UNIVERSIDADE DE SÃO PAULO
ESCOLA POLITÉCNICA

Projeto de Formatura

Corretor Gramatical Acoplável ao OpenOffice.org

CoGrOO 2.0

DIOGO ABREU MEYER PIRES
FÁBIO WANG GUSUKUMA
MARCELO SUZUMURA
WILLIAM DANIEL COLEN DE MOURA SILVA

São Paulo
2006

DIOGO ABREU MEYER PIRES
FÁBIO WANG GUSUKUMA
MARCELO SUZUMURA
WILLIAM DANIEL COLEN DE MOURA SILVA

Corretor Gramatical Acoplável ao OpenOffice.org CoGrOO 2.0

Projeto de Formatura apresentado à disciplina PCS 2050/2502 – Laboratório de Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Orientador: Prof. Dr. Jorge Kinoshita
Co-orientador: Prof. M.Sc. Carlos Eduardo Dantas de Menezes

São Paulo
2006

FOLHA DE APROVAÇÃO

Diogo Abreu Meyer Pires
Fábio Wang Gusukuma
Marcelo Suzumura
William Daniel Colen de Moura Silva

Corretor Gramatical Acoplável
ao OpenOffice.org
CoGrOO 2.0

Projeto de Formatura apresentado à disciplina PCS 2050/2502 – Laboratório de Projeto de Formatura II, da Escola Politécnica da Universidade de São Paulo.

Aprovado em:

Banca Examinadora

Prof_____

Instituição:_____ Assinatura:_____

Prof_____

Instituição:_____ Assinatura:_____

Prof_____

Instituição:_____ Assinatura:_____

DEDICATÓRIA

Dedicamos aos nossos familiares e amigos que nos apoiaram ao longo de todos esses anos.

AGRADECIMENTOS

Aos professores Jorge Kinoshita e Carlos Eduardo Dantas de Menezes por idealizarem este projeto e nos acolherem como seus orientados nesta longa jornada que há muito começou e que por muito tempo continuará.

Ao atual desenvolvedor da interface gráfica do corretor no OpenOffice.org, Bruno Cesar Brito Sant'Anna, por seu trabalho incansável durante a elaboração deste Projeto de Formatura, e sua brilhante atuação no Google Summer of Code.

Ao ex-desenvolvedor do corretor, Marcos Yoshio Okamura Oku, por seu apoio e participação na criação da primeira versão do CoGrOO, além do conhecimento legado aos novos integrantes.

Às Profª Laís Nascimento Salvador, uma das líderes da versão inicial do corretor, e Profª Sueli Caramello Uliano, responsável pela elaboração de grande parte das regras implementadas.

Aos ex-desenvolvedores da interface e do instalador do CoGrOO 1.0, Gabriel Bakiewicz, Fábio Blanco, Edgard Lemos, José Fontebasso Neto e André Felipe Santos.

Ao Marcelo Bonilha (*in memoriam*), que nos deixou importante documentação utilizada até hoje.

À Escola Politécnica da Universidade de São Paulo, por nossa formação acadêmica.

À Financiadora de Estudos e Projetos (FINEP) pela concessão de bolsas de estágio e iniciação científica e pelo apoio financeiro para a realização do CoGrOO 1.0.

Aos entusiastas de software livre e do nosso corretor, cujas sugestões têm ajudado a torná-lo ainda melhor.

"Fomos levados ~~à crer~~."
Anônimo

RESUMO

Esta monografia descreve a construção de uma nova versão do corretor gramatical para língua portuguesa brasileira acoplável ao OpenOffice.org. Contempla brevemente as motivações e os objetivos para a construção de um corretor gramatical livre, apresenta as características do novo projeto e mostra as vantagens a serem introduzidas nesta nova versão em relação à versão anterior. Introduz os conceitos teóricos de processamento de linguagens naturais envolvidos no projeto, para, em seguida, apresentar o processo de construção, de forma detalhada. Mostra também a formulação de uma metodologia de testes, para validação dos módulos que compõem o corretor.

ABSTRACT

This monograph describes the design of a new version of the Brazilian Portuguese grammar checker for OpenOffice.org. Briefly presents the motivations and the objectives for the project of an open source grammar checker, the new project features and the advantages introduced in this new version, in relation to the previous one. Introduces natural language processing theoretical concepts involved in the project, and right after it, the design process, in a thorough way. Also shows the formulation of a test methodology, so the modules that are part of the grammar checker can be validated.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

1 INTRODUÇÃO

O OpenOffice.org está sendo adotado em grande escala atualmente tanto por usuários pessoais quanto por usuários corporativos. Seu sucesso está se devendo ao fato de ele ser uma suíte completa de escritório, que inclui editor de textos, planilha de cálculos, editor de apresentações, entre outros; e também ao fato de ter custo zero, podendo ser obtido gratuitamente para uso. Tudo isso ainda se soma ao fato do OpenOffice.org ter funcionalidades e interface gráfica muito similar a competidores proprietários, o que leva a uma curva de aprendizado relativamente curta; e ser multi-plataforma, o que permite que seja utilizado em Windows, Linux e Solaris.

No entanto, havia uma deficiência no OpenOffice.org, que era a falta de um corretor gramatical [SOUZA], suprida pela construção da primeira versão do CoGrOO[1] [KINOSHITA 2005, KINOSHITA 2006]. A falta de um corretor gramatical constituía uma desvantagem competitiva em relação aos concorrentes proprietários, que geralmente possuíam tal ferramenta de correção.

1.1 Corretores automáticos de erros em textos escritos

Textos escritos estão sujeitos a diversos tipos de erros [NABER, 2003]:

- erros ortográficos: palavras grafadas incorretamente, por exemplo, por troca de letras de mesmo som. Exemplo: sugeito, cuja grafia correta é, na verdade, sujeito.

1 Corretor Gramatical acoplável ao OpenOffice, software produzido com o patrocínio da FINEP, Chamada Pública MCT/FINEP/Software Livre CT-INFO/FINEP-01/2003.

Tais erros são relativamente fáceis de serem corrigidos através de comparações com dicionários de palavras grafadas corretamente.

- erros gramaticais: a estrutura da sentença não segue a norma padrão da língua. Por exemplo, em "Nós foi levado à crer." existem dois desvios gramaticais, sendo a primeira ocorrência em "Nós foi" na qual a flexão do verbo não concorda com o pronome e a segunda é "à crer", na qual foi indevidamente utilizada a crase antes de verbo. Este tipo de erro é mais complexo, pois envolve a análise da estrutura da sentença, o que por sua vez envolve a análise do contexto, não somente das palavras, mas de um conjunto de palavras e as relações entre elas.

- erros de estilo: em certos contextos, o uso de expressões sintáticas complexas e palavras incomuns prejudicam o entendimento do texto. Exemplo: "Tergiversação, procrastinação e postergação comuns atos são na vida dos estudantes, recrudescendo ululantemente ao longo dos anos." escrita em uma revista informativa para adolescentes. Estes erros envolvem se a estrutura da sentença e as palavras utilizadas no texto estão adequadas para o domínio para o qual se escreve.

- erros semânticos: são erros dependentes de contexto, que ocorrem em sentenças sem erros ortográficos, gramaticais ou de estilo, mas que apresentam idéias incoerentes, como por exemplo, "O carro escreve muito bem.". Tais erros são muito difíceis de serem detectados, pois envolve a análise semântica da sentença.

- erros de apresentação: são erros de disposição gráfica textual que não seguem padrões estabelecidos de datilografia, como a inserção indevida de espaço entre uma vírgula e a palavra imediatamente anterior.

Desses erros, os ortográficos são passíveis de serem corrigidos por uma ferramenta de

correção automática computacional e tais ferramentas estão presentes na maioria dos editores de texto existentes no mercado, tanto livres quanto proprietários, dado que o problema não é inerentemente muito complexo.

Os erros gramaticais também são corrigíveis por meio de ferramentas automáticas, mas tais ferramentas estão presentes em sua maior parte nos editores de texto proprietários. Uma vez que a dificuldade de se corrigir esses erros é alta, não há uma quantidade grande, nem conhecimento estabelecido para a construção de ferramentas desse tipo. A complexidade é elevada porque envolve uma série de grandes dicionários léxicos e sintáticos, além de envolver também o aspecto de eficiência computacional, já que um corretor deve ser um processo secundário e, assim, não pode ser intrusivo nas máquinas onde será executado, ou seja, não pode comprometer o desempenho dos computadores dos usuários.

Ferramentas para correção de erros de estilo e erros semânticos ainda não tem no momento da escrita deste texto um desenvolvimento avançado, embora existam alguns desenvolvimentos nesse sentido. Constituem os corretores automáticos mais difíceis de serem construídos, pois envolve a extração e análise de sentido do texto inserido num determinado contexto da língua.

Erros de apresentação são erros simples e geralmente corrigidos por corretores gramaticais.

1.2 Objetivo do trabalho

O objetivo deste trabalho é descrever o processo de construção de um corretor gramatical para a língua portuguesa brasileira, que tenha a capacidade de apontar eventuais desvios gramaticais contidos numa sentença, com o menor número possível de falsos positivos (indicações de erros não existentes).

Para todos os desvios gramaticais encontrados na sentença, devem ser disponibilizados: uma mensagem explicativa sobre o desvio gramatical, uma indicação do local de ocorrência do desvio na sentença e, para alguns casos específicos, sugestões de correção.

Como requisito, destaca-se que este corretor deve ser acoplável a ferramentas de *desktop*, em especial o editor de textos *Writer* do *OpenOffice.org*.

Este corretor, CoGrOO 2.0, será o produto de uma reengenharia de sua versão anterior, a 1.0, concebida para corrigir diversos problemas estruturais encontrados.

1.3 Melhorias a serem introduzidas nesta nova versão

1.3.1 Plataforma computacional

1.3.1.1 Versão 1.0: linguagem de programação Perl

A primeira versão do corretor gramatical foi inteiramente construída na linguagem Perl. O Perl é uma linguagem de programação de uso geral que segue o paradigma estruturado, muito versátil, e com aplicações variadas que vão desde programação web até programas para desktop com interface gráfica. No entanto, o Perl foi escolhido para ser a linguagem utilizada na programação da primeira versão por causa de sua característica mais marcante e razão para a qual foi projetada: ser uma linguagem voltada para processamento de textos.

Na atividade de programação, as facilidades oferecidas para o processamento de texto muito contribuíram para que o grupo de desenvolvimento pudesse se focar no cumprimento dos requisitos funcionais do projeto, cujo horizonte ainda não era determinado e nem mesmo

conhecido, uma vez que o projeto foi pioneiro na área de corretores gramaticais livres.

Um problema inerente ao Perl (e não exclusivamente do Perl, mas de todas as linguagens ditas "dinâmicas") é que o Perl não verifica os tipos das variáveis em tempo de compilação, fazendo com que erros imprevisíveis ocorram durante a execução do programa.

Se por um lado o Perl oferecia bom suporte ao processamento de textos, por outro causou grande dificuldade de rastreamento do código produzido ao longo do desenvolvimento, por causa da falta de um ambiente integrado de desenvolvimento (IDE) adequado. Esse fato dificultou demasiadamente a adição de novas funcionalidades e a manutenção do código do corretor. Com a introdução de mudanças no programa, mesmo com apenas algumas dezenas de milhares de linhas de código, o código foi tornando-se extremamente complexo.

Um outro problema surgiu quando da abertura do código para a comunidade: a linguagem Perl tinha suporte nativo somente em plataformas Linux. Em plataformas Windows, havia-se a necessidade de se instalar uma versão de Perl, o que ocasionou um problema para os usuários menos preparados, gerando um desconforto que é considerado "desnecessário".

1.3.1.2 *Versão 2.0: linguagem de programação Java*

Um requisito não funcional do CoGrOO 2.0 seria então a necessidade de ser escrito em uma linguagem de programação de maior visibilidade e maior difusão entre o público e suportada em uma gama maior de plataformas se comparada à linguagem Perl.

Os usuários se mostram muito mais receptivos ao Java, que pode ter diversos outros usos num sistema computacional desktop, como a visualização de applets em navegadores da Internet, ao contrário do que ocorria com o Perl, que tinha que ser instalado somente para a

utilização do CoGrOO, na maior parte dos usuários.

Além da vantagem para o usuário final, a linguagem Java oferece diversos outros pontos favoráveis, como: paradigma de desenvolvimento orientado a objetos, verificação de tipos em tempo de compilação e tratamento de exceções, plataformas integradas de desenvolvimento estabelecidas no mercado, integração mais fácil com o OpenOffice.org em relação à integração realizada com Perl (CoGrOO 1.0).

A orientação a objetos facilita o trabalho de projeto e programação em times, através da definição de interfaces de comunicação entre módulos, o que proporciona uma maior independência entre os membros da equipe. Em outras palavras, acarreta maior possibilidade de execução de tarefas de desenvolvimento de forma paralela, melhorando a dinâmica e o fluxo de trabalho.

A verificação de tipos em tempo de compilação adiciona um nível a mais de segurança em relação às linguagens ditas dinâmicas, pois a análise do código pelo compilador Java pode indicar erros cometidos pelo programador antes de se obter o código executável. Nas linguagens dinâmicas, os erros acontecem durante a execução do programa, que se for, por exemplo, num ambiente de testes, as causas são investigadas e corrigidas, mas pode também ocorrer num ambiente de produção, expondo o usuário a funcionamentos erráticos e imprevisíveis do programa, além de, às vezes, expô-lo também a mensagens incompreensíveis.

O tratamento de exceções traz o benefício de se poderem tomar decisões de como o programa deve proceder em caso de funcionamentos imprevistos. Assim, o programa pode tentar se recuperar do problema causando o mínimo de incômodo ao usuário, seja por meio de mensagens adequadas ou por procedimentos de contorno para recuperação do erro sem que o

usuário perceba o problema, se isso não for necessário.

O uso de um ambiente integrado de desenvolvimento traz sem dúvida inúmeras vantagens em relação ao desenvolvimento com um editor de texto padrão. Geralmente, pode-se executar o programa sem sair do ambiente de desenvolvimento, existem facilidades de geração e auto-completamento de código, a sintaxe da linguagem pode ser destacada por meio de texto colorido, etc. Essas funcionalidades agilizam a programação e ajudam o programador a evitar erros.

Por fim, a integração com o OpenOffice.org é facilitada, pois ele oferece suporte a extensões construídas em Java por meio de uma Interface de Programação da Aplicação (API) padronizada. No entanto, essa integração será ainda provisória e restrita em funcionalidade, pois funcionalidades específicas para o corretor gramatical, como o assinalamento dos erros através de uma indicação visual no texto (a ondulação verde, tal como no corretor gramatical do Microsoft Word) e interação com o erro através do clique direito do mouse, ainda não são possíveis de serem feitas.

1.3.2 Maior portabilidade entre idiomas humanos, pelo uso do OpenNLP

O *OpenNLP* possui uma série de módulos para o processamento de linguagens naturais, baseados em Java, que fazem uso do pacote de aprendizado de máquina *Maxent*.

A arquitetura do *OpenNLP* é desvinculada de idiomas. Teoricamente bastaria elaborar os dados de treinamento para o *Maxent*, em qualquer língua, que teríamos então um separador de sentenças, separador de *tokens*, localizador de nomes, etiquetador morfológico, agrupador e analisador sintático simples.

Ocasionalmente pode ser útil especializar os algoritmos de aprendizado de máquina para levar em consideração particularidades dos dados de treinamento e do idioma, como foi feito no caso do CoGrOO. Isto é facilmente feito se implementando as interfaces, ou estendendo as classes, do *Maxent*.

Neste projeto, será desenvolvida uma arquitetura que integra os módulos do *OpenNLP* de forma útil ao corretor gramatical, tendo em vista a generalização das estruturas facilitando futuros trabalhos de migração para outros idiomas.

1.4 Tipos de desvios gramaticais detectados

Nesta seção serão apresentados os principais tipos de desvios gramaticais a serem apontados pelo CoGrOO, acompanhados de exemplos de sentenças inadequadas e adequadas, para demonstrar o leque de atuação desejado para o produto.

Deve-se ressaltar que a lista não é de forma alguma completa, dada a extensa coleção de tipos de desvios gramaticais que deve ser detectada pelo corretor gramatical. Optou-se por evidenciar apenas os tipos mais representativos de cada desvio detectado.

Também deve ser dito que a classificação dos desvios em tipos não é de forma alguma absoluta, podendo existir desvios que se encaixariam em mais de uma categoria, mas que, no entanto, se optou por alocá-los em um dos tipos arbitrariamente.

1.4.1 Colocação pronominal

- uso de próclise

Inadequada	Não meto-me no que não sou chamado.
Adequada	Não me meto no que não sou chamado.

1.4.2 Concordância nominal

- concordância de gênero entre substantivos e adjetivos

Inadequada	A menina feio não sabe de nada.
Adequada	A menina feia não sabe de nada.

- concordância de número entre substantivos e adjetivos

Inadequada	As faces rosada das crianças.
Adequada	As faces rosadas das crianças.

- concordância de gênero entre artigo e substantivo

Inadequada	As professores não deram aulas.
Adequada	Os professores não deram aulas.

- concordância de número entre artigo e substantivo

Inadequada	Recebemos os cartão de crédito pelo correio.
Adequada	Recebemos os cartões de crédito pelo correio.

1.4.3 Concordância entre sujeito e verbo

Inadequada	Eles foi comprar banana.
Adequada	Eles foram comprar banana.

Inadequada	Ele foram comprar banana.
Adequada	Ele foi comprar banana.

1.4.4 Concordância verbal

- "fazer" indicando tempo

Inadequada	Fazem três dias que não o vejo.
Adequada	Faz três dias que não o vejo.

- "haver" no sentido de existir

Inadequada	Haviam ratos na cozinha.
Adequada	Havia ratos na cozinha.

1.4.5 Crase (uso combinado de preposição e artigo)

- ocorrência de crase antes de substantivo masculino singular ou plural

Inadequada	Refiro-me à trabalhos remunerados.
Adequada	Refiro-me aos trabalhos remunerados.

- ocorrência de crase antes de verbo

Inadequada	Fomos levados à crer.
Adequada	Fomos levados a crer.

- ocorrência de crase em expressões indicativas de horas

Inadequada	As duas horas estaremos partindo.
Adequada	Às duas horas estaremos partindo.

- ocorrência de crase em expressões indicativas de modo, tempo, lugar etc.

Inadequada	Os ônibus estacionaram a direita do pátio.
Adequada	Os ônibus estacionaram à direita do pátio.

- ocorrência de crase antes de pronomes de tratamento

Inadequada	Enviei os documentos à Vossa Excelência.
Adequada	Enviei os documentos a Vossa Excelência.

- ausência de crase em expressões como "em relação à", "com relação à" e "devido à"

Inadequada	Devido as cobranças injustas, ficamos no vermelho.
Adequada	Devido às cobranças injustas, ficamos no vermelho.

- expressão "de segunda a sexta" e variantes

Inadequada	O balcão de informações atende de segunda à sexta-feira.
Adequada	O balcão de informações atende de segunda a sexta-feira

1.4.6 Regência nominal

- "valorização de"

Inadequada	Hoje se busca a valorização ao corpo em detrimento do espírito.
Adequada	Hoje se busca a valorização do corpo em detrimento do espírito.

1.4.7 Regência verbal

- "assistir" com o sentido de presenciar

Inadequada	Os participantes poderão assistir a apresentação dos premiados.
Adequada	Os participantes poderão assistir à apresentação dos premiados.

- "evitar" ou "usufruir"

Inadequada	Nós usufruímos da alegria de estar com vocês.
Adequada	Nós usufruímos a alegria de estar com vocês.

- "habituar"

Inadequada	Eu habituei-me com o silêncio.
Adequada	Eu habituei-me ao silêncio.

- "namorar"

Inadequada	Júlio namorou com Marina durante três anos.
Adequada	Júlio namorou Marina durante três anos.

- "obedecer" e "desobedecer"

Inadequada	Se não obedecermos as lideranças, será o caos.
Adequada	Se não obedecermos às lideranças, será o caos.

- "preferir"

Inadequada	Eu prefiro festa do que balada.
Adequada	Eu prefiro festa a balada.

1.4.8 Erros comuns na língua portuguesa falada no Brasil

- uso de "meio" como adjetivo

Inadequada	Compramos meio porção de queijo.
Adequada	Compramos meia porção de queijo.

- uso de "meio" como advérbio

Inadequada	A conclusão está meia confusa.
Adequada	A conclusão está meio confusa.

- emprego de "mim" como pronome pessoal do caso reto

Inadequada	Para mim estudar, silêncio é necessário.
Adequada	Para eu estudar, silêncio é necessário.

- emprego de "mim" e "ti"

Inadequada	Entre eu e ele só há discórdia.
Adequada	Entre mim e ele só há discórdia.

- emprego de vírgulas para separar expressões como "ou seja" e "no entanto" do fluxo da sentença

Inadequada	O trabalho mais importante, ou seja a coleta de dados, já foi iniciado.
Adequada	O trabalho mais importante, ou seja, a coleta de dados, já foi iniciado.

- redundância no uso do verbo "haver"

Inadequada	Eu nasci há 10000 anos atrás.
Adequada	Eu nasci há 10000 anos.

2 FUNDAMENTAÇÃO CONCEITUAL

Este capítulo conceitua processamento de linguagens naturais e aponta as dificuldades que o tornam complexo. É apresentado conceitos de processamento de linguagens naturais, o que é um corpus e n-gramas. Por fim, apresenta o princípio da máxima entropia e seu *framework*, a partir de um exemplo.

2.1 Processamento de linguagens naturais

Processamento de linguagens naturais (PLN) é um dos campos de estudo das grandes áreas de "Inteligência Artificial" e "Lingüística". Ele estuda os problemas de geração e compreensão automáticas das linguagens humanas.

- Geração: converter informações de um banco de dados de um computador para uma linguagem inteligível para um ser humano.
- Compreensão: converter fragmentos de linguagens humanas para representações mais formais, que podem ser manipuladas por programas.

O PLN é tido como um problema IA-completo (analogamente a um NP-completo, de tempo polinomial não-determinístico), porque o reconhecimento de linguagens naturais parece requerer um conhecimento extensivo sobre o "mundo externo" e a habilidade para manipulá-lo. A definição de "compreensão" é um dos maiores problemas no processamento de linguagens naturais.

2.1.1 Alguns problemas que tornam complexo o PLN

2.1.1.1 Detecção de limites das palavras/sentenças

Em uma linguagem falada, geralmente não há espaços entre as palavras; os limites entre elas dependem de qual escolha faz mais sentido gramatical e de acordo com o contexto. Na forma escrita, algumas linguagens como o chinês também não apresentam separação entre as palavras.

Os exemplos a seguir mostram alguns problemas na detecção de limites de sentenças nos textos:

"Sr. Silva estava jogando futebol."

"Meu e-mail é cogroo@cogroo.com.br, e meu site é www.cogroo.com.br."

"O computador novo custará R$ 2.500,00."

No primeiro exemplo se poderia confundir o ponto em "Sr." com uma marca de fim de sentença.

No segundo temos diversas marcas, como o traço em "e-mail" e os pontos em "cogroo@cogroo.com.br" e "www.cogroo.com.br".

Já o terceiro a confusão pode ocorre em "R$ 2.500,00", onde existem diversas marcas que podem separar *tokens* (",") ou sentenças (".").

2.1.1.2 Eliminação de ambigüidades dos sentidos das palavras

Muitas palavras possuem mais de um significado; é necessário escolher o significado que faz mais sentido em cada contexto.

"Nada como voltar para casa!" (substantivo)

"Ele casa na semana que vem." (verbo)

2.1.1.3 Ambigüidades sintáticas

A gramática para linguagens naturais é ambígua, isto é, é comum haver inúmeras possíveis "árvores de análise sintática" para uma dada sentença. Escolher a mais apropriada requer informações contextuais e semânticas.

"O chapéu do ursinho de pelúcia marrom."

- O chapéu é marrom?
- O ursinho é marrom?
- O chapéu é de pelúcia?
- O ursinho é de pelúcia?

2.1.1.4 Entradas incorretas ou irregulares

Sotaques estrangeiros ou regionais; ruídos durante a fala; erros de digitação ou gramaticais no OCR (Optical Character Recognition – Reconhecimento Óptico de Caracteres); palavras estrangeiras e nomes próprios.

"O q vc disse que ia fzr?"

"Bill Gates é o fundador da Microsoft."

2.2 PLN estatístico

O processamento estatístico de linguagens naturais utiliza métodos estatísticos, probabilísticos e estocásticos para resolver algumas das dificuldades discutidas na seção 2.1.1, especialmente aquelas que aparecem devido ao tamanho das sentenças – já que, quanto maiores elas forem, maior será o número de análises possíveis.

Os métodos para remover essas ambigüidades costumam envolver o uso de corpora e modelos de Markov. A tecnologia para o processamento estatístico de linguagens naturais provém principalmente do aprendizado de máquina e de uma técnica chamada data mining, que consiste em analisar grandes quantidades de informações coletadas, de forma que se possa extrair novas informações.

2.2.1 *Corpus*

Um corpus é um grande e estruturado conjunto de textos (geralmente processados e armazenados eletronicamente). Um corpus pode conter textos em uma única linguagem (corpus monolingüístico) ou textos em inúmeras línguas (corpus multilingüístico). Corpora multilingüísticos formatados especialmente para comparação lado-a-lado são chamados corpora paralelos alinhados.

Com o intuito de se fazer os corpora mais úteis às pesquisas lingüísticas, eles são normalmente sujeitos a um processo chamado anotação. Um exemplo de anotações no corpus é a etiquetagem morfológica, na qual as informações morfológicas de cada palavra são adicionadas ao corpus na forma de etiquetas. Um outro exemplo é a indicação da primitiva ("base") de cada palavra.

2.2.2 n-grama

Um n-grama é uma subseqüência de n itens de uma dada seqüência. Esta idéia remonta a um experimento de Claude Shannon. Sua idéia era a seguinte: dada uma seqüência de letras (por exemplo, a seqüência "por ex"), qual poderá ser a próxima letra? Por meio de treinamento, pode-se obter uma distribuição de probabilidade para a próxima letra: a=0,4, b=0,00001, c=0....; em que as probabilidades de todas as "próximas letras" possíveis somem 1.0.

A utilização de n-gramas é uma técnica popular no processamento de linguagens naturais. No reconhecimento de voz, os fonemas são modelados utilizando-se uma distribuição de n-gramas. No campo da análise morfossintática, cada n-grama deve representar uma seqüência composta de n palavras. Para uma seqüência de palavras como "O menino é legal", os trigramas seriam: "O menino é" e "menino é legal". Para seqüências de caracteres como "bom dia", os trigramas que podem ser gerados são "bom", "om ", "m d", " di" e "dia". Dependendo da utilização, pode-se realizar um pré-processamento para se eliminar espaços ou não. Em quase todos os casos, a pontuação é removida por pré-processamento.

Os n-gramas podem ser utilizados em praticamente qualquer tipo de dado. Eles têm sido aplicados, por exemplo, para análise e síntese de informações a partir de imagens da Terra feitas por satélite, para tornar possível descobrir de qual lugar da Terra uma certa imagem é proveniente.

Convertendo uma seqüência original de itens em n-gramas, ela se torna parte de um espaço vetorial, permitindo, assim, que a seqüência seja comparada a outras seqüências de um modo eficiente. Por exemplo, se convertêssemos strings com as letras do alfabeto completo

em trigramas, obteríamos um espaço vetorial de dimensão 26^3 (a primeira dimensão medindo o número de ocorrências de "aaa", a segunda de "aab", e assim por diante, com todas as combinações possíveis de três letras). Pode-se notar que a utilização desta representação dá margem a uma perda de informações a respeito de uma string. Por exemplo, as duas cadeias "abcba" e "bcbab" geram os mesmos 2-gramas. Porém, sabemos empiricamente que se duas cadeias de um texto real possuem representações vetoriais similares (de acordo com determinados critérios), então elas apresentarão uma grande probabilidade de serem semelhantes.

2.2.3 O princípio da máxima entropia

A noção de entropia é originária de estudos de termodinâmica, em que é utilizada para mensurar o grau de desordem de um sistema. O conceito de entropia da informação, introduzido por Shannon em seu artigo *A Mathematical Theory of Communication* (Uma teoria matemática da comunicação), em 1948, refere-se à incerteza probabilística - pode-se dizer que há três tipos de incerteza: a determinística, em que não são conhecidos os estados que um sistema pode assumir; a entrópica, em que são conhecidos os estados possíveis, mas não as probabilidades de ocorrência de cada um deles; e a probabilística, em que são conhecidos não só os estados possíveis, mas também a distribuição de probabilidade para eles - associada a uma distribuição de probabilidade. De modo geral, cada distribuição reflete diferente grau de incerteza, e diferentes graus de incerteza estão associados a diferentes distribuições, conforme pode ser visto na Figura 1 [MATOS, 2002]. Com base nisto, "ao se fazerem inferências baseadas em informações parciais, a distribuição de probabilidade que apresenta a maior entropia, sujeita a tudo que é conhecido, deverá ser escolhida, pois esta é a única afirmação que se pode fazer; utilizar qualquer outra distribuição indica que se

assumiram informações arbitrárias, as quais, por hipótese, não se tem." [JAYNES, 1957, GOOD, 1963 apud RATNAPARKHI]. Um exemplo simples pode ser observado quando se lança uma moeda, sem saber se é viciada ou não: a probabilidade mais imparcial - ou seja, sem considerar algo que não se sabe *a priori*, como, por exemplo, a moeda ser viciada - a ser atribuída a cada evento possível é de 1/2, retratando a incerteza por meio de uma distribuição uniforme.

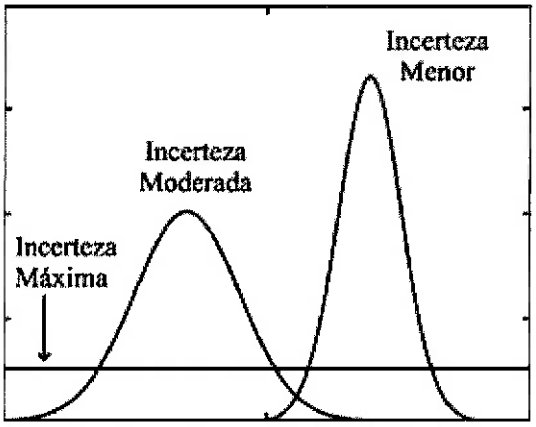


Figura 1 - Representação da incerteza com distribuição de probabilidades

2.2.3.1 A aplicação do princípio da máxima entropia no processamento de linguagens naturais

"Muitos problemas em processamento de linguagens naturais (PLN) podem ser reformulados como problemas de classificação, nas quais a tarefa é observar algum contexto lingüístico $b \in B$ e predizer classes lingüísticas corretas $a \in A$. Isto envolve construir o classificador $cl : B \rightarrow A$ que por sua vez pode ser implementado com uma distribuição p com uma probabilidade condicional, de forma que $p(a \mid b)$ é a probabilidade da classe a dado algum contexto b.

Representam-se evidências com funções chamadas de predicados contextuais ou features. Se $A = \{a_1 ... a_n\}$ representa o conjunto de possíveis classes que se está interessado em predizer, e B representa o conjunto de possíveis contextos ou material textual que se pode observar, então um predicado contextual é uma função:

$$cp : B \rightarrow \{true, false\}$$

que retorna *true* ou *false*, correspondendo à presença ou ausência de informações úteis em algum contexto, ou *histórico* $b \in B$. O conjunto exato de predicados contextuais $cp_1 ... cp_m$ que está disponível para uso varia de acordo com o problema, mas em cada problema, esses predicados devem ser fornecidos pelo usuário. Predicados contextuais são usados em *features*, que são funções da forma

$$f : A \times B \rightarrow \{0,1\}$$

Todas as *features* apresentadas aqui têm a forma

$$f_{cp,a'}(a,b) = \begin{cases} 1, \text{se } a = a' \text{ e } cp(b) = true \\ 0 \text{ caso contrário} \end{cases}$$

O conjunto de *features* utilizado em um problema é determinado por uma estratégia de seleção de *features*, que, em geral, é específico para cada problema."[RATNAPARKHI, 1998]

2.2.3.2 Uma aplicação simples sob o framework de máxima entropia

O exemplo a seguir ilustra o uso da máxima entropia em um problema bastante simples. Suponha que a tarefa seja de estimar a distribuição de probabilidade conjunta p

definida sobre $\{x,y\}\times\{0,1\}$. Suponha ainda que os únicos fatos conhecidos sobre p são que $p(x,0)+p(y,0)=0.6$, e que $p(x,0)+p(y,0)+p(x,1)+p(y,1)=1.0$ (Apesar de a restrição $\sum_{a,b} p(a,b)=1$ ser implícita, uma vez que p é uma distribuição de probabilidade, ela será tratada como uma restrição imposta externamente, para fins ilustrativos).

Em uma tarefa de predição, x e y seriam observações mutuamente exclusivas, e 0 e 1 seriam as duas saídas mutuamente exclusivas em que estaríamos interessados Por exemplo, suponha que a tarefa atual seja determinar a probabilidade com que estudantes do primeiro ano não recebam notas "A", e suponha que se atribua a seguinte interpretação ao espaço de eventos $\{x,y\}\times\{0,1\}$:

x = estudante está no primeiro ano

y = estudante não está no primeiro ano

0 = a nota do estudante é A

1 = a nota do estudante não é A

Assim, o fato observado de que "60% de todos os estudantes receberam uma nota A" seria implementado com a restrição $p(x,0)+p(y,0)=0.6$. E o fato implícito de que "100% de todos os estudantes receberam uma nota A ou não A" seria implementado com a restrição $\sum_{a,b} p(a,b)=1$. O objetivo nessa modelagem é estimar completamente a distribuição p, de modo que questões como "Qual é a porcentagem (estimada) de estudantes do primeiro ano não receberam uma nota A?" possam ser respondidas por um cálculo de uma probabilidade, como $p(x,1)$.

Tabela 1 - A tarefa é encontrar a distribuição de probabilidade p que atenda às restrições feitas.

$p(a,b)$	0	1	
x	?	?	
y	?	?	
Total	0.6		1.0

A tabela 2.1 representa a distribuição de probabilidade p como quatro células preenchidas com um "?", cujos valores devem estar de acordo com as restrições. Evidentemente, há infinitos modos de se preencher as células da tabela; um desses modos é apresentado na tabela 2.2. Contudo, o Princípio da Máxima Entropia recomenda a atribuição de valores apresentada na tabela 2.3, que é o modo mais imparcial (i.e., que não utiliza nenhuma outra restrição que *a priori* não se conhece) de fazê-lo.

Tabela 2 - Um modo de satisfazer às restrições.

$p(a,b)$	0	1	
x	0.5	0.1	
y	0.1	0.3	
Total	0.6		1.0

Tabela 3 - O modo recomendado pelo princípio da entropia máxima.

$p(a,b)$	0	1	
x	0.3	0.2	
y	0.3	0.2	
Total	0.6		1.0

Formalmente, sob o *framework* de máxima entropia, o fato

$$p(x,0)+p(y,0)=0.6$$

é implementado como uma restrição sobre a expectativa do modelo p da *feature* f_1:

$$E_p f_1 = 0.6$$

em que

$$E_p f_1 = \sum_{a\in\{x,y\},b\in\{0,1\}} p(a,b) f_1(a,b)$$

e também f_1 é definida como segue:

$$f_1(a,b) = \begin{cases} 1, \text{se } b = 0 \\ 0, \text{caso contrário} \end{cases}$$

De modo similar, o fato

$$p(x,0)+p(y,0)+p(x,1)+p(y,1)=1.0$$

é implementado como a restrição

$$E_p f_2 = 1.0$$

em que

$$E_p f_1 = \sum_{a\in\{x,y\},b\in\{0,1\}} p(a,b) f_2(a,b)$$

$$f_2(a,b)=1$$

O objetivo sob o framework da máxima entropia é, então, maximizar

$$H(p) = -\sum_{a\in\{x,y\},b\in\{0,1\}} p(a,b)\log p(a,b)$$

sujeito às restrições acima.

3 ESPECIFICAÇÃO

O corretor gramatical deve implementar uma série de funcionalidades. Estas estão descritas nos requisitos funcionais do sistema. Para ser eficiente, no entanto, o corretor gramatical deve também atingir os requisitos não funcionais também descritos.

3.1 Requisitos funcionais

3.1.1 Interface externa

O software deverá prover uma interface para a qual se submete uma sentença e que, após a analise dessa sentença, retornará, no caso de se encontrar desvio gramatical:

- A posição (em índice de caracteres) do inicio e do fim do desvio na sentença;
- Uma explicação a respeito do desvio, caso encontrado;
- Sugestão para contornar o problema, se possível;

Essa interface deve ser flexível e acoplável a qualquer aplicativo no qual a funcionalidade de correção gramatical seja desejada. Em especial, neste trabalho, será implementada a interface com o *OpenOffice.org Writer*.

3.1.2 Processamento de linguagens naturais

3.1.2.1 Dicionários

Para o processamento do texto deverá ser construída uma base de dados contendo palavras com anotações morfológicas, do tipo classe, gênero, número, pessoa etc., e alguma forma de relacionamento entre palavras derivadas.

Deverão ser disponibilizadas as seguintes formas de consulta:

Dada uma palavra, determinar quais suas possíveis classificações morfológicas. Exemplo: "meninas", derivada de "menino", substantivo feminino plural; "casa", palavra primitiva, substantivo feminino singular, ou proveniente do verbo casar, terceira pessoa do singular do presente do indicativo.

Dada uma primitiva e sua classificação morfológica, determinar sua flexão para a classificação. Exemplo: "menino" + "substantivo feminino singular" resultaria em "menina".

O CoGrOO 1.0 dispõe de uma base de dados desse tipo que poderá ser adaptada e utilizada nesta nova versão do CoGrOO.

Deverá ser criada uma interface de leitura dessa base de dados, que poderá no futuro ser estendida para outras bases de acordo com a necessidade.

3.1.2.2 Corpus para entrada estatística

Para treinamento dos modelos, deve-se obter um corpus com texto classificados morfossintaticamente e primitivas das palavras.

Para o CoGrOO 1.0 foi obtido um texto desse tipo já pré-processado. Ele poderá também ser usado na nova versão.

Deverá ser criada uma interface de leitura desse corpus que poderá ser estendia para diversos formatos de corpus, conforme necessário.

3.1.2.3 Separador de sentenças

Deverá ser construído um módulo separador de sentenças. Para esse módulo a entrada é uma cadeia de caracteres que representa um texto. A saída deverá ser uma coleção de cadeias de caracteres que representam as sentenças encontradas no texto, na mesma ordem da cadeia de entrada.

Deverão ser considerados como separadores de sentenças os seguintes sinais gráficos: ponto final (.), ponto de interrogação (?) e ponto de exclamação (!).

3.1.2.4 Separador de tokens

Deverá ser construído um módulo separador de *tokens*. *Tokens* são os diferentes elementos que compõe uma sentença. Entre eles marcas de pontuação, palavras, números etc.

Nesse módulo, a entrada deve ser uma sentença na forma de cadeia de caracteres. A saída, uma coleção ordenada de cadeias de caracteres, que representam cada *token*.

3.1.2.5 Etiquetador morfológico

Deverá ser construído um módulo capaz de decidir a classificação morfológica de cada palavra de acordo com o contexto em que a palavra ocorre na sentença.

Sua entrada será uma coleção ordenada de cadeias de caracteres, representando cada *token* da sentença. A saída, uma coleção de etiquetas morfológicas que melhor se enquadra a cada um dos *tokens* de entrada, sendo cada etiqueta da coleção associada a um *token* de entrada.

3.1.2.6 Agrupador

Deverá ser construído um módulo capaz de agrupar *tokens* etiquetados que podem ser considerados um componente da sentença. Deverão ser identificados sintagmas nominais e verbais.

Sua entrada deverá ser uma coleção de *tokens* etiquetados morfologicamente. A saída, etiquetas que indicam as formações de sintagmas.

3.1.2.7 Analisador sintático simples

Deverá ser construído um módulo capaz de identificar agrupamentos que formam sujeito e verbo.

Sua entrada deverá ser uma coleção de *tokens* etiquetados morfologicamente e indicações de *tokens*. A saída, etiquetas que indicam as formações de sujeito verbo.

3.1.2.8 Analisador de desvios gramaticais

Deverá ser construído um módulo que identifica desvios gramaticais. Este módulo deve prover meios de entrar regras sem necessariamente conhecer as estruturas internas do corretor gramatical.

A entrada deve ser a estrutura da sentença contendo *tokens*, etiquetas, sintagmas e agrupamentos sujeito verbo.

A saída uma estrutura de dados contendo a mensagem de erro, o trecho da sentença com problema e sugestões de como contornar o desvio gramatical.

3.2 Requisitos não funcionais

Os requisitos não funcionais não necessariamente precisaram ser alcançados nesse projeto que tem como ênfase a construção de um corretor gramatical seguindo os requisitos funcionais.

No entanto será feito um estudo de como cumprir os requisitos não funcionais, uma vez que falhas nestes requisitos tornariam o corretor gramatical inutilizável.

3.2.1 Tempo de resposta

Por lidar com o usuário, o tempo de resposta do corretor gramatical deve ser pequeno o bastante para não lhe causar desconforto.

Como métrica, deverá ser comparado com o tempo de resposta do CoGrOO 1.0, que se apresenta, em média, inferior a um segundo por sentença.

3.2.2 Consumo de recursos do processador e memória

Por um lado se sabe que o processamento necessário em um corretor gramatical é intenso, dada a grande quantidade de rotinas de inteligência artificial necessárias para verificação de textos. No entanto este processamento deve ser otimizado para causar o mínimo impacto ao usuário. Para ele, o sistema com ou sem o corretor gramatical rodando deve ter a mesmo desempenho.

Uma medida adotada é que o gasto de processamento e de memória se mantenha inferior ao consumido pelo próprio editor de textos ao qual o corretor esteja acoplado.

3.2.3 Espaço em disco

Um dos grandes problemas do corretor gramatical estatístico é que necessita de um extenso banco de dados, que pode atingir dezenas de mega *bytes*. Para ser de fácil distribuição, inclusive ser incluído nos próprios instaladores de aplicativos cuja funcionalidade de correção gramatical fosse interessante, o pacote de instalação deve ser pequeno.

Dado o grande desafio de se obter tamanhos realmente pequenos, este projeto vai tomar por base levantamentos feitos acerca do tamanho dos instaladores de softwares similares, que é de dez mega *bytes*.

3.2.4 Estabilidade

Deve ser estável o suficiente para não comprometer o bom funcionamento do sistema. Para o usuário, seu uso deve ser transparente. Erros devem ser apenas armazenados em arquivos de log.

3.2.5 Taxa de acertos

a) Separador de sentenças

Deve atingir taxas em torno de 98% de acerto na separação de sentenças em textos, que foi o resultado obtido experimentalmente usando o algoritmo de entropia máxima [RATNAPARKHI, 1998]

b) Separador de *tokens*

Não foram encontrados dados experimentais, no entanto considerando a similaridade deste com o separador de sentenças, foi estimada como alvo a mesma taxa de acerto, 98%.

c) Etiquetador morfológico

Deve atingir taxa acima de 95%. Esse processamento é complexo. Etiquetadores morfológicos baseados em estatísticas têm taxa de acerto entre 95 e 97% [BRILL, 1992]

d) Agrupador

Por sua simplicidade, e por ter um algoritmo semelhante ao do analisador sintático simples também será adotada como alvo 88% de acerto, conforme experiências com entropia máxima. [RATNAPARKHI, 1998]

e) Analisador sintático simples

Será adotada como alvo a taxa de um agrupador (88%), pois sua implementação é semelhante.

f) Analisador de desvios gramaticais

Devem ser considerados, nesse caso, comparativos com outros softwares do tipo. Deve ser priorizado o mínimo de falsos positivos já que estes tornam incômodo o uso do corretor.

3.2.6 Segurança de dados

Por tratar de dados muitas vezes particulares, outras até confidenciais, o sistema de transmissão de dados deve ser seguro. Não deve existir, nas versões para usuários, nenhum tipo de log que possa conter dados de entrada do usuário.

4 PLANEJAMENTO

No sentido de se agregar um maior controle ao projeto, optou-se por seguir uma metodologia de desenvolvimento de software estabelecida no mercado.

Foi escolhido o *Rational Unified Process* (RUP) como guia norteador, pela sua versatilidade e adaptabilidade a projetos de diferentes portes e complexidades.

O CoGrOO 2.0 pode ser considerado um projeto de baixa-média complexidade, de baixa interatividade com clientes externos, mas de uso intensivo de processamento e acesso a dados.

Considerando a complexidade do projeto, o RUP foi utilizado no desenvolvimento com um baixo nível de cerimônia, não sendo produzidos muitos artefatos, mas sim somente os que eram considerados mais relevantes, como o documento de visão, de riscos e de negócios.

Algumas práticas do RUP foram adotadas para guiar o desenvolvimento, como a divisão de tarefas entre os desenvolvedores e a construção iterativa do programa. Basicamente, o projeto foi todo dividido em módulos, como evidenciado na seção 6.1 e pôde-se ter um desenvolvimento paralelo de muitos desses módulos. Para cada um desses módulos, houve o desenvolvimento iterativo, iniciando-se o desenvolvimento sempre com uma versão mais simples do módulo, e em seguida, passando a um refinamento mais cuidadoso para melhoria de funcionalidade e robustez.

Um outro ponto importante do RUP é a integração constante entre os módulos, para validação da arquitetura como um todo, que foi conseguida tão logo se obteve uma versão funcional de cada módulo. Em alguns casos até mesmo se colocou uma solução provisória, com abordagens inocentes do problema, por exemplo, para o módulo separador de tokens, em que se colocou no lugar do módulo definitivo um algoritmo que simplesmente dividia a sentença nos espaços que separam cada palavra. Evidentemente, tal abordagem é inocente no sentido de que esse módulo falso não considera sinais de pontuação, como vírgulas e pontos, sinais estes que adicionam a complexidade ao processo de tokenização. Assim, a integração pode ser feita sem grandes prejuízos e tão logo se puderam detectar falhas estruturais no programa.

4.1 Divisão de tarefas

Com a separação de módulos funcionais, pôde-se dividir o projeto em algumas áreas principais e a conseqüente atribuição de responsabilidades das tarefas aos membros do projeto. Em geral, as responsabilidades eram principalmente do membro ao qual a tarefa foi designada, mas na prática, houve sempre a ajuda de um ou mais membros da equipe para a execução de cada tarefa.

Ao Diogo, foram incumbidas as tarefas de projeto de implementação de testes para os módulos funcionais que compõem o corretor gramatical e para esta tarefa contou com a ajuda de William, que possuía experiência adquirida com o projeto da versão 1.0 do CoGrOO. Ficou responsável pela implementação da classe *Tag*. Também executou tarefas referentes ao módulo que sucede a separação de tokens e antecede a etiquetação morfológica, chamado de pré-etiquetação.

Para o Fábio, foram delegadas as tarefas de projeto e implementação dos módulos

agrupador e analisador sintático simples. Para estas tarefas teve ajuda de William, que foi o membro responsável pelo aprendizado do OpenNLP.

Marcelo foi designado para a elaboração da parte de aplicação de regras gramaticais do corretor. Esta envolvia o porte do formato antigo das regras, legado do CoGrOO 1.0, e o planejamento e implementação da estrutura das regras. Também englobava a tarefa de elaborar o construtor de árvores e o aplicador das regras. Recebeu ajuda do William no projeto da estrutura das regras e do Fábio na implementação dos construtores de árvores e aplicadores de regras.

Por fim, o William foi incumbido da tarefa de estudar a viabilidade de utilização do OpenNLP e Maxent na implementação dos módulos componentes do corretor. Após a confirmação da viabilidade, foi encarregado de estudar o uso do OpenNLP, além de estendê-lo para suprir as necessidades do grupo. Construiu os módulos de separação de tokens e etiquetador morfológico, além de transmitir o conhecimento adquirido em seus estudos de OpenNLP.

4.2 Cronograma

O cronograma foi estabelecido se levando em conta que haveriam reuniões semanais de acompanhamento. Nessas reuniões seriam exibidos aos orientadores os avanços do projeto e se necessário redirecionar os avanços de acordo com os resultados. Também foi importante na elaboração do cronograma as datas alvo em que se deveria entregar algum relatório ou elaborar alguma apresentação para avaliação da disciplina Laboratório de Projeto de Formatura II.

Vide cronograma nos apêndices (pág.127).

5 TECNOLOGIA

Decidiu-se desenvolver este projeto em linguagem de programação *Java 5.0,* utilizando como ambiente de desenvolvimento (IDE) principal o *Eclipse 3.2;* para facilitar o controle dos códigos do projeto, foi utilizada a tecnologia de controle de versões, comumente chamada de *CVS.*

Todos estes softwares podem ser instalados em Linux ou Windows e são gratuitos; no presente momento, pode-se dizer que também são softwares livres (à exceção do ambiente de programação Java; contudo a Sun comprometeu-se a licenciar todas as ferramentas da tecnologia Java como software livre [SUN OPENS JAVA]). Estes podem ser baixadas dos endereços: *http://java.sun.com/,* http://www.eclipse.org/ e *http://www.cvsnt.org.*

A seguir, encontra-se um detalhamento sobre as linguagens, os padrões, os protocolos, as ferramentas e as bibliotecas que foram usadas para o projeto e o desenvolvimento do CoGrOO 2.0.

5.1 Linguagens, padrões e protocolos

5.1.1 Java (versão 5.0)

Java é uma linguagem de programação orientada a objetos que é executada sobre uma máquina virtual, desta forma é possível desenvolver aplicações Java para diversos tipos de hardwares e sistemas operacionais, bastando apenas que exista uma máquina virtual

desenvolvida para esses sistemas. As principais características de Java é sua portabilidade, recursos de rede, segurança e, como foi citada anteriormente, a orientação a objetos.

Dentre as muitas linguagens existentes atualmente Java foi a escolhida pelo grupo por diversas razões, tais como:

- Simplicidade: Comparando Java a C++, por exemplo, podemos dizer que ambas as linguagens são muito simples, entretanto algumas diferenças tornam a primeira mais simples, pois comparada com outras linguagens bem divulgadas e utilizadas pode-se dizer que Java é simples, pois não possui sobrecarga de operadores, *structs*, *unions*, aritmética de ponteiros, herança múltipla, diretivas de pré-processamento e a memória alocada dinamicamente não precisa ser gerenciada pelo desenvolvedor.
- Orientação a Objetos: Este é um ponto forte que possui em relação às outras linguagens bastante difundidas, o desenvolvimento em Java é orientado a objetos e isto significa que existe uma organização do projeto em classes.
- Execução de *Multithread*: Em Java existe a possibilidade de programar sistemas que executam diferentes funções ao mesmo tempo nas máquinas em que estão instalados. Desta maneira é possível utilizar todos os benefícios que os sistemas operacionais atuais oferecem ao programador e alcançar rendimentos muito maiores para os sistemas.
- Controle de Exceções: Atualmente o tratamento das exceções geradas pelo sistema tornaram-se muito importantes, pois a interação com o usuário a cada dia deve ser mais transparente. Antigamente quando uma exceção acontecia o computador parava de executar as tarefas que estava executando e o usuário era obrigado a reiniciar a máquina, o controle de exceções visa eliminar este desgaste do cliente.

- *Garbage Collector*: Linguagens mais antigas e famosas, como por exemplo o C++, atribuíam a tarefa de limpeza das informações geradas pelo programa ao programador, portanto não eram poucos os programadores que implementavam códigos em que a limpeza não era realizada, ocupando mais espaço em memória e diminuindo a performance da máquina do cliente. Java trouxe consigo a ferramenta *garbage collector*, esta ferramenta simplesmente faz a limpeza de informações inúteis sem que o desenvolvedor precise se preocupar com isso, desta forma a performance do programa aumenta e o desenvolvimento torna-se mais fácil de ser realizado. [INFOWESTER, 2003]

Devido a todos estes pontos positivos existentes no Java o grupo optou por utilizá-la no desenvolvimento da segunda versão do corretor gramatical.

5.1.2 XML Schema Definition

O *XML Schema Definition* também é conhecido como XSD e foi proposto inicialmente pela Microsoft, tornando-se uma recomendação oficial do W3C apenas em Maio de 2001.

Ele foi criado para auxiliar na escrita de arquivos XML, onde é utilizado como um *template* e o conteúdo do XML deve seguir o padrão definido por este arquivo. Portanto o XSD define blocos de construção para o arquivo XML.

O que um XSD define [DICAS-L, 2005]:

- elementos que podem aparecer em um documento

- atributos que podem aparecer em um documento

- que elementos são elementos filhos
- a ordem dos elementos filhos
- o número de elementos filhos
- se um elemento é vazio ou pode incluir texto
- tipos de dados para elementos e atributos
- valores padrão e fixos para elementos e atributos

XSD foi utilizado na seção de regras do projeto para que um padrão fosse criado e outros programadores pudessem criar novas regras de uma maneira fácil e que fossem entendidas corretamente pelo CoGrOO.

5.1.3 XMLRPC

O *XMLRPC* é um protocolo de chamada de procedimento remoto, para ser usado em chamadas sobre a Internet. Este protocolo é simples e é codificado em XML.

A implementação deste protocolo é a transformação de dados para arquivos XML que são enviados do cliente para o servidor, ou vice-versa, e decodificados para dados novamente [XMLRPC, 1998].

5.2 Ferramentas e bibliotecas

5.2.1 Eclipse 3.2

O *Eclipse* é uma ferramenta cuja funcionalidade não é somente para o

desenvolvimento de códigos Java, mas também para visualização de base de dados, diagramas *UML*, arquivos *XSD*, edição de arquivos *XML*, *JSP*, *HTML* e criação de arquivos em diversas outras linguagens como *PHP*, *C*, *Pascal* e muitas outras funcionalidades. Ele possui como principal atrativo a personalização do ambiente de desenvolvimento através de *plugins*, que permitem a customização de sua interface, dependendo das necessidades de cada projeto. Por exemplo, para *PHP* existe um *plugin* chamado *PHPEclipse,* que possibilita a visualização de código direto em um *browser* interno no Eclipse. Estes *plugins* são normalmente desenvolvidos pela comunidade de código aberto.

5.2.2 CVS

O *CVS* (*Concurrent Versions System*) é um sistema que auxilia no controle de versões dos arquivos de um projeto. Seu uso permite que diversas versões de um projeto sejam armazenadas em um repositório de dados, armazenando apenas as diferenças entre as versões e os arquivos comuns, permitindo assim gerenciamento eficiente dos arquivos gerados para o sistema. O *CVS* também possibilita que diversos desenvolvedores trabalhem em conjunto em um projeto de forma harmônica, sendo que cada um possui sua própria área de trabalho e o gerenciamento das alterações de cada programador é feito por uma interface com o intuito de evitar conflitos entre os arquivos e perda de código.

5.2.3 Altova XMLSpy

Para a edição de arquivos *XML Schema* usamos o software *Altova XMLSpy*, versão de avaliação. Ele possui uma interessante interface gráfica para criação visual de *XML Schema*, tornando o trabalho de criação desse tipo de arquivo relativamente simples, deixando o projetista livre para criar o *design* de acordo com as necessidades sem conhecer a fundo a

sintaxe XSD. Foi usado no projeto de regras o *XML Schema*.

5.2.4 JUnit

JUnit é uma ferramenta existente, atualmente, no Java para criação de testes. Ela é um *framework open source*, e portanto existe a possibilidade de alterar seu código para as necessidades do seu projeto.

Um ponto positivo do *JUnit*, e que pesou na decisão do grupo, é a integração que existe entre ele e a IDE Eclipse, além do fato de alguns membros do grupo já conhecem a ferramenta *JUnit* e por isso não seria preciso gastar tempo em pesquisas para desenvolver os testes.

Por estas razões o grupo decidiu criar testes utilizando esta tecnologia e foi possível determinar erros existentes em algumas classes e arrumá-los antes de enviar as correções para os outros integrantes do projeto.

5.2.5 OpenNLP

O *OpenNLP* é um conjunto de bibliotecas, codificadas em Java, sobre linguagem natural, e que são muito utilizadas por projetos *open source*. Estas bibliotecas visam dar uma idéia da estrutura básica para pesquisadores e desenvolvedores colaborarem em projetos de código livre sobre linguagens naturais. [OPENNLP, 2006]

A descoberta deste pacote durante a execução do projeto foi de suma importância para o sucesso do grupo em realizar as tarefas dentro do cronograma estimado. Os módulos implementados pelo corretor gramatical foram os do etiquetador, do *Chunker*, do *Tokenizer*, do *NameFinder* e do *ShallowParser*. A única dificuldade encontrada pelo grupo foi a escassa documentação existente para implementar cada módulo. Um subproduto do projeto é a

documentação, no próprio código, dos módulos implementados, permitindo que mais projetos utilizem este ótimo *framework*.

5.2.6 Jude

Jude, que significa *Java and UML Developers' Environment*, é um editor para Modelagem de Dados (UML) criada com Java e de uso fácil e intuitivo. Com o *Jude* é possível realizar uma modelagem de dados complexa e apresentar os dados para o usuário de forma clara. Uma das vantagens deste aplicativo é seu *layout* ser bem intuitivo. O *Jude* possui à sua esquerda uma Árvore de Estrutura com todos os Dados à disposição do usuário para se criar diagramas, mapas etc.

A adoção dele se deve ao fato de ser independente de sistema operacional pois foi construído em Java e dentre as opções gratuitas foi a melhor encontrada.

Os diagramas aos quais o *Jude* possui suporte são:

- Classes
- Casos de Uso
- Seqüência
- Colaboração
- Estados
- Atividades
- Componentes

- Implantação

- Geração de modelos de códigos-fonte Java

- Importação de códigos-fonte Java

- Geração automática de diagramas de classe

6 PROJETO

O CoGrOO é um aplicativo cuja arquitetura é relativamente simples, que em termos lógicos pode ser dividida nas seguintes camadas:

- camada de interface com o usuário / interface com o OpenOffice.org: onde estão contidas as classes que promovem a ligação entre o OpenOffice.org e o usuário com a camada de negócios.
- camada de negócios: onde se localizam os módulos de análise e verificação dos textos. Representa o núcleo do programa.
- camada de dados: são os modelos e dicionários

6.1 *Arquitetura de* software

O CoGrOO foi implementado seguindo-se a arquitetura mostrada na Figura 2.

Nesta figura estão representados os módulos básicos para o processamento do texto desde quando é inserido pelo usuário até o retorno com os erros presentes nesse texto. São eles:

- Separador de *tokens*: cada sentença é dividida em seus respectivos *tokens*.
- Etiquetador morfológico: responsável pela análise morfológica, atribui a cada *token*

uma etiqueta, a qual identifica classe, número, gênero, etc. Para tal, são feitos cálculos estatísticos, cujos parâmetros foram obtidos de um corpus já etiquetado.

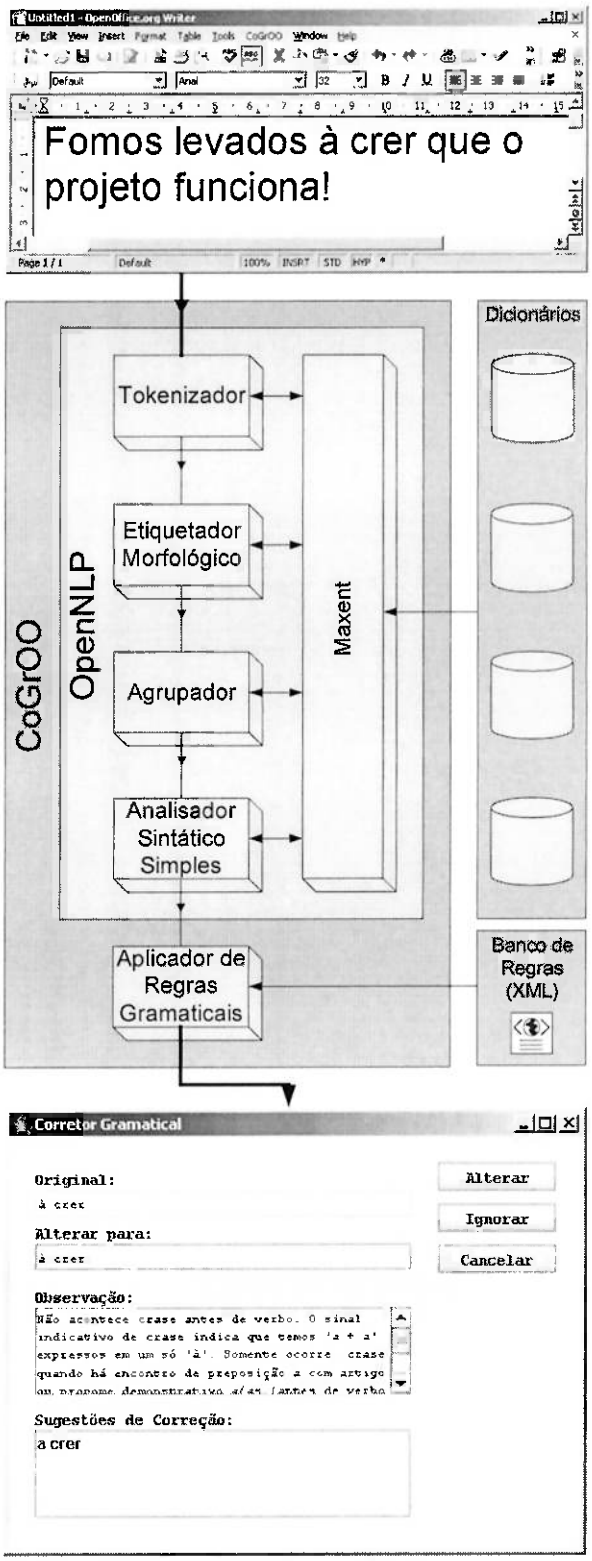


Figura 2 - Arquitetura

- Agrupador: trata-se de um identificador de sintagmas; os padrões (traduzidos em seqüências de etiquetas morfológicas) são aqueles presentes no corpus, mas apenas os que apresentam taxa de acerto acima de uma nota de corte definida arbitrariamente.

- Analisador sintático simples: identifica sujeito e verbo contidos em uma sentença; é semelhante ao agrupador quanto ao método - padrões formados por seqüências de etiquetas morfológicas obtidos de um corpus, filtrados por uma nota de corte.

- Detector de erros: finalmente, este módulo aplica as regras (sua estrutura e funcionamento será mostrado no item 7.4) de erro sobre as sentenças analisadas.

O diagrama de classes referente aos módulos do CoGrOO é mostrado na Figura 3.

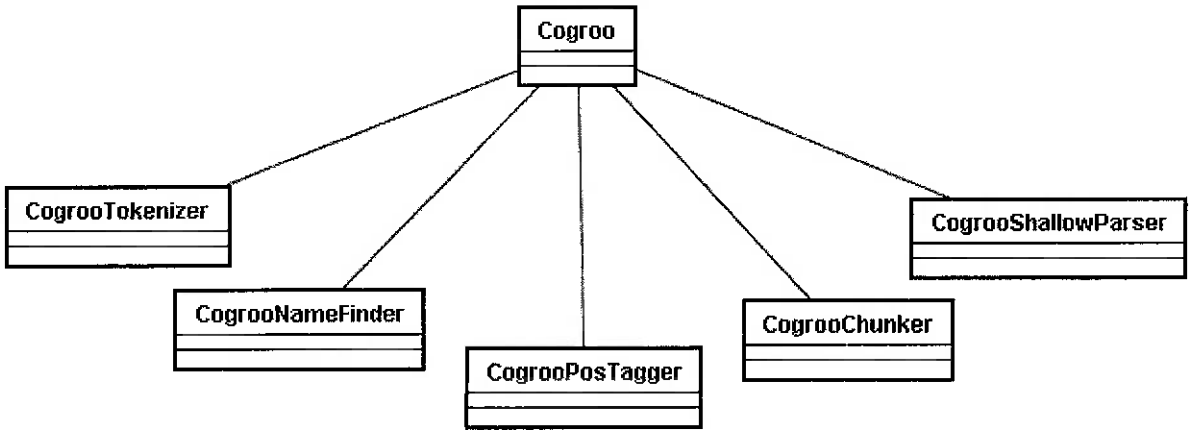


Figura 3 - Diagrama de classes dos módulos de análise de texto

6.2 *Especificação das classes de entidade*

Uma vez definida a arquitetura básica do sistema, é necessário detalhar a estrutura de dados interna do programa, cujas classes são especificadas a seguir:

- Corpus: uma classe que serve de interface entre o programa e o arquivo contendo sentenças para o levantamento das estatísticas.
    - Atributos:
        - contador de sentenças: o número de sentenças contidas no corpus
        - localização das sentenças: um vetor com o mesmo número de posições quanto o número de sentenças existentes no arquivo; o valor de cada elemento representa a posição de início de cada sentença dentro desse corpus
        - número da sentença atual: um contador interno das sentenças, cujo valor é atualizado conforme as sentenças são lidas
        - arquivo do corpus: uma referência ao arquivo "físico" do corpus; permite acesso aleatório a qualquer parte do corpus independentemente do momento do acesso
- Sentença
    - Atributos:
        - sentença: a sentença original inserida pelo usuário
        - tokens: a sentença dividida em um vetor de tokens
        - sintagmas: a sentença dividida em um vetor de sintagmas

- Token
    - Atributos:
        - lexema: o próprio token escrito pelo usuário (palavra, pontuação, etc.)
        - primitiva: no caso das palavras, é aquela da qual se originou o lexema, que não apresenta flexões (de gênero, número, etc.)
        - etiqueta de agrupamento: provê informações se o token é início ou continuação de um sintagma, ou nenhum dos dois.
        - etiqueta sintática: indica se o token é núcleo de um sujeito, de uma locução verbal, ou nenhum dos dois
        - sintagma: indica a qual sintagma pertence o token
        - localização: indica início e fim de um token em uma sentença
- Sintagma
    - Atributos:
        - tokens: um vetor apontando para os tokens que formam o sintagma
        - primeiro token: informação sobre o índice do primeiro token deste sintagma na sentença
        - etiqueta morfológica: indica o tipo de sintagma, e suas flexões

(gênero, número, etc.)

- etiqueta sintática: indica se representa um sujeito ou um verbo

- Etiqueta
  - Atributos:
    - etiqueta: a etiqueta em formato de texto

Isso pode ser representado pelo diagrama de classes presente na Figura 4.

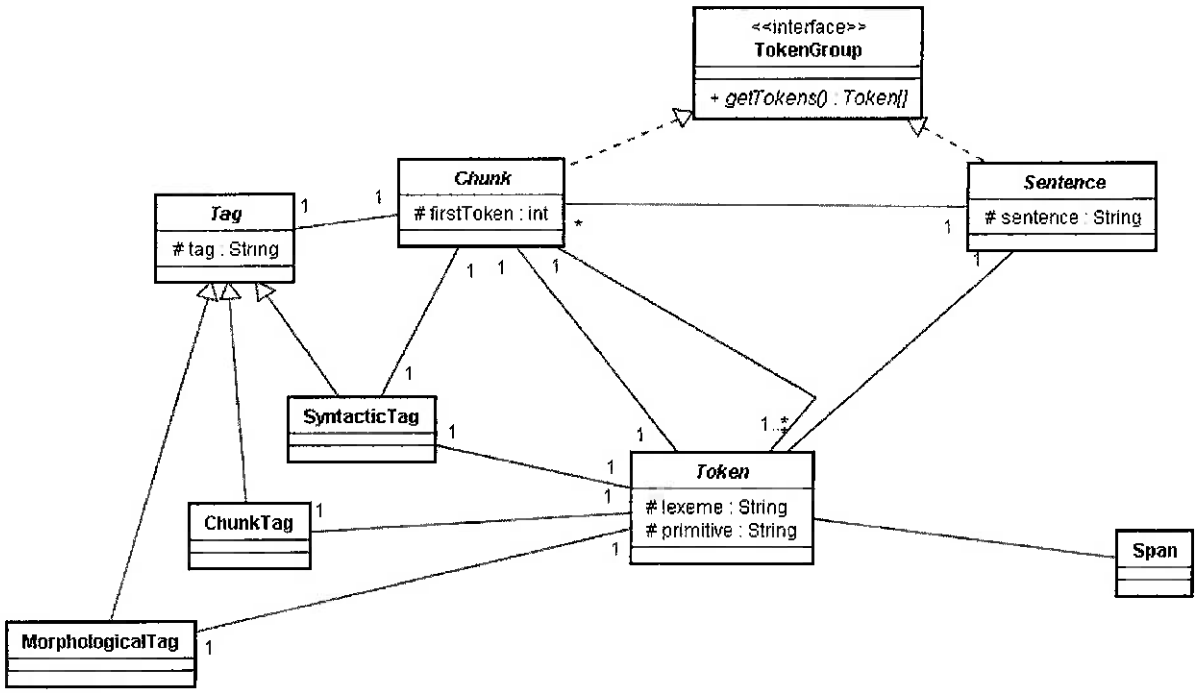


Figura 4 - Diagrama das classes de entidade no CoGrOO 2.0

6.3 Pacotes

Os pacotes foram divididos como mostra a Figura 5.

O pacote de integração (*integration*) representa a camada de interface com os usuários.

O pacote de verificação gramatical (*grammarchecker*) utiliza as ferramentas de processamento de linguagens naturais (presentes em tools) – isto é, o separador de sentenças, o separador de *tokens*, o etiquetador morfológico, o identificador de sintagmas e o analisador sintático simples -, junto com estas últimas, representa o núcleo do programa. Esse núcleo também utiliza as classes presentes nos outros pacotes, que representam a estrutura de dados interna utilizada (entidades básicas como Sentença, Etiqueta Morfológica, etc., e os fluxos de dados provenientes do *corpus*). Já a camada de dados, não representada na figura, são arquivos externos utilizados para o armazenamento das estatísticas da língua portuguesa, das regras de verificação gramatical, etc.

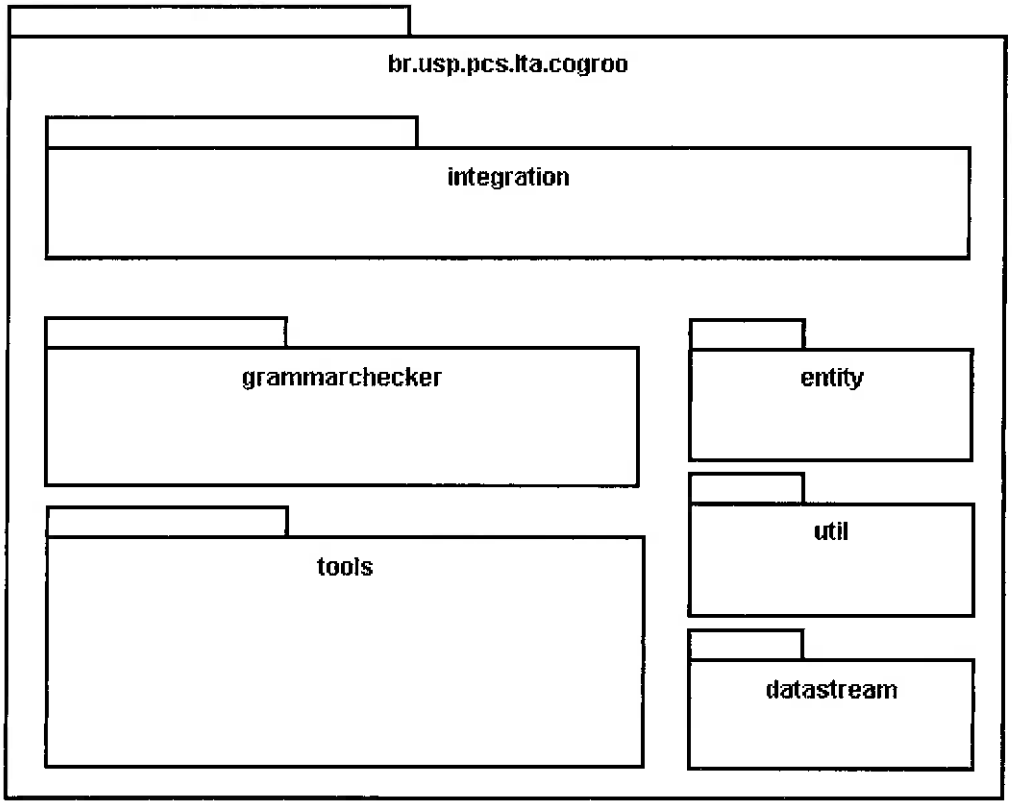


Figura 5 - Pacotes principais do CoGrOO 2.0

6.4 *Diagramas de seqüência*

6.4.1 Treinamento

Foi feito o diagrama de seqüência do treinamento do Agrupador, pois este é o mais básico implementado para o aprendizado do CoGrOO (Figura 6). O treinamento dos outros módulos é praticamente o mesmo, embora haja geradores de contexto e fluxos de dados e eventos específicos para cada um.

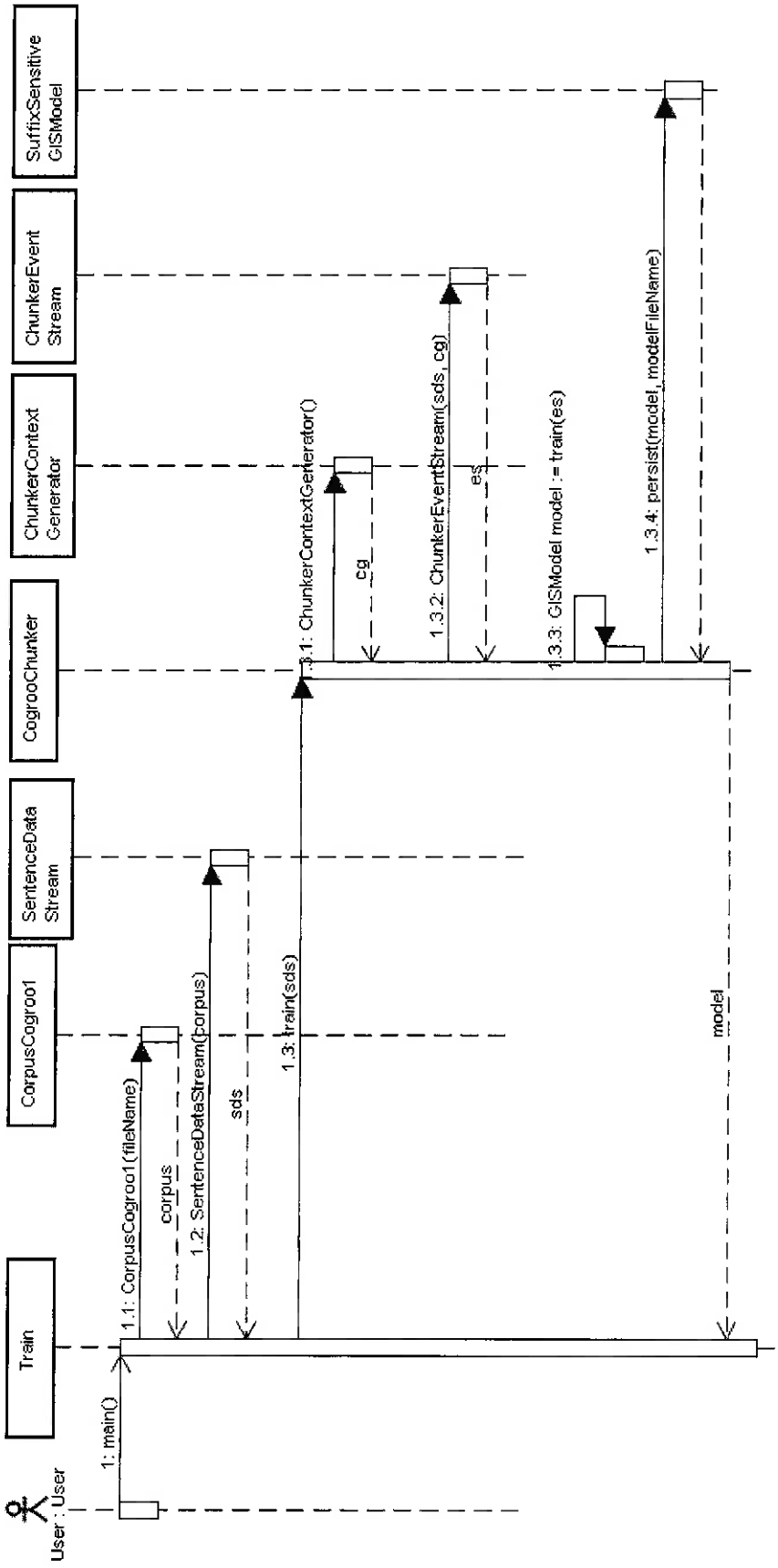


Figura 6 - Diagrama de seqüência de treinamento do chunker

6.4.2 Execução

Foi feito também o diagrama de seqüência da execução do CoGrOO propriamente dita (Figura 7). Nota-se que há certa diferença entre a implementação do programa e a arquitetura inicialmente projetada, pois a classe Cogroo faz a intermediação da comunicação entre os módulos, em vez de haver uma comunicação direta entre os módulos.

6.5 Modelo de integração com o OpenOffice.org

A comunicação entre o OpenOffice.org e o CoGrOO se dá através de um serviço disponibilizado na rede, seguindo um modelo de cliente-servidor.

As requisições de correção de desvios gramaticais em uma sentença são executadas pela interface gráfica do CoGrOO acoplada ao OpenOffice.org, passando-se para o servidor apenas a sentença a ser corrigida.

Esse modelo de comunicação impõe um baixo acoplamento entre o cliente, o OpenOffice.org, e o servidor, que é o CoGrOO propriamente dito, que faz o serviço de correção gramatical.

Outros programas, em tese, poderiam se comunicar da mesma forma que o OpenOffice.org se comunica com o servidor do CoGrOO e, desse modo, também poderiam se beneficiar do serviço de correção gramatical. Esse nível de flexibilidade é atingido graças à padronização de interfaces e o modelo cliente-servidor utilizado.

Maiores detalhes da implementação da arquitetura de comunicação serão expostos na seção 7.6.

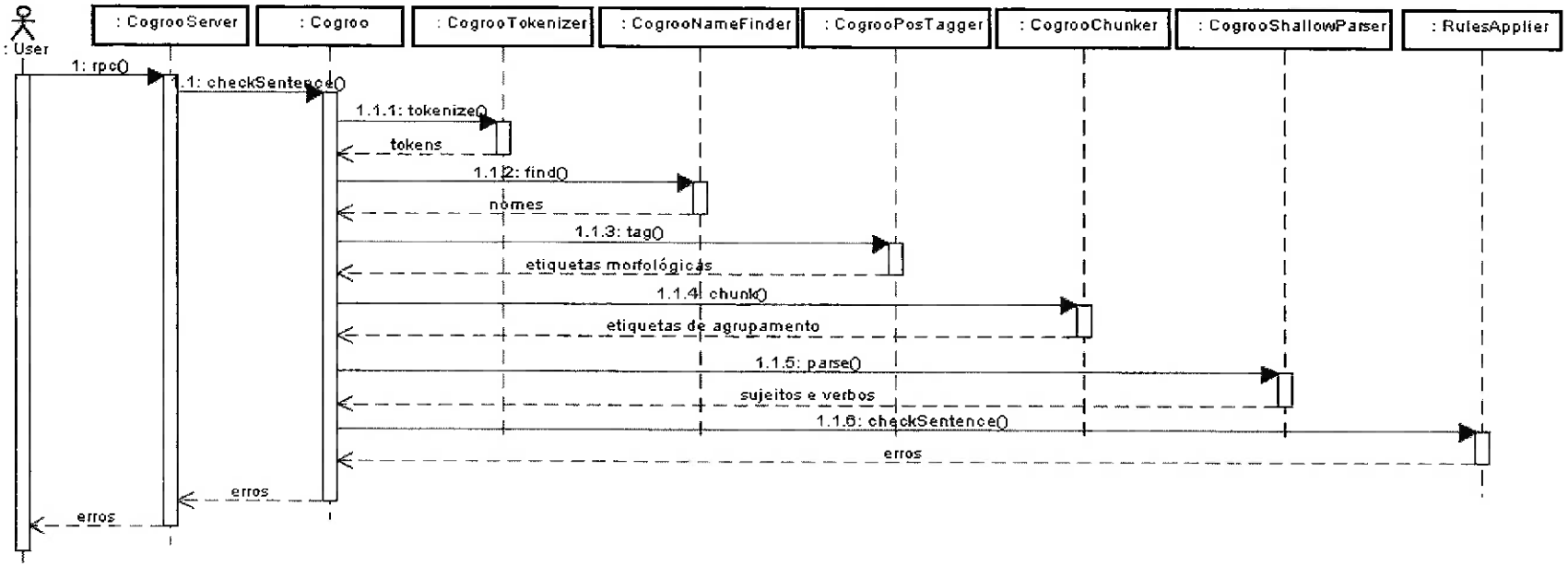


Figura 7 - Diagrama de seqüência de execução

7 IMPLEMENTAÇÃO

Algumas implementações que merecem destaque estão nos itens a seguir. Entre eles o driver de acesso a corpora e a dicionários, os DataStream's, e a classe Sentence. Também é descrito como o OpenNLP foi modificado para atender os requisitos.

7.1 Infra-estrutura de acesso ao Corpus

Corpus lingüístico é um arquivo que obedece a algum formato (*Constraint Grammar*, *VISL*, *Tree Bank*, *CoGrOO 1.0 Constraint Grammar*). *Corpora* úteis para o CoGrOO seriam os constituídos de sentenças extraídas de textos. Estas sentenças devem estar separadas em *tokens* e cada *token* ter sua anotação morfológica. Cada *token* também deve ter uma anotação acerca de seu papel sintático na sentença.

Alguns problemas:

- Diversos formatos de arquivo e não existem *parsers* estabelecidos,
- Durante os treinamentos/testes são feitos acessos intensos ao *Corpus*; dado seu tamanho, esse acesso pode se tornar muito lento, principalmente para buscar uma sentença especifica.

7.1.1 Classe abstrata Corpus

No CoGrOO 2.0 definimos uma classe abstrata a qual demos o nome *Corpus* (pacote

br.pcs.usp.lta.cogroo).

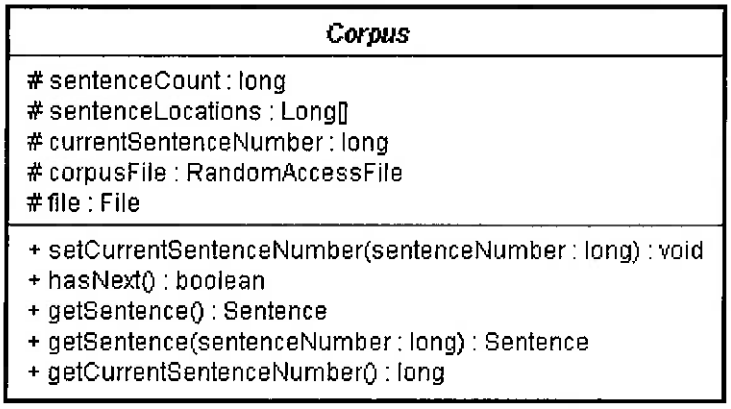


Figura 8 - Classe abstrata *Corpus*

Classes que são estendidas desta abstrata são obrigadas a de alguma forma completar os seguintes campos:

- *sentenceCount: long*
- *sentenceLocations: Long[]*
- *currentSentenceNumber: long*
- *corpusFile: RandomAccessFile*
- *file: File*

Para isso, o construtor deve ler todo o arquivo do corpus, guardando todas as posições do disco no qual se inicia uma sentença. Para isso se usa o *RandomAccessFile*, e seus métodos *getFilePointer()* e *seek()*.

Com isso é possível rapidamente localizar qualquer sentença do corpus com apenas o índice dessa sentença.

Veja o exemplo, supondo que *sentenceLocations* já foi populado:

```
long sentencePosition = sentenceLocation[10];
// sentencePosition tem a posição da sentença 10
System.out.println(sentencePosition);
// imprime 45128, por exemplo,
corpusFile.seek(sentencePosition);
String line = corpusFile.readLine();
System.out.println(sentence);
// imprime o conteúdo da linha que se inicia em 45128
```

As informações *sentenceLocations* são persistidas em disco de forma que na próxima vez que se instancie a classe *Corpus* para um *corpus* já lido não seja necessário fazer todo este processamento novamente.

A classe abstrata *Corpus* também obriga que sejam implementados métodos para se obter uma sentença especifica, iteradores sobre a seqüência de sentenças etc.

7.1.2 Classe CorpusCogroo1

Esta classe estende (*extends*) a classe abstrata Corpus. Ela especifica a classe abstrata para o Corpus com o formato *CoGrOO 1.0 Constraint Grammar*.

Nesse corpus temos uma sentença por linha no seguinte formato:

```
Romário:{Romário}_[PROP_M/F_S_]_@SUBJ> não:{não}_[ADV_]_@ADVL>
se:{se}_[PERS_M/F_3S_ACC_]_@ACC>-PASS
exercitou:{exercitar}_<fmc>_[V_PS_3S_IND_VFIN_]_@FMV
em:{em}_<sam->_[PRP_]_@<ADVL as:{o}_<-
sam>_<artd>_[DET_F_P_]_@>N cobranças:{cobrança}_[N_F_P_]_@P<
de:{de}_[PRP_]_@N< falta:{falta}_[N_F_S_]_@P< e:{e}_<co-
prparg>_[KC_]_@CO pênalti:{pênalti}_[N_M_S_]_@P< $.:[-PNT_ABS]
```

Sintaxe:

```
termo:{primitiva}_<auxiliar>[etiqueta morfológica]_etiqueta sintática
```

A classe *CorpusCogroo1* portanto se encarrega de interpretar este formato de corpus e

instanciar corretamente a classe *Sentence*, tornando o acesso ao corpus transparente para o resto do sistema.

7.1.3 DataStreams de acesso ao corpus

O *Maxent* prevê que os dados de entrada para treinamento implementem a seguinte interface:

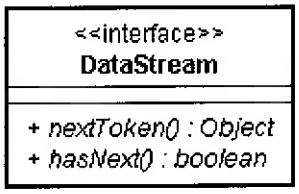


Figura 9 - Interface *DataStream*

Portanto foi necessário criar classes que implementam esta interface para as mais diversas finalidades.

7.1.3.1 SentenceDataStream

Este *DataStream* é implementado de tal forma que o *nextToken: Object* retorne um objeto do tipo *Sentence*.

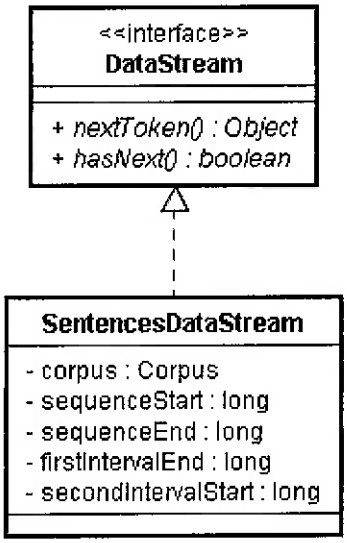


Figura 10 - Classe *SentencesDataStream*

Um *SentencesDataStream* é construído a partir de um objeto do tipo Corpus.

Os campos *sequenceStart* e *sequenceEnd* definem o intervalo que o *DataStream* pode iterar nas sentenças do Corpus. Os campos *firstIntervalEnd* e *secondIntervalStart* são especialmente úteis para os testes, quando se deseja pular um trecho do corpus para mais tarde usar este trecho para comparação.

7.1.3.2 PlainTextByLineDataStream e NameFindDataStream

Similarmente foram implementados o *PlainTextByLineDataStream*, que no lugar de retornar *nextToken* retornar *Sentence*, retorna uma *String* representando a sentença em texto simples.

O *NameFindDataStream* coloca marcações no início e fim de seqüências de nomes próprios. É útil quando se faz o treinamento do módulo *NameFinder*.

7.2 Infra-estrutura de acesso aos dicionários

O *Maxent* prevê dois tipos de dicionários. *Dictionary* (*opennlp.tools.ngram*) e *TagDictionary* (*opennlp.tools.postag*).

Dictionary é basicamente uma lista de palavras na qual se pode consultar se esta palavra existe ou não. Existe uma *MutableDictionary*, que estende *Dictionary* com métodos para criar dicionários persistentes.

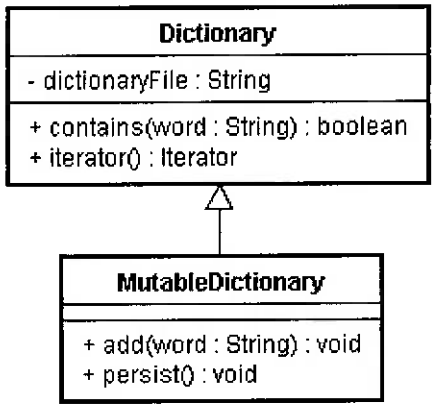


Figura 11 - Classes *Dictionary* e *MutableDictionary*

TagDictionary é uma interface com a seguinte assinatura:

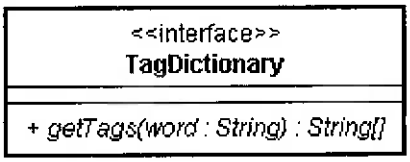


Figura 12 - Interface *TagDictionary*

A classe *TagDictionaryWriter* implementa uma forma de persistir *TagDictionary* em disco.

No entanto estes dicionários são muito fracos para o CoGrOO. Portanto estendemos esta interface para a da seguinte forma:

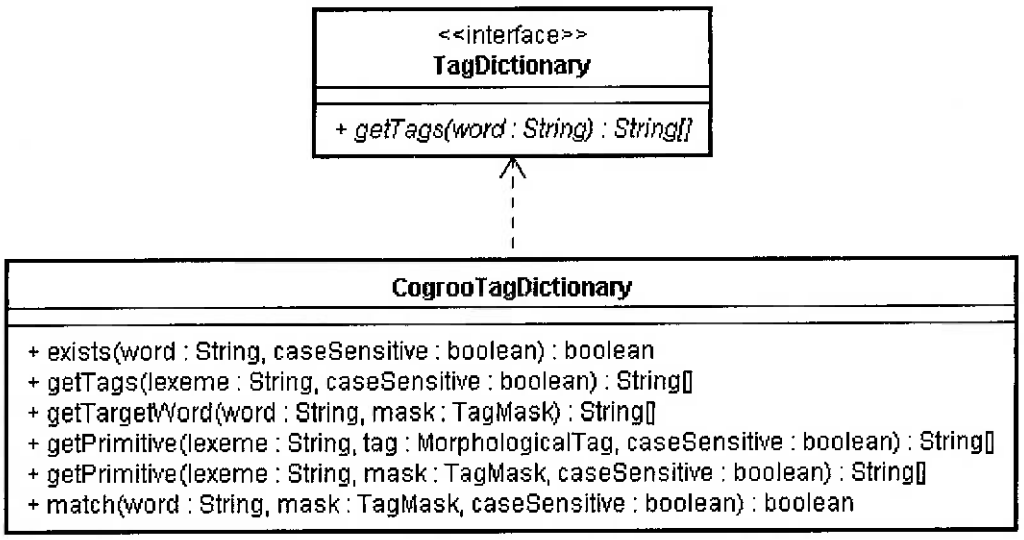


Figura 13 - Classe *CogrooTagDictionary*

Este *CogrooTagDictionary* se faz útil em diversas etapas do processamento do corretor gramatical.

Quase todos os métodos dessa implementação possuem como argumento um *caseSensitive: boolean*, útil para fazer buscas levando em consideração se a busca é sensível deve diferenciar entre letras maiúsculas e minúsculas.

- *exists(word : String, caseSensitive : boolean) : Boolean*

  Se word existir no dicionário, retorna verdadeiro, se não, falso.

- *getTags(lexeme : String, caseSensitive : boolean) : String[]*

  Busca no dicionário as possíveis etiquetas para *lexeme*.

- *getPrimitive(lexeme : String, tag : MorphologicalTag, caseSensitive : boolean) : String[]*

  Busca a primitiva de *lexeme*, dada a etiqueta morfológica de *lexeme*.

- *getPrimitive(lexeme : String, mask : TagMask, caseSensitive : boolean) : String[]*

  Busca as primitivas de lexeme, dado a mascara da etiqueta morfológica e lexeme.

- *getTargetWord(primitive : String, mask : TagMask) : String[]*

  Dada uma primitiva e uma máscara de etiqueta morfológica, retorna a palavra associada. Exemplo: *primitive*: "menino", *mask*: "female plural" retorna "meninas".

- *match(word : String, mask : TagMask, caseSensitive : boolean) : boolean*

  Verdadeiro caso exista uma combinação *word* e *mask*, falso caso não exista.

7.3 Modificações no OpenNLP

Foram feitas diversas modificações no *OpenNLP*. Podemos dividir estas modificações entre as que alteraram a estrutura de treinamento, as que alteraram as *features* de cada módulo, as que alteraram os dados usados para o processamento das *features* de cada módulo etc.

7.3.1 Automação do treinamento

Originalmente o treinamento dos módulos do OpenNLP era feito individualmente pela linha de comando.

Para agilizar o trabalho, e também tornar transparente o reuso do CoGrOO 2.0, foram desenvolvidas classes de suporte ao treinamento.

Foram criadas as classes *TrainInput* e *TrainOutput*, com os dados de entrada e saída mais usuais dos treinamentos dos módulos. Para os módulos que precisam de dados que estas classes não possuem, foram implementadas outras classes que estendem estas duas.

7.3.1.1 TrainInput

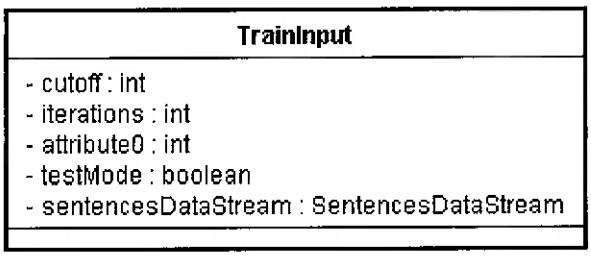


Figura 14 - Classe *TrainInput*

- *cutoff : int*

  Nota de corte de *features*. É parâmetro de treinamento do *Maxent*. Quanto maior, as *features* de maior relevância terão mais destaque e as fracas serão eliminadas.

- *iterations : int*

  Número de iterações na aproximação da função de entropia máxima. É parâmetro de treinamento do *Maxent*, quanto menor, mais rápido o treinamento, mas menos preciso o modelo.

- *testMode : boolean*

  Se é modo teste ou não. Caso seja, os modelos serão gerados e devolvidos no *TrainOutput*, mas não serão persistidos. Caso não seja, os modelos serão persistidos.

- *sentencesDataStream : SentencesDataStream*

  O *DataStream* de onde tirar os dados de treinamento. A partir do *SentencesDataStream* é possível obter qualquer outro tipo de *DataStream* usado no CoGrOO 2.0.

7.3.1.2 TrainOutput

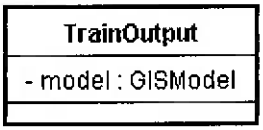


Figura 15 - Classe *TrainOutput*

- *model : GISModel*

  O modelo do *Maxent* criado no treinamento.

7.3.1.3 Extensões do TrainInput

- *DicTrainInput*

  Inclui os nomes dos arquivos nos quais se encontram as listagens de dicionários. São três, abreviaturas, palavras etiquetadas morfologicamente e de primitivas com etiquetas morfológicas para associar a palavra.

- *TokenizerTrainInput*

  Inclui dois dicionários, o de palavras conhecidas e o de abreviaturas.

- *NameFinderTrainInput*

  Inclui o dicionário palavra etiquetadas, que é útil para verificar se um *token* pode ser classificado como nome próprio.

- *TaggerTrainInput*

  Inclui o dicionário palavra etiquetadas e o de palavras conhecidas.

7.3.1.4 Extensões do TrainOutput

- *DicTrainOutput*

  Inclui dois dicionários, de palavras conhecidas e de abreviaturas.

7.3.2 Estrutura de treinamento

Foi estabelecida uma classe que centraliza todas as funções de treinamento dos módulos. *Train* (*br.usp.pcs.lta.cogroo.training*).

Associada a essa classe foi criado um arquivo do tipo ".properties" onde é possível configurar como será o treinamento dos modelos e criação dos dicionários.

É possível configurar todo o treinamento usando "cogroo.properties": o *corpus* que será usado, o número de sentenças individualmente para cada módulo, se será usado um dicionário já persistido ou se será criado um novo, notas de corte e de iterações do *Maxent*, entre outros.

Para poupar tempo a arquitetura de treinamento prevê que dados de saída sejam reaproveitados na entrada do treinamento de outros módulos. São reaproveitadas as instâncias dos dicionários e do corpus, uma vez que o carregamento desses pode levar vários minutos.

7.3.2.1 Alterações dos dados de entrada

Cada um dos módulos do *OpenNLP* originalmente espera como entrada um arquivo texto com formatação especial.

Por exemplo, o detector de sentenças espera um arquivo texto, com certa codificação, que contenha em cada linha uma sentença. O tokenizador espera um texto, em que cada linha existe uma sentença, e na linha seguinte, diversos pares de índices, um par para cada *token*,

indicando onde começa e termina cada *token*. O etiquetador, uma sentença com os *tokens* separados por espaço seguido de anotações sintáticas.

Para evitar a elaboração desses diversos arquivos foi criada a classe *Corpus* e os diversos *DataStreams* que os utilizam.

Então todos os módulos de treinamento do *OpenNLP* tiveram que ser estendidos para suportar os novos *DataStreams*. Com isso foi notável o ganho na facilidade e transparência do treinamento.

7.3.3 Alterações de *features*

O *OpenNLP* vem com uma configuração de *features* padrão, que pode ser estendida de acordo com as necessidades.

Dadas as particularidades da língua portuguesa, e dos recursos de dicionários que dispomos, nos foi conveniente trabalhar nessas *features*.

7.3.3.1 *Separador de sentenças*

Foi adicionado o acesso a dicionários para acrescentar as seguintes *features*:

- Se a seqüência de caracteres, à direita ou à esquerda, do candidato a marca de fim de sentença está presente no dicionário;
- Se o candidato a marca de fim de sentença foi classificado, no dicionário, como tal.

7.3.3.2 *Separador de tokens*

Similarmente ao detector de sentenças, o dicionário ajudou acrescentando as seguintes *features:*

- Se a sequência de caracteres à direita ou à esquerda do candidato a delimitador de *token* está no dicionário;
- Se o conjunto existe no dicionário;
- Foi estendido para aceitar casos de hífen.

7.3.3.3 Agrupador de nomes próprios

Com os dicionários foi possível incluir a seguinte *feature*:

- Se cada componente da seqüência já foi classificado com nome próprio anteriormente;
- Se o conjunto já foi classificado como nome próprio anteriormente.

7.3.3.4 Etiquetador morfológico

Foram incluídas *features* para:

- Dar maior suporte a nomes próprios agrupados
- Vincular letras maiúsculas a nome próprio;
- Auxiliar etiquetagem de palavras compostas;
- Auxiliar etiquetagem de palavras com hífen.

7.3.3.5 *Agrupador e analisador sintático simples*

Não foi necessário adicionar *features* a estes módulos.

7.4 Regras

7.4.1 Metodologia geral de aplicação das regras

Os desvios gramaticais são identificados na sentença por meio da utilização de busca por padrões.

Os padrões das regras de desvios gramaticais são compostos de uma combinação de elementos, que indicam se determinado *token* da sentença pode ser identificado por uma máscara.

As máscaras podem ser de três tipos: máscara de lexema (*LexemeMask*), máscara de primitiva (*PrimitiveMask*) e máscara de etiqueta (*TagMask*). A máscara de lexema indica se o próprio texto da máscara ocorre numa palavra da sentença. A máscara de primitiva identifica um *token* cuja palavra que lhe dá origem é a mesma da máscara. A máscara de etiqueta identifica etiquetas morfológicas ou sintáticas das palavras ou grupos de palavras, como o gênero e número para um substantivo ou o tempo e a pessoa para um verbo.

Um exemplo típico de regra seria o seguinte:

"a" "a" verbo

Essa regra representa um "à" (a craseado) seguido de um verbo. A regra é representada por dois "a"'s normais porque o CoGrOO quebra contrações em suas palavras componentes. No caso, o "à" é representado por "a" "a".

Têm-se então três elementos nessa regra: o primeiro "a", que é um elemento composto

de uma máscara de lexema, o segundo "a", que representa o mesmo caso do "a" anterior (uma máscara de lexema) e o verbo, que representa uma máscara de etiqueta.

A partir de um *token* n arbitrário na sentença na qual se procuram os desvios gramaticais, é feita a combinação do *token* n com o "a". Caso haja a combinação do *token* n com a máscara "a", prossegue-se na busca e examina-se o *token* posterior ao *token* inicial examinado (*token* n + 1). Se o *token* n + 1 se combinar com a segunda máscara, examina-se mais um *token* (*token* n + 2). Se o *token* n + 2 for um verbo, segundo o processo de etiquetamento morfológico do CoGrOO, ocorre uma combinação e diz-se que essa regra foi encontrada na sentença, indicando que existe um desvio gramatical nos *tokens* n a n + 2. Note que se em qualquer dos *tokens* n a n + 2 a ocorrência dos padrões falhar, a regra não se aplica na sentença (se aplicada com base no *token* n – pode acontecer da regra ocorrer na sentença se aplicada a partir de outro *token* arbitrário diferente de n, seguindo o mesmo processo aqui descrito).

7.4.2 Formato das regras

O elemento principal da regra é o padrão que é procurado na sentença, como descrito na seção anterior, mas existem outros atributos de uma regra que também são importantes e serão apresentados a seguir.

Cada regra é composta dos seguintes atributos:

- Método: indica o escopo de atuação da regra, que determina se a regra pode ocorrer em qualquer local da sentença (geral), somente dentro de sintagmas (local-sintagma) ou entre sujeito e verbo.
- Tipo: indica qual o tipo de desvio gramatical detectado pela regra. Por exemplo,

regras de crase.

- Grupo: indica a que grupo de regras a regra pertence. Por exemplo, ocorrência de crase antes de verbo.
- Mensagem: uma mensagem explicativa sobre o desvio gramatical detectado pela regra.
- Padrão: o padrão de desvio gramatical propriamente dito, que é procurado na sentença.
- Fronteiras: indica o limite inferior e superior para a marcação do desvio contado em *tokens*. Por exemplo, o par de fronteiras da regra ["a" "a" verbo] é (0, -1), o que indica que deveriam ser marcados como erro o "a" "a", segundo a Tabela 4.

Tabela 4 - Exemplo de uso do atributo "fronteiras"

Fronteira	"a"	"a"	verbo
Inferior	0	1	2
Superior	-2	-1	0

Note que se a fronteira da regra fosse (0, 0), tanto o "a" craseado quanto o verbo seriam indicados como erro. Assim se podem controlar quais *tokens* devem ser indicados ao se marcar a ocorrência de desvio.

- Sugestões: são padrões de sugestões para correção do desvio gramatical.
- Exemplos: são pares de exemplos de sentenças inadequadas e adequadas, que são utilizadas principalmente para teste da regra, mas servem também como uma indicação na prática de que tipo de desvio gramatical a regra detecta.
- Histórico de modificações: indica o autor da modificação, a data da modificação e pode apresentar um breve comentário sobre a modificação feita na regra.

7.4.3 Estruturação das regras

O formato das regras é delineado por um arquivo esquema de XML (*XML Schema*), com o qual é possível definir uma estrutura para a descrição das regras propriamente ditas.

As regras são então escritas num arquivo XML com base na estrutura imposta no esquema. Diz-se então que o arquivo XML é uma implementação do esquema.

Com a utilização de um esquema XML, é possível automatizar o processo de entrada das regras no programa através de dois passos: a geração de classes em Java que representarão a estrutura das regras e seus atributos e pela facilitação do carregamento das regras no CoGrOO. No processo de carregamento ainda se tem a vantagem de se poder aplicar uma validação nas regras, através da combinação do esquema com o arquivo das regras, fazendo com que o sistema rejeite regras não-conformes com a especificação do CoGrOO e assim evitando um funcionamento errático e imprevisível na detecção dos desvios gramaticais.

Pelo fato da sintaxe de um esquema XML ser razoavelmente complexa, utilizou-se um programa denominado Altova XMLSpy 2006, versão de avaliação. O programa oferece inúmeras facilidades, como a definição do esquema de modo visual, evitando assim erros de sintaxe.

A estrutura definida no esquema é formada pelos atributos descritos na seção Formato das regras.

A edição da estrutura das regras pode ser vista na Figura 16 e a estrutura do padrão de desvio gramatical, na Figura 17.

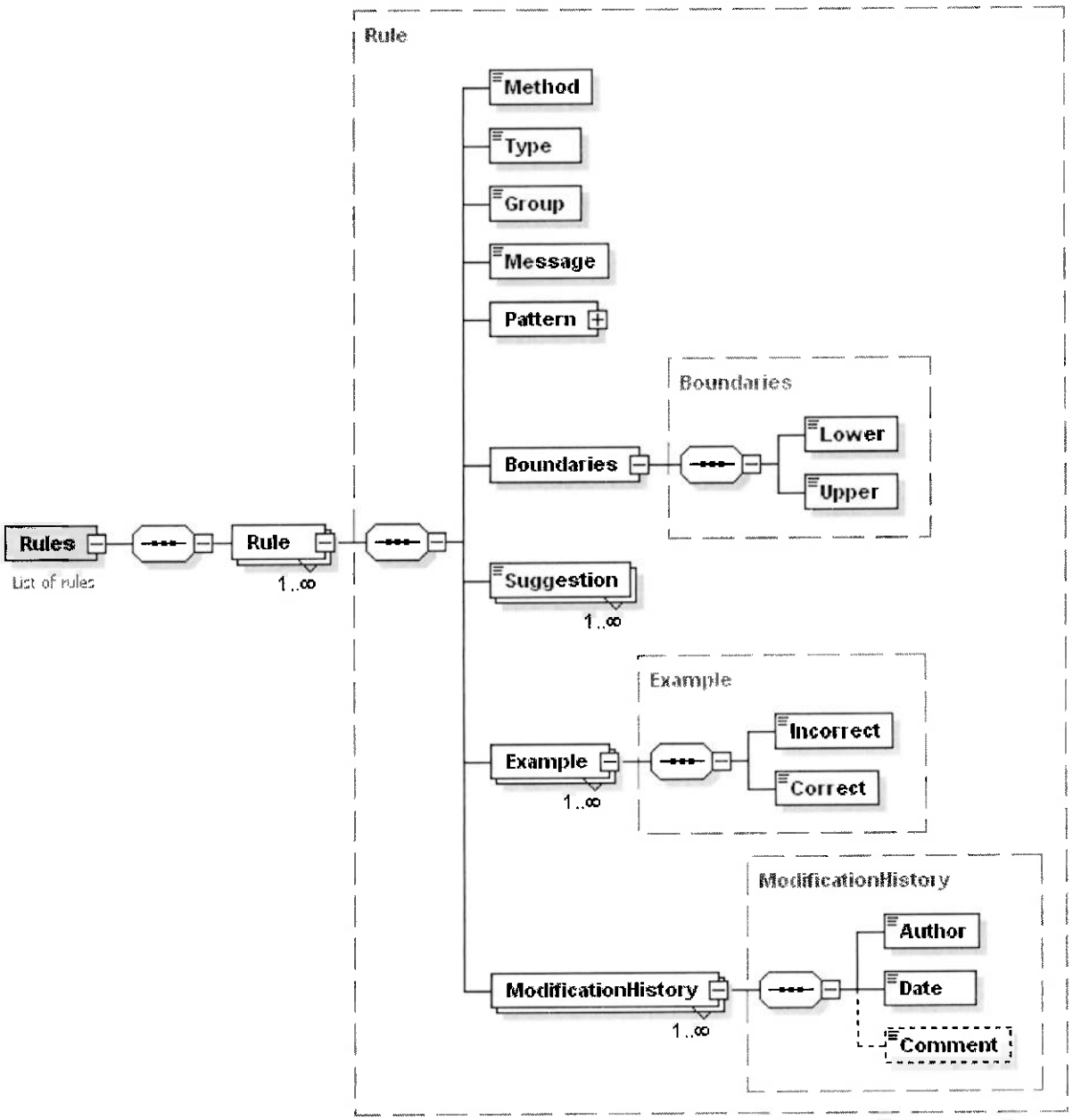


Figura 16 - Construção da estrutura das regras no programa Altova XMLSpy 2006

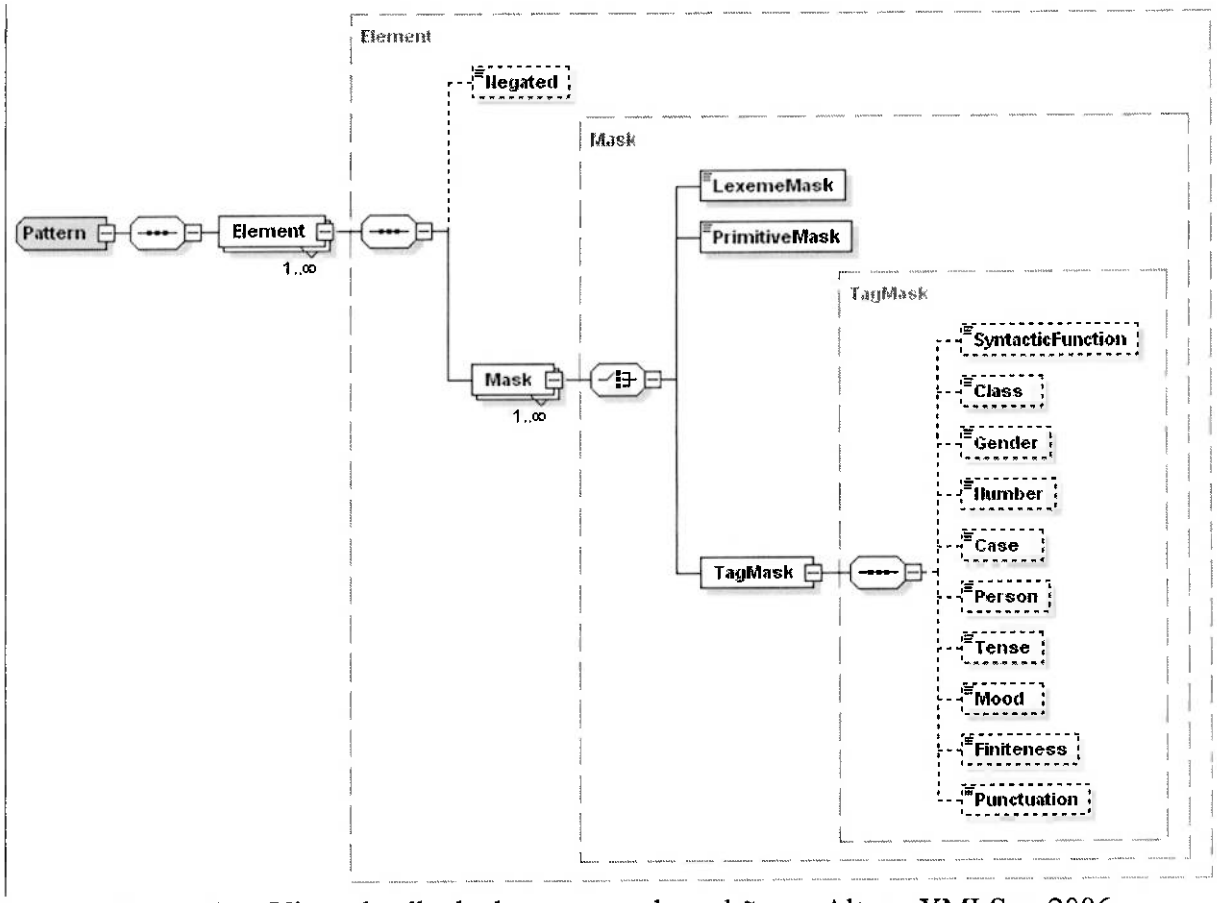


Figura 17 - Visão detalhada da estrutura do padrão no Altova XMLSpy 2006

As classes que receberão os atributos das regras escritas com base no esquema são então geradas automaticamente por uma tecnologia denominada de *XML Binding*.

O esquema é passado por um compilador de *XML Binding*, que interpreta as etiquetas nele contido e é capaz de gerar classes que representam os atributos das regras. Desse modo, são geradas algumas classes Java, como a classe *Pattern*, que contém os atributos que representam o padrão da regra e a classe *Example*, que contém o par de sentenças inadequada e adequada, entre outras.

Os processos descritos anteriormente, de definição do esquema, a escrita do arquivo de implementação do esquema e a compilação das classes, constituem o tempo de compilação do

processo de aplicação de regras, indicado na Figura 18.

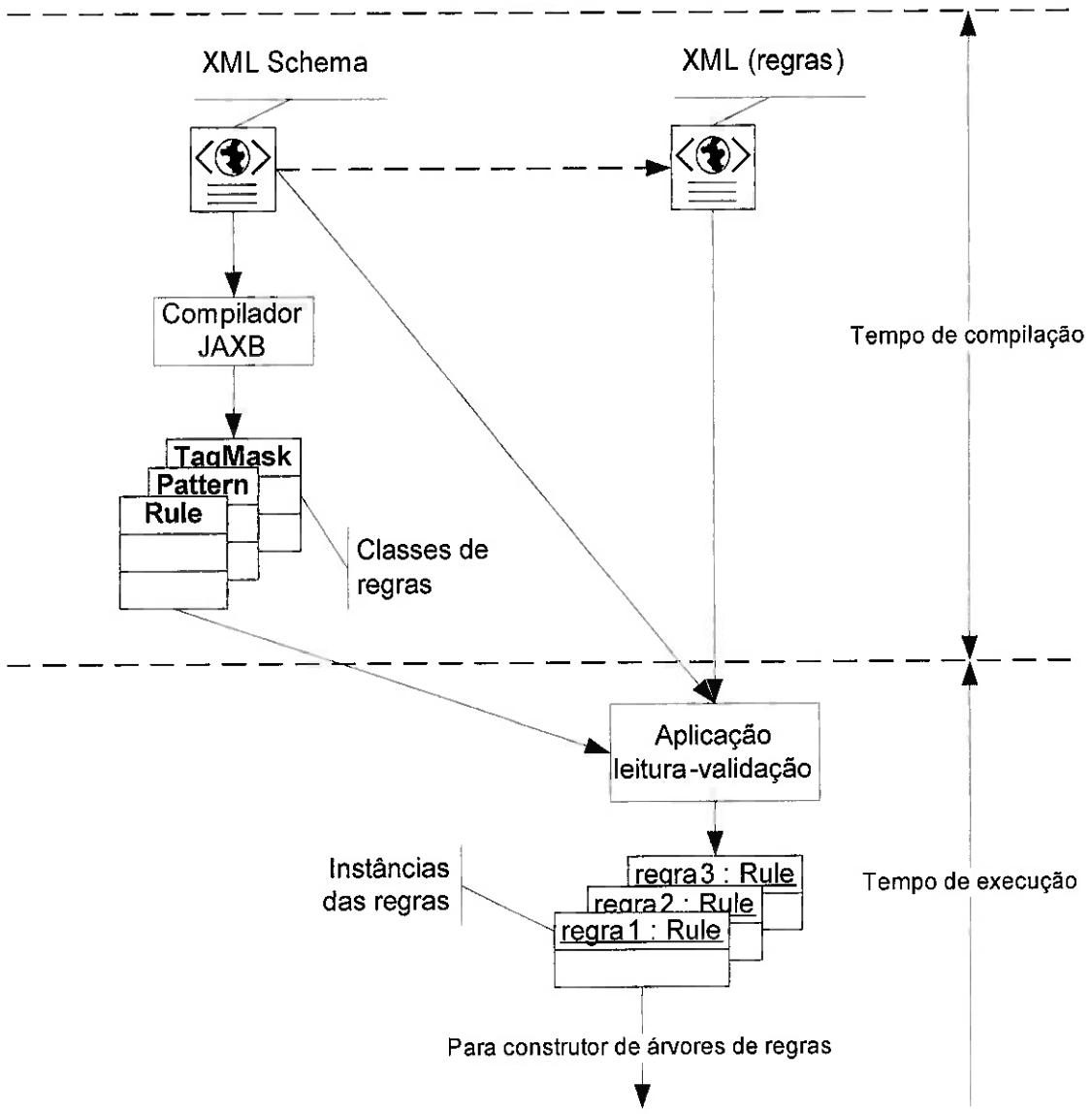


Figura 18 - Compilação da estrutura, leitura e validação das regras

7.4.4 Construção das árvores de regras

De posse do esquema, do arquivo de regras e das classes geradas a partir do esquema, podem-se carregar automaticamente as regras para uso pelo CoGrOO. A partir deste ponto, a aplicação de regras no CoGrOO está em tempo de execução, sendo constituído dos processos

indicados na Figura 18 e Figura 19.

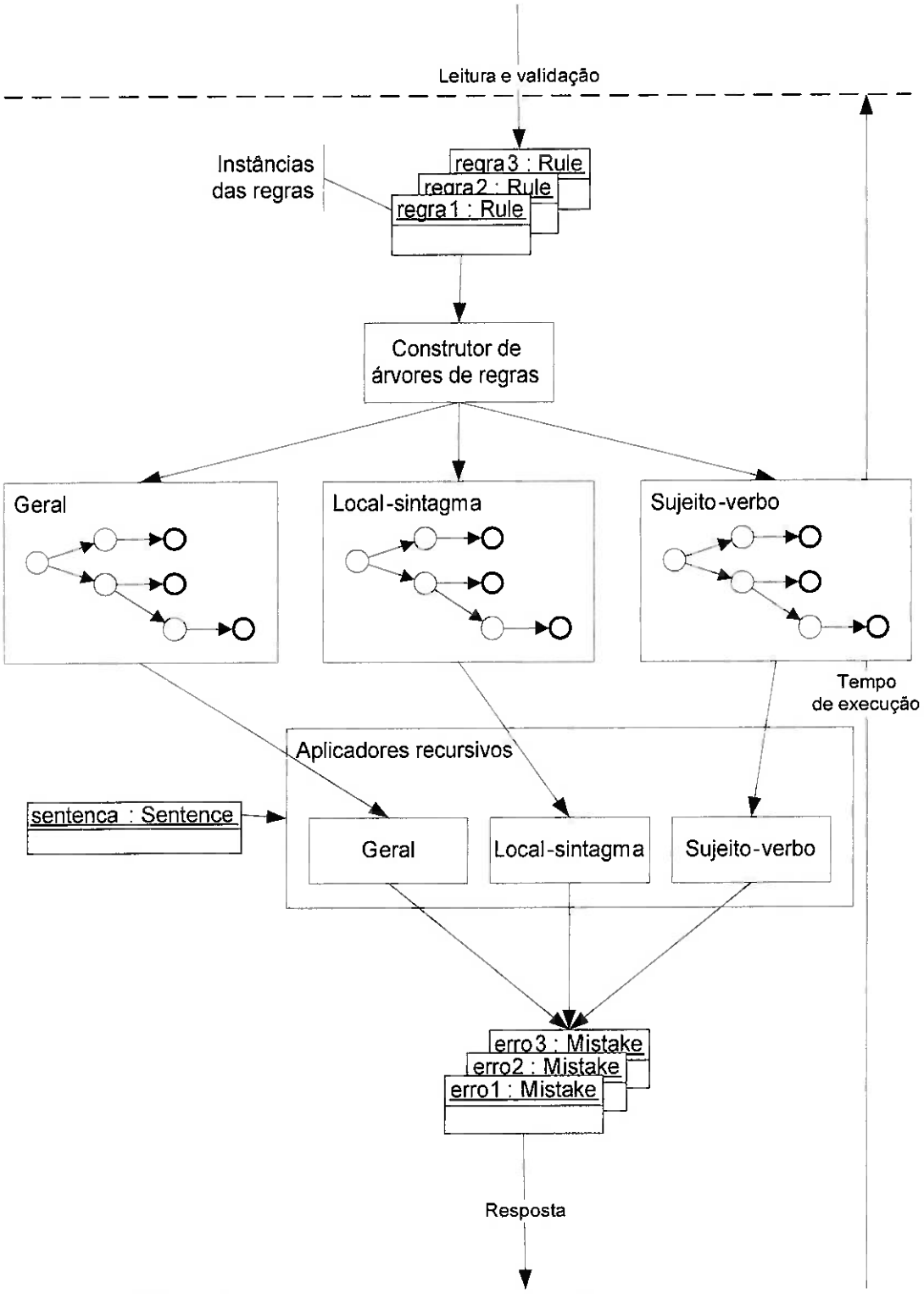


Figura 19 - Construtores de árvores e aplicadores de regras

No processo de carregamento das regras, é aplicada uma validação que indica se as regras estão em conformidade com o esquema.

As informações contidas no arquivo de regras são então utilizadas para construir instâncias da classe *Rule* (regra), que contém as informações necessárias para a aplicação das regras nas sentenças.

O exemplo da regra ["a" "a" verbo], mostrado na seção 7.4.1, é uma visão simplista do método de aplicação de regras. Para que se possam aplicar as regras na sentença, é necessário construir árvores que determinam se o padrão ocorre ou não na sentença, se aplicado a partir de um *token* arbitrário.

A aplicação das regras consiste então de algoritmos que transitam sobre árvores de estados (um autômato) não-determinísticos cujas transições são disparadas pela identificação de *tokens*.

Para exemplificar o processo de construção das árvores, serão utilizadas seis regras de método geral (que existem de fato no CoGrOO). Elas podem ser vistas na Tabela 6 (descritas em formato simplificado seguindo o código apresentado na Tabela 5):

Tabela 5 - Legenda para o padrão das regras simplificado

Símbolo	***Significado***
"x"	lexema
|	alternativa
(x)	etiqueta
{x}	primitiva

Tabela 6 - Regras para o exemplo de construção de árvore

Regra
"a" "a" (substantivo masculino singular)

Regra
"a" "a" "Vossa" "Senhoria|Majestade|Eminência|Excelência|Reverendíssima|Santidade"
"a" "a" (verbo)
{preferir} "mais"
{preferir} (substantivo) "de" "o" "que" (substantivo)
(verbo 3ª pessoa plural) "anexa" (determinante feminino plural)

São construídas tantas árvores quantos forem os métodos possíveis no atributo método descrito no esquema das regras. No CoGrOO, são três, já citadas: geral, local-sintagma e sujeito-verbo.

O algoritmo construtor de regras processa uma regra de cada vez e constrói estados para cada elemento da regra sendo processada. O estado é colocado na árvore correspondente ao método da regra sendo processada.

O algoritmo de construção das árvores consiste basicamente dos seguintes passos:

- Estado atual é o inicial (0);
- Para cada regra;
    - Obtém-se o padrão da regra;
    - Para cada elemento do padrão;
        - Verificar se já não existe estado atingível com o elemento sendo processado;
        - Se sim, estado atual se torna o estado atingível por meio da detecção do elemento;

- Se não, colocar um novo estado atingível por meio da detecção do elemento, a partir do estado atual; Estado atual é o estado novo;

Por esse algoritmo obtém-se a árvore da Figura 20, para as seis regras utilizadas como exemplo.

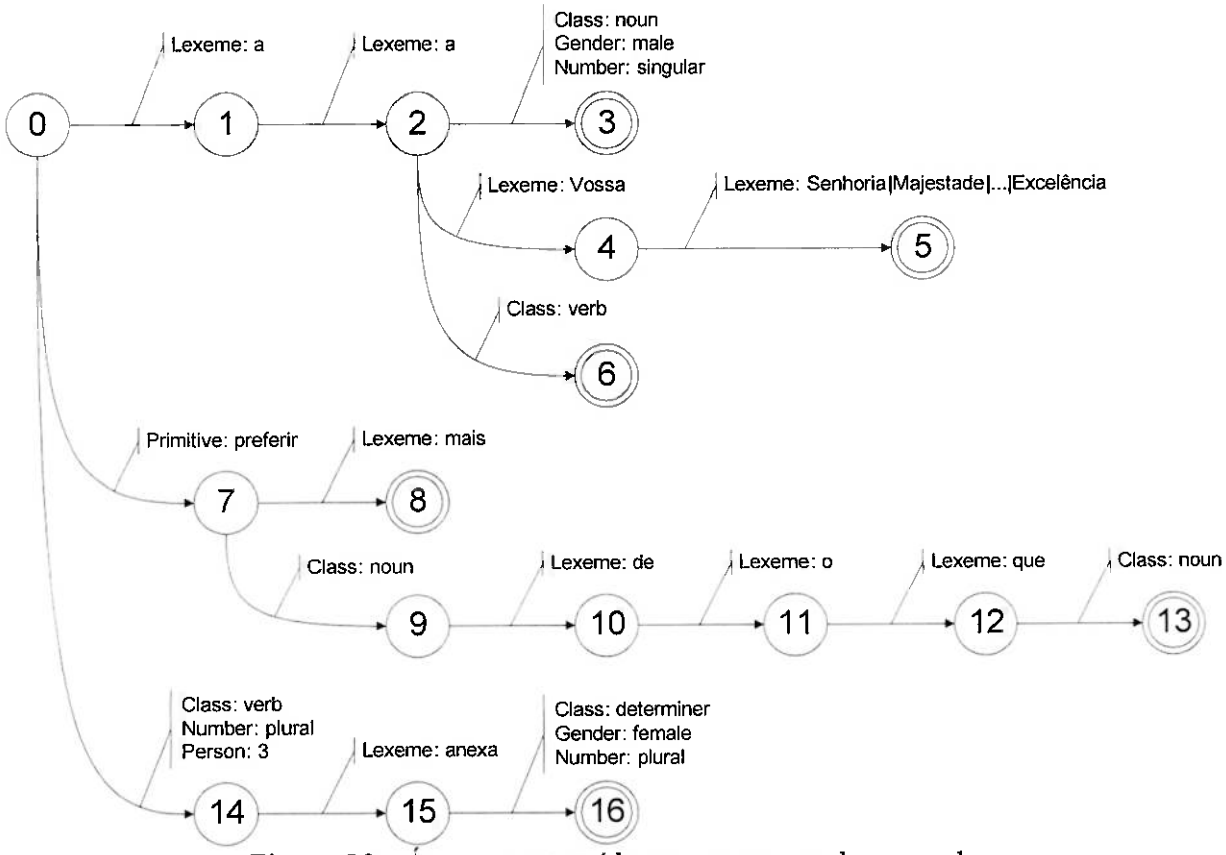


Figura 20 - Árvore construída para as regras de exemplo

7.4.5 Aplicadores de regras

Os aplicadores de regras são algoritmos recursivos que transitam sobre as árvores, dado um *token* ou agrupamento e um estado de uma das árvores, e verificam se o *token* ou agrupamento provoca a transição para um estado seguinte na árvore.

O número de aplicadores no CoGrOO é igual ao número de árvores geradas, que por

sua vez é igual ao número de métodos definidos nas regras.

O processo será descrito para a árvore de regras do método geral, mas o princípio de aplicação vale para os outros métodos, apenas trocando-se *token* por agrupamento no caso da árvore de sujeito-verbo. No caso da árvore local-sintagma, o princípio de aplicação ainda se dá sobre os *tokens*, mas somente nos *tokens* dentro de cada agrupamento.

O processo inicia-se pelo primeiro *token* da sentença e o estado inicial de uma das árvores. Todos os caminhos que saem do estado inicial são testados e os estados cuja transição seja ativada são tidos como novos estados atuais, nos quais o mesmo processo de transição é aplicado novamente, desta vez utilizando o próximo *token* da sentença. Este fato faz com que a aplicação de regras seja um processo recursivo, pois se tem novamente um *token* e um estado atual a ser testado.

Um exemplo de aplicação, para o mesmo conjunto de regras definido na seção 7.4.4 é apresentado para facilitar a compreensão do processo.

Sentença: "Preferem anexa as cartas de apresentação."

A preparação da sentença no CoGrOO para a aplicação de regras oferece o seguinte resultado (simplificado – apresenta somente as informações relevantes para o aplicador de escopo geral):

Tabela 7 - Lexemas, primitivas e etiquetas morfológicas da sentença de exemplo

Lexema	***Primitiva***	***Etiqueta morfológica***
Preferem	preferir	verbo 3ª pessoa do plural presente do indicativo
anexa	anexar	verbo 3ª pessoa do singular presente do indicativo
as	o	determinante feminino plural
cartas	carta	substantivo feminino plural
de	de	preposição
apresentação	-	substantivo feminino singular

Lexema	*Primitiva*	*Etiqueta morfológica*
.	-	pontuação absoluta

A aplicação começa no primeiro *token* da sentença e pelo estado inicial dá árvore. As transições possíveis são:

Tabela 8 - Primeira iteração na aplicação de regras

Token em questão			
"Preferem", primitiva: "preferir", verbo 3ª pessoa do plural presente do indicativo			
Estado atual	*Transita por*		*Transita para estado*
0	Lexema	"a"	1
	Primitiva	"preferir"	7
	Etiqueta	verbo conjugado na 3ª pessoa do plural	14

Dadas essas três possibilidades, vê-se que a segunda e a terceira são possíveis, pois o *token* "Preferem" tem como primitiva a palavra "preferir" e é um verbo conjugado na 3ª pessoa do plural. Neste caso, são ativadas essas duas transições e os próximos estados que serão usados como base de aplicação no próximo passo do aplicador são os estados 7 e 14.

O próximo *token*, "anexa" e o primeiro dos estados de chegada da transição do passo anterior (estado 7) são escolhidos como base da aplicação de regras. Para essa configuração, as transições possíveis são:

- o lexema do *token* é "mais"

- o *token* é um substantivo

Tabela 9 - Segunda iteração na aplicação de regras

Token em questão		
"anexa", primitiva: "anexar", verbo 3ª pessoa do singular presente do indicativo		
Estado atual	*Transita por*	*Transita para estado*

7	Lexema	"mais"	8
	Etiqueta	substantivo	9

Neste passo, verifica-se que nenhuma transição é ativada e não há mais como se encontrar nenhuma regra por esse caminho da árvore.

O aplicador examina então o caminho possibilitado pelo estado 14, utilizando o mesmo *token* como base.

Tabela 10 -Continuação da segunda iteração na aplicação de regras

Token em questão			
"anexa", primitiva: "anexar", verbo 3ª pessoa do singular presente do indicativo			
Estado atual	***Transita por***		***Transita para estado***
14	Lexema	"anexa"	15

Por esse caminho, o *token* "anexa" e a transição por lexema "anexa" habilitam a mudança para o estado 15.

Toma-se como base agora o *token* seguinte ao "anexa", que é o *token* "as" e o estado 15.

Tabela 11 - Terceira iteração na aplicação de regras

Token em questão			
"as", primitiva: "o", determinante feminino plural			
Estado atual	***Transita por***		***Transita para estado***
15	Etiqueta	determinante feminino plural	16

Aqui, ocorre transição para o estado 16, pois o *token* "as" é um determinante feminino plural. O estado 16 é final, o que indica que um desvio gramatical foi encontrado entre o *token* que iniciou o processo de aplicação de regras ("Preferem") e este último *token*

examinado ("as").

A classe que representa um estado final contém uma referência para a regra por ela procurada e assim se consegue todas as informações relevantes sobre a regra de desvio, que são os atributos de uma regra, como os descritos na seção Formato das regras.

Neste ponto, o aplicador recomeça todo o ciclo: o estado considerado volta a ser o inicial (0) e o *token* escolhido para verificação da transição é o *token* seguinte ao que deu início ao processo, neste caso o "anexa".

Sucessivamente, cada *token* é escolhido como base do processo a partir do estado 0, o processo recursivo é aplicado e, assim, a sentença toda é varrida em busca de desvios gramaticais.

7.5 Corretor gramatical

É uma classe controladora que faz a integração dos módulos do corretor gramatical.

Primeiramente, carrega todos os recursos necessários ao seu funcionamento, como os dicionários e modelos do OpenNLP, além das árvores de regras.

Em seguida, cada um dos módulos é invocado e é passado como parâmetro um objeto da classe *Sentence*. Esse objeto é preenchido gradualmente conforme é processado por cada módulo.

Primeiramente, é assumido que o OpenOffice.org enviará uma sentença por vez para ser examinada pelo CoGrOO. Portanto, o primeiro módulo a processar a sentença recebida é o separador de *tokens*. Os *tokens* separados são armazenados no objeto sentença.

Em seguida, a sentença é enviada ao módulo de pré-etiquetamento, responsável por

separar as palavras contraídas, como "do", que é, na verdade, a preposição "de" mais o artigo "o"; além de procurar nomes na sentença.

O próximo módulo é o etiquetador morfológico que determina as classes morfológicas. O atributo *morphologicalTag* de cada *token* é preenchido nesta etapa.

O agrupador determina quais conjuntos de tokens podem fazer parte de um sintagma nominal.

Por fim, o analisador sintático simples procura agrupamentos que podem ser sujeito e verbo.

Com o objeto sentença preenchido com as informações relevantes, o aplicador de regras pode se utilizar de todo o resultado da análise gramatical para determinar os desvios gramaticais.

7.6 Integração com o OpenOffice.org

Uma chamada remota de procedimento (RPC) é efetuada pela interface gráfica de usuário do corretor gramatical no OpenOffice.org para o servidor que executa o CoGrOO. O CoGrOO recebe a requisição e recupera a sentença submetida pelo usuário e é capaz de aplicar as análises gramaticais nesta sentença. Em seguida, o CoGrOO monta objetos que representam desvios gramaticais, padronizados por meio da implementação de contratos (interfaces) entre o cliente e o servidor, e os envia de volta ao cliente.

É responsabilidade do cliente manipular esses objetos de desvio gramatical e exibir as informações relevantes contidas nesses objetos na interface gráfica do usuário.

8 TESTES E RESULTADOS

8.1 Metodologia de teste

Foram criados testes para cada módulo interno do corretor gramatical. São eles: separador de sentenças, separador de *tokens*, etiquetador morfológico, agrupador e analisador sintático simples. O objetivo principal desta tarefa foi verificar o funcionamento de cada módulo e estimar a qualidade do CoGrOO em cada etapa de tratamento de uma sentença.

Os testes foram desenvolvidos utilizando a tecnologia JUnit. Para auxiliar a visualização dos resultados de acurácia foi desenvolvida uma classe de relatórios.

Para realizar a validação das informações testadas foi preciso um corpus anotado. Os testes utilizaram os módulos desenvolvidos pelo grupo e o resultado que cada módulo devolveu foi comparado com a anotação do corpus. Para cada resultado diferente, foi considerado que o módulo não funcionou corretamente e nesse caso foi contabilizado um erro para este módulo. Caso o resultado seja idêntico, é considerado um acerto para o módulo em teste. No final do teste são contadas as quantidades de erros e acertos e exibidos em um relatório a porcentagem de acurácia daquele módulo.

Um requisito para o projeto dos testes é que os dados usados como corpo de prova sejam dados nunca vistos pelo módulo sendo testado. Ou seja, o corpo de prova não pode ser usado no treinamento do módulo. Por isso é empregada a metodologia 1/9, que consiste em se utilizar um único corpus, do qual utilizamos 90% da informação para treinamento e os outros

10% como corpo de prova. Dado esse cenário, a parte referente ao corpo de prova (10%) é variada até que sejam realizados dez testes, e sempre utilizando as outras nove partes para treinamento em cada iteração do teste.

8.2 Resultados esperados

Para avaliação da qualidade do corretor gramatical foram criados testes que informam a acurácia obtida por cada módulo. Desta maneira foi possível analisar estes resultados e buscar maneiras de aperfeiçoar as rotinas executadas nos tratamentos de sentenças. Entretanto a elaboração destes códigos não foi elementar e alguns problemas foram detectados na comparação entre o valor obtido e o valor esperado pelos testes, em alguns casos foi possível alterar a verificação para que o valor obtido entre o número de respostas corretas sobre o número de inválidas fosse aproximado do real valor que o módulo exercia, entretanto em outros casos não foi possível realizar estas mudanças e por isso alguns resultados estão abaixo de números considerados razoáveis.

As taxas de acerto que se desejava obter encontram-se nos requisitos não funcionais, item 3.2.5.

A importância de resultados altos deve-se porque os módulos são chamados em cascata e os erros de um são propagados para os demais módulos. Quanto menor fosse a taxa obtida, maior a probabilidade do resultado final ser ruim o bastante para o corretor gramatical tornar-se inútil na correção de textos.

Um exemplo disto é utilizar uma estimativa de 95% de acertos em um texto com duzentas palavras, com esta porcentagem teríamos a marca de dez palavras analisadas de forma errada neste texto, o que significa um erro a cada vinte palavras. Ao passo que com uma estimativa de 90%, teríamos um erro a cada dez palavras, o que se torna um valor não

aceitável.

8.3 Separador de sentenças

O teste criado para o separador de sentenças faz a validação de uma sentença que o sistema encontra com uma sentença do corpus anotado. Para este teste se agrupa três sentenças do corpus com a intenção de formar uma seqüência de texto. Este texto então é enviado ao modulo separador de sentenças. O resultado correto é o módulo retornar exatamente três sentenças, já que o texto é constituído de três sentenças. No caso de ter devolvido mais sentenças, o teste considera que houve um falso positivo e, se ao contrario, retornar menos de 3 sentenças considera-se que houve um falso negativo. Se não houve nem um falso positivo e nem um falso negativo é verificado se a segunda sentença retornada pelo modulo é diferente da segunda sentença informada pelo corpus, caso sejam diferentes considera-se que houve um erro.

Deste modo todo o corpus é testado e ao final do teste o relatório exibe a quantidade de sentenças testadas, a acurácia, a quantidade de falsos negativos e a quantidade de falsos positivos.

Tabela 12 - Exemplo de resultado correto e incorreto para o separador de sentenças

Sentenças: Fomos levados a crer. Não sabemos de nada. O sr. Madruga é louco.	
Resultado correto	Resultado incorreto
Fomos levados a crer.	Fomos levados a crer.
Não sabemos de nada.	Não sabemos de nada.
O sr. Madruga é louco.	O sr.
	Madruga é louco.

O módulo separador de sentenças foi executado com um corpus de 2130 sentenças o

resultado obtido pode ser considerado bom pois obtivemos 96% de acurácia, sendo que entre os erros foram levantados que 44% deles geraram falsos negativos (número de sentenças gerado pelo separador de sentenças menor que o número de sentenças utilizadas como entrada para este módulo) e outros 44% de falsos positivos (número de sentenças gerado pelo separador de sentenças maior que o número de sentenças utilizadas como entrada), restando 12% de erros no qual a posição em que a cadeia de entrada foi dividida em sentenças não correspondeu às mesmas sentenças que faziam parte da cadeia.

Um exemplo de falso positivo pode ser visto na seguinte sentença que o teste informou que não foi reconhecida corretamente:

Foi utilizada técnica mista, incluindo desenho, pastel, cerografia e fotografismo. Min. Rocha Azevedo, 1, tel. 1). Até 1 de março.

O módulo comparou a segunda sentença existente, Min. Rocha Azevedo, 1, tel. 1), com o resultado da saída do módulo, Min., e visto que estas duas sentenças não são equivalentes um erro foi gerado para exibição no relatório.

8.4 Separador de tokens

O teste criado para o separador de *tokens* faz a validação de um *token* que o sistema encontra comparando com o *token* do corpus anotado. A implementação deste teste recupera cada sentença pertencente ao corpus e utiliza-a no modulo do separador de *tokens* para que ela seja quebrada em *tokens*. Estes *tokens* são comparados com os *tokens* da sentença original do corpus e para cada *token* incorreto é considerado que houve um erro. Um relatório do teste é exibido quando a execução termina informando a quantidade de sentenças analisadas e a acurácia obtida pelo módulo.

Tabela 13 - Exemplo de resultado correto e incorreto para o separador de *tokens*

Sentença: O máximo que consigo ficar sem alguém é dois anos.	
Resultado correto	Resultado incorreto
O	O máximo
máximo	que
que	consigo
consigo	ficar
ficar	com
com	alguém
alguém	é
é	dois
dois	anos
anos	.
.	

O módulo separador de *tokens* foi testado com 2150 sentenças o resultado obtido foi muito ruim, a acurácia foi de 60% e este valor é explicado pelo grande número de problemas que existem quando algum nome próprio existe na sentença e por alguns problemas no corpus que agrupa *tokens*, átomos, quando não deveria estar agrupado. Para corrigir o problema de nome próprio é necessário um maior tempo de desenvolvimento do módulo NameFind existente no OpenNLP.

A seguir são exibidos dois exemplos dos problemas encontrados para realizar a comparação neste módulo:

A direção do novo semanal será assinada por Ewaldo Ruy.

O erro encontrado neste exemplo é relacionado a nome próprio, o teste comparou o *token* Edwaldo Ruy, do corpus, com Edwaldo, gerado pelo módulo, e identificou um erro. O outro exemplo é o seguinte:

Eles se dizem oposição, mas ainda não informaram o que vão combater.

Neste exemplo o sistema identifica o erro pois o *token* o, gerado pelo módulo, é comparada ao *token* o que, anotado no corpus, e por isso um erro é encontrado. Percebemos neste exemplo que existe um problema com a anotação do corpus e isto interfere nos resultados de acurácia do módulo.

8.5 Etiquetador morfológico

O teste criado para o etiquetador morfológico faz a validação de uma etiqueta que o sistema encontra comparando com uma etiqueta do corpus anotado. Para a realização deste teste é recuperada cada sentença do corpus e enviada para o módulo do etiquetador morfológico que a processa, associando uma etiqueta morfológica para cada *token.* Essas etiquetas são comparadas com as etiquetas anotadas no corpus e caso não sejam iguais em todos os *tokens* da sentença, considera-se que houve um erro. Assim, depois de todas as sentenças testadas é gerado um relatório que contém informações sobre a quantidade de sentenças e a acurácia obtida pelo teste.

Tabela 14 - Exemplo de resultado correto e incorreto para o etiquetador morfológico

Sentença: A Fifa e a CBF deveriam entrar de sola.			
Resultado correto		Resultado incorreto	
Lexema	Etiqueta Morfológica	Lexema	Etiqueta Morfológica
A	determinante feminino singular	A	determinante feminino singular
Fifa	**nome próprio feminino singular**	**Fifa**	**substantivo feminino singular**
e	conjunção coordenativa	e	conjunção coordenativa
a	determinante feminino singular	a	determinante feminino singular
CBF	**nome próprio feminino singular**	**CBF**	**substantivo feminino singular**
deveriam	verbo na terceira pessoa do plural do pretérito do subjuntivo	deveriam	verbo na terceira pessoa do plural do pretérito do subjuntivo
entrar	verbo infinitivo	entrar	verbo infinitivo

de	preposição	de	preposição
sola	substantivo feminino singular	sola	substantivo feminino singular
.	pontuação absoluta	.	pontuação absoluta

Executando este teste com 36169 *tokens*, o resultado obtido pode ser considerado ruim pois foi obtido a acurácia de 91%, totalizando 32914 *tokens* etiquetados corretamente e 3255 que foram identificados erroneamente. Este grande número de *tokens* identificados de forma errada possui algumas explicações, entre elas não podíamos nos abster de citar que a implementação do módulo etiquetador não está perfeita mas outros problemas relacionados aos dados fornecidos pelo corpus também possuírem pequenas falhas e alguns *tokens* poderem ser classificados de varias formas e a maneira como foi etiquetado pelo módulo ser diferente do valor esperado pela base de informações.

Um exemplo que exibe um problema na etiqueta fornecida pelo corpus na qual o sistema de teste considerou que houve um erro do módulo é o seguinte:

> Mesmo a simples tentativa de se documentar esse ponto deixa uma pessoa exposta a acusações de condescendência e, assim, os brancos de fato conseguiram cooptar os julgamentos de valor.

O teste exibiu uma mensagem de erro durante a classificação dos *tokens* desta sentença quando tentou comparar a etiqueta fornecida para a palavra "documentar". O módulo gerou uma etiqueta de verbo no infinitivo enquanto o corpus estava anotado que o valor correto seria um verbo na terceira pessoa do singular e no infinitivo, como a comparação requisita que ambos os resultados sejam idênticos, foi gerado um erro neste *token*. Acreditamos que o resultado deste módulo seria muito melhor com um corpus com menos incorreções que o utilizado.

8.6 Agrupador

O teste criado para o agrupador faz a validação de um agrupamento que o sistema determina comparando com uma compilação de informações contidas no corpus anotado. Este teste busca todos os *tokens* de uma sentença, recupera de qual classe morfológica cada *token* pertence e qual a sua finitude, se possuir, e com estas informações descobre os agrupamentos contidos na sentença. Estes agrupamentos são comparados com os agrupamentos obtidos no corpus e caso sejam diferentes é considerado que houve um erro. Após percorrer todas as sentenças é exibido um relatório contendo os valores da quantidade de *tokens* testados, a acurácia do modulo e a quantidade de erros que existiram neste teste.

Tabela 15 - Exemplo de resultado correto e incorreto para o agrupador

Sentença: Pegam os dois pimpolhos Joana e Luca e se mandam para Lake Powell, Arizona.			
Resultado correto		Resultado incorreto	
Lexema	Etiqueta de Agrupamento	Lexema	Etiqueta de Agrupamento
Pegam	Núcleo do Sintagma Verbal	Pegam	Núcleo do Sintagma Verbal
os	Inicio do Sintagma Nominal	os	Inicio do Sintagma Nominal
dois	Continuação do Sintagma Nominal	dois	Continuação do Sintagma Nominal
pimpolhos	Núcleo do Sintagma Nominal	pimpolhos	Núcleo do Sintagma Nominal
Joana	**Continuação do Sintagma Nominal**	**Joana**	**Núcleo do Sintagma Nominal**
e	Outros	e	Outros
Lucas	Núcleo do Sintagma Nominal	Lucas	Núcleo do Sintagma Nominal
e	Outros	e	Outros
se	Núcleo do Sintagma Nominal	se	Núcleo do Sintagma Nominal
mandam	Núcleo do Sintagma Verbal	mandam	Núcleo do Sintagma Verbal
para	Outros	para	Outros
Lake	Núcleo do Sintagma Nominal	Lake	Núcleo do Sintagma Nominal
Powell	Núcleo do Sintagma Nominal	Powell	Núcleo do Sintagma Nominal

,	Outros	,	Outros
Arizona	Núcleo do Sintagma Nominal	Arizona	Núcleo do Sintagma Nominal
.	Outros	.	Outros

Após a execução do teste neste módulo foi obtido o bom resultado de 97% de acurácia, do qual 771 foram erros de etiquetas dentre as 36417 testadas. O resultado não foi melhor pois as etiquetas para informação do agrupamento foram obtidas automaticamente, o que certamente introduz erros ao sistema e além disso os treinamentos para este módulo foram realizados com uma quantidade menor de sentenças do que outros módulos pois os arquivos gerados por ele eram extremamente grandes, o que tornaria o CoGrOO uma ferramenta mais pesada para instalação nos clientes, e devido a isto talvez a parcela do corpus utilizada não representasse um espaço amostral significativo.

Um exemplo de erro encontrado na comparação entre a etiqueta gerada e a original é exibido a seguir:

Pegam os dois pimpolhos Joana e Luca e se mandam para Lake Powell, Arizona.

O erro encontrado ocorreu no *token* Joana pois a etiqueta original é "continuação do sintagma nominal" enquanto a gerada foi "núcleo do sintagma nominal".

8.7 Analisador sintático simples

O teste criado para o analisador sintático simples faz a validação de sujeito e verbo que o sistema determina comparando com o sujeito e verbo contido no corpus anotado. Para cada sentença encontrada no corpus é efetuada uma busca para recuperar a classe morfológica do token e sua finitude, caso possua. A partir dessas informações é possível obter a etiqueta sintática e comparar com as etiquetas obtidas das anotações do corpus. Caso alguma etiqueta

na sentença seja diferente da etiqueta do corpus é considerado que houve um erro e ao final do corpus um relatório é elaborado com dados sobre a quantidade de tokens, a acurácia do modulo e a quantidade de erros relatados pelo teste.

Tabela 16 - Exemplo de resultado correto e incorreto para o analisador sintático simples

Sentença: Aquele pensamento provocou-me um arrepio estranho e delicioso.			
Resultado correto		Resultado incorreto	
Lexema	Etiqueta de Agrupamento	Lexema	Etiqueta de Agrupamento
Aquele	Nada	Aquele	Nada
pensamento	Sujeito	pensamento	Sujeito
provocou-me	Verbo	provocou-me	Verbo
um	Nada	um	Nada
arrepio	**Nada**	**arrepio**	**Sujeito**
estranho	Nada	estranho	Nada
e	Nada	e	Nada
delicioso	Nada	delicioso	Nada
.	Nada	.	Nada

Para este módulo foi obtido o bom resultado de 96% de acurácia, dos quais 1111 foram erros de etiquetas morfológicas dentre as 36417 testadas. Os problemas encontrados para o valor da acurácia são os mesmos que os encontrados no módulo agrupador pois ambos os módulos foram implementados de forma similar e resultados semelhantes eram esperados.

Um exemplo de comparação que foi considerado como erro é o seguinte:

Em a volta de uma viagem a o exterior, vale a pena trazer uma impressora matricial.

O módulo retornou "nada" para a etiqueta sintática do *token* "vale", enquanto o valor anotado do corpus era "sujeito".

8.8 Cumprimento dos requisitos não funcionais

Alguns requisitos não funcionais ficaram bastante prejudicados dado que nesse projeto se privilegiou os requisitos funcionais, para validar as idéias e se alcançar os resultados em tempo hábil.

Um dos que nos pareceu aceitável foi o tempo de resposta das requisições (corretor gramatical já iniciado), que está similar ao do CoGrOO 1.0, que consideramos nosso alvo, e também de similares.

Consumo de processamento e memória deixou a desejar. O sistema leva muito tempo e consome muito processamento durante a inicialização. Em um microcomputador relativamente moderno a inicialização pode levar 40 segundos, consumindo cem por cento do processamento.

Outro quesito que falhou foi o consumo de memória. O *Maxent* carrega todas as bases de dados em memória durante a inicialização. Isto está consumindo uma quantidade considerável de memória *heap* da máquina virtual *Java*, aproximadamente 128 MB.

Quanto ao quesito espaço em disco, o CoGrOO 2.0 ocupa 20 MB. É um resultado satisfatório com relação ao CoGrOO 1.0 que ocupava bem mais do que isso. No entanto era desejável tamanho menor.

O controle de erros do Java, associado com as políticas utilizadas na codificação, fizeram o sistema se manter disponível mesmo em casos de falha. Ainda é preciso avaliar melhor este quesito, mas nos testes já feitos o sistema se mostrou estável e em caso de falha se recuperou sem causar desconforto ao usuário.

No quesito segurança de dados o corretor também falhou. Não foi implementada comunicação segura entre cliente-servidor, dado que isto não se mostrou como enfoque do projeto e que pode ser facilmente implementado no futuro. Existem diversos *logs* durante a execução que também não foram removidos e serão mantidos até que o sistema esteja com boa usabilidade.

9 TRABALHOS FUTUROS

O projeto CoGrOO 1.0 representou avanços significativos tanto para o PLN quanto para a comunidade que usa e desenvolve Software Livre.

É possível que a nova versão seja mais significativa ainda devido a sua facilidade de expansão, tornando-se um padrão para corretores gramaticais.

Contudo, muita coisa tem que ser feita para que isso se torne uma realidade; este capítulo discutirá 4 assuntos que receberão atenção futura:

- Otimizações (tempo e armazenamento);
- O porte para outra língua;
- Implementação de regras de apresentação e
- Como suplantar alguns problemas advindos, principalmente, relacionados à qualidade dos recursos lingüísticos disponíveis.

9.1 Otimizações

No desenvolvimento desse projeto se procuraram validar as idéias e tecnologias envolvidas no desenvolvimento do corretor gramatical, levando-se em conta principalmente seus requisitos funcionais.

Vários quesitos não funcionais falharam e eles seriam o enfoque de mais curto prazo no projeto.

9.1.1 Diminuir o tempo de inicialização

Verificou-se que a maior parte do tempo de inicialização do sistema é gasto com a leitura das bases de dados.

Os dicionários e modelos do *Maxent* são persistidos em disco. Quando eles são usados, o *Maxent* trás todo o conteúdo de volta para um objeto na memória.

Pelo conhecimento adquirido na elaboração do projeto, uma solução para este problema seria colocar estes objetos em um banco de dados, por exemplo SQL, e estender o *Maxent* para acessar estes objetos diretamente no banco de dados sem precisar carregar o objeto em memória.

9.1.2 Otimizar os modelos para poupar processamento

Para o treinamento dos modelos foram usados valores arbitrários para o número de sentenças de treinamento e notas de corte.

Certamente poderia diminuir a quantidade de dados dos modelos. Para isso poderia criar rotinas que experimentam diversas configurações e traçam a curva de aprendizagem do modelo. Quando a máquina adquirir conhecimento satisfatório e mais dados não colaborarem mais com o desempenho, poderia estabelecer este ponto de treinamento como ideal.

9.1.3 Reduzir o tamanho do pacote

Poderia diminuir o tamanho do pacote do corretor gramatical com alguns procedimentos:

- Compactar os dicionários usando algum dos algoritmos de compressão conhecidos, no entanto de forma a manter seus requisitos funcionais e não funcionais de tempo de acesso e gasto de memória;
- Diminuir os modelos do *Maxent* conforme descrito no item 9.1.2.

9.2 Porte para outras línguas

Intenciona-se que o CoGrOO torne-se uma arquitetura padrão para corretores gramaticais livres. Para a sua difusão, torna-se importante ter um porte para a língua inglesa e um para a língua espanhola.

A língua inglesa é falada, como primeira língua, por 354 milhões de pessoas no mundo inteiro, e, como segunda língua, por mais de 1 bilhão de pessoas [INGLÊS]. Já o Espanhol é falado como primeira língua por cerca de 420 milhões de pessoas, sendo difícil precisar quantas outras o falam como segunda língua [ESPANHOL].

Mesmo para estas línguas não existe ainda um corretor gramatical abrangente, mas apenas protótipos [LINGUCOMPONENT].

A implementação para um novo idioma significa:

- Obter um Corpus Lingüístico com anotações morfológicas, sintáticas e de primitivas para o idioma em questão;
- Obter um dicionário com anotações morfológicas, sintáticas e de primitivas para o idioma em questão;
- Implementar a classe abstrata Corpus definida nesse projeto, que será o driver de leitura do novo Corpus;

- Implementar a classe abstrata Tag definida nesse projeto, para as tags morfológicas e sintáticas da língua em questão;
- Popular os dicionários com os dados obtidos, usando os Writers do CoGrOO;
- Executar o treinamento de cada um dos módulos e verificar o desempenho usando os módulos de testes do CoGrOO;
- Fazer o ajuste fino dos módulos problemáticos especializando as features de treinamento, repetir o treinamento;
- Escrever as regras de erro na sintaxe descrita por esse projeto, também é possível escrever módulos de verificação novos de acordo com necessidade;
- Verificar se todas as regras funcionam de acordo, caso necessário retornar ao ajuste fino dos módulos.

9.3 Regras de apresentação

Regras de apresentação são referentes a convenções tipográficas. São muito mais simples de se implementar que as regras para detecção de erros gramaticais.

Como o intuito do projeto CoGrOO era o de produzir um Corretor Gramatical, ignorou-se a implementação destas no presente trabalho. Contudo, ao analisar o desempenho de um dos principais corretores gramaticais da nossa língua, o ReGra [REGRA], fica evidente a importância de não as ignorar em um produto: cerca de 40% dos erros corretamente assinalados pelo ReGra eram inadequações a apresentação [ULIANO].

Deve-se ressaltar que o número de Falsos Positivos* (vide glossário) encontrados por regras estilísticas é muito pequeno, perto dos encontrados pelas regras gramaticais. Assim,

implementar um módulo aplicador de regras estilísticas contribuirá para melhorar a relação entre verdadeiros e falsos positivos assinalados num texto.

Para exemplificar, segue uma lista de regras de apresentação:

- Uma palavra não pode aparecer duas ou mais vezes seguidas;
- Não deve haver espaço antes de um símbolo de pontuação (ponto final, vírgula, ponto e vírgula, dois pontos, exclamação, interrogação);
- Após um sinal de pontuação deve haver um espaço antes da próxima palavra;
- Não deve haver espaço após o abre parêntesis;
- Não deve haver espaço após o abre aspas;
- Não deve haver espaço antes do fecha parêntesis;
- Não deve haver espaço antes do fecha aspas;
- Deve haver espaço antes e depois de um hífen;

9.4 Problemas conhecidos

Conhecendo o funcionamento da versão atual do corretor gramatical, e suas limitações, existe dificuldade no tratamento de alguns tipos de erros gramaticais.

Estes problemas estão suspensos até que se encontre alguma forma eficiente de resolvê-los. A seguir uma lista com alguns erros desses tipos:

- Tendo em vista que os módulos do CoGrOO se baseiam em um *corpus* "correto", a

identificação de padrões sobre um texto "incorreto" leva a situações complicadas para a detecção de erros. Tome-se como exemplo a situação:

"Dificuldades na obtenção de financiamento das obras, levaram..." (vírgula separando sujeito e verbo)

Em um *corpus* correto não é de se esperar que haja uma vírgula separando sujeito e verbo; portanto, dificilmente o token [Dificuldades] será identificado como sujeito do verbo [levaram], o que torna inviável (até este momento) a detecção de um erro como este.

- Outro problema existente é a anotação inserida no *corpus*. No seguinte exemplo, o *token* foi identificado de maneira incorreta:

"O máximo que consigo ficar sem alguém é dois anos." *(token = "O máximo")*

Este problema ocorre pois o *corpus* é etiquetado automaticamente. Para minimizar este problema seria necessário que o *corpus* fosse corrigido manualmente.

- Limitações nas análises, principalmente do Identificador de sintagmas e do Detector de sujeito e verbo, e análise sintática incompleta: não se propôs até o momento identificar elementos como objeto direto, indireto, predicativo do sujeito, etc., inviabilizando a implementação de regras como a seguinte

"Os carros e as equipes de restabelecimento ficam estrategicamente distribuídas..." (distribuídos)

, em que [distribuídas] deve concordar com [Os carros e as equipes]. Além disso, a identificação das funções sintáticas de uma palavra na frase, em tese, só pode acontecer com maior precisão quando todos os outros elementos da sentença também são identificados, o que indica que mesmo a identificação de sujeito e verbo do jeito que está implementada está aquém do que poderia ser.

Neste item provavelmente se encontra o maior número de situações em que o

CoGrOO ainda apresenta grandes limitações.

- Ambigüidades sintáticas: como foi dito anteriormente, algumas sentenças podem dar origem a árvores sintáticas diferentes, e optar por uma delas só é possível quando há informações de natureza semântica. Uma vez que análises neste nível de profundidade não estão no escopo do projeto, situações assim não podem ser corrigidas adequadamente.

10 CONCLUSÕES

O principal objetivo, que era construir um corretor gramatical acoplável ao OpenOffice.org, que fosse multiplataforma e desvinculado de idioma específico, foi atingido.

Todos os requisitos funcionais foram implementados, mas alguns requisitos não funcionais não foram atingidos. Este fato se deveu principalmente pelo fato do grupo ter priorizado o cumprimento dos requisitos funcionais e assim obter uma versão totalmente funcional do corretor. Sugestões de como contornar os problemas e conquistar os requisitos não funcionais pendentes já foram identificadas.

A arquitetura tem um bom grau de generalização, facilitando o porte para outros idiomas. A forte modularização, obtida com a orientação a objetos, trouxe diversos benefícios entre eles: facilidade de extensão, manutenibilidade e rastreamento de problemas.

A arquitetura do novo corretor gramatical é fortemente tipada, graças ao uso da linguagem Java. Os benefícios do uso do Eclipse como ambiente de desenvolvimento ficou evidente, principalmente por facilitar o trabalho em equipe. Tudo isto tornou o desenvolvimento bem menos propenso a erros.

Pretende-se continuar o desenvolvimento deste corretor seguindo as idéias propostas para o cumprimento dos requisitos não funcionais. Este desenvolvimento se mostra necessário, dada a grande expectativa que existe em torno do projeto CoGrOO (um exemplo disso foi o convite recebido feito pelo comitê de organização da "III Conferência Latino-americana de Software Livre", para que um membro da equipe de desenvolvimento

palestrasse sobre o desenvolvimento do CoGrOO 2.0. Neste evento, o grupo soube que a versão 1.0 já está sendo usada em 3000 máquinas das instalações da Usina Hidrelétrica de Itaipu [MARTINS]).

REFERÊNCIAS*

BRILL, E. **A Simple Rule-Based Part Of Speech Tagger**. Proceeding of ANLP-92, 3rd Conference of Applied Natural Language Processing, Trento, Italy, 1992. p. 152-155.

DICAS-L Tutorial XML-Schema, 2005: Disponível em: <http://www.dicas-l.com.br/dicas-l/20050326.php>. Acesso em: 29 nov. 2006.

ESPANHOL. Disponível em: <http://en.wikipedia.org/wiki/Spanish_Language>. Acesso em: 29 nov. 2006.

INFOWESTER Linguagem Java, 2003: Disponível em: <http://www.infowester.com/lingjava.php>. Acesso em: 29 nov. 2006.

INGLÊS. Disponível em: <http://en.wikipedia.org/wiki/English_language>. Acesso em: 29 nov. 2006.

JUDE: Disponível em: <http://jude.change-vision.com>. Acesso em: 29 nov. 2006.

KINOSHITA, J.; SALVADOR, L. N.; MENEZES, C. E. D. **CoGrOO – Um Corretor Gramatical para a língua portuguesa, acoplável ao OpenOffice**. Anais da "XXXI Latin American Informatics Conference, CLEI 2005", Cali, Colômbia, 2005 (versão eletrônica disponível em http://cogroo.incubadora.fapesp.br/portal/down/Doc/Artigo_Clei_2005.pdf).

KINOSHITA, J.; SALVADOR, L. N.; MENEZES, C. E. D. **CoGrOO: a Brazilian-Portuguese Grammar Checker based on the CETENFOLHA Corpus**. "Proceedings of the 5th International Conference on Language Resources and Evaluation, LREC 2006", Gênova, Itália, 2006 (versão eletrônica disponível em http://cogroo.incubadora.fapesp.br/portal/down/Doc/LREC_2006.pdf).

LINGUCOMPONENT Sub-Project: Grammar Checking. **Apresenta os esforços da Comunidade de Software Livre para produzir um Corretor Gramatical para o a suíte de escritório OpenOffice.org**. Disponível em: <http://lingucomponent.openoffice.org/grammar.html>. Acesso em: 02 dez. 2006.

---

* De acordo com:

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. NBR 6023: informação e documentação: referências: elaboração. Rio de Janeiro, 2002

MARTINS, M. S. **Comunicação pessoal de Marcos Siríaco Martins para William Colen Silva**, durante o evento: "III Conferência Latinoamericana de Software Livre (Latinoware)", data: 16-17 de novembro de 2006.

MATOS, R. S.; VEIGA, Á. **Otimização de entropia: implementação computacional dos princípios MaxEnt e MinxEnt**. Pesqui. Oper., Rio de Janeiro, v. 22, n. 1, 2002. Disponível em: <http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0101-74382002000100003&lng=en&nrm=iso>. Último acesso em: 11/2006. doi: 10.1590/S0101-74382002000100003.

NABER, D. **A Rule-Based Style and Grammar Checker**. Diplomarbeit Technis Fakultät, Universität Bielefeld, Alemanha, 2003

OPENNLP: Disponível em: <http://opennlp.sourceforge.net/>. Acesso em: 29 nov. 2006.

RATNAPARKHI, A. **Maximum Entropy Models for Natural Language Ambiguity Resolution**. Ph.D. Dissertation. University of Pennsylvania, Julho de 1998 (versão eletrônica disponível em ftp://ftp.cis.upenn.edu/pub/ircs/tr/98-15/98-15.pdf)

REGRA NILC. Disponível em: <http://www.nilc.icmc.usp.br/nilc/projects/regra.htm>. Acesso em: 29 nov. 2006.

SOUZA, F. J. F. **Adoção do OpenOffice.org**. Palestra na "Semana de Software Livre do Legislativo", disponível em <http://www.tc.df.gov.br/tcbrasil/tcdf/TCDF_OORev.pdf>. Acesso em: 02 dez. 2006.

SUN OPENS JAVA. Disponível em: <http://www.sun.com/2006-1113/feature/> Acesso em: 02 dez. 2006.

ULIANO, S. C.; MENEZES, C. E. D.; GUSUKUMA, F. W. **Uma análise do CoGrOO, um Corretor Gramatical acoplável ao OpenOffice**. Artigo não publicado. Disponível em: <http://cogroo.incubadora.fapesp.br/portal/down/Doc/analise>, último acesso em 29/11/2006.

XML RPC. Disponível em: <http://www.xmlrpc.com/>. Acesso em: 29 nov. 2006.

GLOSSÁRIO

- **Agrupador**: no presente caso, é aquele que indica os sintagmas presentes em uma sentença, e os *tokens* que os formam.
- **Analisador sintático simples**: é um módulo responsável por obter informações sintáticas a partir das palavras e de suas classes morfológicas.
- **Análise morfológica**: é o ato de se examinar as palavras presentes em uma sentença, de modo a determinar suas classes gramaticais.
- **Análise sintática**: é o ato de se examinar uma sentença, de modo a identificar e estabelecer sua estrutura, i.e., as relações entre os elementos que a constituem.
- ***Chunker***: vide agrupador.
- **Classe morfológica ou gramatical**: é a classificação de uma palavra segundo seu significado e/ou função. Na língua portuguesa, são dez as classes: substantivo, artigo, adjetivo, numeral, pronome, verbo, advérbio, preposição, conjunção e interjeição. As seis primeiras classes são variáveis (i.e., flexionam-se em gênero, número, etc.) e as quatro últimas são invariáveis.
- ***Corpus* lingüístico**: é um arquivo que obedece a um certo formato. *Corpus* útil para o CoGrOO seriam os constituídos de sentenças extraídas de textos. Estas sentenças devem estar separadas em *tokens* e cada *token* ter sua anotação morfológica. Cada *token* também deve ter uma anotação acerca de seu papel sintático na sentença.
- **Etiqueta**: neste projeto, é a entidade representativa de alguma informação referente à entidade que a possui.

- **Etiqueta morfológica**: é a entidade que representa a classe morfológica de uma palavra no sistema.
- **Etiqueta sintática**: é a entidade que representa a função sintática de uma palavra ou de um sintagma.
- **Falsos negativos**: o analisador de desvios não encontra um erro onde o erro existe.
- **Falsos positivos**: o analisador de desvios afirma haver um erro onde não existe. É o tipo de erro cometido por corretores gramaticais que mais perturba os usuários, pois o usuário pode trocar uma estrutura correta por uma incorreta ou mesmo ficar receoso quanto à veracidade dos outros erros (outros falsos positivos ou desvios verdadeiros) assinalados pelo corretor, abalando sua confiança no mesmo.
- **Função sintática**: é o papel desempenhado por uma palavra ou um agrupamento de palavras na estrutura de uma frase, que indica sua relação com os outros elementos desta.
- ***Remote Procedure Call*** (RPC): é um protocolo que permite chamadas remotas de sub-rotinas.
- **Separador de *tokens***: separa a sentença em seus elementos constituintes mais fundamentais, como palavras e sinais de pontuação.
- ***Shallow parser***: vide analisador sintático simples.
- **Sintagma**: é uma unidade formada por uma ou várias palavras que, juntas, desempenham uma função na frase. A combinação das palavras para formarem as frases não é aleatória; precisam obedecer a determinados princípios da língua. As

palavras se combinam em conjuntos, em torno de um núcleo. E é esse conjunto (o sintagma) que vai desempenhar uma função no conjunto maior, que é a frase.

- ***Token***: é a menor unidade de dados em que uma sentença pode ser dividida; no presente caso, pode representar uma palavra, um sinal de pontuação, um número, etc.

- ***Tokenizador***: vide separador de *tokens*.

APÊNDICE A – Features coletadas em exemplos em português

Este texto mostra algumas *features* coletadas nos diferentes módulos do *OpenNLP* para exemplos de textos em português brasileiro.

Separador de sentenças

Objetivo: Decidir se marcas de fim de linha estão separando linhas no contexto.

Estratégia:

1. Caracteres do tipo [.!?] são candidatos a separadores de sentença.
2. Ao redor de cada candidato, serão levantadas as *features* envolvidas.

Exemplo:

```
Ele foi procurar uma casa. Ele vai se casar com a Sra. Maria.
012345678901234567890123456789012345678901234567890123456789
          1         2         3         4         5         6
```

A. Caracter "." na posição 25:

Feature	*Significado*
sn	Sulfixo inicia com " "
eos=.	Caracter candidato a marcador de fim de sentença: "."
x=casa	Prefixo é "casa"
4	Tamanho do prefixo 4
v=uma	Seqüência anterior ao prefixo é "uma"
s=	Sufixo é " "
n=Ele	Seqüência posterior ao sufixo é "Ele"
ncap	Seqüência posterior ao sufixo se inicia com maiúscula.

B. Caracter "." na posição 53:

Feature	*Significado*
sn	Sulfixo inicia com " "
eos=.	Caracter candidato a marcador de fim de sentença: "."
x=Sra	Prefixo é "Sra"
abb	Prefixo é abreviatura
3	Tamanho do prefixo 3
xcap	Prefixo se inicia com maiúscula
v=a	Seqüência anterior ao prefixo é "a"
s=	Sufixo é " "
n=Maria.	Seqüência posterior ao sufixo é "Maria."
ncap	Seqüência posterior ao sufixo se inicia com maiúscula

C. Caracter "." na posição 60:

Feature	*Significado*
eos=.	Caracter candidato a marcador de fim de sentença: "."
x=Maria	Prefixo é "Maria"
5	Tamanho do prefixo 5
xcap	Prefixo se inicia com maiúscula
v=Sra.	Seqüência anterior ao prefixo é "Sra. "
s=	Sufixo é " "
n=	Seqüência posterior ao sufixo é " "

A partir dos dados de treinamento e com as *features* levantadas o *OpenNLP* sugere que os prováveis marcadores de fim de sentença são os nas posições 25 e 60. Ficamos, portanto, com as seguintes sentenças.

Ele foi procurar uma casa.
Ele vai se casar com a Sra. Maria.

Separador de tokens

Objetivo: Além dos espaços muitos outros símbolos podem separar tokens na frase. Exemplo "escritório, é" são três tokens.

Estratégia:

1. Caracteres de espaço sempre separam tokens;

2. Caracteres não alfanuméricos podem separar *tokens*

As *features* aqui coletam dados para decidir quando um caractere não alfanumérico separa *token.*

Exemplo:

```
O Linux, o sistema operacional, é livre.
012345678901234567890123456789012345678901234567890
          1         2         4         5
```

Em seguida, o tratamento da vírgula, caractere 40. Decide se "operacional," deve ser quebrado em dois *tokens* ou não.

Feature	***Significado***
p=o	Prefixo "o"
s=peracional,	Sulfixo "peracional,"
dn=sb	Grupo "operacional," não existe no dicionário.
d=p	Prefixo existe no dicionário.
p1=o	Prefixo "o"
p1_alpha	Prefixo é alfabético
p2=bok	Seqüência antes do prefixo não existe
p1f1=op	Prefixo + sufixo "op"
f1=p	Sufixo "p"
f1_alpha	Sufixo é alfabético
f2=e	Caractere depois do sufixo "e"
f2_alpha	Caractere depois do sufixo é alfabético
f12=pe	Sufixo mais caractere seguinte "pe"

Feature	*Significado*
p=op	Prefixo "op"
s=eracional,	Sulfixo "eracional,"
dn=sb	Grupo "operacional," não existe no dicionário.
p1=p	Prefixo "p"
p1_alpha	Prefixo é alfabético
p2=o	Caractere antes do prefixo "o"
p2_alpha	Caractere antes do prefixo é alfabético
p21=op	Anterior ao prefixo + prefixo "op"
p1f1=pe	Prefixo + sufixo "pe"
f1=e	Sufixo "e"
f1_alpha	Sufixo é alfabético
f2=r	Caractere depois do sufixo "r"
f2_alpha	Caractere depois do sufixo é alfabético
f12=er	Sufixo mais caractere seguinte "er"

E segue até acabar a seqüência, sendo que as duas últimas:

Feature	*Significado*
p=operaciona	Prefixo "operaciona"
s=l,	Sulfixo "l,"
dn=sb	Grupo "l," não existe no dicionário.
p1=a	Prefixo "a"
p1_alpha	Prefixo é alfabético
p2=n	Caractere antes do prefixo "n"
p2_alpha	Caractere antes do prefixo é alfabético
p21=na	Anterior ao prefixo + prefixo "na"
p1f1=al	Prefixo + sufixo "al"
f1=l	Sufixo "l"
f1_alpha	Sufixo é alfabético
f2=,	Caractere depois do sufixo ","
f12=l,	Sufixo mais caractere seguinte "l"

Feature	*Significado*
p=operacional	Prefixo "operacional"
s=,	Sulfixo ", "
dn=sb	Grupo ", " não existe no dicionário.
d=s	Prefixo existe no dicionário
p1=l	Prefixo "l"
p1_alpha	Prefixo é alfabético
p2=a	Caractere antes do prefixo "a"
p2_alpha	Caractere antes do prefixo é alfabético
p21=al	Anterior ao prefixo + prefixo "al"
p1f1=l	Prefixo + sufixo "l"
f1=,	Sufixo ","
f2=bok	Fim do token alcançado

Por fim as *features* seriam processadas pelo *Maxent*, comparadas com os contextos armazenados no modelo, concluiria que "operacional, " fica:

[operacional] [,]

Etiquetador morfológico

Objetivo: Palavras de mesma grafia podem ser classificadas morfologicamente de diferentes formas de acordo com o contexto em que estão. Por exemplo "casa", que pode ser substantivo ou verbo (casar).

[Ele] [foi] [procurar] [uma] [casa] [.]

Estratégia:

1. Para cada *token*, levantar as características desse *token*, dos *tokens* vizinhos e da sentença em que se encontra
2. Colocar também as decisões vizinhas

Exemplo: processamento de "casa" em "Ele foi procurar uma casa."

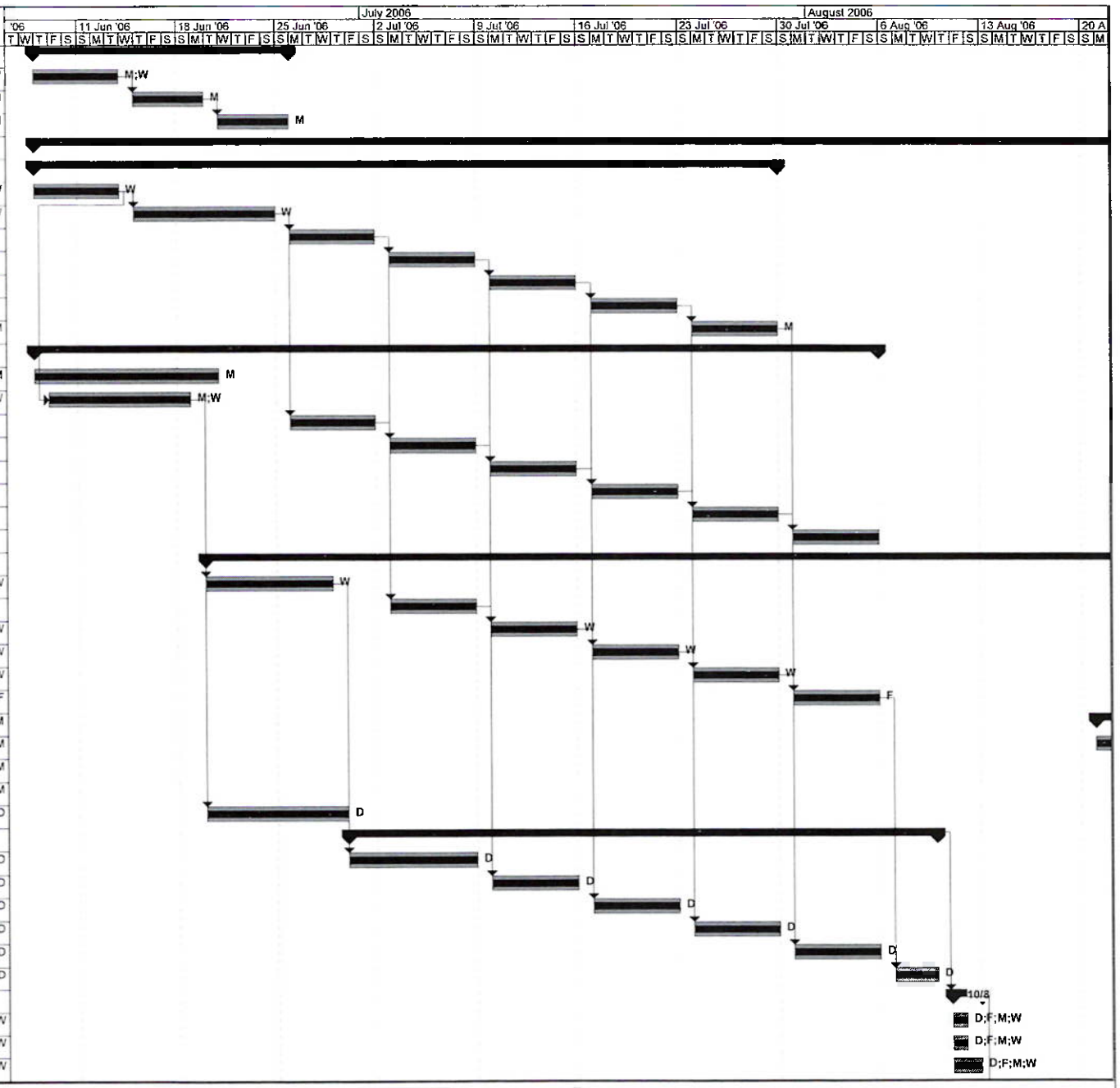
Feature	*Significado*
w=casa	O próprio token
p=uma	Token anterior
t=DET_F_S	Tag escolhida para a token anterior
pp=procurar	Token anterior da anterior
t2=V_INF_,DET_F_S	Tag para o token anterior ao anterior
n=.	Próximo token.
nn=*SE*	Próximo é começo fim de sentença.

As *features* seriam processadas pelo *Maxent*, comparadas com os contextos armazenados no modelo, concluiria que "casa", neste contexto, é:

substantivo feminino singular

APÊNDICE C – Cronograma

ID	Task Name	Duration	Start	Finish	Predec	Resource Initials
1	**RUP Phases Planning**	**19 days**	**Wed 7/6/06**	**Sun 25/6/06**		
2	Elaboration	7 days	Wed 7/6/06	Tue 13/6/06		M;W
3	Construction	6 days	Wed 14/6/06	Mon 19/6/06	2	M
4	Transition	6 days	Tue 20/6/06	Sun 25/6/06	3	M
5	**Elaboration**	**103 days**	**Wed 7/6/06**	**Sun 17/9/06**		
6	**Requirements**	**53 days**	**Wed 7/6/06**	**Sat 29/7/06**		
7	Corpus Access	7 days	Wed 7/6/06	Tue 13/6/06		W
8	Filters	11 days	Wed 14/6/06	Sat 24/6/06	7	W
9	Dictionaries (OpenNLP)	7 days	Sun 25/6/06	Sat 1/7/06	8	
10	Machine Learning (Oper	7 days	Sun 2/7/06	Sat 8/7/06	9	
11	Tagger	7 days	Sun 9/7/06	Sat 15/7/06	10	
12	Chunker	7 days	Sun 16/7/06	Sat 22/7/06	11	
13	Rule Applier	7 days	Sun 23/7/06	Sat 29/7/06	12	M
14	**Modeling**	**60 days**	**Wed 7/6/06**	**Sat 5/8/06**		
15	UML modeling tool sele	14 days	Wed 7/6/06	Tue 20/6/06		M
16	**Corpus Access**	11 days	Thu 8/6/06	Sun 18/6/06	7	M;W
17	Filters	7 days	Sun 25/6/06	Sat 1/7/06	8	
18	Dictionaries	7 days	Sun 2/7/06	Sat 8/7/06	9	
19	**Machine Learning**	7 days	Sun 9/7/06	Sat 15/7/06	10	
20	Tagger	7 days	Sun 16/7/06	Sat 22/7/06	11	
21	Chunker	7 days	Sun 23/7/06	Sat 29/7/06	12	
22	**Rule Applier**	7 days	Sun 30/7/06	Sat 5/8/06	13	
23	**Coding**	**91 days**	**Mon 19/6/06**	**Sun 17/9/06**		
24	Corpus Access	10 days	Mon 19/6/06	Wed 28/6/06	16	W
25	Filters	7 days	Sun 2/7/06	Sat 8/7/06	17	
26	Dictionaries	7 days	Sun 9/7/06	Sat 15/7/06	18	W
27	Machine Learning	7 days	Sun 16/7/06	Sat 22/7/06	19	W
28	Tagger	7 days	Sun 23/7/06	Sat 29/7/06	20	W
29	Chunker	7 days	Sun 30/7/06	Sat 5/8/06	21	F
30	**Rule Applier**	**29 days**	**Sun 20/8/06**	**Sun 17/9/06**		M
31	Technology	14 days	Sun 20/8/06	Sat 2/9/06		M
32	Port	4 days	Sun 3/9/06	Wed 6/9/06	31	M
33	Applier	11 days	Thu 7/9/06	Sun 17/9/06	32	M
34	Database technology selecti	11 days	Mon 19/6/06	Thu 29/6/06	16	D
35	**Testing**	**42 days**	**Thu 29/6/06**	**Wed 9/8/06**		
36	Corpus Access	10 days	Thu 29/6/06	Sat 8/7/06	24	D
37	Filters	7 days	Sun 9/7/06	Sat 15/7/06	25	D
38	Dictionaries	7 days	Sun 16/7/06	Sat 22/7/06	26	D
39	Machine Learning	7 days	Sun 23/7/06	Sat 29/7/06	27	D
40	Tagger	7 days	Sun 30/7/06	Sat 5/8/06	28	D
41	Chunker	4 days	Sun 6/8/06	Wed 9/8/06	29	D
42	**Lifecycle Architecture Mile**	**3 days**	**Thu 10/8/06**	**Sat 12/8/06**	**35**	
43	Check Vision	2 days	Thu 10/8/06	Fri 11/8/06		D;F;M;W
44	Check Requirements	2 days	Thu 10/8/06	Fri 11/8/06		D;F;M;W
45	Architecture validation	3 days	Thu 10/8/06	Sat 12/8/06		D;F;M;W



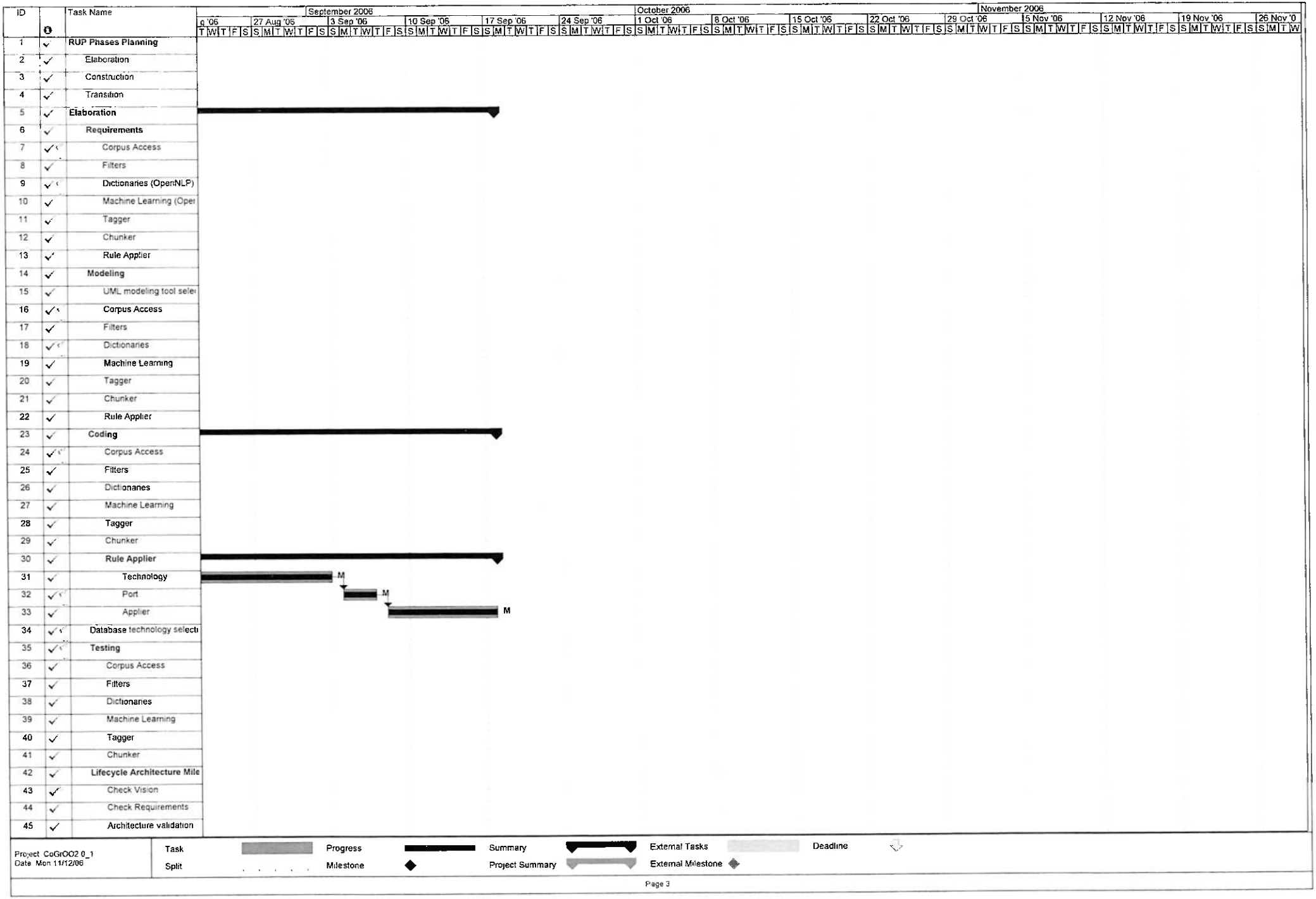
Project: CoGrOO2 0_1
Date: Mon 11/12/06

Task | Split | Progress | Milestone | Summary | Project Summary | External Tasks | External Milestone | Deadline

ID		Task Name	Duration	Start	Finish	Predecessors	Resource Initials
46	✓	**Construction**	**64 days**	**Sun 13/8/06**	**Sun 15/10/06**	42	
47	✓	**Requirements**	**49 days**	**Sun 13/8/06**	**Sat 30/9/06**		
48	✓	Corpus Access	7 days	Sun 13/8/06	Sat 19/8/06		W
49	✓	Filters	7 days	Sun 20/8/06	Sat 26/8/06	48	W
50	✓	Dictionaries	7 days	Sun 27/8/06	Sat 2/9/06	49	
51	✓	Machine Learning	7 days	Sun 3/9/06	Sat 9/9/06	50	
52	✓	Tagger	7 days	Sun 10/9/06	Sat 16/9/06	51	
53	✓	Chunker	7 days	Sun 17/9/06	Sat 23/9/06	52	
54	✓	Rule Applier	7 days	Sun 24/9/06	Sat 30/9/06	53	M
55	✓	**Modeling**	**49 days**	**Sun 20/8/06**	**Sat 7/10/06**		
56	✓	Corpus Access	7 days	Sun 20/8/06	Sat 26/8/06	48	F,M
57	✓	Filters	7 days	Sun 27/8/06	Sat 2/9/06	49	F,M
58	✓	Dictionaries	7 days	Sun 3/9/06	Sat 9/9/06	50	
59	✓	Machine Learning	7 days	Sun 10/9/06	Sat 16/9/06	51	
60	✓	Tagger	7 days	Sun 17/9/06	Sat 23/9/06	52	
61	✓	Chunker	7 days	Sun 24/9/06	Sat 30/9/06	53	
62	✓	Rule Applier	7 days	Sun 1/10/06	Sat 7/10/06	54	M
63	✓	**Coding**	**46 days**	**Sun 27/8/06**	**Wed 11/10/06**		
64	✓	Corpus Access	7 days	Sun 27/8/06	Sat 2/9/06	56	W
65	✓	Filters	7 days	Sun 3/9/06	Sat 9/9/06	57	
66	✓	Dictionaries	7 days	Sun 10/9/06	Sat 16/9/06	58	
67	✓	Machine Learning	7 days	Sun 17/9/06	Sat 23/9/06	59	
68	✓	Tagger	7 days	Sun 24/9/06	Sat 30/9/06	60	
69	✓	Chunker	7 days	Sun 1/10/06	Sat 7/10/06	61	
70	✓	Rule Applier	4 days	Sun 8/10/06	Wed 11/10/06	62	M
71	✓	**Testing**	**43 days**	**Sun 3/9/06**	**Sun 15/10/06**		
72	✓	Corpus Access	7 days	Sun 3/9/06	Sat 9/9/06	64	
73	✓	Filters	7 days	Sun 10/9/06	Sat 16/9/06	65	
74	✓	Dictionaries	7 days	Sun 17/9/06	Sat 23/9/06	66	
75	✓	Machine Learning	7 days	Sun 24/9/06	Sat 30/9/06	67	
76	✓	Tagger	7 days	Sun 1/10/06	Sat 7/10/06	68	
77	✓	Chunker	7 days	Sun 8/10/06	Sat 14/10/06	69	F
78	✓	Rule Applier	4 days	Thu 12/10/06	Sun 15/10/06	70	M
79	✓	Beta packing (installer)	15 days	Sun 1/10/06	Sun 15/10/06		M
80	✓	**Transition**	**14 days**	**Mon 16/10/06**	**Sun 29/10/06**	46	
81	✓	Packing (installer)	14 days	Mon 16/10/06	Sun 29/10/06		M
82	✓	Beta test (user view)	14 days	Mon 16/10/06	Sun 29/10/06		D,F,W
83	✓	Beta test (build stability)	14 days	Mon 16/10/06	Sun 29/10/06		D,F,W
84	✓	**Documentation**	**39 days**	**Mon 16/10/06**	**Thu 23/11/06**	46	
85	✓	RUP documents revision	14 days	Mon 16/10/06	Sun 29/10/06		D,F,M,W
86	✓	**Final Report**	**39 days**	**Mon 16/10/06**	**Thu 23/11/06**		
87	✓	Scope	4 days	Mon 16/10/06	Thu 19/10/06		D,F,M,W
88	✓	Definition of the docume	4 days	Mon 16/10/06	Thu 19/10/06		D,F,M,W
89	✓	Text writing	35 days	Fri 20/10/06	Thu 23/11/06	87,88	D,F,M,W

Project: CoGrOO2 0_1
Date: Mon 11/12/06

Task	Progress	Summary	External Tasks	Deadline
Split	Milestone	Project Summary	External Milestone	


ID
Task Name
September 2006
October 2006
November 2006
6 '06
27 Aug '06
3 Sep '06
10 Sep '06
17 Sep '06
24 Sep '06
1 Oct '06
8 Oct '06
15 Oct '06
22 Oct '06
29 Oct '06
5 Nov '06
12 Nov '06
19 Nov '06
26 Nov '0
1 RUP Phases Planning
2 Elaboration
3 Construction
4 Transition
5 Elaboration
6 Requirements
7 Corpus Access
8 Filters
9 Dictionaries (OpenNLP)
10 Machine Learning (Oper
11 Tagger
12 Chunker
13 Rule Applier
14 Modeling
15 UML modeling tool selec
16 Corpus Access
17 Filters
18 Dictionaries
19 Machine Learning
20 Tagger
21 Chunker
22 Rule Applier
23 Coding
24 Corpus Access
25 Filters
26 Dictionaries
27 Machine Learning
28 Tagger
29 Chunker
30 Rule Applier
31 Technology
32 Port
33 Applier
34 Database technology selecti
35 Testing
36 Corpus Access
37 Filters
38 Dictionaries
39 Machine Learning
40 Tagger
41 Chunker
42 Lifecycle Architecture Mile
43 Check Vision
44 Check Requirements
45 Architecture validation
M
M
M
Project: CoGrOO2 0_1
Date: Mon 11/12/06
Task
Split
Progress
Milestone
Summary
Project Summary
External Tasks
External Milestone
Deadline


ID
Task Name
September 2006
October 2006
November 2006
27 Aug '06
3 Sep '06
10 Sep '06
17 Sep '06
24 Sep '06
1 Oct '06
8 Oct '06
15 Oct '06
22 Oct '06
29 Oct '06
5 Nov '06
12 Nov '06
19 Nov '06
26 Nov '0
46 Construction
47 Requirements
48 Corpus Access
49 Filters
50 Dictionaries
51 Machine Learning
52 Tagger
53 Chunker
54 Rule Applier
55 Modeling
56 Corpus Access
57 Filters
58 Dictionaries
59 Machine Learning
60 Tagger
61 Chunker
62 Rule Applier
63 Coding
64 Corpus Access
65 Filters
66 Dictionaries
67 Machine Learning
68 Tagger
69 Chunker
70 Rule Applier
71 Testing
72 Corpus Access
73 Filters
74 Dictionaries
75 Machine Learning
76 Tagger
77 Chunker
78 Rule Applier
79 Beta packing (installer)
80 Transition
81 Packing (installer)
82 Beta test (user view)
83 Beta test (build stability)
84 Documentation
85 RUP documents revision
86 Final Report
87 Scope
88 Definition of the docume
89 Text writing
W
F;M
M
D;F;W
D;F;M;W
Project: CoGrOO2 0_1
Date: Mon 11/12/06
Task
Split
Progress
Milestone
Summary
Project Summary
External Tasks
External Milestone
Deadline

7 **Corpus Access**
Corpus Access was not really done in 1 day.
William had already done it before the elaboration of this cronogram

9 **Dictionaries (OpenNLP)**
OpenNLP - Maximum Entropy training

16 **Corpus Access**
CA will be done experimentally to see how much time does the modelling of a module actually takes. It will also serve as a bulletproof test to William CA requirements planning.

18 **Dictionaries**
This hiatus between CA and the rest of the modelling was designed mostly because the semestral alumni will be in the end of 9th term period, which is full of tests and projects

24 **Corpus Access**
- Bulletproof test to see whether the modelling is adequate or not. to know if M&F's modelling enables a good coding procedure
- CA will be done mostly by Marcelo.

32 **Port**
Port rules from the old CoGrOO format to the new format.

34 **Database technology selection**
Possibly none required anymore
We intend to adopt the technology that is used by OpenNLP

35 **Testing**
9/1 tests = cut corpus in 10 parts, use 9 for training and use 1 as an input to test the functional module.

71 **Testing**
This time the test will be done changing the part used as an input. Each module will be tested 10 times total.

89 **Text writing**
This task will be expanded in the future.

APÊNDICE D – Conteúdo do CD

Pastas e arquivos importantes

./cogroo/data

Pasta onde ficam os arquivos necessários para a geração de dicionários e modelos.

- abr.txt
  - lista de abreviaturas.
- dic.txt
  - texto do dicionário morfológico.
- flex.txt
  - texto do dicionário de flexões.
- corpus_cut
  - corpus CentenFolha 1.0 reformatado.

./cogroo/models/cogroo/portuguese

Pasta com os modelos e dicionários usados na execução.

./cogroo/src

Códigos-fonte do corretor gramatical.

./cogroo/bin

Binários do corretor gramatical.

./cogroo/lib

Bibliotecas necessárias para execução ou treinamento.

./cogroo-gui

Códigos-fonte da interface gráfica.

Arquivos de Configurações

./cogroo/src/br/usp/pcs/lta/cogroo/cogroo.properties

Configurações de treinamento e execução. Indica nomes dos dicionários e modelos.

Indica configurações de treinamento.

./cogroo/src/log4j.properties

Configurações do logger

Scripts

./cogroo/CogrooServer.bat ou ./cogroo/CogrooServer.sh

Inicia servidor Corretor Gramatical (para o OpenOffice.org).

./cogroo/CogrooCommandLine.bat ou ./cogroo/CogrooCommandLine.sh

Inicia CoGrOO em linha de comando: o usuário pode digitar sentenças na linha de

comando para o CoGrOO corrigir.

./cogroo/CogrooViewer.bat ou ./cogroo/CogrooViewer.sh

Inicia CoGrOO com interface gráfica.

./cogroo/CogrooTrain.bat e ./cogroo/CogrooTrain.sh

Gera novos modelos e dicionários (pode levar horas).

./cogroo/createCogrooModelFolders.bat e

./cogroo/createCogrooModelFolders.sh

Cria pastas para os modelos.

MANUAL DO USUÁRIO

Instalação da GUI no OpenOffice.org Writer

1. Abrir o OpenOffice.org Writer

2. Ir em Ferramentas > Gerenciador de pacotes

3. Clicar em Adicionar

4. Selecionar o arquivo CoGrOO_v2.0.zip e clicar em Abrir

5. Fechar o Gerenciador

6. Reiniciar o OpenOffice.org Writer

7. Ao final destes passos, terão sido criados uma barra de ferramentas com um botão para acessar o corretor e um menu CoGrOO.

Usando o CoGrOO no OpenOffice.org Writer

1. No menu CoGrOO, selecione Configurações

2. Clique Default e depois OK. Caso use um computador remoto como servidor, digite o IP deste.

3. Certifique-se que o servidor CoGrOO esteja sendo executado. Veja no item Scripts como proceder.

4. Pronto, basta executar o corretor clicando no botão ou selecionando no menu CoGrOO.