de índice 2 na figura 4.14 é 1, porque neste segmento a AruleList tem duas regras, sendo que aquela que começa primeiro é a do segmento de índice 1. Neste exemplo de processamento, o benefício está em não processar um segmento, mas, considerando que a frase era, por exemplo, "O Alberto e o João vão para a Coreia de o Sul", verifica-se que, neste caso, seria benéfico não processar oito segmentos.

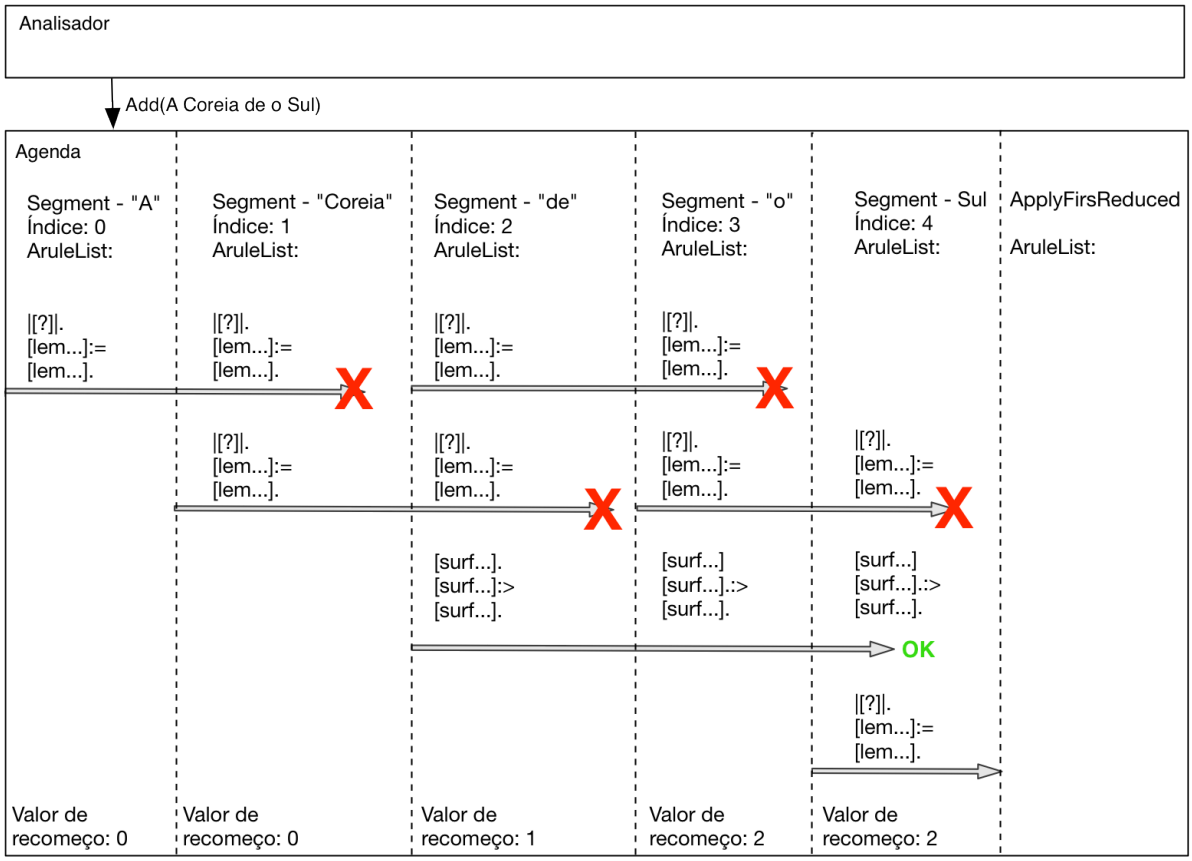

Figura 4.14: Primeiro passo do processamento da frase “A Coreia de o Sul”, com as duas optimizações

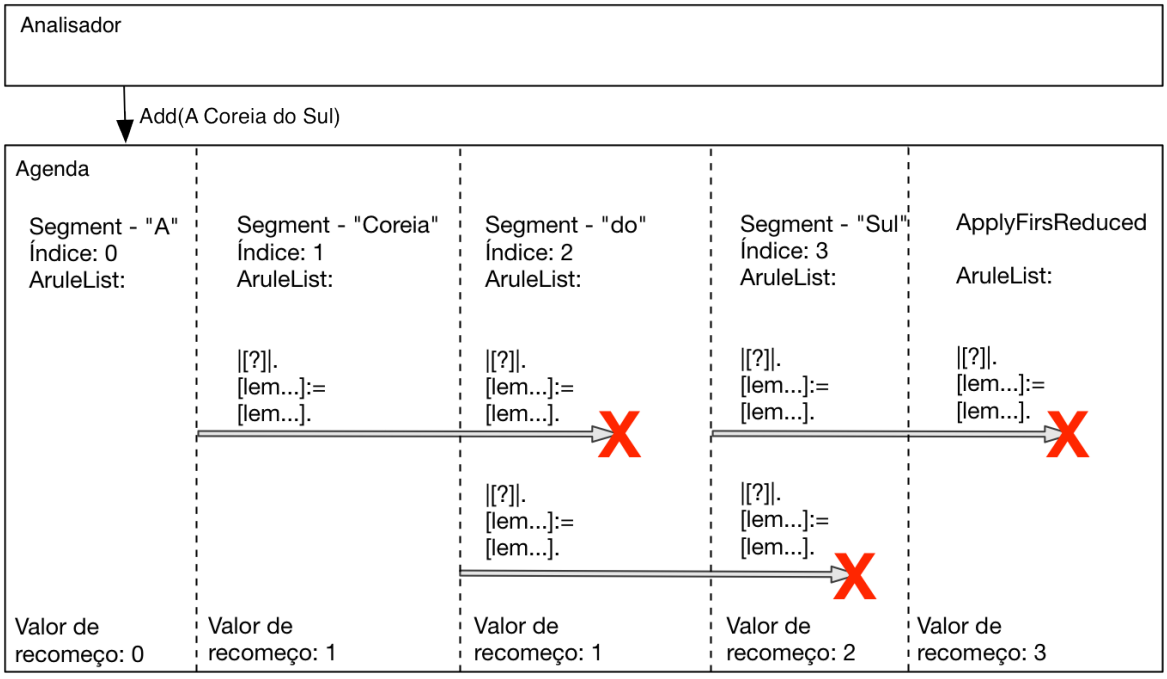

Figura 4.15: Segundo passo do processamento da frase “A Coreia de o Sul”, com as duas optimizações

4.9.2 Optimização dos índices de regras

Em 3.2.2 foi apresentada a indexação das regras, tendo sido também descritas as propriedades dos vários índices de regras presentes no sistema. Posteriormente, foi realizado um estudo sobre o número de regras associado às formas superficiais no índice de formas superficiais constantes e lemas variáveis. Analisando as regras do sistema, observa-se que existem 1796 formas superficiais no respectivo índice. Na tabela 4.4, apresentam-se as 10 formas superficiais com mais regras associadas.

Palavra	Número de regras
a	433
em	370
por	242
de	215
the	101
com	94
sem	41
até	34
são	32
la	31

Tabela 4.4: Número de regras associadas a cada forma superficial

É possível observar que o número de regras associadas às formas superficiais não é uniforme, havendo formas superficiais que têm um elevado número de regras associadas, quando comparadas com as restantes. Um exemplo disso mesmo é a forma superficial "a" que tem 433 regras associadas. O segmento de forma superficial "a" é frequente no texto e, sempre que este é processado, as 433 regras associadas são candidatas a emparelhar. Para diminuir o número de regras candidatas, as regras com mais de um item que tenham a forma superficial constante e o lema variável são indexadas num novo índice, um índice que tem em consideração a forma superficial do primeiro item e do segundo item das regras. Como se pode verificar abaixo, as seguintes regras são associadas no índice das formas superficiais à forma superficial "a":

```
join:
[surface='a'],
[surface='baixa'],
[surface='altitude']
:>
[surface=@@+,lemma='a baixa altitude',CAT='adv',DEG='pst'].

[surface='a'],
[surface='bom'],
[surface='passo']
:>
[surface=@@+,lemma='a bom passo',CAT='adv',DEG='pst'].
```

Com a introdução do novo índice, estas regras passam a estar associadas ao novo índice. A primeira regra é associada à concatenação das formas superficiais "a" e "baixa" e a segunda é associada à concatenação das formas superficiais "a" e "bom".

Com este novo índice, é possível reduzir o número de regras que têm a mesma forma superficial indexada, o que reduz o número de regras candidatas a gerir pelo algoritmo de aplicação de regras.

Capítulo 5

Avaliação

Para medir o desempenho de um sistema, é necessário definir qual o conjunto de critérios a medir e, posteriormente, fazer uma análise dos resultados. Neste caso, em que o sistema a avaliar é uma evolução do sistema RuDriCo, avalia-se o desempenho do sistema final com base no desempenho do sistema original, como se pode observar na secção 5.1. Uma vez que este sistema é baseado em regras e a sintaxe destas difere de um sistema para o outro, é realizada também uma breve avaliação da nova sintaxe, como se verifica em 5.2.

5.1 Avaliação do desempenho

Na secção 5.1.1, apresenta-se a metodologia de avaliação do desempenho do sistema, que inclui a definição do ambiente de avaliação e a avaliação original do RuDriCo.

Como as alterações foram realizadas sequencialmente, são efectuadas avaliações a estados intermédios do sistema. Realiza-se ainda uma avaliação depois da alteração do processamento da entrada e saída do sistema em 5.1.2. Esta avaliação realiza-se dado que esta alteração tem influência no tamanho dos ficheiros que o sistema processa. Consequentemente, a introdução de camadas no sistema tem impacto no desempenho do mesmo, pelo que se realiza uma nova avaliação, apresentada na secção 5.1.3. Depois de introduzidos os contextos e a nova representação dos itens, realiza-se uma avaliação em 5.1.4. Na secção 5.1.5, apresenta-se a avaliação do sistema após as propriedades automáticas serem introduzidas. Este estado do sistema é avaliado, pois esta alteração não foi realizada com o intuito de aumentar o desempenho do sistema, mas apenas para reduzir o trabalho do utilizador quando este escreve regras e para reduzir o número de variáveis nas mesmas. Finalmente, em 5.1.6, é feita a avaliação global do sistema RuDriCo após todas as alterações e optimizações introduzidas, ou seja, a avaliação do RuDriCo2.

5.1.1 Metodologia da avaliação

Para medir o desempenho do RuDriCo2, usam-se dois critérios: o tempo de CPU, medido em segundos, e a memória ocupada, medida em Megabytes. Considere-se ainda que não foi possível medir os casos em que o sistema usa menos de 1Mb, pelo que, nas tabelas de resultados, estes casos são apresentados como "$<$ 1 Mb".

Estes dois critérios são analisados em duas fases de execução: quando o sistema pré-processa as regras e quando processa o texto de entrada.

Para gerar os resultados, definiu-se um ambiente de processamento idêntico ao usado no início deste trabalho, o que corresponde ao conjunto de 3096 regras usadas no L^2F. Os ficheiros usados para a avaliação

são excertos do CETEMPúblico[1], sendo que cada excerto tem um tamanho diferente. Os ficheiros estão descritos na tabela 5.1 e são caracterizados pelo tamanho em Kilobytes e pelo número de frases.

Ficheiro	Nº frases	Tamanho (Kb)
part08-1.xml	1	17
part08-10.xml	10	241
part08-100.xml	100	1 524
part08-500.xml	500	7 203
part08-1000.xml	1000	14 308
part08-5000.xml	5000	70 634
part08-10000.xml	10000	140 905
part08-50000.xml	50000	716 195
part08-100000.xml	100000	1 436 545

Tabela 5.1: Ficheiros para testar o sistema RuDriCo

De seguida, apresenta-se a avaliação do sistema original para posterior comparação, na tabela 5.2.

Ficheiro	Tempo (s)	Tamanho (Mb)
part08-1.xml	0.16	< 1.00
part08-10.xml	1.96	10.60
part08-100.xml	8.33	34.90
part08-500.xml	38.00	143.00
part08-1000.xml	78.00	275.00
part08-5000.xml	392.00	1234.00
part08-10000.xml	782.75	2560
part08-50000.xml	não processa	-
part08-100000.xml	não processa	-

Tabela 5.2: Avaliação do desempenho do sistema RuDriCo

Observe-se que, devido ao método de leitura e escrita do sistema, o sistema torna-se incapaz de processar os dois maiores ficheiros.

5.1.2 Alteração da metodologia usada para leitura e escrita de ficheiros

No RuDriCo, a entrada e a saída são processadas com um *xml dom parser*. O RuDriCo2 processa o ficheiro de entrada com um *xml sax parser* e processa a saída manualmente. Para avaliar esta alteração ao sistema, testa-se o processamento dos nove ficheiros num ambiente de avaliação especial, no qual o sistema não aplica qualquer regra, ou seja, testa-se apenas a leitura e a escrita dos ficheiros. Depois deste teste, é realizada a avaliação do sistema como foi descrita na secção anterior.

A tabela 5.3 contém os resultados da memória utilizada na leitura e escrita dos ficheiros, sendo possível verificar que, com esta alteração, já é possível processar todos os ficheiros de avaliação. Note-se que a memória utilizada pelo RuDriCo depois da nova alteração foi reduzida. A redução não é constante de ficheiro para ficheiro, mas, analisando a memória ocupada do ficheiro part08-5000.xml, esta é aproximadamente 116 vezes menos do que a memória usada no sistema original.

Como um dos objectivos deste trabalho é aumentar o desempenho do RuDriCo, na tabela 5.4 apresentam-se os resultados do tempo de leitura e escrita de ficheiros. Note-se que esta alteração no método de leitura e escrita aumentou também o desempenho do RuDriCo. Observando os tempos de processamento, verifica-se que quanto maior é o ficheiro, maior é o aumento desse desempenho. Por comparação ao maior ficheiro que o RuDriCo original processa, o ficheiro part08-10000.xml, o sistema

[1]Corpus de Extractos de Textos Electrónicos MCT/Público

	Original (Mb)	Alteração do processamento de entrada e saída (Mb)	%
part08-1.xml	<1.00	<1.00	-
part08-10.xml	<1.00	<1.00	-
part08-100.xml	14.00	<1.00	-
part08-500.xml	132.00	<1.00	-
part08-1000.xml	292.00	1.80	0.62
part08-5000.xml	1433.00	12.20	0.85
part08-10000.xml	2252.8	30.00	1.33
part08-50000.xml	não processa	159.50	-
part08-100000.xml	não processa	332.60	-

Tabela 5.3: Memória utilizada para a leitura e escrita

	Original (s)	Alteração do processamento de entrada e saída (s)	%
part08-1.xml	0.01	0.01	100.00
part08-10.xml	0.09	0.04	44.44
part08-100.xml	0.54	0.20	37.04
part08-500.xml	2.44	0.93	38.11
part08-1000.xml	4.81	1.81	37.63
part08-5000.xml	23.56	8.96	38.03
part08-10000.xml	51.83	17.90	34.54
part08-50000.xml	não processa	97.21	-
part08-100000.xml	não processa	201.00	-

Tabela 5.4: Tempo de leitura e escrita em segundos

tornou-se aproximadamente 2.9 vezes mais rápido. Dado que o desempenho da escrita e leitura de ficheiros aumentou, o tempo de processamento dos ficheiros de avaliação diminuiu, como se pode observar na tabela 5.5. No primeiro ficheiro, o tempo de processamento não diminuiu, porque o ficheiro tem uma única

	Original(s)	Alteração do processamento de entrada e saída (s)	%
part08-1.xml	0.16	0.16	100
part08-10.xml	1.96	1.82	92.72
part08-100.xml	8.33	7.74	92.20
part08-500.xml	38.00	35.99	94.70
part08-1000.xml	78.00	74.00	94.87
part08-5000.xml	392.00	368.36	94.07
part08-10000.xml	782.75	728.75	93.10
part08-50000.xml	não processa	3667.73	-
part08-100000.xml	não processa	7188.05	-

Tabela 5.5: Tempo de processamento dos ficheiros de avaliação

frase e o processamento de uma frase não é suficiente para se notar os ganhos desta alteração. Os restantes ficheiros reduziram o tempo uniformemente, em média, para 93.71% do tempo do RuDriCo original.

Consequentemente, a memória utilizada no processamento de ficheiros também é reduzida, como mostra a tabela 5.6. Assim, verifica-se que a redução da memória ocupada não é constante em todos os ficheiros, sendo esta proporcional ao tamanho do ficheiro, por exemplo, no processamento do ficheiro de 500 frases, a memória ocupada passa para 12.38% da memória que o sistema original ocupa, mas, no processamento do ficheiro de 1000 frases, a memória ocupada foi reduzida para cerca de 7.13%.

Esta alteração não teve impacto no pré-processamento de regras, portanto, não é realizada uma comparação com o sistema original.

	Original (Mb)	Alteração do processamento de entrada e saída (Mb)	%
part08-1.xml	< 1.00	< 1.00	-
part08-10.xml	10.60	9.80	92.45
part08-100.xml	34.90	12.40	35.53
part08-500.xml	143.00	17.70	12,38
part08-1000.xml	275.00	19.60	7.13
part08-5000.xml	1234.00	56.40	4.57
part08-10000.xml	2560	102.20	3.99
part08-50000.xml	não processa	507.30	-
part08-100000.xml	não processa	1023.60	-

Tabela 5.6: Memória utilizada no processamento dos ficheiros de avaliação

5.1.3 Introdução de camadas

O conceito de camada permite agrupar as regras do sistema em várias camadas. A introdução deste conceito é descrito na secção 4.2.

As regras utilizadas para avaliação estão distribuídas por 35 ficheiros e estão organizadas pela funcionalidade das regras. Com a introdução das camadas, as regras são organizadas em camadas. As regras de desambiguação não podem ser divididas em várias camadas, dado que geram resultados diferentes. As restantes regras ficam com a organização original, em que cada camada de regras tem as regras correspondentes a um ficheiro.

Ao observar o algoritmo de processamento de frases quando foram adicionadas as camadas, na secção 4.2, conclui-se que o número de regras por camada tem influência no desempenho do sistema, tendo sido realizado um estudo para descobrir o número óptimo de regras por camada. Neste estudo, utilizaram-se todas as regras do sistema, excepto as regras de desambiguação. Para as restantes 2330 regras, em vez de cada ficheiro corresponder a uma camada, as regras são divididas em camadas de igual tamanho de modo a encontrar o número óptimo de regras por camadas. Os testes foram realizados processando o ficheiro part08-1000.xml e os resultados estão na tabela 5.7. Note-se que, no caso de existir apenas uma

Regras/Camada	Tempo (s)
1	146.00
2	75.00
4	40.50
8	23.20
16	15.80
17	15.10
32	9.10
73	7.70
146	6.90
156	6.70
167	6.10
180	6.90
292	7.80
583	8.70
1165	14.90
2330	15.20

Tabela 5.7: Estudo do número de regras óptimo por camada

regra por camada, o sistema demora mais tempo, ao contrário do que acontece quando todas as regras estão na mesma camada. Este facto resulta da estrutura original do algoritmo de processamento de frases resultado ter dois ciclos, um ciclo para as frases e um ciclo para todos os segmentos de uma frase. No

caso da existência de camadas, o algoritmo de processamento de frases passa a ter três ciclos, o ciclo para as frases, o ciclo para as camadas e o ciclo para os segmentos da frase. Esta complexidade adicional no algoritmo é compensada em alguns casos, porque o algoritmo de Agenda aumenta o desempenho quando processa um número mais pequeno de regras. O tempo de processamento utilizado como referência para a análise do desempenho é o tempo do sistema com todas as regras na mesma camada. Analisando a tabela de resultados, verifica-se que, a partir de 17 regras por camada, o sistema aumenta o desempenho e o melhor resultado é atingido quando as camadas têm 167 regras. No caso do sistema ter 167 regras por camada, o desempenho do sistema aumenta cerca de 2.5 vezes.

Após realizado o estudo do número de camadas, o sistema é analisado nas duas fases de processamento: pré-processamento das regras e processamento de ficheiros. Na tabela 5.8, apresentam-se os resultados do tempo de geração do ficheiro optimizado e aí pode-se verificar que as camadas aumentaram este tempo em cerca de 8,2% do tempo original. Dado que as regras só são pré-processadas quando se acrescentam regras e o tempo do pré-processamento de regras é pequeno, a perda de desempenho nesta fase do processamento é insignificante.

	Sem camadas	Com camadas
Tempo(s)	0.49	0.53

Tabela 5.8: Tempo de geração do ficheiro optimizado

Na tabela 5.9, realiza-se uma nova avaliação do sistema com camadas, onde cada camada corresponde às regras de um ficheiro.

	Alteração do processamento de entrada e saída (s)	Camadas (s)	%
part08-1.xml	0.16	0.12	75.00
part08-10.xml	1.82	0.74	40.77
part08-100.xml	7.74	3.36	43.41
part08-500.xml	35.99	15.37	42.70
part08-1000.xml	74.00	30.79	41.61
part08-5000.xml	368.36	152.19	41.32
part08-10000.xml	728.75	301.50	41.37
part08-50000.xml	3667.73	1546.70	42.17
part08-100000.xml	7188.05	2951.20	41.06

Tabela 5.9: Tempo de processamento dos ficheiros de avaliação

Analisando a tabela, verifica-se que todos os tempos baixaram uniformemente com a introdução de camadas, excepto o tempo de processamento do ficheiro com uma só frase. No caso do primeiro ficheiro, o aumento de desempenho é menor do que nos restantes, porque o sistema processa apenas uma frase. Os restantes tempos baixaram em média para cerca de 41.80% do tempo original.

Observe-se na tabela 5.10 que a memória ocupada diminuiu com o conceito de camadas, por exemplo, para o ficheiro de 1000 frases a memória foi reduzida para 70.41% da memória original. Verifica-se também que, quanto maior é o ficheiro a processar, menos se nota a redução de memória.

5.1.4 Introdução de contextos e alteração da representação dos itens

No RuDriCo original não existe o conceito de contexto, mas este é simulado usando variáveis, o que implica que as regras sejam mais trabalhosas e extensas. No RuDriCo original, além de não existirem contextos, é obrigatório designar o lema e a forma superficial em todos os itens. Na secção 4.3, são introduzidos os contextos e é introduzida também uma nova representação para os itens, em que a forma

	Alteração do processamento de entrada e saída (Mb)	Camadas (Mb)	%
part08-1.xml	< 1.00	< 1.00	-
part08-10.xml	9.80	< 1.00	-
part08-100.xml	12.40	6.10	49.19
part08-500.xml	17.70	9.30	52.54
part08-1000.xml	19.60	13.80	70.41
part08-5000.xml	56.40	51.10	90.60
part08-10000.xml	102.20	97.00	94.91
part08-50000.xml	501.30	490.00	97.75
part08-100000.xml	1023.60	998.10	97.51

Tabela 5.10: Memória utilizada no processamento dos ficheiros de avaliação

superficial e o lema são pares propriedade-valor. Com estas duas alterações, a sintaxe fica mais compacta e o uso de variáveis é reduzido.

Para medir o aumento de desempenho do sistema na fase de processamento de ficheiros, são analisados os tempos de processamento dos ficheiros de avaliação. As regras foram convertidas de modo a utilizarem contextos e usufruírem da omissão do lema e da forma superficial. Os novos resultados, juntamente com os resultados do estado do sistema na secção anterior, apresentam-se na tabela 5.11. O tempo de processamento do ficheiro com uma frase é 91.67% do tempo medido na secção anterior. O ganho de desempenho deste ficheiro é menor que os restantes, porque este ficheiro não tem um número de frases suficientemente elevado, para que as alterações se notem no desempenho. Observando os restantes ficheiros, nota-se que esta alteração reduziu o tempo do sistema para metade do tempo.

	Camadas (s)	Alteração da sintaxe (s)	%
part08-1.xml	0.12	0.11	91.67
part08-10.xml	0.74	0.40	53.91
part08-100.xml	3.36	1.69	50.30
part08-500.xml	15.37	7.83	50.95
part08-1000.xml	30.79	15.29	49.66
part08-5000.xml	152.19	76.80	50.46
part08-10000.xml	301.50	154.12	51.12
part08-50000.xml	1546.70	791.00	51.14
part08-100000.xml	2951.2	1611.66	54.61

Tabela 5.11: Tempo de processamento dos ficheiros de avaliação

Como esta alteração reduz o tamanho das regras e diminui o número de variáveis usadas nestas, é esperado que a memória ocupada ao processar os ficheiros de avaliação diminua. Observe-se a tabela 5.12, onde se nota que a memória ocupada reduziu em média para 92.61%.

	Camadas (Mb)	Alteração da sintaxe (Mb)	%
part08-1.xml	< 1.00	< 1.00	-
part08-10.xml	< 1.00	< 1.00	-
part08-100.xml	6.10	5.70	93.44
part08-500.xml	9.30	9.00	96.77
part08-1000.xml	13.80	13.10	94.93
part08-5000.xml	51.10	47.30	92.56
part08-10000.xml	97.00	89.20	91.96
part08-50000.xml	490.00	431.10	87.98
part08-100000.xml	998.10	904.60	90.63

Tabela 5.12: Memória utilizada no processamento dos ficheiros de avaliação

Importa referir ainda que é necessário medir o tempo na fase de pré-processamento de regras, pois a estrutura das regras foi alterada ao serem adicionados os contextos. Na tabela 5.13 apresentam-se os resultados e verifica-se que esta alteração aumentou o tempo de pré-processamento para aproximadamente 102% do tempo usado antes desta alteração.

	Com camadas	Alteração da sintaxe
Tempo(s)	0.53	0.54

Tabela 5.13: Tempo de geração do ficheiro optimizado

5.1.5 Propriedades automáticas e capitalização

No RuDriCo, a capitalização das formas superficiais e dos lemas é realizada com base na propriedade UPC. Esta propriedade está presente em todos os segmentos que o sistema recebe e, consequentemente, está presente em todas as regras que geram novos segmentos. Na secção 4.4.1, é descrita a solução usada para a propriedade UPC não ser usada em todos os segmentos nem em todas as regras que geram segmentos. As propriedades HIGH e LOW também estão em todos os segmentos do RuDriCo original e estas são calculadas com base em variáveis. As três propriedades referidas passaram a ser calculadas internamente, o que reduziu o número de variáveis usadas nas regras. Adicionalmente, apenas as regras de descontracção usam duas propriedades, TOKENS e TOKENE, que passaram também a ser calculadas internamente.

Estas propriedades foram retiradas das regras e, para as regras que geram segmentos, há pelo menos três propriedades que deixaram de ser usadas. Ainda assim, estas propriedades são calculadas internamente, portanto, não é claro que o sistema altere o desempenho. A avaliação do sistema apresenta-se na tabela 5.14, onde se pode verificar que esta alteração não teve um impacto significativo no sistema, o que leva à conclusão que esta alteração reduz o trabalho do utilizador ao escrever as regras sem prejudicar o desempenho do sistema.

	Alteração da sintaxe (s)	Propriedades automáticas (s)	%
part08-1.xml	0.11	0.11	100
part08-10.xml	0.40	0.38	94.50
part08-100.xml	1.69	1.68	99.59
part08-500.xml	7.83	7.58	96.83
part08-1000.xml	15.29	15.03	98.30
part08-5000.xml	76.8	75.40	98.18
part08-10000.xml	154.12	150.64	97.74
part08-50000.xml	791.00	769.03	97.22
part08-100000.xml	1611.66	1530.56	94.97

Tabela 5.14: Tempo de processamento dos ficheiros de avaliação

Com esta alteração, as regras reduzem o tamanho e é esperado que a memória ocupada no processamento também seja reduzida. Na tabela 5.15, apresentam-se os novos valores de memória ocupada, sendo possível concluir que esta alteração se reflectiu na memória ocupada, embora o ganho não seja significativo. Por exemplo, para o ficheiro de 1000 frases, a memória ocupada passa a ser 95.42% da memória ocupada pelo sistema avaliado na alteração anterior. Nota-se que no ficheiro de 100 frases não há redução da memória ocupada, porque o ficheiro não tem tamanho suficiente para esta alteração se reflectir.

	Alteração da sintaxe (Mb)	Propriedades automáticas (Mb)	%
part08-1.xml	< 1.00	< 1.00	-
part08-10.xml	< 1.00	< 1.00	-
part08-100.xml	5.70	5.70	100
part08-500.xml	9.00	8.60	95.56
part08-1000.xml	13.10	12.50	95.42
part08-5000.xml	47.30	44.70	94.50
part08-10000.xml	89.20	84.60	94.84
part08-50000.xml	431.10	410.40	95.20
part08-100000.xml	904.60	850.40	94.01

Tabela 5.15: Memória utilizada no processamento dos ficheiros de avaliação

Como as regras ficaram mais compactas, espera-se que esta alteração tenha impacto no pré-processamento das regras. Na tabela 5.16, apresentam-se os valores da nova medição do tempo de pré--processamento. Verifica-se que esta alteração reduziu o tempo para aproximadamente 91% do tempo medido na secção anterior.

	Alteração da sintaxe	Propriedades automáticas
Tempo(s)	0.55	0.50

Tabela 5.16: Tempo de geração do ficheiro optimizado

5.1.6 RuDriCo2

Após a alteração das propriedades automáticas, foram realizadas outras alterações ao sistema, como a introdução de operadores na sintaxe e as optimizações ao algoritmo de aplicação de regras. Nesta secção, realiza-se a avaliação do RuDriCo após todas as alterações, ou seja, avalia-se o RuDriCo2. Na tabela 5.17, é apresentada a avaliação do tempo de processamento dos ficheiros de avaliação no RuDriCo2 e compara-se com o estado do sistema avaliado na secção anterior.

	Propriedades automáticas (s)	RuDriCo2 (s)	%
part08-1.xml	0.11	0.09	81.82
part08-10.xml	0.38	0.23	60.05
part08-100.xml	1.68	0.89	53.12
part08-500.xml	7.58	3.91	51.50
part08-1000.xml	15.03	7.78	51.76
part08-5000.xml	75.40	38.90	51.59
part08-10000.xml	150.64	78.70	52.24
part08-50000.xml	769.03	397.46	51.68
part08-100000.xml	1530.56	800.86	52.32

Tabela 5.17: Tempo de processamento dos ficheiros de avaliação

Observando os ficheiros com 100 ou mais frases, verifica-se que o sistema ficou cerca de 2 vezes mais rápido do que na secção anterior. Observando a tabela 5.18, esta que contém a avaliação inicial e a avaliação final do sistema, conclui-se que, após todas as alterações e optimizações realizadas ao sistema, o sistema RuDriCo2, excluindo o ficheiro só com uma frase, passou em média a processar os ficheiros de avaliação em 10.42% do tempo inicial. Para o ficheiro de uma só frase, não há um aumento de desempenho idêntico aos restantes, porque o tempo de inicializar o sistema é maior do que o tempo de processamento da frase.

	Original (s)	RuDriCo2 (s)	%
part08-1.xml	0.16	0.09	56.25
part08-10.xml	1.96	0.23	11.56
part08-100.xml	8.33	0.89	10.73
part08-500.xml	38.00	3.91	10.28
part08-1000.xml	78.00	7.78	9.97
part08-5000.xml	392.00	38.90	9.92
part08-10000.xml	782.75	78.70	10.05
part08-50000.xml	não processa	397.46	-
part08-100000.xml	não processa	800.86	-

Tabela 5.18: Tempo de processamento dos ficheiros de avaliação

Na tabela 5.19, apresentam-se os valores da memória utilizada no sistema final e no sistema avaliado na secção anterior. Verifica-se que o RuDriCo2 reduziu a memória ocupada, em média, para 93.70% da

	Propriedades automáticas (Mb)	RuDriCo2 (Mb)	%
part08-1.xml	< 1.00	< 1.00	-
part08-10.xml	< 1.00	< 1.00	-
part08-100.xml	5.70	5.50	96.49
part08-500.xml	8.60	8.20	95.35
part08-1000.xml	12.50	10.00	91.20
part08-5000.xml	44.70	42.30	94.63
part08-10000.xml	84.60	78.10	92.32
part08-50000.xml	410.40	386.00	94.05
part08-100000.xml	850.40	781.20	91.86

Tabela 5.19: Memória utilizada no processamento dos ficheiros de avaliação

memória utilizada no sistema avaliado na secção anterior.

Na tabela 5.20, apresentam-se os valores da memória utilizada no sistema original e no sistema final e pode-se concluir que o sistema aumenta o desempenho quando o número de frases cresce. Para contabilizar a redução de memória, observe-se o último ficheiro que o sistema original processa. Aí, verifica-se que o sistema final reduziu a memória utilizada para 3.05% da memória do sistema original.

	Original (Mb)	RuDriCo2 (Mb)	%
part08-1.xml	< 1.00	< 1.00	-
part08-10.xml	10.60	< 1.00	-
part08-100.xml	34.90	5.50	15.76
part08-500.xml	143.00	8.20	5.73
part08-1000.xml	275.00	10.00	3.64
part08-5000.xml	1234.00	42.30	3.43
part08-10000.xml	2560.00	78.10	3.05
part08-50000.xml	não processa	386.00	-
part08-100000.xml	não processa	781.20	-

Tabela 5.20: Memória utilizada no processamento dos ficheiros de avaliação

Para finalizar a avaliação do desempenho do sistema, apresenta-se, de seguida, a avaliação final da fase de pré-processamento de regras. Houve alterações que tiveram impacto nesta fase de processamento, mas que não foram consideradas fases de avaliação intermédias do sistema, como, por exemplo, as alterações apresentadas na secção 4.8. A tabela 5.21 apresenta o resultado final do tempo de pré-processamento de regras e o tempo anterior, enquanto a tabela 5.22 apresenta o tempo do sistema final e original.

As alterações referidas nesta secção aumentaram o tempo de pré-processamento de regras para 102%

	Propriedades automáticas	RuDriCo2
Tempo(s)	0.50	0.51

Tabela 5.21: Tempo de geração do ficheiro optimizado

	Original	RuDriCo2
Tempo(s)	0.49	0.51

Tabela 5.22: Tempo de geração do ficheiro optimizado

do tempo medido anteriormente. Comparando o tempo de pré-processamento entre o RuDriCo e o RuDriCo2, verifica-se que o RuDriCo2 aumentou o tempo em cerca de 4%, o que corresponde a 0.02s. Esta baixa de desempenho no pré-processamento das regras é desprezável, porque as regras só são pré--processadas quando algo é alterado.

5.2 Avaliação da sintaxe

Ao longo do capítulo 4, realizaram-se alterações à sintaxe original do RuDriCo, alterações essas que têm os seguintes objectivos:

- reduzir o tamanho das regras;
- reduzir o uso de variáveis;
- aumentar a expressividade das regras.

Uma das alterações mais relevantes é a adição de contextos, porque permite que as regras fiquem mais expressivas e que o número de variáveis usadas seja reduzido, o que, por sua vez, torna as regras mais compactas. Outra alteração relevante consiste em colocar a forma superficial e o lema como pares propriedade-valor, pois torna possível ocultar a forma superficial e o lema quando se representa um item. As alterações foram realizadas incrementalmente e a sintaxe do RuDriCo2 é a sintaxe que contém todas as alterações. Observe-se, como exemplo, a regra:

```
S1 [L1,'CAT'/'pre']
S2 ['poder','CAT'/'nou'] ['poder','CAT'/'ver','MOD'/'inf']
S3 [L3,'CAT'/'ver','MOD'/'inf']
-- >
S1*
S2 ['poder','CAT'/'nou']-
S3* .
```

Esta, na sintaxe do RuDriCo2, é escrita da seguinte forma:

```
disamb:
|[CAT='pre']|
[lemma='poder',CAT='nou'] [lemma='poder',CAT='ver',MOD='inf']
|[CAT='ver',MOD='inf']|
:=
[lemma='poder',CAT='nou']-.
```

Comparando as sintaxes, nota-se que, no RuDriCo2, a regra é mais compacta e não é necessário usar uma única variável.

Importa aqui referir que a avaliação da sintaxe é subjectiva, na medida em que algumas das suas propriedades não são contabilizáveis. O critério avaliado é a determinação do quão compactas se tornaram as regras, porque, na secção 2.4, já se havia verificado que a sintaxe do sistema XIP é muito mais compacta e que são usadas menos variáveis. A avaliação deste critério é realizada comparando o tamanho dos ficheiros de regras do RuDriCo original com os ficheiros de regras do RuDriCo2. Note-se que, na conversão das regras do RuDriCo para o RuDriCo2, os operadores disjunção, negação e item opcional não são usados, uma vez que esta conversão foi realizada por um programa que não tem a capacidade de introduzir estes operadores.

Como já foi referido anteriormente, as regras estão divididas em 35 ficheiros, mas, neste caso concreto, as regras são agrupadas em três ficheiros, um ficheiro por tipo de regra. Além disso, os comentários, o caractere espaço e o caractere quebra de linha foram removidos dos ficheiros antes da medição de tamanhos. Na tabela 5.23, é apresentado o tamanho de cada ficheiro.

Tipo de Regras	RuDriCo	RuDriCo2	%
Descontracção	105.70kb	92.90kb	88%
Desambiguação	83.60kb	20.00kb	24%
Contracção	4.20mb	3.30mb	79%

Tabela 5.23: Tamanho dos ficheiros de regras

Observando a tabela, verifica-se que as regras de desambiguação reduzem o tamanho do ficheiro para 24% do tamanho original, sendo este o tipo de regras que apresenta um maior ganho, pois a maioria destas regras usam contextos.

Capítulo 6

Conclusões e Trabalho Futuro

Um dos principais contributos do presente trabalho foi a resolução do baixo desempenho do sistema RuDriCo, sendo que o aumento de desempenho foi obtido através de algumas alterações realizadas ao sistema original, tal como descrito no capítulo 4. As alterações que mais contribuíram para o aumento de desempenho foram:

- a introdução de camadas;
- a introdução de contextos juntamente com a alteração da representação dos itens;
- as optimizações realizadas ao algoritmo de aplicação de regras;
- índice de regras com formas superficiais dos dois primeiros itens.

No que diz respeito à introdução de camadas, esta operação permite que o algoritmo de aplicação de regras fique com melhor desempenho, porque, deste modo, o sistema aplica as regras por camada, em vez de estar constantemente a tentar aplicar todas as regras. Paralelamente, e como as camadas adicionam uma complexidade adicional ao sistema, foi realizado um estudo que mostra que, a partir das 17 regras por camada, o sistema fica mais rápido, tendo o seu pico de desempenho nas 167 regras por camada.

Acerca da introdução de contextos, verificou-se que esta, juntamente com a alteração da representação dos itens, permitiu que fossem usadas menos variáveis nas regras, facto que tornou as regras menos trabalhosas para o sistema.

Já quanto às optimizações realizadas ao algoritmo de aplicação de regras, estas fazem com que o algoritmo reduza o número de passos que faz ao aplicar uma regra.

O novo índice de regras, por sua vez, faz com que o sistema reduza o número de regras candidatas, o que leva a um aumento de desempenho.

De acordo com os resultados apresentados no capítulo 5, pode-se afirmar que o RuDriCo2 é cerca de 10 vezes mais rápido do que o RuDriCo.

A sintaxe do RuDriCo2 é mais expressiva e mais compacta do que a sintaxe do RuDriCo e a introdução dos novos operadores também permite escrever regras que não podem ser escritas no RuDriCo, por exemplo, para simular a negação de uma categoria no RuDriCo, é necessário usar tantas regras quanto as restantes categorias presentes no sistema. Cada tipo de regras ficou com uma sintaxe diferente de modo a ser possível a existência de operadores e verificações específicas a cada tipo de regra. Foram ainda adicionados dois operadores específicos às regras de contracção de segmentos que permitem que estas regras fiquem mais compactas, na medida em que não é necessário escrever a forma superficial no consequente das regras quando esta é a concatenação das formas superficiais dos itens que emparelham no

antecedente. A introdução de contextos juntamente com a alteração da representação dos itens permite que as regras do RuDriCo2 sejam mais simples de escrever e mais compactas, usando menos variáveis. Tome-se como exemplo o facto de, no RuDriCo2, não ser necessário simular contextos recorrendo a variáveis.

Importa ainda referir que a validação das regras do RuDriCo2 foi melhorada em relação à validação das regras do RuDriCo. Assim, o RuDriCo2 valida todas as propriedades e valores contidos nas regras, excepto as formas superficiais e os lemas. No RuDriCo2, se uma variável aparece no consequente e esta não aparece no antecedente, é gerado um erro, dado que a variável é inconsistente. Adicionalmente, é também verificado se todas as regras de contracção se podem aplicar ou não. Estas melhorias à validação das regras ajudam os utilizadores a encontrar erros nas mesmas.

Relativamente ao trabalho futuro, verifica-se que a operação que compara itens com segmentos é realizada com muita frequência no decorrer do sistema. Actualmente, esta operação é uma comparação entre propriedades dos segmentos e propriedades dos itens. Estas propriedades estão representadas num mapa de inteiros, sendo realizadas comparações entre inteiros. Futuramente, pode ser mudada a representação interna das propriedades dos itens e dos segmentos, por exemplo, para arrays de bits onde cada bit representa o valor de uma propriedade. Com esta representação, a comparação entre segmentos e itens resume-se a uma operação binária entre os arrays e é esperado que esta operação seja mais eficiente do que a comparação actual.

Bibliografia

[Brill, 1992] Brill, E. (1992). A simple rule-based part of speech tagger. In Proceedings of the third conference on Applied natural language processing pp. 152–155, Association for Computational Linguistics, Morristown, NJ, USA.

[Church, 1988] Church, K. W. (1988). A Stochastic Parts Program and noun Phrase Parser for Unrestricted Text. In Second Conference on Applied Natural Language Processing pp. 136–143,, Austin,Texas.

[Cole et al., 1995] Cole, R. A., Mariani, J., Uszkoreit, H., Zaenen, A. and Zue, V. (1995). Survey of the State of the Art in Human Language Technology, Center for Spoken Language Understanding CSLU, Carnegie Mellon University, Pittsburgh, PA.

[Cutting et al., 1992] Cutting, D., Kupiec, J., Pedersen, J. and Sibun, P. (1992). A Practial Part-of-Speech Tagger. In Proceedings of the 3rd ACL Conference on Applied Natural Language Processing pp. 133–140,, Trento,Italy.

[Garside et al., 1997] Garside, R., Leech, G. N. and McEnery, T. (1997). Corpus Annotation: Linguistic Information from Computer Text Corpora. Longman.

[Greene and Rubin, 1962] Greene, B. B. and Rubin, G. M. (1962). Automatic Grammatical Tagging of English. Technical Report, Brown University, Providence, RI.

[Hagège et al., 1998] Hagège, C., Meireles, A., Diogo, C., Leite, F., Barão, N. and Cotovio, P. (1998). Actas do XIV Encontro Nacional da Associação Portuguesa de Linguística. In Desambiguador de Etiquetagem Dirigido por Regras Linguísticas vol. II, Universidade de Aveiro, Aveiro.

[Klein and Simmons, 1963] Klein, S. and Simmons, R. F. (1963). A Computational Approach to Grammatical Coding of English Words. In Journal of the Association for Computational MAchinery (10) pp. 334–347,.

[Marques, 2008] Marques, J. (2008). Relatório Da Bolsa. Technical report Instituto Superior Técnico - Universidade Técnica de Lisboa, Portugal.

[Medeiros, 1995] Medeiros, J. C. (1995). Processamento Morfológico e Correcção Ortográfica do Português. Master's thesis Instituto Superior Técnico - Universidade Técnica de Lisboa, Portugal.

[Márquez and Padró, 1997] Márquez, L. and Padró, L. (1997). A Flexible POS Tagger Using an Automaticalluy Acquired Language Model. In Proceedings of the 35th Annual Metting of the Association for Computational Linguistics pp. 238–245,, Madrid.

[Pardal, 2007] Pardal, J. (2007). Manual do Utilizador do RuDriCo. Technical report Instituto Superior Técnico - Universidade Técnica de Lisboa, Portugal.

[Ratnaparkhi, 1996] Ratnaparkhi, A. (1996). A Maximum Entropy Model for Part-of-Speech Taging. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP-96), Philadelphia, PA.

[Ratnaparkhi, 1998] Ratnaparkhi, A. (1998). Maximum Entropy Models for Natural Language Ambiguity Resolution. PhD thesis, University of Pennynsilvania.

[Ribeiro et al., 2003] Ribeiro, R., Mamede, N. J. and Trancoso, I. (2003). Computational Processing of the Portuguese Language: 6th International Workshop, PROPOR 2003, Faro, Portugal, June 26-27, 2003 vol. 2721, chapter Using Morphossyntactic Information in TTS Systems: Comparing Strategies for European Portuguese. : Springer.

[Schmid, 1994a] Schmid, H. (1994a). Part-of-Speech Tagging with Neural Networks. In Proceedings of the 15th International Conference on Computational Linguistics, Kyoto, Japão.

[Schmid, 1994b] Schmid, H. (1994b). Probabilistic Part-of-Speech Tagging using Decision Trees. In Proceedings of the 15th International Conference on new methods in language processing, Manchester, Reino Unido.

[Schulze et al., 1994] Schulze, B. M., Heid, U., Schmid, H., Schiller, A., Rooth, M., Grefenstette, G., Gaschler, J., Zaenen, A. and Teufel, S. (1994). DECIDE. MLAP-Project 93-19 D-1b I STR and RXRC.

[Stolz et al., 1965] Stolz, W. S., Tannembaum, P. H. and Carstensen, F. V. (1965). Stochastic Aproach to the grammatical Coding of English. In Communications Of the ACM 8(6) pp. 399–405,.

[Viterbi, 1967] Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimal decoding algorithm. In IEEE Transactions on Information Theory , 260–269.

[Voutilainen, 1995a] Voutilainen, A. (1995a). A systax-based par-of-speech analyser. In Proceedings of 7th Conference of the European Chapter of The Association for Computational Linguistics, Dublin.

[Voutilainen, 1995b] Voutilainen, A. (1995b). Constraint Grammar: a Language-Independent System for Parsing Unrestricted Text chapter Morphological Disambiguation. : Mouton de Gruyter.

[Xerox, 2003] Xerox (2003). Xerox Incremental Parser – Reference Guide.