FILIPE MIGUEL CASSAPO

UMA SOCIEDADE MULTIAGENTE PARA O MAPEAMENTO AUTOMÁTICO INTELIGENTE DE COMPETÊNCIAS EM AMBIENTE DE COLABORAÇÃO.

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

CURITIBA

2004

FILIPE MIGUEL CASSAPO

UMA SOCIEDADE MULTIAGENTE PARA O MAPEAMENTO AUTOMÁTICO INTELIGENTE DE COMPETÊNCIAS EM AMBIENTE DE COLABORAÇÃO.

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática Aplicada.

Área de Concentração: *Metodologias e Técnicas de Computação*

Orientador: Prof. Dr. Edson Emilio Scalabrin

CURITIBA

2004

Cassapo, Filipe Miguel
Uma Sociedade Multiagente para o Mapeamento Automático Inteligente de Competências em Ambiente de Colaboração. Curitiba, 2004. 310p.

Dissertação – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1.Sistemas Multiagentes Autopoiéticos 2.Mapeamento de Competências 3.Inteligência Artificial 4.Hermenêutica Computacional. I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada II-t

Esta página deve ser reservada à ata de defesa e termo de aprovação que serão fornecidos pela secretaria após a defesa da dissertação e efetuadas as correções solicitadas.

À Minha mãe Gabriela, com todo o meu amor,
Ao amor da minha vida, a minha esposa Danielle,
À Minha querida irmã, Marie-Luce,
Ao meu querido irmão, Marco.

Agradecimentos

Além de todas as pessoas que contribuíram à levar este trabalho a bom termo, agradeço particularmente (em ordem alfabético):

O Prof. Dr. Bortolo Valle, pelo apoio na revisão da conceituação filosófica do trabalho.

O Sr. Edgard R. May, pelo apoio dado através da disponibilização de material sobre a história das ciências, bem como pelo apoio na revisão final deste trabalho.

O Prof. Dr. Edson E. Scalabrin, meu orientador, pelo eficiente e pronto direcionamento na elaboração de toda a pesquisa, bem como pelo forte apoio na participação dos vários congressos nacionais e internacionais onde este trabalho foi apresentado.

O Sr. Jefferson M. Pellissari, a Sra. Márcia B. Cavalcantes, e o Sr. Sandro M. Melhoretto pelo apoio dado na revisão da versão final deste trabalho.

O Pof Dr. José Claudio C. Terra, pelos conselhos, bem como pelas várias oportunidades dadas na vasta rede da Gestão do Conhecimento no Brasil.

O Prof Dr. Marcos H. Schmei, pela visão e pelos ensinamentos.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

ATN(s)	*Augmented Transition Networks* (Redes de Transição Aumentadas)
C.N.R.S.	*Centre National de Recherche Scientifique* (Centro Nacional de Pesquisa Científica)
CoP	Comunidade de Prática (*Community of Practice*)
CR	Computo-Representacionalismo
DAI	*Distributed Artificial Intelligence* (Inteligência Artificial Distribuida)
DC(s)	Dependência(s) Conceitual(ais)
GC	Gestão do Conhecimento (*Knowledge Management*)
ELI	*English Language Interpreter* (Interpretador da Linguagem Inglesa)
HMM	Hidden Markov Models (Modelos de Markov Escondidos)
IA	Inteligência Artificial
IAD	Inteligência Artificial Distribuída
IAS	Inteligência Artificial Simbólica
LAN	*Local Area Network* (Rede Local)
LP	Lógica dos Predicados
MAS	*Multiagent System* (Sistema Multiagente)
M.I.T.	Massachusetts Institute of Technology
MM	Markov Models (Modelos de Markov)
PIB	Produto Interior Bruto
PLN	Processamento de Linguagem Natural
RS(s)	Redes Semânticas
SMA	Sistema Multiagentes (*Multiagent System*)
TI	Tecnologia da Informação (*Information Technology*)

Resumo

O ativo *conhecimento* é o único ativo que se que se multiplica quando divido. Este é o ponto de partido essencial de toda estratégia voltada ao crescimento do Capital Intelectual em uma organização. Ao contrário dos ativos tangíveis, que perdem valor à medida que são compartilhados, os ativos intangíveis (essencialmente, as competências, experiências, modelos mentais e atitudes) aumentam seu valor à medida que são distribuídos entre seres cognitivos. Em uma economia cada vez mais governada pela agregação de inteligência à produtos, serviços e processos, e onde a inovação *em tempo real* é chave para o sucesso dos negócios, a chamada "Gestão do Conhecimento" aparece como *a* nova metodologia essencial de gerenciamento das organizações do século XXI. Para poder, porém, implementar tecnologias de suporte para as práticas de compartilhamento de conhecimento, e aplicar metodologias eficientes em conjunto com estas tecnologias, é necessário ter um entendimento claro da natureza da cognição. Centenas de ano de pesquisas epistemológicas não esgotaram as perguntas essenciais, que continuam sendo feitas por trás da chamada "Gestão do Conhecimento": o que é o *Conhecimento*? Como pode ser *naturalizado*? Este trabalho, sustentado por um estudo epistemológico das noções de conhecimento, percepção, interação lingüística, e inteligência, visa portanto propor a teoria biológica construtivista da agência autônoma (a *Autopoiésis*) como novo quadro epistemológico para as Ciências Cognitivas e a Inteligência Artificial, e mostrar como a implementação e uso de sistemas multiagentes (ou sociedade de agentes de software) pode efetivamente permitir à artefatos tecnológicos (essencialmente nesta pesquisa, os computadores) adquirir uma dimensão hermenêutica em sociedades mistas compostas por pessoas e tecnologias, operacionalizando compartilhamentos de conhecimento à volta de "Mapeamentos Automáticos Inteligentes". Os "Mapeamentos Automáticos Inteligentes" propostos serão definidos a partir das características de serem (i) **construídos de forma automática**, (ii) **de entendimento comum entre todos os agentes** envolvidos no seu uso, (iii) **completos** e **consistentes**, (iv) **atualizados em *tempo real* em**

uma "memória corporativa dinâmica", e, principalmente, (vi) capazes de fornecer **de forma pró-ativa, em uma linguagem natural, informações pertinentes, personalizadas e contextualizadas**. O trabalho experimental relatado, organizado em torno de dois experimentos visando a validar o paradigma epistemológico proposto, focará especificamente na constituição de mapeamentos de competências. As interações constitutivas da inteligência dos mapeamentos discutidos nesta dissertação serão o fruto de um ambiente de colaboração computacional distribuído (Bate-Papo e Comunidades de Práticas). Sendo a pesquisa necessária à realização de mapeamentos automáticos inteligentes, inserida na intersecção de múltiplas disciplinas, como a epistemologia, a psicologia, a lingüística, as ciências da computação, a sociologia e a gestão, as conclusões alcançadas ao termo deste trabalho transcenderão aspectos tecnológicos da informática: além de sustentar a escolha da *Autopoiésis*, como quadro adequado de trabalho para o objetivo de pesquisar o que são "máquinas inteligentes", e ter mostrado pragmaticamente, através de dois experimentos, como sistemas multiagentes autopoiéticos são efetivamente capazes de mapear de forma inteligente e automática conhecimentos, esta pesquisa levará também à uma série de conclusões sobre a natureza da cognição, e da sua construção.

Palavras-chave: 1. Sistemas Multiagentes Autopoiéticos. 2. Mapeamento de Competências. 3. Inteligência Artificial. 4. Hermenêutica Computacional.

Abstract

Knowledge is the only asset which multiplies when shared. This is the essential starting point of any strategy aiming at enabling the growth of the intellectual capital of an organization. Unlike tangible assets, which loose value when they are shared, intangible assets (essentially, competences, experiences, mental models and attitudes) increase their value as they are distributed among cognitive beings. In an economy which appears to be clearly governed by knowledge-intensive products, services and processes, and where *real-time* innovation is required to enable businesses' success, the so-called "Knowledge Management" seems to be *the* new essential methodological tendency for managing the XXIst-century organizations. However in order to implement technologies for supporting knowledge-sharing practices, and to apply efficient methodologies together with these technologies, it is necessary to have a clear understanding of the nature of cognition. Hundreds of years of epistemological researches have not run out the essential questions which continue being asked behind the idea of "Managing Knowledge": what is *knowledge*? How can it be *naturalized*? Based on an epistemological study of the notions of knowledge, perception, linguistic interaction, and intelligence, this work aims at proposing the constructivist biological theory of autonomous agency (the Autopoiesis) as a new framework for both Cognitive Sciences and Artificial Intelligence, as well as to show how the implementation and the use of multi-agent systems (also called software agents' societies) can effectively allow technological artifacts (essentially, inside this work, computers) to acquire a hermeneutic dimension in human-and-technologies' societies, and operationalize knowledge sharing around "Intelligent Automatic Knowledge Mapping". Such "Intelligent Automatic Maps" will be defined with the following properties: they will be (i) **automatically constructed**, (ii) **commonly-understood by all the agents** involved in their use, (iii), **complete and consistent**, (iv) **actualized in *real-time* inside "dynamic corporate memories"**, and, mainly, (v) capable of **providing pro-actively and in natural language, relevant, personalized and**

contextualized pieces of information. The experimental work which will be described, organized around two scenarios aiming at validating the proposed paradigm, will specifically be focused on competence-maps construction. The interactions, which will be constitutive of the maps' intelligence, will be the result of a distributed computational collaborative environment (Chats and Communities of Practice). Since the research work necessary for creating intelligent automatic maps lays at the intersection of multiple disciplines, like epistemology, psychology, linguistics, computer science, sociology, and management, the conclusions we will propose at the end of this dissertation will transcend simple technological aspects of computer science: additionally to the fact of choosing the Autopoiesis as the adequate framework for intelligent-machines-related research-fields, and of pragmatically showing through two experiments how autopoietic multi-agent systems can effectively map knowledge in an intelligent automatic way, this work will also bring some conclusions on the nature of cognition as well as its construction.

Keywords: 1. Multi-agent Autopoietic Systems. 2. Competence Mapping. 3. Artificial Intelligence. 4. Computational Hermeneutics.

Résumé

La *connaissance* est l'unique actif qui se multiplie lorsqu'il est divisé. Tel est le point de partie essentiel de toute stratégie dirigée vers la croissance du Capital Intellectuel d'une organisation. A l'inverse des actifs tangibles, qui perdent leur valeur à mesure qu'ils sont partagés, les actifs intangibles (essentiellement, les compétences, les expériences, les modèles mentaux et les attitudes) augmentent leur valeur lorsqu'ils sont distribués parmi les êtres cognitifs. Dans le contexte d'une économie gouvernée, de forme toujours plus évidente, par l'agrégation d'intelligence aux produits, services et processi, et où l'innovation *en temps réel* est clef pour le succès des affaires, la "Gestion des Connaissances" apparaît comme *la* nouvelle tendance méthodologique de gestion des entreprises du XXIe siècle. Pour pouvoir, cependant, construire de nouvelles technologies de support à la pratique de la Gestion des Connaissances, et appliquer les méthodologies adéquates à l'utilisation de ces technologies, il est nécessaire d'avoir une compréhension claire de la nature de la cognition. Plusieurs centaines d'années de recherches épistémologiques n'ont pas épuisé les questions essentielles qui continuent de se poser en arrière plan de la «Gestion des Connaissances » : qu'est-ce que la *connaissance*? Comment peut-elle être *naturalisée*? Ce travail, basé sur une étude épistémologique des notions de connaissance, perception, interaction linguistique et intelligence, se propose pourtant d'adopter la théorie biologique constructiviste de l'agence autonome (l'Autopoièse) comme nouveau cadre de travail pour les sciences cognitives et l'intelligence artificielle, et de montrer comment la construction et l'utilisation de systèmes multi-agents (ou sociétés d'agents logiciels) peut effectivement permettre à l'artefact technologique (essentiellement, pour ce travail, l'ordinateur) d'acquérir une dimension herméneutique au cœur de sociétés mixtes composées de personnes et de technologies, permettant, de cette façon, l'opérationnalisation du partage des connaissance autour de « Cartographies Automatiques Intelligentes de Connaissances». Les « Cartographies Automatiques Intelligentes » que nous proposerons seront définies à partir des propriétés

suivantes : elles seront (i) **construites de forme automatique**, (ii) **de compréhension commune entre tous les agents** concernés par leur utilisation, (iii) **complètes et consistantes**, (iv) **actualisées *en temps réel* dans une « mémoire corporative dynamique»**, et, principalement, (v) capables de **fournir pro-activement, et en langue naturelle, des informations pertinentes, personnalisées et contextuelles**. Le travail expérimental que nous relaterons, organisé autour de deux scénarios visant à valider le paradigme épistémologique défendu, sera focalisé sur la construction de cartographie de compétences. Les interactions constitutives de l'intelligence des cartographies discutées dans cet essai seront le fruit d'un milieu distribué de collaboration informatique (Conversations *on-line* et Communautés de Pratique). Etant donné que les recherches nécessaires à la création de cartographies automatiques intelligentes se trouvent à l'intersection de multiples disciplines, comme l'épistémologie, la psychologie, la linguistique, l'informatique, la sociologie et la gestion, les conclusions présentées au terme de ce travail transcenderont les simples aspects technologiques de l'informatique: en plus de défendre le choix de l'Autopoièse pour atteindre l'objectif de savoir ce que seraient des « machines intelligentes », et de montrer de forme pragmatique, au travers de deux expériences, comment les systèmes multiagents autopoiétiques sont effectivement capables de cartographier de forme intelligente et automatiques des connaissances, ce travail nous permettra également de mettre en évidence un certain nombre de conclusion au sujet de la nature de la cognition et de sa construction.

Mots-clefs: 1. Systèmes Muliagents Autopoiétiques. 2. Cartographie de Compétences. 3. Intelligence Artificielle. 4. Herméneutique Computationnelle.

Capítulo 1

Introdução

Saber, e, logo, poder agir adequadamente, sempre foi percebido como um diferencial nas sociedades, nas organizações, e para os próprios seres. Por esta razão, da filosofia ântica à epistemologia moderna, a natureza da cognição sempre constituiu um assunto de pesquisa essencial. Com a entrada da nossa sociedade em uma economia desmaterializada, movida pelo valor dos ativos intangíveis[1], as preocupações pelo conhecimento e seu gerenciamento nunca estiveram tão presentes[2]. Apesar de parecer uma preocupação recente, a chamada "Gestão do Conhecimento" (GC) não constitui uma novidade: ela sempre foi desempenhada nas sociedades, de forma oral ou por meio de artefatos (tabletes de barro, livros, ou computadores). O que gerou efetivamente a revelação das "sociedades do conhecimento"[3], é a aparição de uma capacidade tecnológica nova em lidar de forma sistemática com a informação: existe hoje a capacidade operacional de obter instantaneamente informações, à qualquer momento, em qualquer lugar. Para poder, porém, sustentar metodologias de GC, reflexões epistemológicas e pesquisas tecnológicas são necessárias. **Estes serão os objetivos deste trabalho: propor a teoria da Autopoiésis como novo quadro epistemológico de**

[1] Karl E. SVEIBY, em "The new organizational wealth." [SVE98], propõe o conceito de "capital intelectual", como diferença entre o valor de mercado das organizações e seu valor contábil. O "ativo intangível" constituindo tal capiral é dividido em Estrutura Externa (marca, relação com cliente, etc.), Estrutura Interna (processos, infra-estruturas de informação, etc.), e Competência Individual (treinamentos, experiências, etc.)

[2] A literatura que pode ser consultada a respeito é vasta. Pode-se recomendar, para uma boa introdução, "Working Knowledge: How Organizations Manage What They Know" de DAVENPORT [DAV98], "The Knowledge-Creating Company" de NONAKA [NON95], e "The fifth discipline: The art and the practice of the Learning Organization" de SENGE [SEN90].

[3] A expressão "sociedade do conhecimento" foi proposta em 1990 pelo reconhecido filósofo da administração Alvin Toffler. Segundo este teórico, os ativos intangíveis ganharam importância e aquilo que está presente na organização, mas não se materializa, se tornou sua principal fonte de riqueza [TOF90].

pesquisa, e apresentar as sociedades de agentes de software como aplicação pragmática deste novo paradigma.

1.1. Gestão do Conhecimento e Competitividade no século XXI

Segundo um estudo realizado pelo IDC [IDC99], o re-trabalho intelectual e a ineficiência no uso dos ativos informacionais custou mais de US$ 30 bilhões em 2003 as "500 *Fortunes*", sendo que os trabalhadores do conhecimento representam hoje, em media, 40% da força de trabalho[4]. A evolução do conhecimento humano e os êxitos tecnológicos decorrentes são eles mesmos constitutivos da nova competitividade sustentada por novos modelos de gestão. Terra, em [TER02], p.26-28, argumenta que ser competitivo no século XXI significa:

- **Produzir soluções e serviços ricos em conhecimento**: o poder de compra de bens manufaturados caiu de aproximadamente 75% nos últimos 40 anos, enquanto que os preços de produtos de saúde e educação (ricos em conhecimento) cresceram aproximadamente três vezes mais do que a inflação.[5]
- **Ser capaz de inovar de forma contínua**: um estudo de Reichheld[6] sobre empresas de alta tecnologia mostrou que os produtos lançados nos últimos cinco anos são responsáveis por 49,1% das vendas dos líderes do mercado, enquanto que os mesmos são responsáveis por apenas 10,7% das vendas totais para o grupo de companhias com menos de 30% de fatia de mercado.
- **Investir no capital intelectual**: foi mostrado que para cada ano adicional de estudos aumenta a produtividade de 8,5% no setor industrial e de 13% no setor de serviços.[7]
- **Proteger o capital intelectual**: as receitas de exportações com *royalties* e licenciamentos nos Estados Unidos aumentaram de US$ 16,6 bilhões em 1990 para US$ 27,3 bilhões em 1996[8]. O número de aplicações mundiais de patentes foi multiplicado por 7 entre 1985 e 2000[9].

---

[4] "IDC estimates that the Fortune 500 companies are operating at a $12 billion knowledge deficit in 1999. This figure is expected to increase to $31.5 billion by 2003 as the percentage of knowledge workers in the work force increases from 20% in 1999 to over 40% in 2003", [IDC99], p. 2.
[5] Dados estratos de "The manufacturing paradox", The Economist, 1º de Novembro de 2001.
[6] Reicheld, F., The Loyality Effect, Havard Business Review, 1996.
[7] The Economist, Setembro 1996, p. 12.
[8] Maskus, K., Intellectual Property Rights in the Global Economy, Institute of International Economics, 2000, p.8
[9] Ver World Intellectual Property Organization, 1985 e 2000.

1.2. Necessidade de Mapeamentos Automáticos e Inteligentes

Neste contexto, investimentos em tecnologias *inteligentes* da informação são fundamentais[10]. Precisa-se entender, por "*inteligentes*", o fato que tais tecnologias devem ter o objetivo, não de fornecer caoticamente sempre mais informações, mas, ao contrário, de evitar a sobrecarga cognitiva dos trabalhadores intelectuais. Com o número crescente de pessoas atualmente "conectadas" à Internet (em 2002 cerca de 460 milhões[11]), a quantidade de informações disponíveis e acessíveis mundialmente está também crescendo de forma exponencial: **a Internet possuía em 2002 cerca de 17 bilhões de páginas disponíveis com um crescimento 7,3 milhões de páginas por dia[12]. Estes números permitem entender que mapeamentos e resgates não automatizados e não personalizados de informações estão se tornando completamente ineficientes e inviáveis economicamente.**

No século XXI, deve-se esperar de um mapeamento de informações realmente eficiente no suporte de uma estratégia de transferência do conhecimento que seja:

i. **Construído de forma automática**;
ii. **De entendimento comum entre todos os agentes** envolvidos no seu uso;
iii. **Completo** (possuindo todas as informações disponíveis) e **consistente** (todas as informações que possui são o reflexo da realidade);
iv. **Atualizado em "tempo real"** e **disponível em uma "memória corporativa dinâmica"[13];**
v. Capaz de fornecer de forma **pró-ativa, em linguagem natural,** informações **pertinentes**, **personalizadas** e **contextualizadas**.

Mapeamentos respondendo à estas características serão chamados neste trabalho de "Mapeamentos Automáticos Inteligentes". Eles deverão ser capazes de observar o mundo (real ou virtual), para perceber as informações presentes, detectar os interesses dos agentes (humanos ou virtuais) envolvidos, e fornecer pró-ativamente, em linguagem natural, informações personalizadas, nos momentos os mais adequados para os agentes envolvidos no seu uso.

---

[10] São investidos cerca de US$ 7 bilhões por ano em TI nos Estados Unidos (cerca de 7% do PIB), ver PC Magazine, iBiz, 21 de Julho 2001 p.7.
[11] Ver TEHAN, R. Internet Statistics: Explanation and Sources in CRS Report for Congress, US Department of State, 2002., p. 6.
[12] TEHAN, R. Internet Statistics: Explanation and Sources in CRS Report for Congress, US Department of State, 2002., p. 11.
[13] Faze-se aqui referência à noção de "Memória Dinâmica", proposta por Schank em [SCH82] e [SCH99], definindo a cognição como capacidade de experimentação, memorização, e lembrança do memorizado.

1.3. Formulação da Proposta

O objetivo deste trabalho será de propor a teoria da agência autônoma (a Autopoiésis) como novo quadro epistemológico para as Ciências Cognitivas e a Inteligência Artificial, e mostrar como sociedades de agentes de software podem adquirir uma dimensão hermenêutica, produzindo interações pró-ativas entre Humanos e Softwares, e operacionalizando "Mapeamentos Automáticos Inteligentes". Na parte experimental deste trabalho, focaremos especificamente na constituição de mapeamentos de competências. As interações constitutivas da inteligência dos mapeamentos acontecerão em um ambiente de colaboração computacional distribuído (Bate-Papo[14] e Comunidades de Prática[15]). Para suportar interações entre humanos e máquinas, e propiciar o entendimento e a produção de sentenças escritas em linguagem natural, proporemos os princípios de Neutralização Semiológica e Semiotização Natural, como extenções da teoria das dependências conceituais de Schank [SCH81].

1.4. Pesquisa Realizada e Contribuição

Sendo o objetivo deste trabalho inserido na intersecção de múltiplas disciplinas, como a psicologia, a lingüística, a ciência da computação, a sociologia e a gestão, foi necessário, para atingi-lo, encaminhar uma pesquisa multi-disciplinar envolvendo:

i. **A epistemologia e as teorias e filosofias do conhecimento**, de forma a poder abordar adequadamente a noção de conhecimento, para sustentar a tecnologia cognitiva de mapeamento automático inteligente proposta.

ii. **A lingüística**, de forma a poder possibilitar interações semióticas ricas e naturais entre agentes reais e artificiais.

iii. **As ciências cognitivas**, **e, especificamente, a Autopóiesis**, como teoria construtivista da cognição, de forma a poder propiciar um entendimento da essência do ser cognitivo, das noções de "Agente", e "Sociedade de Agentes".

iv. **A inteligência artificial e as ciências da informação e da computação**, para poder operacionalizar, através do uso da tecnologia, um sistema de mapeamento automático inteligente de competências.

---

[14] O Bate-Papo (ou *Chat*) é um ambiente colaborativo distribuído em uma rede de computadores, permitindo "conversar" em tempo real.

[15] Uma "Comunidade de Prática" (ou *Community of Practice* - CoP) é um espaço virtual permitindo o agrupamento assíncrono de pessoas possuindo objetivos comuns a serem desenvolvidos de forma colaborativa.

As principais contribuições desta pesquisa são:

i. **A proposta de um novo quadro teórico de trabalho para as tecnologias inteligentes**, **baseado na teoria cognitiva da *Autopoiésis***, descartando o computo-representacionalismo (CR) e o empirismo ontológico, para tornar-se para a fenomenologia transcendental e para a teoria da autonomia e da enação.

ii. **A proposta e a implementação de uma arquitetura multiagente** baseada no quadro teórico adotado**, permitindo operacionalizar o mapeamento automático inteligente de competências** em qualquer sociedade cognitiva (serão analisados os cenários "Incubadora de Fábricas de Software" e "*Home-care*" [16]).

1.5. Delimitações do Tema

Devem-se ressaltar alguns aspectos teóricos que não serão abordados neste trabalho:

i. **Existência empírica das competências mapeadas**: os agentes implementados não possuirão a capacidade de verificar se as competências que mapeiam são realmente possuídas (empirismo ontológico), ou se trata de quimeras. Os agentes possuirão apenas "crenças" derivadas dos fenômenos percebidos.

ii. **Consistência lógica interna dos mapeamentos**: não será abordada também a coerência interna dos mapeamentos, e não serão tratados inconsistências lógicas ou vínculos lógicos entre as competências mapeadas.

Além destas restrições teóricas, o trabalho será também delimitado por uma série de escolhas empíricas:

i. **Desempenho operacional**: o objetivo da implementação de um protótipo foi de mostrar a realizabilidade computacional do paradigma proposto, sem, porém, levar em consideração problemas de performance ou otimização.

ii. **Implementação do protótipo computacional**: na operacionalização do protótipo, foi optado para a realização de uma arquitetura completa e operacional, envolvendo todos os agentes requeridos, sem, porém, aprofundar a implementação específica de cada agente.

[16] O *Home-care* é uma forma tratamento médico visando a tirar o mínimo o paciente do seu ambiente familial.

1.6. Organização da Dissertação

O trabalho de pesquisa realizado foi globalmente desempenhado em três grandes etapas, expostas através dos três principais capítulos desta dissertação:

- **Capítulo 2, Sustentação Epistemológica**. Nesta parte, será realizada uma apresentação das principais teorias do conhecimento, da fenomenologia ântica à hermenêutica contemporânea, passando pelo cognitivismo clássico. Será também proposta uma análise das principais teorias da lingüística (abordagem estrutural, semiótica, e abordagem transformacional), bem como suas implicações para o processamento de linguagens naturais. Será finalmente detalhada a teoria da Autopóiesis e suas implicações para as noções de Agente, Sociedade de Agentes, Emergência e Autonomia.
- **Capítulo 3, Mapeamento Automático Inteligente de Competências em Ambiente de Colaboração.** Este capítulo detalhará a operacionalização computacional de mapeamentos automáticos inteligentes, aplicando o paradigma escolhido da *Autopoéisis*. Para descrever as enações hermenêuticas de uma sociedade composta por agentes virtuais e humanos, serão expostas duas formas essenciais de interação: a *Neutralização Semiológica* (do real para o artificial) e a *Semiotização Natural* (do artificial para o real).
- **Capítulo 4, Pragmática e Discussão.** Esta última parte visará em propor a análise de dois cenários nos quais foi aplicada experimentalmente a arquitetura de agentes descrita no capítulo 3: um cenário de “Incubadora de Fábricas de Software”, e um outro de “*Home-care*”. Será também discutida a possibilidade de expandir a aplicação da arquitetura à outros domínios cognitivos, bem como outros tipos de tarefas e outras enações semióticas (reconhecimento de voz, deteção de movimentos, atividade cerebral, etc.)

Este trabalho será finalmente concluído por uma série de considerações sobre a natureza da cognição e da agência, bem como a proposta de algumas perspectivas sobre a naturalização da cognição e suas implicações hermenêuticas.

Capítulo 2

Sustentação Epistemológica

Para poder abordar adequadamente a realização de um mapeamento inteligente, capaz de exibir as características de ser (i) **construído de forma automática**, (ii) **de entendimento comum entre todos os agentes** envolvidos no seu uso, (iii) **completo** e **consistente**, (iv) **atualizado em "tempo real" em uma "memória corporativa dinâmica"**, e, principalmente, (vi) capaz de fornecer **de forma pró-ativa, em uma linguagem natural, informações pertinentes, personalizadas e contextualizadas**, é necessário voltar-se para a noção de conhecimento, para não só entender a sua formação, mas também perceber sua transmissão através das interações mediatas entre os seres cognitivos. Portanto, este capítulo proporá uma análise das principais abordagens da noção de conhecimento, bem como da natureza da interação lingüística, para propor a Autopoiésis como escolha paradigmática do caráter da agência cognitiva, apresentar a noção de hermenêutica do objeto tecnológico, e descrever o conhecimento como fenômeno transcendental emergente em uma sociedade autopoiética.

2.1. Teorias do Conhecimento

A principal pergunta que inflama debates filosóficos a cerca da cognição é a do acesso a uma realidade objetiva e compreendida. A partir do momento que o animal, na escala da evolução, se dotou de uma percepção semiótica[17], e aplicou tal percepção a si próprio (processo de auto-consciência), ele começou a se perceber como inserido em um meio que, naturalmente, desejou entender, além de sempre tentar aprimorar sua concepção da natureza e

[17] Chamamos de percepção semiótica a aparição de representações construídas e articuladas, bem como intenções, que podemos aproximadamente situar no coração do processo de hominização, entre os primeiros australopitecos (há 3 milhões de anos), e a desaparição do homo de neandertal (há 50.000 anos).

da posição dele, como ser, neste "mundo" que se oferecia a sua investigação. Serão propostos, a seguir, quatro grandes abordagens da noção de cognição: o conhecimento fenomenológico de Aristóteles, a ruptura Cartesiana e o conhecimento geométrico, o cognitivismo clássico e a naturalização da mente por meio da máquina de Turing, e a fenomenologia contemporânea como proposta de um relativismo hermenêutico do conhecimento.

2.1.1. Conhecimento Fenomenológico e Filosofia Ântica

Não poderíamos abordar a noção de conhecimento sem referenciar a filosofia de Aristóteles, e propor a noção de conhecimento fenomenológico. Aristóteles nasceu em 374 A.C. em Estagira. Órfão aos 17 anos, quitou a Macedônia para Atenas, onde se tornou aluno de Platão durante 20 anos na Academia. Viajou para Assos, na Ásia Minora, onde fundou seu próprio centro de ensino e pesquisa, e escreveu seu diálogo sobre a filosofia, a "*Charte de Assos*". Após ter sido o preceptor do filho de Philippe II, voltou para Atenas, onde ensinou a filosofia para os "Peripateticianos". Vítima de uma acusação de impiedade, Aristóteles fugiu em 323 A.C. para Chalcis, onde morreu no ano seguinte.

A filosofia Aristotélica é fundada sobre uma fratura epistemológica. De um lado, uma ruptura com o **Idealismo Platônico**[18], e a intuição profunda do que os seres reais são efetivamente os indivíduos, e de um outro lado, uma herança **eleática**, querendo entender o *Ser* como um princípio transcendental superando os indivíduos particulares, expressa por Parmênides[19] na fórmula fundadora da ontologia ocidental "O Ser é" – [MOR85], p. 54:

> « O *Ser* de Parmênides é exatamente conforme á exigência da identidade, que é a lei do pensamento: eternamente um e idêntico à si-próprio, ele é exclusivo de toda mudança e de toda diversidade. A concepção eleática do *Ser* leva à conseqüências inadmissíveis : 1) na ordem física, a impossibilidade da mudança ilustrada por Zenão; 2) na ordem lógica, a impossibilidade da predicação, do julgamento de atribuição[...]. Se o *Ser*, efetivamente, é exclusivo de toda diversidade, a ciência se esgota nesta dupla asserção tautológica: "o ser é", e "o não ser não é"; e até supondo que se distingue uma pluralidade de sujeitos, não se poderia, sem contradição, afirmar de um sujeito um atributo outro que ele mesmo; será necessário se contentar de falar: o homem é homem, o branco é branco; nunca teremos o direito de falar: o homem é branco. »

[18] Para Platão, a verdadeira aspiração da alma é o conhecimento das Idéias, perfeitas e imutáveis, cujas encarnações naturais são corrompidas, e das quais a nossa natureza corporal nos afasta. [PAS86], p. 30. Mas se tratando da verdade, Aristóteles prefere sacrificar os vínculos de amizade: "*amicus plato, magis amica veritas*."

A solução lógica adotada por Aristóteles para autorizar um acesso discursivo ao mundo é a das **categorias**. O "Ser" deve se falar em dois sentidos: o ser "**substantivo**" (afirmação de existência), e o ser "**copulativo**" (afirmação de uma relação). Esta distinção não é uma dicotomia entre essência e existência: para Aristóteles, só pode ter essência através da existência. O ser copulativo denota simplesmente um vínculo que não pode ser concebido sem os termos que o compõe. O sujeito é sempre concreto, e apenas ele *é*, no sentido pleno do termo: ele é a **substância primeira**. A atribuição é uma abstração, ou **substância segunda**. A existência da substância segunda permite realizar duas formas de predicação: *segundo a essência* (o que o ser *é* por substância: "o homem é um animal") , e *segundo o acidente* (o que o ser *é* por acidente: "este homem é grande"). A abertura da predicação por acidente em **nove categorias universais** permite completar a estrutura discursiva de Aristóteles (Figura 2.1).

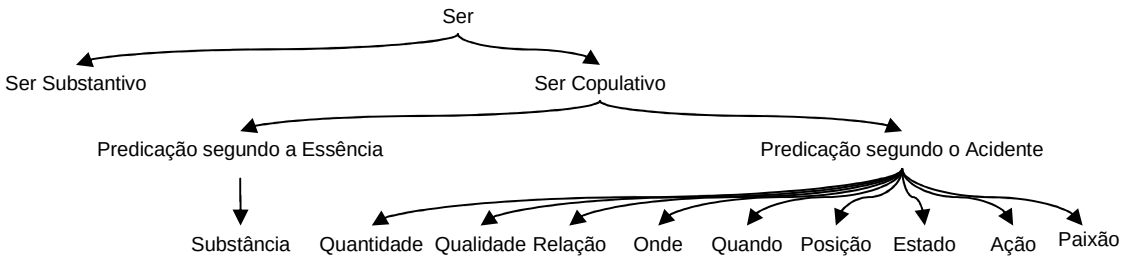


Figura 2.1: As Categorias Aristotélicas [BAC97a], p24.

A partir do entendimento discursivo do mundo através das categorias, Aristóteles propõe a noção de ciência como conhecimento do que é, não por acidente, mas por essência, distinguindo, portanto, a verdade da opinião. A busca da verdade universal deve apoiar-se no uso de uma estrutura lingüística rigorosa que não pode ser a indução, mas sim o **silogismo**, composto por uma majora, uma minora e uma conclusão que derive da verdade acidental da minora em função da verdade universal da majora: «Todos os homens são mortais, ora Sócrates é homem, logo Sócrates é mortal.» Assim, o *conhecimento* consiste, para Aristóteles, em elaborar cadeias de silogismos para acessar a verdade através do uso correto do discurso.

Para enfrentar o eleatismo físico, Aristóteles elabora uma teoria da **mudança**: «Necessariamente, o que advem, provém ou do ser, ou do não-ser. Porém, um como outro são igualmente impossíveis. Não é o ser, efetivamente, que pode advir, vir a ser, porque já é; e do

[19] Parmênides, filósofo grego do século V A.C., formulou no seu poema "Da Natureza" a proposta de uma

não-ser, nada poderia advir.» ([ARI90], p. 27). Tal teoria separa a **Matéria** (o que algo é por essência: a madeira de uma caixa de madeira), da **Forma** (o que algo é por acidente: a forma de caixa da caixa de madeira), a **Potência** (a forma que pode ser, mas ainda não é), do **Ato** (a forma é atualizada na matéria), e articula-se segundo o paradigma das quatro causas: a causa **material** (ser em função da sua matéria), a causa **formal** (ser em função da sua forma), a causa **eficiente** (ser em função da atualização da forma), e a causa **final** (ser em função do seu objetivo). Conseqüentemente, o conhecimento aristotélico é dado em um **Cosmo**, um mundo ordenado onde cada coisa está no seu lugar em função da sua causa final, e não em um universo homogêneo sem objetivos (caso da física cartesiana).

Finalmente, os instrumentos essenciais de acesso inteligível ao mundo são para Aristóteles a nossa alma e os nossos sentidos. A sensação é uma alteração da alma da **aptidão** (o inteligível em potência) ao **exercício** (o inteligível em ato). A alma recebe através dos sentidos a percepção como a matéria recebe a forma (a alma se in-forma). A informação da alma tem a particularidade de ser empírica: se percebo uma pedra branca, não é devido ao fato da minha percepção ser branca, mas sim por que a pedra é realmente branca.

A ciência aristotélica coloca em evidência o conceito de **conhecimento fenomenológico**, e mostra que tal conhecimento sustenta uma racionalidade completa. O conhecimento fenomenológico é o *prima* do ver e do falar: percebo as coisas do jeito que são, e articulo a minha percepção com as minhas palavras (ou seja, não posso conhecer o que não posso falar). Tal modo de racionalidade possui suas regras (lógica proposicional), seus axiomas (não contradição e identidade), e propõe asserções baseadas em uma percepção imediata, categorizada por meio de uma ontologia natural. É importante reconhecer que **a psicologia de senso comum**, que sempre foi aristotélica, é a forma mais eficiente conhecida até hoje de lidar de forma preditiva com o conhecimento e o comportamento das pessoas.

Apesar da ciência moderna ter excomungado o fenomenológico por ser subjetivo e não quantificável, pode-se analisar a **Inteligência Artificial Simbólica** (a IAS, que pretende ser uma naturalização científica da cognição) como ingenuamente fenomenológica, no sentido aristotélico do termo. Isto se nota nas características *top-down* das suas teorias, bem como nas suas formas de validação empírica. Entende-se por *top-down* que o cientista começa por observar, por introspecção em linguagem natural, uma "característica" do raciocínio humano, para depois implementar um sistema simbólico simulando tal fenômeno, e finalmente validar

---

realidade imutável, por oposição a visão transformacionista dos Milesios ([GLE97] p.50 a 53).

sua teoria em experimentos *ad-hoc*. Isto é o caso dos **sistemas especialistas**, que podem ser qualificados de aristotélicos por excelência, e cuja construção denota claramente o uso de uma abordagem aristotélica do conhecimento logicamente articulado, até se as lógicas hoje usadas são geralmente mais complexas do que o silogismo[20]. Que se trata de encadeamento direto ou reverso, o sistema especialista é uma série de aplicação de regras extraídas lingüisticamente do "especialista". O "**raciocínio baseado em casos**" é um outro exemplo claro de abordagem empírica aristotélica, sendo ele uma aplicação simbólica automática de uma "propriedade" do raciocínio humano, induzida por introspecção: a assimilação de um caso presente à casos passados na tarefa de resolver problemas. Finalmente, a utilização recente de "**ontologias**" em IAS representa uma volta clara à concepção aristotélica do mundo: entende-se, no uso e no compartilhamento de ontologias, que os sistemas de informação possuem categorizações imediatas e naturais do mundo que, para poder ser compartilhadas, deverão, no mínimo, poder ser *traduzidas*, em termo dos seus conceitos.

2.1.2. Ruptura Cartesiana e Impossibilidade de uma Ciência da Mente

Além de ser os modos inteligíveis e sensíveis privilegiados do relacionamento humano com a Natureza, o conhecimento **idealista** (de Platão) e o conhecimento **fenomenológico** (de Aristóteles), bem como a visão cosmológica do Mundo, reinaram sobre a filosofia e a ciência durante cerca de nove séculos[21]! Com a decadência do império Romano, o crescimento do império Bizantino, e, especificamente, a conversão de Constantino ao Cristianismo em 324, a concepção Platônica do conhecimento foi recuperada pelas autoridades religiosas para justificar o **dogma da contemplação divina**, e condenar as tentações da carne, da qual a exploração do mundo através dos sentidos fazia parte. Nas palavras de Santo Agostinho:

> « Agora menciono uma outra forma de tentação ainda mais variada e perigosa. Pois acima da tentação carnal, que se baseia nas delícias e prazeres sensuais [...], existe também a tentação da mente, que, utilizando-se dos cincos sentidos, motivada por vaidade e curiosidade, realiza experimentos com auxilio do corpo, em busca de conhecimento e sabedoria.» [GLE97], p. 95

[20] Faze-se aqui referência, entre outros, à lógica dos predicados (que não é aristotélica), às lógicas temporais, e lógicas para-consistentes.

[21] Pode-se considerar que a inércia dogmática Aristotélica e Platônica durou da conversão de Constantino (324) até os manuscritos de Roger Bacon (1250). [GLE97] p.94 – 97.

Com Santo Tomás de Aquino (1225 – 1274), a ciência e o conhecimento Aristotélicos são novamente propostos de forma dogmática, para fundar uma "**cosmologia cristã**"[22]. Apesar da inércia monolítica do pensamento medieval, alguns pensadores como o franciscano Roger Bacon, já anunciavam no século XIII uma próxima ruptura: "Parem de ser dominados por dogmas, olhem para o mundo!" [GLE97], p.99. Bacon, nos seus manuscritos, já vinha propondo a matemática e a experimentação como instrumentos de exploração da Natureza, abrindo conseqüentemente o caminho para uma concepção objetivista do conhecimento.

A real ruptura na tradição epistemológica ocidental deve ser datada de 1637, ano no qual Descartes propõe "O Discurso do Método". Nascido em 1596 em uma família nobre, Descartes recebeu os melhores ensinamentos da época [ROD71]. Pouco atraído pela vida mundana, ele decide abandonar sua carreira militar, para se retirar na solidão em Holanda. Esgotado pelas intermináveis argumentações lógicas da Escolástica, Descartes decide fundar, para superar a metafísica contraditória e desordenada da sua época, um **método** para assegurar um acesso confiável à verdade. Este método, que deve ser entendido como uma verdadeira ascese do conhecimento, é a "**Dúvida Metodológica**". Através dela, Descartes começa por reduzir as percepções sensíveis à simples ilusões, argumentando que os sonhos parecem tão verdadeiros quanto a realidade. Radicalizando sua ascese, através da **dúvida hiperbólica**, ele desconfia depois das idéias puras, como as da matemática, argumentando que um *Mau Gênio* poderia equivocar meu raciocínio. A dúvida cartesiana torna-se então simultaneamente uma negação de uma epistemologia idealista e de uma ontologia empirista, como analisou Husserl em "Pesquisas Lógicas" [HUS59]. Se a suspensão do julgamento já era um clássico do cepticismo (Hipóteses Pyrrhonianas [PAS86], p. 58), a revogação do sensível para melhor entender o empírico é uma originalidade cartesiana, que permite fundar o ***Cogito***, como consciência transcendental essencial à existência do conhecimento:

> «Nesta dúvida hiperbólica, um pedestal subsista, inalterável, sobrevivendo a toda dúvida possível: o *Ego*. Efetivamente, a dúvida hiperbólica apenas é uma dúvida por que é minha: sou a última instância onde a dúvida pode se produzir. A dúvida se torna contraditória e se dilui se o cepticismo não chega à conclusão indefectível de que o Ego conduzindo a dúvida é ele mesmo fora de dúvida. O cepticismo ântico era praticável por que não radical. O cepticismo radical é contraditório. O radicalismo das Meditações devia necessariamente chegar ao autor da dúvida, o Ego Cogito, que ia se tornar a fonte de toda certeza.» [BAC97], p46

[22] Seguindo o modelo proposto por Ptolomeu, cerca de 10 séculos antes, a Terra "volta [ocupar] seu trono no

A partir da existência da idéia de infinito em mim, Descartes atualiza a **Prova Ontológica**[23] de Santo Agostinho, e coloca Deus como pivô do conhecimento, sendo ele a garantia da existência das idéias claras e distintas, que colocou em mim de forma inata para me ajudar a entender o mundo. A essência do Cogito passa então ser a clareza matemática.

Se o *Cogito Ergo* permite reabilitar o mundo das minhas idéias, como estados do pensamento, Descartes deve encontrar uma outra solução para reabilitar também o meu entendimento do mundo externo, e não cair no solipsismo. Para tal, Descartes propõe um duplo dualismo. O **dualismo metafísico** opõe ontologicamente o **Espírito**, cuja essência é puro pensamento, e a **Matéria**, cuja essência é pura extensão. O **dualismo epistemológico** opõe a **objetividade fenomenológica**, que, provinda dos meus sentidos, é confusa e inexata, e a **objetividade científica**, que, provinda do meu raciocínio matemático, é clara e exata. Conseqüentemente, o entendimento da natureza não deve decorrer da observação direta pelos sentidos, mas sim de construções matemáticas *a priori*. Sendo o homem a coincidência da objetividade científica (graças a sua mente) e da objetividade fenomenológica (a partir do seu corpo), Descartes funda a abordagem moderna do conhecimento científico, como movimento do Conhecer para o Ser[24], sendo uma física matemática especulativa o prima do saber, e uma física matemática experimental a busca por indícios na Natureza. O movimento de Galileu, fundador da Revolução Coperniciana[25] do século XVI, que observa os corpos celestes através da luneta astronômica, é um correlato da epistemologia cartesiana que consiste em deixar de confiar nos sentidos, para utilizar um intermediário matematicamente construído para acessar a realidade. As conseqüências do cartesianismo são tão numerosas, que basta olhar ao nosso redor para lembrar a todo instante da geometrização da Natureza.

Para o objetivo de naturalizar a cognição, e realizar "mapeamentos inteligentes de conhecimento", **a epistemologia Cartesiana possui conseqüências desastrosas**. Nesta visão,

---

centro do universo" [GLE97], p.96.

[23] A prova ontológica consiste em argumentar que uma idéia, como representação, só existe em mim se existe uma realidade formal que possui tanto realidade quanto a minha idéia. Como a idéia de infinito está em mim, e que sou um ser finito, deve necessariamente existir um outro ser possuindo tanto realidade formal como a minha idéia de infinito. Este ser infinito e perfeito só pode ser Deus, que colocou em mim de forma inata todas as outras idéias claras e distintas, para me permitir discernir claramente as coisas do mundo [PAS86], p.90.

[24] O movimento do Conhecer ao Ser representa uma ruptura completa com a racionalidade Aristotélica, que procede do Ser para o Conhecer.

[25] No século XVI, Copérnico decide basear-se na observação do movimento dos corpos celestes para propor um modelo heliocêntrico "hipotético" do Universo ("De revolutionibus orbium coelestium", 1548), em ruptura com a visão Aristotélica de Cosmos, e o modelo de Ptolomeu. Kepler formula alguns anos depois as suas leis sobre os movimentos orbitais dos corpos, e Galileu finalmente apresenta em 1632 a noção de "Universo" Copernicano, não mais como hipótese, mas como realidade.

só pode efetivamente se alcançar um conhecimento real do mundo através da matematização *a priori* do objeto a ser conhecido. Porém, o nosso objeto sendo o próprio conhecimento, e o conhecimento sendo consciência, devemos, portanto, ser capazes de espacializar o *Ergo Cogito*. Este objetivo é obviamente contraditório e inatingível, desde o dualismo metafísico: o Cogito é puro pensamento, e, logo, não pode ser espacializado, *ergo*, ele não pode ser conhecido. Na mesma linha, Kant afirma, no século XVIII, **a impossibilidade de uma ciência** da mente ([RIV92], p.313). Aprofundando a espacialização cartesiana do conhecimento, Kant detalha que todo fenômeno é dado em um *a priori* espácio-temporal. Como só pode ter ciência que de fenômenos espacializados, toda percepção, para poder ser objeto da ciência, deve ser medida e projetada em um espaço cartesiano[26]. Como os fenômenos da consciência são dados na única dimensão do tempo, eles não podem ser espacializados, e logo, não pode existir uma ciência da mente.

2.1.3. Cognitivismo Clássico e Inteligência Artificial Simbólica

Toda a problemática das ciências cognitivas[27] é conseqüência direta das epistemologias cartesiana e kantiana. A própria sistematização da "Gestão do Conhecimento" constitui uma tentativa de **naturalização da cognição**, uma forma de gerenciamento científico. Para, porém, efetivamente chegar a um primeiro paradigma epistemologicamente sustentado e fisicamente realizado de naturalização da mente, será necessária uma caminhada filosófica e científica de cerca de 320 anos, do discurso do método, à proposta de uma "**Inteligência Artificial**"[28] na *Dartmouth Conference* de 1956.

A epistemologia Cartesiana, se pode ser entendida como um golpe fatal a toda tentativa de espacialização dos fenômenos mentais, constitui na realidade um ponto de partida para atingir este objetivo: se a natureza deve ser entendida *a priori* através da quantificação, uma naturalização cartesiana dos fenômenos mentais deverá, também, se basear em uma teoria lógico-matemática da mente. Descartes ele mesmo já anunciava a idéia de **inteligência**

[26] Para KANT, toda síntese objetiva deve ser dada simultaneamente no espaço e no tempo, para formar um "diverso espacio-temporal", sobre o qual o órgão da ciência, a matemática, poderá operar.

[27] As Ciências Cognitivas nasceram por volta de 1940, com o objetivo de propor uma solução ao problema de naturalização da mente, constituindo, portanto, uma ruptura com a tradição dualista cartesiana. Segundo J. Stewart, as ciências cognitivas são "o projeto de constituir a mente e os processos mentais como objetos das ciências naturais, se relacionando com a filosofia , as ciências sociais e humanas e a biologia". [STE99], p. 1

[28] Em 1956, McCarthy propõe pela primeira vez o termo "Inteligência Artificial" como tópico da "*Dartmouth Conference*", a primeira conferência devotada ao assunto. Nesta conferência, Newell, Shaw e Simon, da *Carnegie Institute of Technology*, hoje *Carnegie Mellon University*, demonstram o primeiro programa de IAS, o Teorista Lógico (*Logic Theorist*), capaz de demonstrar automaticamente teoremas da lógica das proposições.

mecânica, quando descrevia o corpo humano como « uma máquina, que, sendo criada das mãos de Deus, é incomparavelmente melhor ordenada e possui em si movimentos mais admiráveis do que qualquer outra destas que podem ser inventadas pelos homens» ([DEC63], Discurso do Método, 5ª parte). Assim, Pascal criava em 1642, a primeira máquina de calcular mecânica, que Leibniz melhoraria em 1673. Leibniz, precursor da noção de inteligência mecânica, já tinha a visão de uma máquina universal de *cálculo* do raciocínio, cujos argumentos poderiam ser mecanicamente decididos. Nas palavras de Norbert Wiener:

> «A filosofia de Leibniz é baseada em dois conceitos fortemente ligados – o de um simbolismo universal e o de uma máquina de raciocínio. Daí decorrem a notação matemática e a lógica simbólica de hoje. Assim, da mesma forma que o cálculo aritmético pode se submeter a uma mecanização que progrediu do ábaco até as calculadoras ultra-rápidas atuais, o *calculus ratiocinator* de Leibniz contém os germes da *machina ratiocinatri*, a máquina que raciocina. [...] Conseqüentemente, não é nem um pouco surpreendente que a mesma mola intelectual que conduziu ao desenvolvimento da lógica matemática tenha no mesmo tempo conduzido à mecanização ideal ou real dos processos do pensamento.» [WIE48], p. 18.

Com as "Pesquisas sobre as leis do pensamento" [BOO54], Boole propõe em 1854, através da matematização da lógica[29], uma abordagem cartesiana do raciocínio. Criticando e aperfeiçoando o pensamento de Boole, Frege funda em 1884 a lógica matemática moderna[30] e elabora a primeira teoria coerente do cálculo dos predicados e das proposições, como instrumentos do estudo matemático das leis do pensamento. A partir dos trabalhos de Frege, os filósofos do Círculo de Viena[31] fundam o **positivismo lógico** como nova epistemologia, rejeitando toda proposição incoerente do ponto de vista da lógica dos predicados. No mesmo momento, visando originalmente "salvar o paraíso que Cantor nos deixou"[32] ([BAC97], p29), o matemático alemão Hilbert se lança na elaboração de um procedimento de redução das entidades matemáticas abstratas às entidades aritméticas concretas e finitas. Se o "**Programa de Hilbert**" fracassou a partir da formulação em 1932 por Gödel do **teorema de**

[29] O trabalho de Boole, a álgebra binária, visando originalmente a representar as "leis do pensamento", continua até hoje de ser maciçamente usado em matemática, lógica e ciências da computação.
[30] Ver os "Fundamentos da aritmética" [FRE84].
[31] O Circulo de Viena foi um grupo de intelectuais que se deu nos anos 1920 a missão de constituir um saber organizado a partir das descobertas físico-matemáticas modernas, adotando o positivismo lógico, traduzido pela aplicação da lógica simbólica aos modos de pensamento. Seus principais membros foram os matemáticos Hans Hahn (1879-1934) e Friedrich Waismann (1896-1959), os lógicos Moritz Schlick (1882-1936), Kurt Gödel (1906-1978), Rudolf Carnap (1891-1970) e Alfred Tarski (1902-1983), o economista Otto Neurath (1882-1945).
[32] No início do século XX, a erupção de objetos matemáticos ideais, como os transfinitos, despertou muitas dúvidas quanta a existência real destas construções, agravadas pelo fato que vários contradições e paradoxos eram formulados a partir da manipulação destes objetos teóricos delicados.

incompletude[33], ele levou a conceitualização do primeiro elemento essencial da inteligência mecânica: **a aritmetização do raciocínio**. Efetivamente, Hilbert mostrou como o raciocínio sobre entidades abstratas podia ser reduzido a uma manipulação finita formal de símbolos, independentemente de todo conteúdo semântico: é o nascimento do **representacionalismo simbólico**. Com a invenção em 1936 de uma máquina abstrata capaz de realizar tal manipulação formal finita de símbolos, a "**Máquina Universal de Turing**"[34], Alan Turing aporta a contribuição essencial de **mecanização do raciocínio**. O formalismo mecânico Turingiano, além de tornar a manipulação formal de símbolos automática, permite naturalizar o cálculo, ou seja, produzi-lo apenas pela ação das forças físicas da natureza. McCullogh e Pitts colocam em 1943 a pedra final no edifício da naturalização mecânica do pensamento, através da criação do "Neurônio Formal" [MAC43]. A partir de observações fisiológicas, eles propõem um modelo físico simplificado de rede de neurônios binários, percorrida por fluxos elétricos, na qual pode-se perceber a propagação física do raciocínio lógico. Desta forma, constitui-se uma **fisiologização da psicologia**, acrescentando o elemento final da materialização da mente: a dimensão espacial. O cérebro é considerado a partir daí como um dispositivo lógico mecânico calculatório complexo. A mecanização artificial efetiva do raciocínio pode então ser realizada no computador digital, ou máquina de Von Neumann, apresentada no simpósio de Hixon em 1948[35].

A teoria cognitiva decorrente é o **simbolismo computo-representacional** de Fodor, que funda a primeira abordagem científica moderna da cognição no seu artigo "The Language of Thoughts" [FOD75]. Para Fodor, pensar significa computar, ou seja, executar uma manipulação formal finita de símbolos, independentemente de qualquer semântica. A cognição torna-se um dispositivo mecânico provido de sensores que transformam a realidade objetivamente percebida em símbolos formais materializados pelos estados internos da máquina, e provê ações no mundo real a partir de atuadores, cuja ação é governada pelos resultados da execução do programa da máquina (Figura 2.2). É a "primeira cibernética"[36].

---

[33] O teorema de indecidibilidade de Gödel [AND54] mostra que todo sistema formal é intrinsecamente incompleto do ponto de vista das deduções que pode operar, desde a não decidibilidade de fórmulas verdadeiras bem formadas em um sistema possuindo uma aritmética.

[34] A máquina de Turing descreve o comportamento do computador digital, como uma cabeça de leitura se deslocando em uma banda finita, escrevendo ou apagando símbolos pertencendo à um alfabeto finito, em função do que é lido, do estado interno da cabeça, e do "algoritmo" seguido. [TUR95]

[35] J. Von Neumann, "The General and Logical Theory of Automata", in Jeffpress A., Cerebral Mechanisms in Behavior. The Hixon Symposium, California Institute of Technology, 1948, Wiley & Sons, NY, 1951; p. 22-24.

[36] A primeira cibernética (do grego *kubernètès* – o homem de direção), das conferências de Macy de 1947, foca essencialmente na retro-alimentação (*feed-back*) ao serviço da transformação de uma entrada (*input*) em uma

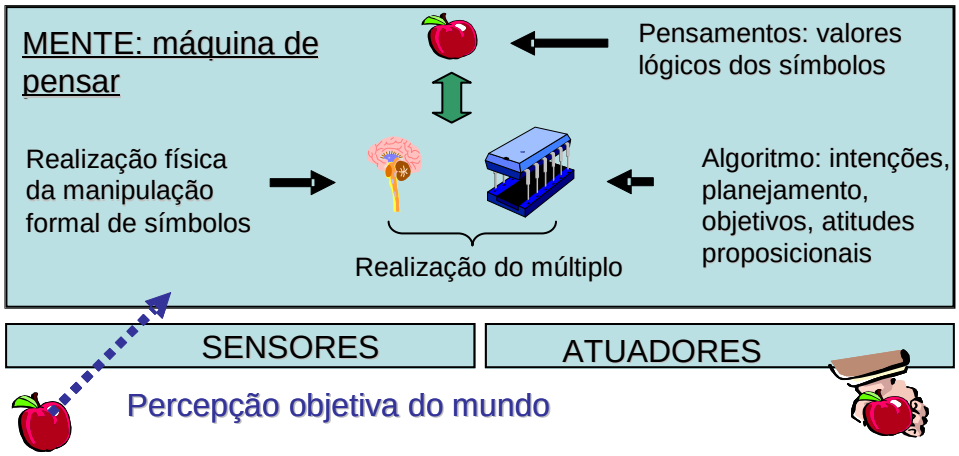


Figura 2.2: O Computo-Representacionalismo Fodoriano

A abordagem Fodoriana é extremamente sedutora por que é simultaneamente não reducionista (existe um nível lógico independente do físico) e materialista (a explicação psicológica pode ser reduzida a um determinismo mecânico). Com os avanços tecnológicos[37] da IAS, na segunda parte do século XX, a naturalização efetiva da mente parecia alcançada...

2.1.4. Ancoragem dos Símbolos e Fenomenologia Contemporânea

Como já comentado, a abordagem epistemológica da IAS é absolutamente aristotélica, baseada em uma concepção objetivista do mundo. Sendo esta constatação, a fatalidade das várias frustrações pragmáticas[38] às quais a IAS levou aparece muita mais evidente. Efetivamente, uma abordagem baseada em uma introspecção lingüística subjetiva devia necessariamente fracassar, por ser um sensualismo empírico ingênuo.

De forma mais detalhada, o ponto central das dificuldades do funcionalismo de Fodor é o próprio axioma principal de sua teoria: o **formalismo puro da manipulação simbólica**. Na máquina de Turing, os símbolos são manipulados mecanicamente e independentemente de qualquer valor semântico. Mas se os símbolos são desprovidos de interpretações, como podem então fazer sentido? Esta pergunta fundadora é conhecida como o ***symbol-grounding-***

---

saída (*output*), em um sistema que deve atingir um objetivo pré-definido. Trata-se claramente de uma abordagem comportamentalista (*behaviorista*).

[37] Referenciamos aqui a resolução automática de problemas (anos 50), a orientação objeto (anos 60), o processamento de linguagem natural (anos 70), os sistemas especialistas (anos 80), e os sistemas baseados em conhecimento e os sistemas ontológicos (anos 90).

[38] Faze-se aqui referência ao fracasso do sistema de resolução automática de todos os problemas (General Problem Solver), às dificuldades pragmáticas encontradas nos sistemas de processamento de linguagem natural (elipses, metáforas, anti-frases, fórmulas cínicas, etc.), e nos famosos sistemas especialistas, que sempre precisam de “mais uma regra”, para firmar seu “entendimento” ontológico do domínio.

problem, cuja expressão mais famosa é o "Problema da câmara Chinesa"[39] de Searle. Em última análise, o formalismo Turingiano implica uma **heteronomia** do raciocínio mecânico: só existe sentido, por que um agente externo (heterônomo) pode emprestar temporariamente e subjetivamente uma semântica. Em outras palavras, a máquina de Turing não se percebe si-próprio, ela é sempre percebida**. O teorema de incompletude** de Gödel, o problema de parada da máquina de Turing, e o problema da independência do contexto de Shanon, são conseqüências diretas do formalismo simbólico da máquina de Turing:

> «Da mesma forma que Gödel, frente ao problema da "decisão" colocado por Hilbert, tinha chegado ao teorema de incompletude da lógica aritmetizada, Turing chega também a uma resposta negativa no problema de "parada da máquina de Turing": não existe um procedimento mecânico capaz de determinar se o cálculo efetuado terá um fim, ou se entrará em *loop* indefinidamente.» [HAV95], p. 249.

Se a dificuldade do CR encontra-se na sua heteronomia, e na sua abordagem objetivista e ontológico-empirista, deve-se buscar uma solução alternativa na **autonomia**, e numa abordagem **fenomenológica relativa** da cognição. Apesar do sucesso da epistemologia cartesiana, a abordagem fenomenológica não morreu no século XVII, e Kant, na sua oposição entre o nômeno (a coisa em si) e o fenômeno (a legitimização cientifica da coisa), é um precursor da fenomenologia moderna [KAN81]. Se Hegel, escapando ao dogmatismo absoluto do saber, propõe em 1807 uma fenomenologia da mente como "ultrapassagem de si-próprio" [HEG07], o fundador da fenomenologia moderna ficará sem dúvida o filosofo alemão Edmund Husserl.

Nascido em Prostejov, na Moravia, em 1859, Husserl foi o aluno do matemático Weierstrass e do psicólogo Brentano[40]. Ele levou a vida toda uma carreira estudiosa de professor e abriu, a partir de 1901, uma via nova para as teorias do conhecimento: a fenomenologia como ciência dos fenômenos. O ponto de partida de Husserl é: "*toda consciência é consciência de*" [HUS01]. Para poder, portanto, se concentrar no fenômeno,

---

[39] O *symbol-grounding-problem* (problema de ancoragem dos símbolos) é exposto no artigo "Minds, Brains and Programs" [SEA80], da seguinte forma: uma pessoa, fechada em uma câmara onde só existem duas "portinholas", possui um livro de regras que indica exatamente quais símbolos chineses escrever e passar na segunda portinhola ao receber determinados símbolos através da primeira portinhola, e dá portanto a impressão que "a câmara entende o chinês". Porém, isto é valido apenas para um observador externo: a pessoa dentro da sala não compreende uma só palavra. Como é o caso para uma máquina Fodoriana, a câmara não é cognitiva.

[40] Brentano foi um dos primeiros científicos a definir claramente a *intencionalidade* como objeto da psicologia, colocando que os objetos intencionais (os estados da mente, como, por exemplo o "unicorne") possuem propriedades intensionais (não substitualidade dos idênticos e não generalização existencial).

como é dado à minha consciência, Husserl propõe o método da **redução fenomenológica**[41], como suspensão na crença ingênua em uma qualquer objetividade externa a minha consciência: a redução Husserliana é uma neutralização, ou suspensão da atitude natural de percepção objetiva do mundo através dos sentidos. Esta fenomenologia é conseqüentemente diferente da abordagem Aristotélica. É importante entender que a redução Husserliana não leva à um novo solipsismo, mas, ao contrário, a noção chave de **intencionalidade**, como abertura ao dado. Nesta epistemologia, o ato cognitivo é, ao mesmo momento, **intenção** (movimento de dirigir-se á) e **intencionado** (objeto de direção), independentemente da forma de ser de uma realidade objetiva. A consciência aparece desta forma como uma **transcendência imanente**: transcendência na sua relação com um mundo *além* de si, imanente por que esta projeção acontece dentro de si. Finalmente, Husserl argumenta que **a projeção intencional é universal**, sendo ela a forma primordial de ser de toda consciência.

A fenomenologia transcendental resolve, em função da sua própria natureza, o *symbol-grounding-problem*: como toda consciência é consciência *de*, a intencionalidade possui necessariamente uma dimensão semântica. Realizando uma análise crítica da IAS, pode-se questionar a razão de ter desprovido o símbolo de senso, se ele pode ser puro senso. Por que ter, também, introduzida uma heteronomia, se a cognição pode e deve ser autônoma? Pode-se afirmar que a primeira cibernética deixou escapar seu objetivo original ao esquecer a noção de autonomia. A adoção da fenomenologia transcendental como abordagem epistemológica da noção de conhecimento, para o projeto de constituir mapeamentos automáticos inteligentes, é decisiva. Efetivamente, esta escolha implica que tal sistema deverá (i) possuir uma fenomenologia perceptiva lingüística, (ii) ser autônomo por essência, para se inscrever de forma cognitiva nas suas interações com os usuários. Por este motivo, a próxima parte deste capitulo será dedicado ao estudo da natureza da interação lingüística, e a parte seguinte abordará a noção de autonomia como atributo essencial da cognição.

2.2. Lingüística

Intimamente vinculado a noção de cognição, encontra-se nas epistemologias ânticas, medievais, renascentistas e modernas a noção comunicação, ou, de forma mais genérica, de interação semiótica. Portanto, uma sustentação teórica da constituição de mapeamentos

[41] A redução Husserliana deve ser entendida no sentido etimológico do termo: *re-ducere*, ou seja, levar de volta ao essencial, que é a essência do fenômeno como fenômeno percebido, e não como independência objetiva.

automáticos de conhecimento, se ela deseja ser completa, deve necessariamente incluir uma análise da linguagem natural, nas suas dimensões coletiva e individual. Tal análise se torna ainda mais imprescindível, desde a afirmação da necessidade de interações pró-ativas, e contextualizadas garantindo a inserção hermenêutica do objeto tecnológico que é o mapeamento na fenomenologia perceptiva do usuário. Serão propostas, a seguir, três grandes paradigmas da lingüística: a abordagem estrutural de Saussure, a semiótica Peirciana, e as gramáticas gerativas de Chomsky. Esta análise nós levará integrar aspectos sociais e coletivos, bem como individuais e perceptivos, no processamento de linguagem natural.

2.2.1. Abordagem Estrutural e Relativismo do Signo

Da mesma forma que Descartes representa uma ruptura epistemológica fundamental, a abordagem estrutural proposta por Saussure no seu "Curso de Lingüística Geral" [SAU76] foi decisiva na história da lingüística. Ferdinand de Saussure nasceu em 1857 em Genebra. Após ter estudado em Leipzig, ele ensinou a gramática comparada em Paris e depois em Genebra. É durante este período, de 1907 a 1911, que deu uma aula de Lingüística Geral, cujos elementos foram publicados por seus alunos, após sua morte em 1913.

A lingüística pré-saussuriana pode ser divida em três grandes fases. Da Grécia ântica até a segunda metade do século XVIII, a lingüística é essencialmente gramática, ou seja, um estudo lógico das regras que permitem diferenciar as formas corretas de se expressar. Trata-se então de uma "disciplina essencialmente normativa" [SAU76], p. 14. Com as grandes descobertas e o desenvolvimento do comércio internacional, a normalização dogmática baseada no Grego e no Latim passa ser questionada, pois não é aplicável aos idiomas amer-indianos. Influenciada pelo crescimento das ciências investigativas, a lingüística se torna no fim do século XVIII filologia, sendo seu objetivo fixar, comentar e interpretar textos. Com a descoberta do Sânscrito[42], Franz Bopp inaugura em 1816 a gramática comparativa, desenhando árvores genealógicas evolutivas das línguas para esclarecer os seus relacionamentos parentais. Trata-se de uma lingüística evolucionista, no sentido Darwinista.

Influenciado pelos trabalhos de Boas[43] e Sapir[44], Saussure introduz com o **estruturalismo lingüístico** uma grande quebra de paradigma. Ele começa por diferenciar a

[42] O Sânscrito é uma língua indo-aryana, fonte comum do Grego e do Latim.
[43] O antropólogo americano f. Boas (1858 – 1942) é responsável pelo repertório de todas as línguas amer-indianas conhecidas na época. Seu livro "Handbook of American Indian Languages" (1911) inaugura a etno-lingüística, sendo seu objetivo o estudo de uma língua viva no seu contexto sócio-cultural.

língua da fala, sendo a **língua** o aspecto coletivo da linguagem, e a **fala** uma instanciação psico-física individual da língua. Para Saussure, a lingüística deve ser o estudo da língua:

> «O estudo da linguagem possui duas partes: uma, essencial, tem por objeto a língua, social por essência e independente do individuo; [...] a outra, secundária, tem por objeto a parte individual da linguagem: a fala. [...] Sem dúvida, estes objetos são estreitamente vinculados. [...] Mas isto não impede as duas coisas de serem absolutamente distintas. [...] A língua existe na coletividade na forma de uma soma de impressões depositada em cada cérebro. [...] Não existe nada de coletivo na fala; as suas manifestações são individuais e momentâneas. [...] Podemos conservar o nome de lingüística para as duas disciplinas. [...] Mas não poderemos confundir [a lingüística da fala] com a lingüística propriamente dita, cuja língua é o único objeto.» [SAU72], p. 37.

Saussure coloca depois uma distinção entre a abordagem **diacrônica** (o estudo evolutivo de uma língua) e **sincrônica** (o estudo da língua no seu contexto sócio-cultural, em uma dada época, em um dado lugar) da lingüística: «É sincrônico todo o que se reporta ao aspecto estático da nossa ciência, e diacrônico tudo o que é voltado à evoluções.» [SAU72], p117. A partir destas premissas, Saussure define a forma racional de estudo da lingüística (Figura 2.3).

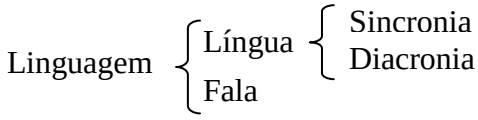


Figura 2.3: Forma Racional de Estudo da Lingüística [SAU72], p.139.

Definindo a lingüística como o **estudo sincrônico da língua**, Saussure introduz a noção de signo. O **signo** Saussuriano (Figura 2.4) é a junção de dois elementos inseparáveis: o **significante** (o que é percebido através da fala ou da escrita), e o **significado** (o que é entendido cognitivamente a partir do signo). O vínculo entre significante e significado, ou seja, o signo, é arbitrário: ele é uma convenção sócio-cultural. Poderíamos aqui reconhecer um simbolismo representacional, comparável ao CR. Porém, como o significante Saussuriano é linear na dimensão exclusiva do tempo, o signo Saussuriano demora epistemologicamente Kantiano, e não pode ser adequado a uma abordagem espacializante da mente.

---

[44] O lingüista americano E. Sapir, quando afirmou que todos os elementos de uma língua são interdependentes e possuem valores relativos em constante mutação na estrutura lingüística, fundou o estruturalismo lingüístico.

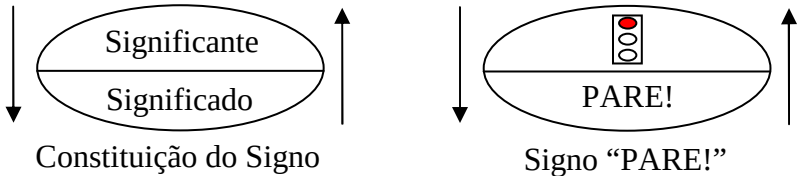


Figura 2.4: Natureza do Signo Lingüístico para Saussure

Dois conceitos Saussurianos fundamentais, na direção de uma espacialização do fenômeno lingüístico, são as noções de **estrutura** e de **valor do signo**. Para introduzi-los, Saussure utiliza a metáfora da partida de xadrez. Em um determinado momento do jogo, cada peça possui um valor relativo na estrutura de interdependência de todas as peças, e o conhecimento dos fatos que levaram a esta determinada situação do jogo é completamente irrelevante para determinar este valor. Da mesma forma, o signo possui um valor relativo em uma estrutura lingüística, que pode ser definido apenas por um estudo sincrônico. O arbitrário do signo e seu valor relativo na estrutura lingüística, quando colocados no contexto de uma fenomenologia da percepção, levam a uma visão **relativista coletiva da articulação do mundo**, radicalmente diferente do empirismo ontológico Aristotélico (Figura 2.5).

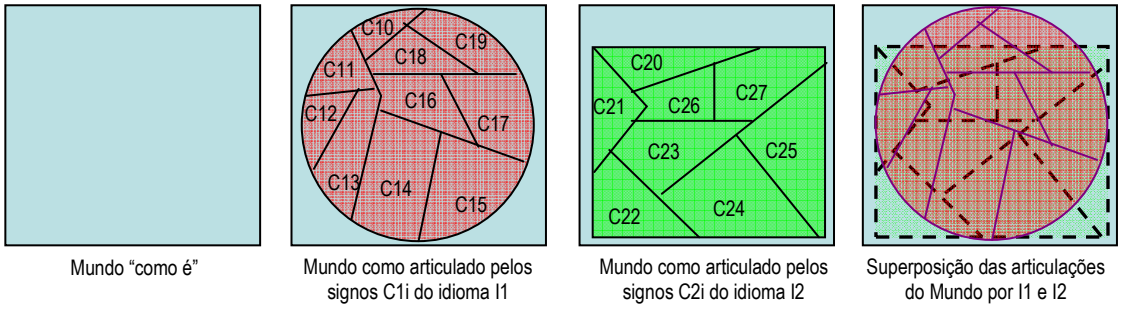


Figura 2.5: Relativismo Lingüístico Saussuriano e Fenomenologia Perceptiva

2.2.2. Semiótica e Percepção Sígnica Individual

A abordagem Saussuriana permite evidenciar aspectos coletivos da comunicação, destacando as noções de estrutura e valor sincrônico do signo, e corroborando à uma visão fenomenológica do Conhecimento, na medida que constitui um relativismo hermenêutico[45].

[45] Trata-se de um relativismo hermenêutico, na medida que a língua necessita ser estudada no seu contexto histórico-social, para poder ser compreendida. A hermenêutica defina-se como um ramo da fenomenologia, considerando que a semântica é dependente da cultura, e sempre necessita interpretações [HAV99], p. 220.

Entretanto, sendo seu objetivo o estudo sincrônico da língua, Saussure devia necessariamente deixar de lado reflexões mais profundas sobre os aspectos individuais e perceptivos do signo[46]. Para o projeto de espacialização da cognição, entendida como uma transcendência imanente à *minha* consciência, como para a realização de mapeamentos inteligentes pró-ativos e personalizados, que fornecem informações, adquirindo uma dimensão hermenêutica para *mim*, é porém necessário possuir uma teoria da lingüística com dimensões individuais e perceptivas. A teoria de maior impacto neste sentido é a **semiótica Peirciana**, que se constitui como uma verdadeira ciência da comunicação e da percepção sígnica.

O filósofo, matemático e lógico Americano Charles S. Peirce nasceu em Cambridge em 1859. Físico e Químico de profissão, ele é o principal fundador da semiótica que apresentou como teoria da significação [PEI36]. Para poder abordar suas noções de **Signo** e **Semiose**, é necessário entender a epistemologia de Peirce. Toda sua articulação sistematicamente ternária é a conseqüência inevitável da sua proposta epistemológica, segundo a qual o mundo é dado em três categorias: a **primeiridade** (o possível geral não marcado pela existência), a **segundidade** (o descontinuo, no qual a potencialidade se torna atualidade), e a **terceridade** (o objeto na sua permanência existencial). A classificação Peirciana das ciências é uma primeira tríade conseqüente destas categorias (Figura 2.6).

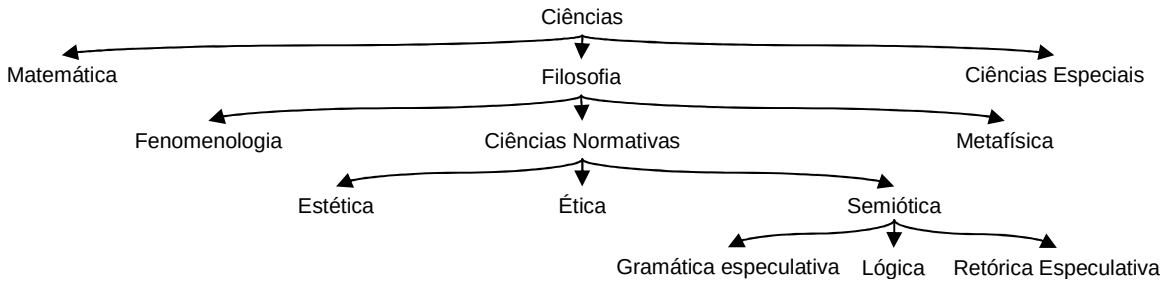


Figura 2.6: Classificação das Ciências Segundo Peirce [JOR00], p.7.

Na filosofia, as ciências normativas investigam como deve reagir a mente ao impacto dos fenômenos (dados como primeiridade). A Semiótica Geral é, nas ciências normativas, uma lógica que deve ser entendida não apenas como uma ciência das condições necessárias para chegar à conclusões verdadeiras, mas, "num sentido mais amplo, uma ciência das leis

[46] Até se se pode argumentar que Saussure aborda aspectos perceptivos individuais nos capítulos "representação da língua pela escritura" e "princípios da fonologia", deve-se reconhecer que estas abordagens são extremamente específicas, e não entendem propor uma teoria da lingüística como *fenômeno geral perceptivo individual*.

necessárias do pensamento [...] que trata das condições gerais dos signos como signos [...] e também das condições necessárias para a transmissão de significado de uma mente a outra" [JOR00], p. 4.

A abordagem da percepção fenomenológica de Peirce se dá também em uma tríade, cujos elementos são o ***percepto*** (o que se apresenta para ser percebido), o ***percipuum*** (o modo como o *percepto* é acolhido e filtrado de forma não controlada pelos sentidos), e o ***julgamento perceptivo*** (modo como o percipuum é imediatamente absorvido nos modelos mentais interpretativos do receptor). Esta abordagem, propondo uma tricotomia entre o físico, o sensorial e o cognitivo, constitui um relativismo fenomenológico em dois níveis: nas limitações perceptivas dos sentidos que filtram sistematicamente o *percepto* para torná-lo *percipuum* (sobre o qual não se tem controle), e nas pré-conceitualizações da mente que recebe o *percipuum*, para torná-lo *julgamento*. Em outras palavras, "só percebemos o que estamos equipados para interpretar, […] só entendemos o que podemos compreender: […] [é] essa visão, que une o físico ao sensório e ao mental" [SAN98], p. 99. Desta forma, apesar de parecer uma abordagem ontológica empírica do tipo sujeito-objeto-representação, a *percepção fenomenológica* Peirciana sustenta perfeitamente a fenomenologia transcendental Husserliana. O **Signo Triádico** é uma conseqüência direta desta fenomenologia (Figura 2.7).

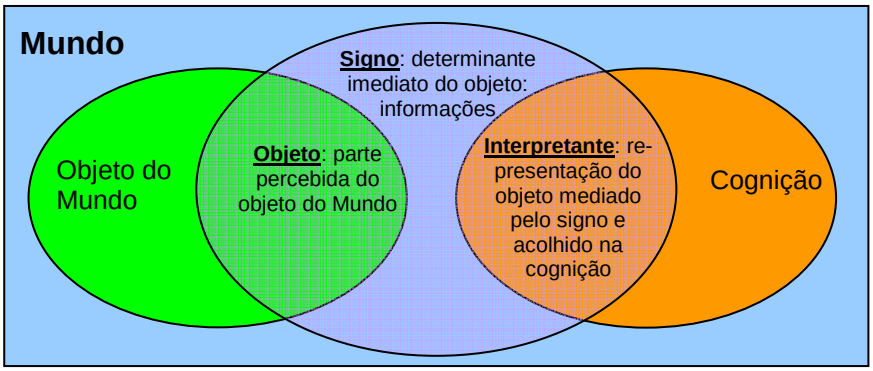


Figura 2.7: O Signo Triádico Peirciano [CAS03], p.6.

Nas Palavras de Peirce, "um *signo* intenta representar, ao menos em parte, um objeto que é, então [...] a causa ou determinante do signo, mesmo que o signo represente o objeto falsamente. Entretanto, dizer que ele representa seu objeto implica que ele afete uma mente de tal modo que [...]determine, naquela mente, algo que é *mediatamente* próprio ao objeto. Essa determinação da qual a causa imediata ou determinante é o signo e da qual a causa mediada é

o objeto pode ser chamada de interpretante." [JOR00], p. 11. Cabe ressaltar que o ***interpretante*** Peirciano é de natureza behaviorista: ele é uma convenção formada por "hábito social". Como na lingüística Saussuriana, chega-se à noção de arbitrário do signo. Nota-se também que o *objeto do signo*, como "causa", é independente da existência real (ou física) de um objeto no mundo real.

A **Semiose** é finalmente definida como uma abstração conceitual da qual apenas o ser cognitivo é capaz: a capacidade em passar da experiência fenomenológica direta, para uma re-presentação cognitiva da experiência. A semiose é, nesses termos, um processo que articula as experiências e as significações. A Semiose é sempre dada por três elementos inseparáveis:

> «Por semiose quero dizer [...] uma ação, ou uma influência, a qual é, ou envolve, uma cooperação entre três sujeitos, tal como um signo, seu objeto, e seu interpretante, essa influência tri-relativa não é de qualquer forma reduzível em ações entre pares. Semeiosis, no período grego ou romano, à época de Cícero já, se bem que me recordo, significava a ação de praticamente qualquer espécie de signos; e a minha definição confere a tudo o que assim se comportar a denominação de *signo.*» [JOR00], p. 13.

Apesar de ter sido muitas vezes opostas, a Lingüística Estrutural Saussuriana e a Semiótica Peirciana são perfeitamente compatíveis. O signo Saussuriano é, na sua essência, idêntico ao signo Peirciano, apesar de Saussure ter dado um enfoque mais **estrutural** e Peirce mais **perceptivo**. Por possuir uma alta relevância nestas duas dimensões (estrutura e percepção), as duas abordagens serão essenciais ao desenvolvimento de mapeamentos inteligentes. Cabe ressaltar, porém, que nenhuma delas possui propriedades espacializantes na dimensão da mente (ou interpretante). Uma terceira abordagem da lingüística será portanto necessária ao alcance dos objetivos da constituição de mapeamentos inteligentes.

2.2.3. Abordagem Gerativa e Transformacional

A charneira que deve unir as teorias do conhecimento e a lingüística, para que se solidifiquem mutuamente, encontra-se na formação do Interpretante Peirciano (ou Significado para Saussure). Em plena onda behaviorista[47], Peirce e Saussure assimilaram, logicamente, a

[47] A proposta da Semiótica, bem como o Signo Saussuriano são contemporâneos do Circulo de Viena e do Positivismo Lógico. Sendo o critério de cientificidade desta época o verificacionismo empírico articulado pelo órgão lógico-matemático, a psicologia científica vinculada devia também ser experimental, medível e repetível. Desta forma, J. Watson inaugura em 1913 no artigo "Psychology as the Behaviorist View It" as noções de estímulo, resposta e reflexo condicionado. De forma correlacionado, Peirce explica a formação do vínculo arbitrário entre o signo e o interpretante a partir do condicionamento social operado pelo meio na formação das capacidades lingüísticas da criança.

formação do Interpretante à um condicionamento social advindo do meio ambiente no momento da aprendizagem. Como toda proposta inductivista, tal visão devia cair com a emergência de contra-exemplos. Se a aquisição das capacidades lingüísticas é puro reflexo condicionado, ou seja, repetição do que já foi ouvido ou lido, como explicar a infinita geratividade explosiva das crianças que, por volta 2 anos de idade, começam a produzir espontaneamente frases articuladas nunca antes escutadas? Este é o ponto de partida da proposta, pelo lingüista americano Chomsky, de uma abordagem **gerativa e transformacional** da lingüística.

Noam Chomsky nasceu em Filadélfia em 1928. A partir de 1955, ele passa a ser professor do Massachusetts Institute of Technology (M.I.T.), aonde ensina até hoje. Seus paradigmas lingüísticos, expostos em "*Syntactic structures*" [CHO57] e "*Aspects of the theory of syntax*" [CHO65], corroboraram às teorias computacionalistas da cognição.

Sendo sua crítica essencial às lingüísticas vigentes na primeira parte do século XX as suas fraquezas inductivistas e suas incapacidades em explicar às fases naturais de aquisição da linguagem pelas crianças, Chomsky devia necessariamente exibir uma proposta **dedutivista** por essência, e que podia dar conta da **criatividade lingüística espontânea**. Como notado também pelo psicólogo Vigotsky ([VIG99], p. 157), Chomsky reconhece em primeiro lugar, no desenvolvimento lingüístico das crianças, uma série de ***universais***, independentes das condições socioculturais nas quais são imersas: (i) a palavra-ação ("bola!", significando "Quero a bola!"), (ii) a palavra-associação ("Filipe bola!", significando "Quero a bola"), e (iii) as frases elaboradas. Sendo a constatação da criação explosiva de frases infinitamente originais na fase (iii), Chomsky conclui a existêcia de uma série de regras de produção lingüísticas "pré-programadas" nos seres humanos. Com a descoberta da molécula da DNA por Watson em 1953, e o advento do neo-darwninismo genético[48], Chomsky pode fundar um **paradigma dedutivista** da lingüística, postulando uma transmissão **genética** dos *universais* na espécie humana, sendo o terceiro dele, a capacidade inata de produzir frases a partir de uma série regras profundas, instanciadas superficialmente na especialização sócio-cultural do aparato lingüístico originalmente onipotente (etno-lingüisticamente falando): "A linguagem é inata, inerente à natureza humana e seu maior aprendizado é conseqüentemente natural do desenvolvimento." [CHO57], p. 52.

[48] Graças a descoberta da molécula de DNA, e a aplicação da teoria da informação a biologia, os neo-darwinistas conseguem dar uma explicação molecular, e logo, materialista, as leis de herança observadas por Mendel, e a teoria da seleção natural de Darwin.

O programa de pesquisa Chomskiano visa logo em verificar experimentalmente a existência dos universais, e descobrir a natureza das regras profundas de produção lingüística, que Fodor chamará, alguns anos depois, de “Linguagem dos pensamentos” [FOD75], e Pylyshyn de “Mentalês” (*Mentalese*) [PYL84]. Para Chomsky, a estrutura lingüística **profunda**, por oposição à **superficial**, consistindo nas instanciações fonéticas e semânticas de uma determinada língua natural, é o fato de regras **sintáticas recursivas** chamadas de **gramática gerativa** (Figura 2.8). Uma vez que a criança aprende, por construtivismo social, conceitos e palavras, estes podem ser aplicados à criação de frases originais desde a existência da estrutura profunda: é a passagem do 1º para o 3º universal. Chomsky introduz desta forma as noções de **Competência** (fato da estrutura lingüística profunda) e **Empenho** (fato da estrutura superficial), bem como o conceito de “**falante nativo ideal**” (para o qual o empenho contempla toda a competência) e “**falante nativo concreto**” (especializado sócio-culturalmente a partir do falante ideal). Desta forma, o fenômeno de comunicação aparece como a capacidade de especialização do relacionamento entre a estrutura profunda (regras inatas) e a estrutura superficial (semântica e fonética particular) por **transformação**.

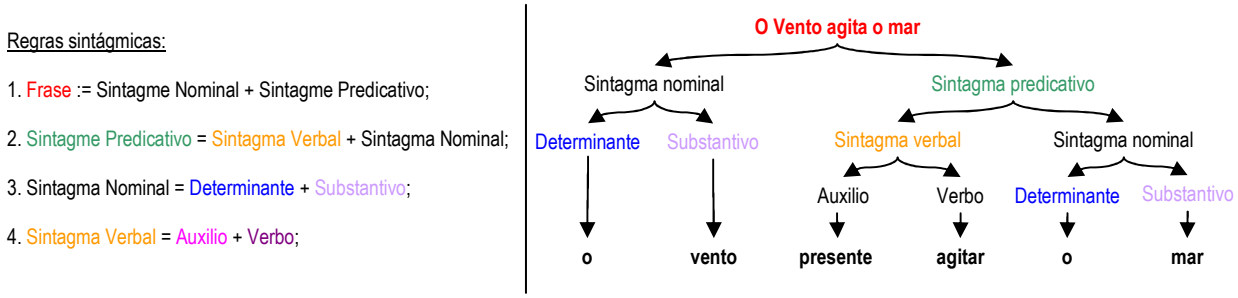


Figura 2.8: As Gramáticas Gerativas Chomskianas

A correlação entre a proposta lingüística de Chomsky e a ciência da computação é direta, da invenção das **linguagens de programação** (gramáticas qualificadas de regulares por Chomsky), aos sistemas de **processamento de linguagens naturais** (que tentem operar com gramáticas sensíveis ao contexto). Um **compilador** é, por exemplo, um autômato finito de essência Chomskiana. Além disto, as interações entre a teoria lingüística de Chomsky e o CR são mutuamente constitutivas, e enxergam na máquina de Von Neumann seu objeto experimental predileto. Da mesma forma que Fodor utiliza as gramáticas gerativas para explicitar a cognição como uma manipulação formal finita de símbolos, ou “Linguagem dos

pensamentos", Chomsky utiliza o Mentalês como apoio epistemológico a sua teoria dos universais. Não é então um acaso se a lingüística Chomskiana lembra tanto a abordagem fenomenológica aristotélica do conhecimento, tanto na predicação a partir de categorias (ou regras sintáticas para Chomsky), como na teoria da informação da alma da aptidão ao exercício (idêntica à passagem da Competência ao Desempenho de Chomsky). Conseqüentemente, sendo uma abordagem fenomenológica aristotélica, como a proposta funcionalista da IAS, inadequada do ponto de vista epistemológico, desde a refutação de um empirismo ontológico ingênuo (através da fenomenologia transcendental), deverão ser guardadas das gramáticas gerativas unicamente **suas propriedades espacializantes**, através da manipulação formal do símbolo, sem nunca cair na armadilha de querer assimilar a cognição a esta manipulação. Em outras palavras, nos mapeamentos inteligentes, o processamento formal de linguagem natural (que será estudado a seguir), apenas servirá de ferramenta para viabilizar interações semióticas naturais, sem nunca ser, por si, responsável pela cognição embutida no sistema computacional. Como será detalhado, na terceira parte deste capítulo, a cognição será o fato da **autonomia autopoiética** de uma **sociedade de agentes**, e não da manipulação funcional de símbolos.

2.2.4. Processamento de Linguagem Natural

Para definir o objetivo da IA, Turing propõe em 1950 um critério de resposta à pergunta "As máquinas podem pensar?", através do "Jogo da Imitação"[49], também conhecido como "**Teste de Turing**" [TUR50]. Sustentando uma abordagem funcionalista simbólica da cognição, este teste inaugura o Processamento de Linguagem Natural (PLN) como disciplina essencial das ciências cognitivas clássicas. Efetivamente, para "provar que algo pensa", Turing afirma que este deve poder nós convencer, através de interações lingüísticas naturais datilografadas, que é humano. A abordagem Turingiana, criticável a partir da argumentação de Searle[50], propõe desta forma uma inteligência desencarnada puramente lingüística.

Embora a máquina de Von Neumann tinha sido originalmente projetada para aplicações numéricas, o PLN, objetivando "a construção de programas capazes de interpretar

[49] O "Jogo da Imitação" é definido por Turing da seguinte maneira: três pessoas, um homem (A), uma mulher (B) e um interrogador (C) encontram-se em salas separadas e podem se comunicar através de mensagens datilografadas. O objetivo do jogo para (C) é descobrir através de perguntas e respostas o sexo de (A) e de (B). (B) tenta ajudar (C) através das suas afirmações, e (A) tenta enganá-lo. Se, substituindo (B) por uma máquina, (C) acerta ou erra da mesma forma em descobrir o sexo de (A) e (B), sem mesmo notar que (B) não é humano, o Teste de Turing é concluinte para a máquina substituindo (B). [TUR50], p. 39-40.

e gerar informações em linguagem natural" [VIE00], p. 2, cresceu rapidamente a partir de 1950. Em 1958, McCarthy inventa a linguagem **LISP** (*List Processing*), com o intuito de operacionalizar o processamento recursivo de listas de símbolos. Na mesma época, Masterman projeta a estrutura de **Redes Semânticas** (RS) para representação do conhecimento. Para Nijholt [NIJ88], um propulsor essencial da área de PLN foi a guerra fria[51], incentivando aplicações militares de criptologia e tradução automática. Em 1965, Weizenbaum constrói em LISP o programa **ELIZA**[52], capaz de dialogar em inglês sobre qualquer tópico, através de simples regras de manipulação simbólica. As gramáticas gerativas de Chomsky [CHO65] são, em 1965, uma contribuição essencial. Em 1969, Schank propõe o modelo de **Dependências Conceituais** (DCs). Wood descreve em 1970 as ATNs (*Augmented Transition Networks*) para a compreensão da linguagem natural. O desenvolvimento por Colmerauer em 1972 da linguagem **PROLOG** (*Progammation Logique*) orienta a criação de sistemas de PLN baseados em lógica dos predicados (LP). Minksy cria em 1975 o conceito de "***Frames***" para representação semântica de objetos. Os anos 80 vêem o aperfeiçoamento da geração de texto em linguagem natural (POLITICS ou TEL-SPIN[53] são exemplos clássicos) bem como uma série aplicações comerciais de tratamento de texto. Os anos 90 estenderam os sistemas de PLN à indexação e recuperação de informação em texto, a partir de modelos estocásticos como os ***Markov Models*** (MM) e ***Hidden Markov Models*** (HMM) [JUR00], além de ver a aparição de numerosas aplicações de reconhecimento e geração de fala.

Todos os sistemas de PLN, qualquer que seja o paradigma que implementam, são conseqüências diretas das lingüísticas Saussuriana e Peirciana, e acabam tendo propriedades espacializantes diretamente conectadas a noção de gramática gerativa. Desta forma, pode-se especificar o processo de PLN a partir de cinco etapas essenciais: (i) **aquisição**, (ii) **análise léxico-morfológica**, (iii) **análise sintática**, (iv) **análise semântica**, e (v) **pragmática** (Figura 2.9). A fase (i) consiste em transformar uma série de sinais visuais (escrita) ou auditivos (fala) em morfemas[54], a fase (ii) reconhece[55] cada morfema como elemento de um léxico, a fase (iii)

---

[50] Trata-se da "Câmara Chinesa" e do problema de ancoragem dos símbolos à uma semântica [SEA80].
[51] Os projetos envolvendo tradução se multiplicaram até chegar-se, em 1966, a uma situação que contabilizava mais de 20 milhões de dólares gastos no Estados-Unidos.
[52] ELIZA se tornou um brinquedo popular na sua versão que "simulava" o diálogo de um psicoterapeuta.
[53] POLITICS [SCH81], p 308-317, é um programa capaz de ler fatos extraídos de jornais e fazer prognósticos ou responder a perguntas sobre futuras ações políticas. TELL-SPIN [SCH81], p. 197-226 é um programa capaz de inventar e contar histórias em linguagem natural, após ter interagido com um usuário.
[54] O morfema divide um fluxo contínuo de sinais (fonemas) em unidades morfológicas semânticas.
[55] O tratamento morfológico reconhece a classe, o gênero, o número, etc, do morfema, levando em consideração flexões gramaticais.

realiza árvores de derivação Chomskianos para relacionar os morfemas, a fase (iv) constrói uma representação semântica baseada em um determinado paradigma de PLN (DCs, RSs, ATNs, MMs, HMMs, etc.), e a fase (v) resolve ambigüidades e finalmente encaminha ações computacionais pragmáticas a partir do entendimento da frase e do contexto.

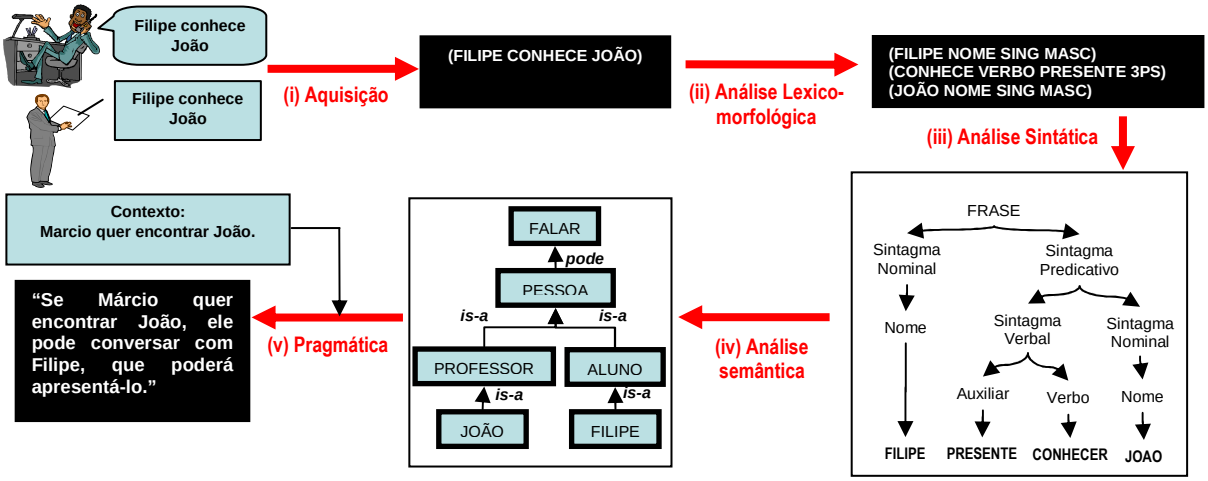


Figura 2.9: Etapas Genéricas do PLN.

Para a construção de mapeamentos em ambiente de colaboração computacional escrita, as etapas (i) e (ii) são triviais. Sendo a etapa (iii) chomskiana por essência, apenas as fases (iv) e (v) deverão ser objeto de uma escolha. A Figura 2.10 mostra alguns paradigmas à disposição para a representação computacional de conhecimentos declarativos: a *LP* consiste em representações Fregianas, as *RSs* mapeiam relacionamentos semânticos em uma rede de conceitos, os *Frames* se traduzem por uma modelagem orientada-objeto, as ATNs permitem processar frases com autômatos finitos a transição de estados, e as DCs modelam recursivamente o sentido pragmático de frases. No caso da criação de mapeamentos cognitivos, desde a escolha de uma abordagem fenomenológica transcendental do conhecimento, o PLN deve apenas ser um **meio**, e não um fim. Como já argumentado, a cognição não deve ser o fato da natureza das interações semióticas (visão heterônoma), mas sim da natureza da agência em si (visão autônoma). Por esta razão qualquer um dos paradigmas apresentado na Figura 2.10 é válido para a realização de mapeamentos inteligentes. Como uma escolha deve ser feita, será usado o paradigma das DCs (como será apresentado no capítulo 3) por dois motivos: sua orientação à modelagem do ***entendimento*** das frases, com a transformação de qualquer linguagem natural em uma meta-linguagem

conceitual vinculando recursivamente primitivas semânticas e conceitos, e sua simplicidade e eficiência de implementação.

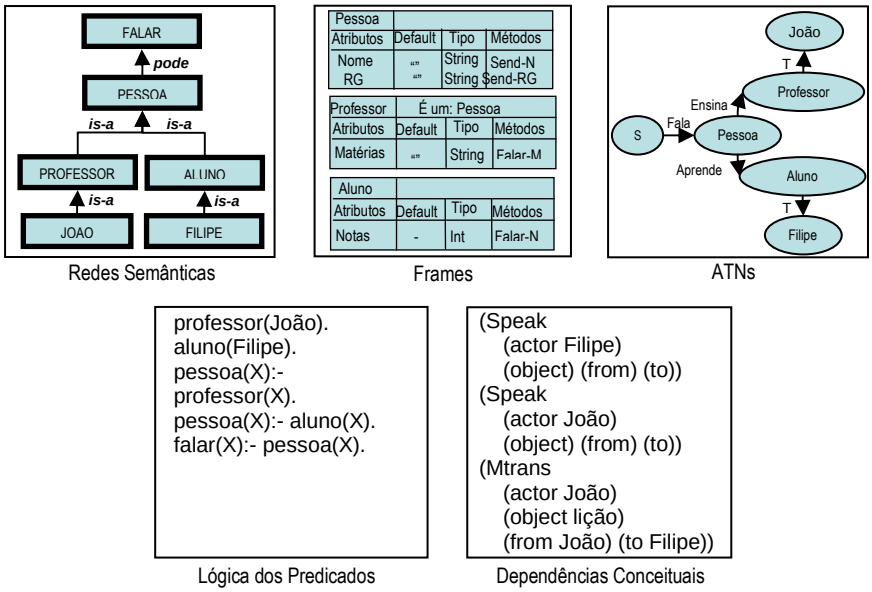


Figura 2.10: Paradigmas de Representação Computacional da Linguagem Natural.

2.3. Agência Autopoiética, Autonomia e Emergência

A fenomenologia transcendental Husserliana aparece muitas vezes como apenas um exercício intelectual, certo interessante, porém desprovido de qualquer possibilidade de aplicação pragmática. Ao contrário, da mesma forma que a ascese metodológica cartesiana leve a espacialização do conhecimento, percebido como objetividade ontológica, através do simbolismo funcional, a redução fenomenológica leva a proposta de uma "fenomenologia biológica [...] como segunda cibernética" (segundo Moreno [MOR94], p. 24), também chamada de "construtivismo não objetivista radical" (segundo Stewart [STE99], p83), mais conhecida como teoria da ***Autopoiésis*** (segundo Maturana e Varela [MAT76]). Como último elemento de sustentação epistemológica à construção de mapeamentos inteligentes de conhecimento, serão detalhados a seguir o paradigma da Autopoiésis, e suas conseqüências sobre a noção de **Agente Autônomo** em IA. Tal orientação nos levará finalmente a propor

uma abordagem relativista nova da TI no seu apoio à GC, a hermenêutica do objeto computacional, ou "**gênesis tecnológica do sentido**"[56].

2.3.1. Autopóiesis

Como enfatizado previamente, o foco dado na inteligência simbólica e lingüística tem como conseqüência a desencarnação da cognição, que, se representa para Fodor o fim de "um chauvinismo antropocêntrico", gera também o paradoxo de uma espacialização da consciência fora do corpo, que é, porém, o seu lugar biológico. A partir desta constatação, os neurobiólogos e filósofos Humberto Maturana e Francisco Varela estabeleceram uma teoria **encarnada** da cognição, visando a responder, de forma não reducionista, e sem cair na armadilha funcionalista, às perguntas "**O que é a vida?** Que tipo de máquinas são os seres vivos? Qual é a sua fenomenologia, incluindo reprodução e evolução, a partir da sua organização unitária? O que é próprio dos sistemas vivos desde a sua origem, e permanece invariável durante as suas sucessivas gerações?". A resposta que encontrariam seria o próprio fato da organização do ser vivo, que seria denominado *Autopoiésis*.

O trabalho de parceria entre Maturana e Varela começou nos anos 60 no Chile, onde fundaram uma comunidade de pesquisa a respeito dos atributos essenciais dos sistemas vivos, que levar-lhes-á, nos anos 70, a propor o paradigma da *Autopoiésis* [MAT76]. A partir dos anos 80, Varela passa interessar-se mais especificamente para a cognição biológica como "encarnada", sendo diretor de pesquisa no C.N.R.S. na França, aonde morreu em 2001, após ter dedicado sua existência ao entendimento do vivo. Maturana continua professor na Universidade do Chile, em Santiago, aonde recebeu o Premio Nacional de Ciências em 1994.

A abordagem essencial de Maturana e Varela consiste em estudar as propriedades do cognitivo como propriedades do **biológico**, tentando isolar as condições mínimas necessárias para afirmar de uma entidade que é viva. Reivindicando-se de uma epistemologia fenomenológica Husserliana, eles começam por claramente negar toda objetividade ontológica, e enfatizar que o entendimento da cognição como um fenômeno biológico deve levar em conta o **observador** como um sistema vivo bem como seu papel. Definem depois a noção essencial de **unidade** para a operação de **distinção** do observador:

---

[56] *Technological Genesis of Meaning*, como descrito por Havelange em "*Mémoire collective: la constitution technique da la cognition*" [HAV99b], p. 105.

> «A operação cognitiva básica que realizamos como observadores é a operação de distinção. Através dessa operação, especificamos uma unidade como uma entidade distinta do seu meio ambiente, caracterizamos ambos unidade e ambiente com as propriedades as quais esta operação lhes fornece e especificamos sua diferenciação. Uma unidade assim especificada é uma unidade simples que define através de suas propriedades o espaço no qual ela existe e o domínio fenomenal que ela pode gerar na sua interação com outras unidades.» [MAT80]:XIX

A partir desta caracterização, Maturana e Varela introduzem uma dicotomia entre **estrutura** e **organização**, afirmando que "as relações entre os componentes que definem uma unidade composta (sistema) como uma unidade composta de um tipo em particular, constituem a sua **organização**", e que "os componentes atuais (com todas as suas propriedades incluídas) e as atuais relações existentes entre eles, que realizam concretamente o sistema como um membro em particular da classe de unidades compostas a qual ela pertence pela sua organização, constituem a sua **estrutura**." [MAT80]:XIX. Esta distinção permite evitar um reducionismo da noção de vida a "vida como a conhecemos", sendo que as características mínimas de um sistema vivo (e, logo, cognitivo) deverão ser buscadas na sua organização e não sua estrutura. Como o nota Moreno, o dualismo organização-estrutura abre o novo programa de pesquisa da "**vida artificial**", a partir do momento que o entendimento de *organização* é puramente formalista: "Tudo o que acontece em um sistema não é um problema físico, mas sim dinâmico" (Prigogine citado em [MOR94], p. 35).

A partir destes elementos essenciais, Maturana e Varela definem finalmente os seres vivos como um tipo particular de **máquinas homeostáticas**[57,] denominadas de *autopoiéticas*. Etimologicamente falando, *Autopoiésis*, do grego *autos* e *poiein* significa "auto-produção". O termo é definido pelos autores da seguinte forma: "**um sistema autopoiético é organizado como uma rede de processos de produção de componentes que (i) regeram continuamente através das suas transformações e interações a rede que os produziu, e (ii) constituem o sistema na qualidade de unidade concreta no espaço aonde existem, especificando o domínio topológico onde se realizam como rede**" [VAR79], p. 45. A idéia de Autopoiésis expande a de homeostase em duas direções: ela coloca todas as referências da homeostase **internas** ao sistema, e afirma ou produz a **identidade** do sistema. Conseqüentemente, o sistema autopoiético, além de ser capaz de se auto-produzir

[57] Sistemas em que o mecanismo responsável pela sua estabilidade é interno ao próprio mecanismo, ou seja, nos quais as fronteiras são definidas pela própria organização, são tipos especiais de máquinas chamadas de homeostáticas.

continuamente de forma fiel, sempre se mantém topologicamente diferenciado do seu meio através de uma fronteira que é a sede de intercâmbios dissipativos com o meio no qual existe. Para os autores, tal definição é suficientemente abrangente para abraçar todos os sistemas vivos, da célula a uma sociedade, passando pelos animais biológicos. Por oposição aos sistemas autopoiéticos, os sistemas não-vivos (e, logo, não cognitivos), são qualificados de **alopoiéticos**. Tais sistemas, **heterônomos** na sua essência, sempre precisam de uma causa externa para desempenhar transformações: sua estabilidade, sua conservação e sua delimitação topológica não são definidas de forma autônoma. Tipicamente, um **sistema especialista** é alopoiético: nada nele produz as regras que irão guiar seu desempenho, sendo que todas elas são inseridas de forma heterônoma a partir de um especialista humano (que é, ele, autopoiético). Do ponto de vista biológico, a Autopoiésis joga abordagens filogenéticas da vida ao segundo plano. O que importa no ser é muito mais a autoconservação que a transmissão de características genéticas por reprodução: **a mitose é muita mais adequada à definição da vida do que a meiose e a reprodução**.

A definição da vida como *Autopoésis* implica claramente uma mudança de perspectiva de "**Cognição = Computação**" para "**Cognição = Autonomia**" [CAS2003b], p. 4. O entendimento da noção autopoiética de *autonomia*, necessita o detalhamento dos conceitos de **fechamento operacional** e **emergência**, que será realizado a seguir.

2.3.2. Enação, Fechamento Operacional e Autonomia

Para que uma máquina seja auto-determinada através das relações entre os processos de produção que a constrói de forma invariante, ela deverá, segundo os autores, exibir as características de: (i) não ter "entradas" ou "saídas", (ii) ser **autônoma** e ter **individualidade**.

Para chegar à noção de autonomia, o pesquisador deve se desfazer das algemas funcionalistas da primeira cibernética, que levam a uma percepção comportamentalista da cognição. Desta forma, Varela questiona a nossa interpretação do cachorro que vira a cabeça na nossa direção "para poder nos ver melhor" [VAR79]. Para o autor, imputar uma intenção ao cachorro é negar sua autonomia, na medida que consiste em considerar que produz uma "saída" comportamental em função do "estímulo" recebido. Ao contrário da visão heterônoma funcionalista do tipo "entrada-saída", o autor propõe então perceber a fenomenologia do sistema autopoiético em termos de "**perturbação-dissipação**". Nesta perspectiva, o que leva

o cachorro a virar a cabeça é **seu mecanismo interno de equilibração** que dissipa a **perturbação** recebida, de forma a poder manter sua **identidade autopoiética**:

> «Nas interações entre os seres vivos e o meio ambiente dentro da congruência estrutural, as perturbações do ambiente não determinam o que acontece com o ser vivo; ao contrário é a estrutura do ser vivo que determina o que deverá ocorrer com ele. Esta interação não tem uma dimensão instrutiva, porque ela não determina as mudanças que deverão ocorrer [...]. Neste sentido, [...] as mudanças que resultam da interação entre os seres vivos e os seus ambientes são certo ocasionadas por agentes perturbadores, porém determinadas pela estrutura do sistema perturbado.» [MAT92], p. 96.

A **ontogêneses** do sistema autopoiético, ou seja, a conservação da sua individualidade topológica e estrutural, não distingue as perturbações advindas do meio ambiente das internas: os dois tipos de perturbação são amarrados na fenomenologia da máquina viva.

A partir da distinção entre acoplamento por entrada-saída e por perturbação-dissipação, os autores introduzem a noção de **enação**. Se a percepção não consiste em uma recepção sensorial ontologicamente objetiva do "mundo como é", isto significa que o mundo percebido é tanto construído pela minha percepção que a minha percepção é por ele construída: quando vejo uma mesa, a mesa do mundo é tão responsável por minha percepção, que minha percepção é responsável pela mesa do mundo. Esta construção circular cria interações recursivas entre a percepção e o percebido cuja estabilização faz **emergir** o mundo. Isto significa que não existe algo como uma percepção objetiva, mas sim uma fenomenologia particular conseqüência da estabilização de enações com o mundo. O que Husserl colocava nas palavras "toda consciência é consciência de", Maturana é Varela aplicam na definição da cognição como "**emergência enativa**", onde saber significa **estabilizar um mundo percebido** em uma historia de acoplamentos estruturais. Neste paradigma, as categorizações não são pré-dadas pelo mundo, mas co-dadas com o mundo. Como a cognição é emergência enativa, ela deve ser encarnada: "com a palavra *encarnada*, queremos enfatizar que a cognição depende dos tipos de experiências que decorrem do fato de ter um corpo dotado de diversas capacidades sensori-motoras e que tais capacidades individuais se inscrevem elas mesmas em um contexto biológico, psicológico e cultural mais amplo." [VAR91], p 234.

Para finalizar a conceitualização necessária a definição da **autonomia**, os autores introduzem a noção de **fechamento operacional**:

> «O acoplamento por entradas consiste em considerar que o sistema é essencialmente determinado por entradas que refletem certas características do ambiente, são absorvidas como matéria bruta, e trabalhadas no interior. [...] O acoplamento por fechamento operacional, ao contrário, consiste em pensar que o sistema é definido essencialmente por seus diversos modos de coerência interna, decorrente da sua interconectividade.» [VAR79], p. 199.

Para os autores, o determinismo organizacional interno do sistema autopoiético é conseqüência do seu fechamento operacional, definido pelo fato que os procedimentos internos responsáveis pela coesão das partes do sistema e pela sua própria homeostase auto-referenciada são: (i) dependentes recursivamente os uns dos outros para a geração e a realização deles mesmos e (ii) constituem o sistema como uma unidade reconhecível onde existem. Tal fechamento se constitui portanto ele mesmo como uma enação estável dos procedimentos responsáveis pela existência do sistema: um sistema operacionalmente fechado pode ser definido como um **equilíbrio autoprodutivo**. É importante ressaltar que o fechamento operacional não é comparável à noção de retro-ação heterônoma de Wiener[58]: no fechamento operacional, a exterioridade não é determinante.

Os autores chegam então a noção de **autonomia**, como capacidade de subordinar toda transformação à conservação da identidade afirmada pelo próprio funcionamento. Um sistema autônomo é operacionalmente fechado, e suas relações com o mundo são enativas, ou seja, mutuamente constitutivas. Ele possui seus próprios mecanismos internos de resistência às agressões do ambiente. A "**percepção**" que possui não é objetiva, mas sim fenomenológica no sentido que constitui tanto o mundo que é por ele constituída. Para o observador, tal sistema possui uma ***intencionalidade*** "espontânea", na medida que exibe "**vontade**" e "**pró-atividade**". Deve-se ressaltar a importância da teoria da **complexidade** no entendimento dos sistemas autopoiéticos. Em última instância, eles podem ser constituídos por sub-sistemas eles mesmos autopoiéticos, o que torna sua organização **fractal**[59] (Figura 2.11).

[58] Jean-Pierre Dupuy, em [DUP94], p. 69, esclarece o "encontro perdido" da cibernética com seu objetivo na construção equivocada da terminologia "*feed-back*. Epistemologicamente, Wiener define o "*feed-back*" como um mecanismo heterônomo de determinação de um sistema, e propõe a cibernética como a ciência do comando (do latim *kubernètès*, o homem que dirige). Para Dupuy, o fechamento do sistema deve ser auto-determinado para se chegar a noção de cognição: é o que chama de segunda cibernética.

[59] As figuras geométricas Fractais (do latim *frangere*, irregular), principalmente estudadas por Benoit Malbranche entre 1965 e 1975, possuem a propriedade principal de invariabilidade da estrutura a várias escalas.

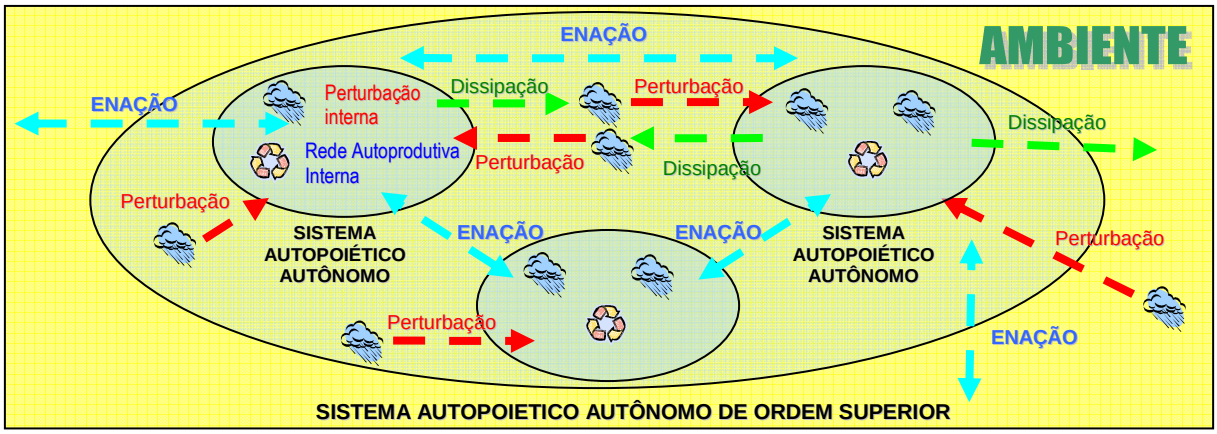


Figura 2.11: Constituição Fractal de uma Máquina Autopoéitica.

Longe de serem uma teoria não determinista, as "**matemáticas do caos**"[60] se interessem especificamente a estabilidade topológica de tais sistemas complexas. O grande desafio da pesquisa em vida artificial, no contexto da Autopoiésis, consiste portanto em encontrar **atratores emergentes estáveis** entre os "aleatórios de fato" dos sistemas autônomos construídos.

2.3.3. Agência e Sociedade de Agentes

Pode-se agora propor o elemento central possibilitando a realização de mapeamentos inteligentes: ***a orientação-agentes***. Tal programação envolve a criação de entidades computacionais **autônomas** em sistemas distribuídos chamados de **Sistemas Multiagente** (SMA, ou *Multiagent Systems* – MAS).

Apesar de serem hoje objetos clássicos da Inteligência Artificial Distribuida[61] (IAD, ou *Distributed Artificial Intelligence* – DAI), agentes e SMAs são ainda alvos de várias dúvidas conceituais, como colocado por Franklin em [FRA96], p.1: "os agentes parecem simples programas de computador: qual seria a diferença?". No mesmo artigo, o autor propõe um estado da arte da noção de agente através de: a definição *AIMA*[62] – "qualquer coisa que pode ser visto como capaz de perceber seu ambiente com sensores e agir nele com atuadores",

[60] As chamadas "matemáticas do caos", ou "caos determinístico" são o resultado do estudo das propriedades dos sistemas dinâmicos dirigidos por equações não lineares, como os sistemas meteorológicos. [SCI91], p. 15.
[61] Segundo Scalabrin, [SCA99], "a IAD, visando à modelagem de um sistema distribuído, tem como base os seguintes preceitos: problemas fisicamente distribuídos e heterogêneos em termos funcionais, redes e visão distribuída, problemas complexos e visão local, sistemas adaptáveis".
[62] Definição dada por Russel e Norvig em [RUS95], p. 33.

a definição *MAES*[63] – "sistemas computacionais presentes em ambientes dinâmicos complexos, que sentem e agem de forma autônoma nestes ambientes, e realizam objetivos ou tarefas para os quais foram desenhados", a definição *KIDSIM*[64] – "entidade de Software persistente dedicada a um objetivo específico", a definição *HAYES-ROTH*[65] – "sistemas realizando de forma contínua três funções: percepção dinâmica das condições do ambiente, ações afetando estas condições, e raciocínio para interpretar percepções, resolver problemas, executar inferências e determinar ações", a definição *WOOLDRIDGE-JENNINGS*[66] – "sistema computacional baseado em hardware ou software possuindo as seguintes características: autonomia, habilidades sociais, reactividade, e pró-atividade". A estas abordagens podem ser acrescentadas as características propostas por Ferber [FER99] – "entidade física ou virtual capaz de agir no seu ambiente, de se comunicar com outros agentes, que é dirigida por um conjunto de tendências, que possui seus próprios recursos, que pode perceber seu meio, e possui uma representação limitada dele, que possui habilidades e pode oferecer serviços, que pode ser capaz de se reproduzir, e cujo comportamento tende a satisfazer seus objetivos", e por Scalabrin [SCA96] – "entidade autônoma que possui conhecimentos especializados, uma representação destes conhecimentos, uma representação do problema a ser resolvido, uma representação parcial do seu ambiente, protocolos de comunicação especializados, e um modelo das suas intenções".

Estas propostas levam a uma definição empírica trivial de agente como um programa exibindo as características originais de: (i) ser **autônomo**, **social**, **pró-ativo** e **persistente**, (ii) possuir **sensores**, **executores**, um **ambiente**, **objetivos**, uma **agenda** própria e **conhecimentos**, (iii) poder **raciocinar**, **dialogar**, **negociar**, **coordenar** e **colaborar**, (iv) precisar de **confiança**. Como estas características são *ad-hoc*, e demonstram uma certa redundância, deve-se efetuar uma redução da entidade agente, para extrair suas propriedades essenciais. Baseando-se na Autopoiésis, chega-se a três características: uma própria - a **autonomia**, e duas emergentes - a **intencionalidade,** e a **identidade topológica**. Pode-se definir a **autonomia** como um homeostatismo autoreferenciado realizado por fechamento operacional, a **intencionalidade** como um fenômeno emergente, para o observador, da enação com o mundo, e a **identidade topológica** como um espaço emergente, para o observador, de

[63] Definição dada por Maes em [MAE95], p. 108.
[64] Definição dada por Smith, Cypher e Spohrer em [SMI94], p. 56.
[65] Definição dada por Hayes -Roth em [HAY95], p. 330.
[66] Definição dada por Wooldridge e Jennings em [WOO95], p. 2.

realização da autonomia. Em conclusão, pode-se afirmar que um **agente de software** é uma **estrutura autopoiética** particular, que: (i) regera continuamente através das suas transformações e interações a rede (de regras) que o produz, e (ii) constitui um sistema na qualidade de unidade concreta no espaço onde existe, especificando o domínio topológico aonde se realiza. A Figura 2.11 resume a arquitetura essencial do agente de Software.

A partir da definição do agente, pode-se realizar uma conceituação dos SMAs. Várias definições empíricas ad-hoc podem ser encontradas na literatura: Durfee, Lesser e Corkill[67] falam de uma "rede fracamente acoplada de entidades capazes de resolver problemas, que trabalham em conjunto para encontrar respostas que vão além das capacidades e conhecimento de cada uma delas", Jennings, Sycara, e Wooldridge[68] de um "sistema composto de múltiplos componentes autônomos demonstrando as seguintes propriedades: cada agente possui capacidades incompletas para resolver um problema, não existe um controle global do sistema, os dados são descentralizados e a computação assíncrona", Scalabrin[69] de um "sistema que comporta os seguintes elementos: um ambiente (E), um conjunto de objetos (O) situados e passivos em E, um conjunto de agentes (A) que são objetos específicos ($A \subseteq O$), um conjunto de relações (R) que liga objetos e agentes, um conjunto de operações (Op), permitindo aos agentes de (A) perceber, produzir, consumir, transformar e manipular objetos de (O)", Shmeil[70] de "sistema composto de: (i) um ambiente *Am*, (ii) um conjunto *Ag* de agentes, e (iii) um conjunto *Ob* de objectos, não agentes". Em resumo, um SMA é classicamente definido como um sistema composto de um conjunto de agentes, capaz de resolver de forma **distribuída** problemas que cada agente envolvido não poderia resolver só, sem que seja necessário qualquer controle centralizado *ad-hoc* do processo.

[67] Definição dada em [DUR89], p. 78.
[68] Definição dada em [JEN98].
[69] Definição dada em [SCA96] e [FER99].
[70] Definição dada em [SHM99].

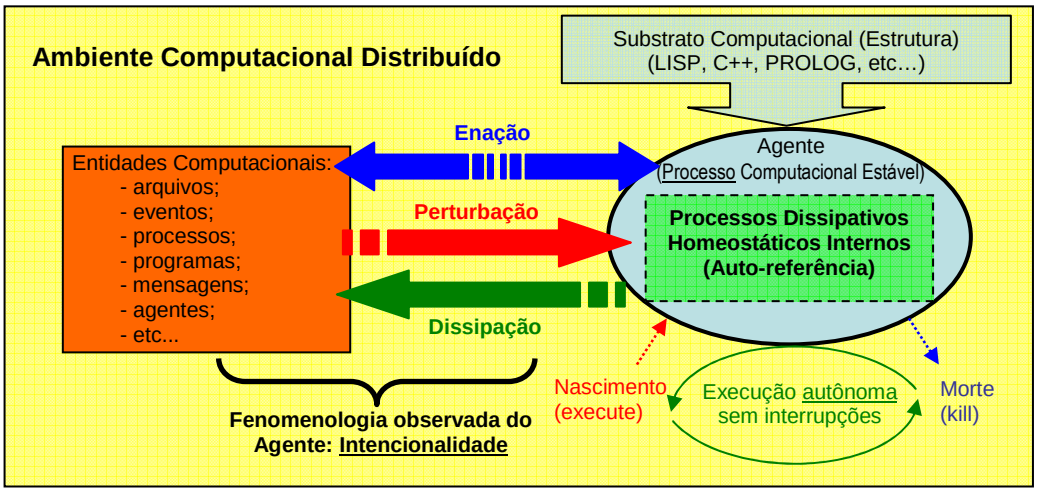


Figura 2.12: Arquitetura Essencial de um Agente de Software.

A partir da conceituação autopoiética da natureza da agência, pode-se definir em última instância um SMA como uma **sociedade autopoiética** (ou seja, uma máquina autopoiética de ordem superior, compostas por sub-máquinas, elas mesmas, autopoiéticas), ou seja, uma organização fractal possuindo três características: **autonomia**, **intencionalidade,** e **identidade topológica.** Chega-se desta forma a conclusão que o SMA e o agente são o mesmo objeto fractal observado a escala diferente. Para não cair na objeção da regressão infinita, deve-se acrescentar que uma máquina autopoiética pode ser composta de partes alopoiéticas, que são, no caso do agente de software, **rotinas simbólicas funcionais**. A Figura 2.13 ilustra a natureza dos SMAs na perspectiva de uma fenomenologia biológica. Müller, em [MUL99] p.75, adota uma proposta similar na descrição da multiagência como uma estrutura dissipativa auto-organizada na qual a resolução de problemas é um fenômeno emergente, acrescentando que toda a problemática, neste paradigma, é garantir a superposição do atrator do sistema dissipativo e da solução do problema, ou seja, a **convergência** da arquitetura[71].

[71] Müller, em [MUL99], p.78, esclarece: “De uma certa maneira, a dissipação é o preço que uma organização deve pagar a segunda lei da termodinâmica em troca de ordem. Para que a entropia diminua a nível macro (SMA), ela deve ser eliminada por mecanismos dissipativos a nível micro (Agente)”

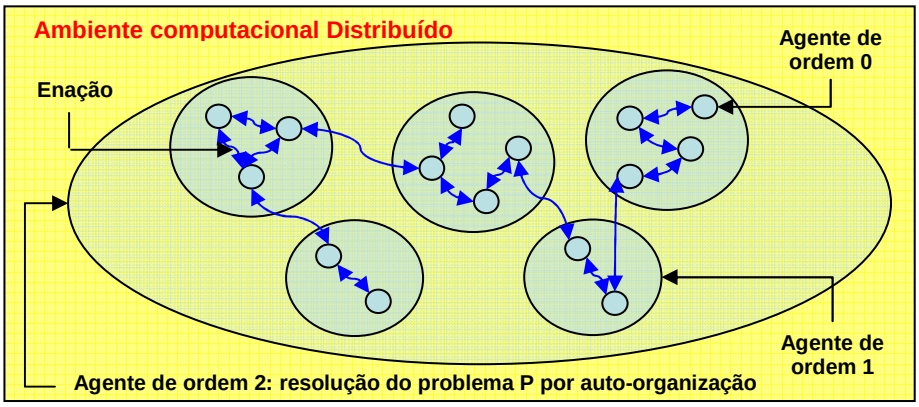


Figura 2.13: Arquitetura Fractal Auto-Organizada de um SMA.

2.3.4. Hermenêutica do Objeto Tecnológico

Duvidar da natureza representacional do saber, e negar, através de um construtivismo relativo da percepção, a possibilidade de acessar ao mundo "como é", constitui uma mudança radical de perspectiva de grande impacto sobre o entendimento do objeto da IA. Na visão ontológica objetivista clássica, o mundo pode ser exatamente percebido através dos sentidos (visão aristotélica, para a qual a linguagem é adequada para o acesso ao mundo), ou através de representações matematicamente construídas (visão cartesiana, para a qual a matematização *a priori* é adequada para o acesso ao mundo). No CR, o raciocínio pode ser mecanizado através da manipulação formal finita de símbolos. O objetivo essencial de tais "ciências cognitivas" consiste em querer "**mecanizar o humano**". Porém, o fracasso desta tentativa, desde a impossibilidade de ter cognição sem interpretação (*symbol-grounding-problem*), e desde a proposta de um construtivismo da percepção (enação) leva a adotar uma perspectiva oposta: "**humanizar a máquina**", ou seja, pensar a **inserção hermenêutica do objeto tecnológico**.

A proposta original de uma abordagem hermenêutica da cognição deve ser atribuída ao filósofo alemão Heidegger, na sua oposição a Husserl, quanta a forma de explanação da experiência vivenciada. Se Husserl se concentra no fenômeno individual, Heidegger propõe a noção de "***Dasein***", como "sentido localizado do ser no mundo" [HEI62]. Para Heidegger, a forma básica da existência é a compreensão, contextualizada em um determinado lugar a um determinado momento. Esta ênfase dada à interpretação situada é chamada de perspectiva **hermenêutica**, também descrita por Annels [ANN96] como um processo interpretativo que busca compreensão e revelação do fenômeno através da linguagem, e por Kvale [KVA96] como o estudo das atividades culturais humanas, buscando o sentido através da interpretação.

A proposta construtivista de Maturana e Varella, o saber como emergência advindo da influência circular acoplando uma máquina autopoiética ao seu ambiente, é claramente hermenêutica, na medida que o mundo é mais "**interpretado**" através da enação, do que objetivamente percebido. Sendo uma sociedade de agentes (ou de seres de forma geral), fractal, o paradigma da Autopoiésis pode ser aplicado de forma geral ao entendimento dos relacionamentos interpretativos em todo tipo de conjunto social:

> «Sistemas sociais são sistemas auto-referenciados baseados em comunicações significativas. Eles usam comunicações para constituir e interconectar os eventos que constroem o sistema. Neste sentido, eles são sistemas autopoiéticos. Eles existem apenas para reproduzirem os eventos que servem como componentes do sistema.» (Luhmann, citado em [PRO00], p. 23)

Neste contexto, devemos nos interrogar a respeito da posição do artefato tecnológico de forma geral, e do computador, de forma específica, em uma sociedade vista como autopoiética. Como todo objeto do mundo, ele é interpretado, e deve logo entrar em enação com os agentes que somos. Através das ciências da informação, descobrimos que esta enação possui uma característica muito particular: **ela é constitutiva de saber**. A partir desta constatação, da visão do artefato computador como "algo que pensa", chega-se à visão de algo que "**dá matéria para pensar**" [CAS03c], p. 12. Esta seria o que Havelange chama a "**gênese artefatual (ou tecnológica) do saber**": « Antropologicamente constitutiva, a técnica intermedia a representação através de uma memória externa inscrita nos objetos materiais. Fenomenologicamente, ela instaura uma gênese técnica da intencionalidade que modifica o compartilhamento tradicional entre o empírico e o transcendental.» [HAV03], p. 5.

Trata-se de uma perspectiva radicalmente diferente para o objeto da IA, cujo objetivo não é mais construir máquinas que pensam, ou imaginar que o pensamento possa ser mecânico, mas sim construir máquinas **hermeneuticamente inseridas** e **adaptáveis**, em sociedades autopoiéticas compostas por pessoas e artefatos. Este novo posicionamento passa antropologicamente por uma transição do modo gráfico de racionalidade para um modo artefatual (ou computacional, no caso da IA). Isto coloca a pedra final do edifício epistemológico necessário à criação de mapeamentos inteligentes: o fato de serem "inteligentes" não deve significar que possuem um processamento de conhecimento (o que não faz sentido na perspectiva adotada), mas sim que devem sempre ser naturalmente (hermeneuticamente) inscritos em uma determinada sociedade, em um determinado lugar, em um determinado momento.

2.4 Conclusão

Para poder sustentar uma estratégia de GC, é necessário realizar mapeamentos de conhecimento, que, para serem eficientes, devem ser (i) **construídos de forma automática**, (ii) **de entendimento comum entre todos os agentes** envolvidos no seu uso, (iii) **completos** e **consistentes**, (iv) **atualizados em "tempo real" em uma "memória corporativa dinâmica"**, e, (v) capazes de fornecer **pró-ativamente, em linguagem natural, informações pertinentes, personalizadas e contextualizadas.** A realização pragmática destes "mapeamentos inteligentes" não é trivial, e necessite uma sólida conceituação epistemológica permitindo tomar decisões essenciais sobre a cognição nas dimensões (i) da própria natureza do conhecimento, (ii) da natureza das entidades capazes de desempenhar a cognição e (iii) da natureza das interações semióticas entre tais entidades.

A Figura 2.14 resume a análise que foi proposta na parte 2.1 sobre teorias do conhecimento, destacando quatro fases filosóficas essenciais: a **percepção ontológica objetivista Aristotélica** (conhecimento como **fenomenológico e lingüístico**), o **dogmatismo scolástico medieval** (conhecimento como **contemplação divina**), o **objetivismo metodológico Cartesiano** (conhecimento como **representação matematicamente construída *à priori***), e a dupla tentativa **computo-representacionalista** e **fenomenológica transcendental** de naturalização da mente. A abordagem escolhida, para a construção de mapeamentos inteligentes, não é a IAS e o computo-representacionalismo, mas sim a visão fenomenológica husserliana de conhecimento como transcendental, complementada pela teoria da **Autopoiésis** para descrever a natureza dos seres cognitivos.

A partir desta escolha, a Figura 2.15 resuma as conclusões atingidas quanta a natureza do signo lingüístico, e das interações semióticas entre seres cognitivos, na parte 2.2. Cabe ressaltar que o acesso semiótico ao mundo é **relativo**, e que não consideramos a linguagem natural como uma ferramenta naturalmente construída para acessar ao mundo "como é", nem como o fato da cognição. Efetivamente, as interações lingüísticas entre máquinas autopoiéticas são apenas uma forma particular de **enação** visando manter a coerência de uma sociedade, como máquina autopoiética de ordem superior. Desta forma, a escolha particular da teoria das **DCs** de Schank para a implementação das interações semióticas entre agentes humanos e agentes virtuais é apenas pragmática sem decorrer de uma obrigação epistemológica.

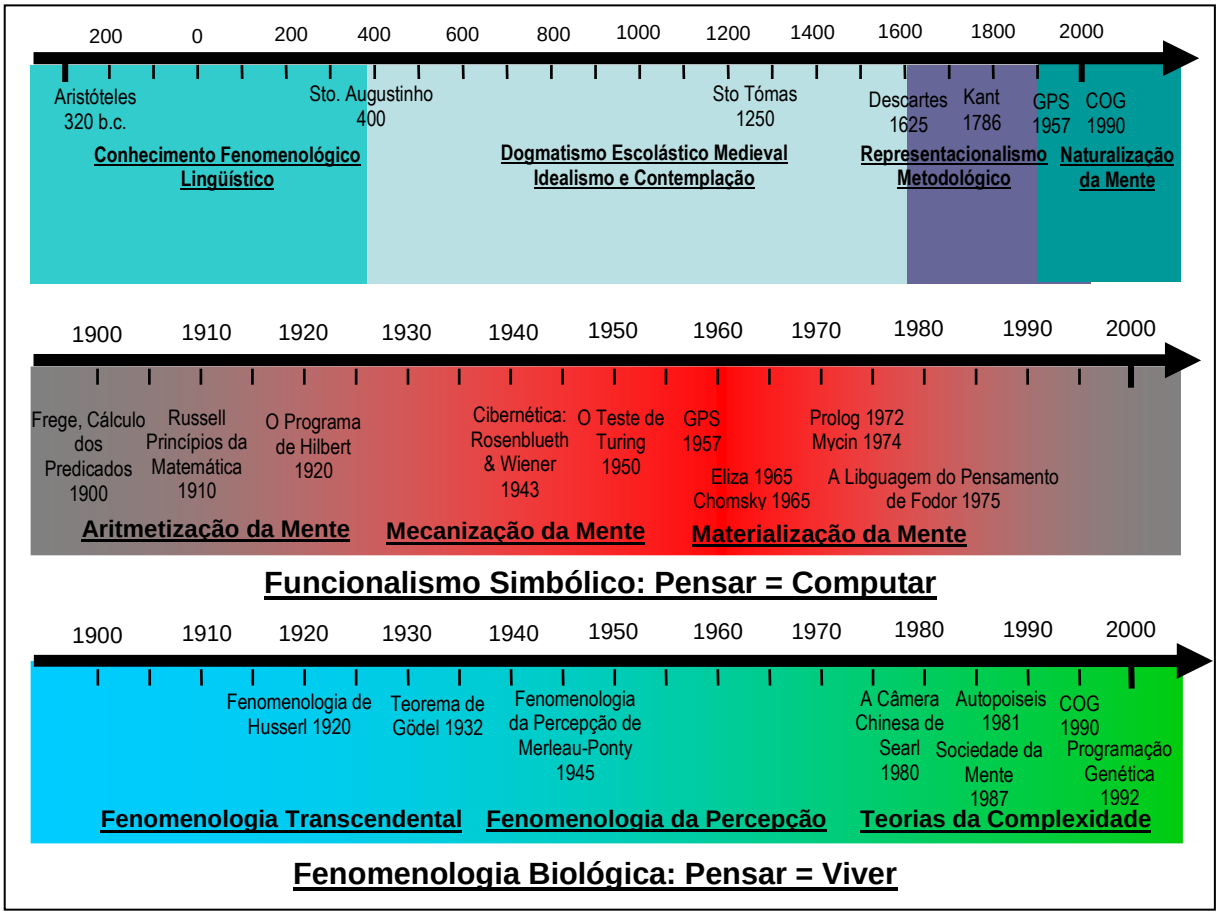
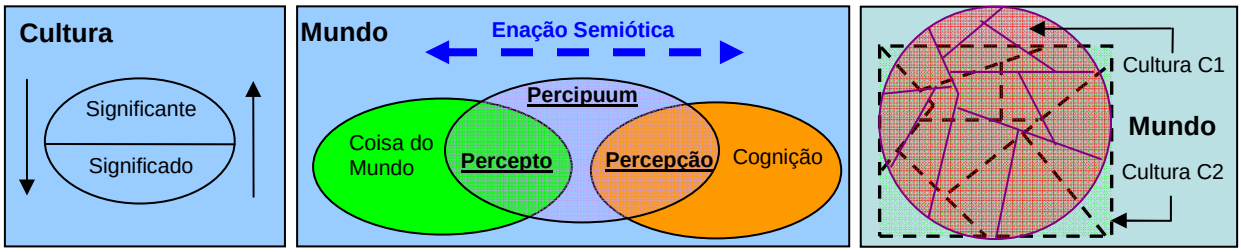


Figura 2.14: Análise Histórica das Teorias do Conhecimento.

Sendo finalmente a escolha **formalista** (a organização pré-vale sobre a estrutura) da Autopoiésis, realizada na parte 2.3, foi também decidida a adoção da **orientação-agentes** na programação de mapeamentos inteligentes. A agência foi definida de forma essencial a partir de três características: a **autonomia** (homeostase auto-referenciada), a **intencionalidade** emergente (ações intencionais emergentes observadas sustentando a homeostase), e a **identidade** topológica (manutenção emergente observada do espaço no qual a homeostase se realiza). A Tabela 2.1 resume finalmente as diferenças epistemológicas e pragmáticas entre a abordagem **heterônoma** da cognição (IAS clássicas, ou primeira cibernética), e a visão **autônoma**.



Signo Lingüístico | Natureza da Enação Semiótica | Relativismo Lingüístico Hermenêutico

Figura 2.15: Signo Lingüístico e Relativismo Hermenêutico das Enações Semióticas.

Será abordada no próximo capítulo a aplicação pragmática das escolhas epistemológicas aqui realizadas, para a implementação de mapeamentos inteligentes de competências em ambientes de colaboração. Um protótipo operacional de SMA será amplamente detalhado e comentado para mostrar como a orientação-agente pode efetivamente tornar possível a programação de entidades computacionais pró-ativas, capazes de apoiar de forma inteligente uma estratégia organizacional de Gestão do Conhecimento.

	Heteronomia	Autonomia
Sustentação Epistemológica	Objetivismo Ontológico	Construtivismo Fenomenológico
Sustentação Científica	Lógica Matemática	Complexidade
Percepção do Mundo	Ontologicamente Objetiva	Fenomenologicamente Construída
Relacionamento c/ o Mundo	Entradas-Saídas	Enação / Ermergência
Lógica de Operação	Função de Transferência	Fechamento Operacional
Conhecimento	Simbólico com Semântica Interpretada	Emergente com Significados Próprios
Metáfora	"Pensar = Computar", "Pensar = Manipular Símbolos"	"Pensar = Viver", "Pensar = Fazer Emergir"
Concretização Tecnológica	Sistemas Simbólicos (Sistemas Especialistas, CBR, etc.)	Agentes e SMAs (organizações autopoiéticas)
Autores Principais	Wiener, Fodor, Chomsky, Turing	Maturana, Varella, Brooks, Minsky.

Tabela 2.1: Heteronomia vs Autonomia.

Capítulo 3

Mapeamento Automático Inteligente de Competências em Ambiente de Colaboração

Após o estudo epistemológico teórico da natureza da cognição e das interações semióticas, este terceiro capítulo apresentará uma forma pragmática de implementação de mapeamentos inteligentes de competências em ambiente de colaboração síncrono (Bate-Papo) ou assíncrono (Comunidades de Prática), a partir dos conceitos até agora estabelecidos. Para tal, a arquitetura do SMA implementada neste trabalho será em primeiro lugar apresentada de forma genérica, e contextualizada através da descrição de cenários clássicos do seu desempenho. Em segundo lugar, a implementação da enação semiótica do real (agentes humanos) para o artificial (agentes de software) será detalhada como "**Neutralização Semiológica**", ou seja, passagem de uma semiótica humana particular para uma semiótica genérica artificial (ou *mapeamento*). Finalmente, a implementação da operação inversa, consistindo em passar do virtual para o real será também explicitada como "**Semiotização Natural**", ou seja, passagem de uma semiótica artificial genérica para uma semiótica humana particular. A descrição das implementações realizadas constituirá desta forma o coração pragmático deste trabalho, que levará, no capítulo 4, à análise e à discussão dos resultados obtidos no uso do SMA implementado.

3.1. Um Ambiente de Colaboração Multiagente

Nas organizações do século XXI, o ativo *Conhecimento* possui um papel fundamental, na medida que representa a única fonte de riqueza que nunca poderá se esgotar, e cujo crescimento sempre será garantido. Porém, para que este ativo possa efetivamente se

multiplicar, ele precisa ser compartilhado, e é unicamente este compartilhamento que garante o aumento do seu valor: o "conhecimento é único ativo que se multiplica quando dividido"[72]. É por este motivo que ambientes de colaboração síncronos e assíncronos são hoje ferramentas empresarias essenciais[73]. Será detalhada, a seguir, uma arquitetura de SMA, que permite automatizar parte destas interações colaborativas geradoras de riqueza, através da sua capacidade de mapear automaticamente (por simples observação) "***quem sabe o que***", e fornecer intencionalmente informações pertinentes, para pessoas pertinentes, à momentos pertinentes, sobre os conhecimentos mapeados na comunidade. Para tal, será necessário explicitar as várias formas de enações semióticas envolvidas na criação de uma meta-organização autopoiética agrupando duas sub-organizações: uma composta por pessoas, e uma outra por agentes de Software.

3.1.1. Uma Arquitetura Multiagente

O problema a ser resolvido consiste em criar um sistema computacional capaz de interagir naturalmente[74] em uma comunidade de profissionais[75] inter-relacionados por tecnologias virtuais de colaboração como Bate-Papos, providenciando, a momentos oportunos, informações pertinentes sobre "*quem sabe o que*". Para os usuários envolvidos, a solução proposta deverá "dar a impressão" que existe um novo "colega conectado", cujo nome é "Agente", e que sempre dá boas dicas para facilitar os relacionamentos entre os membros da rede (conforme ilustrado na Figura 3.1).

Sendo que, em última análise, o problema consiste em inserir hermeneuticamente um mapeamento de competências em uma sociedade autopoiética composta por pessoas e máquinas, a solução adotada para resolvê-lo, desde do estudo conduzido até agora, é a realização de um SMA. Tal SMA deve consistir em um macro-agente de ordem 0, cujas enações com os usuários humanos (que resolvem o problema proposto) são o resultado das enações internas da sociedade de agentes de ordem 1 que o compõe. Sendo a natureza fractal de um sistema autopoiético, cada agente de ordem 1 pode, ele mesmo, ser uma sociedade de

---

[72] Esta lema clássica da economia do conhecimento foi originalmente proposta por K.E. Sveiby em [SVE98], justificando a importância para as organizações da nova economia de medir e gerenciar seus ativos intangíveis.

[73] Vários exemplos desta afirmação (Siemens, Nortel, Texaco, Xerox, etc.), podem ser encontrados em [TER02]. O detalhamento dos vários casos de sucesso de uso de ferramentas de colaboração para sustentar a GC na Siemens pode também ser encontrado em [DAV02].

[74] Deve-se ntender, por interação natural, o fato que o sistema deverá interagir em linguagem natural com os usuários.

agentes de ordem 2, e assim recursivamente até chegar às partes elementares alopoiéticas do SMA, ou seja, o código funcional de manipulação da informação.

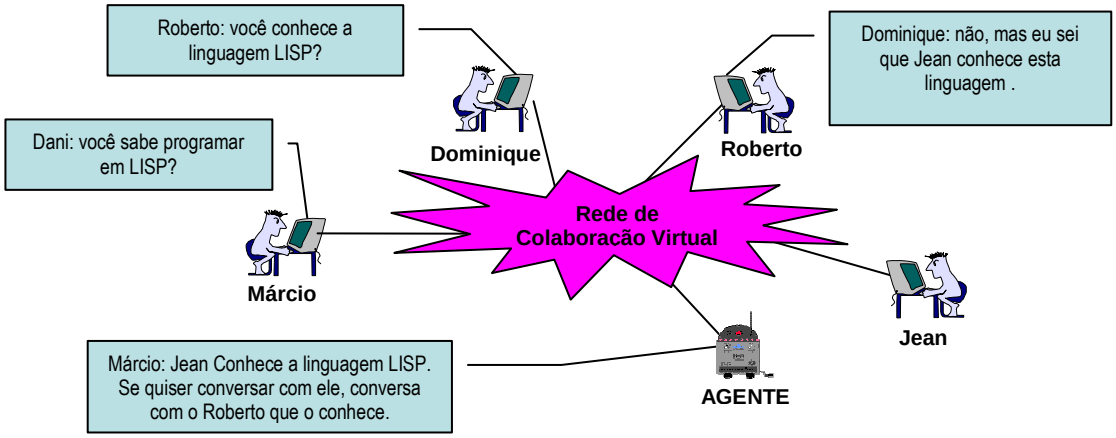


Figura 3.1: Visão do Resultado de um Mapeamento Inteligente de Competências.

A solução proposta, e implementada principalmente em LISP, é detalhada em ordem 1 (sociedade de ordem 1, cujas enações constituem o "AGENTE" da Figura 3.1) na Figura 3.2.

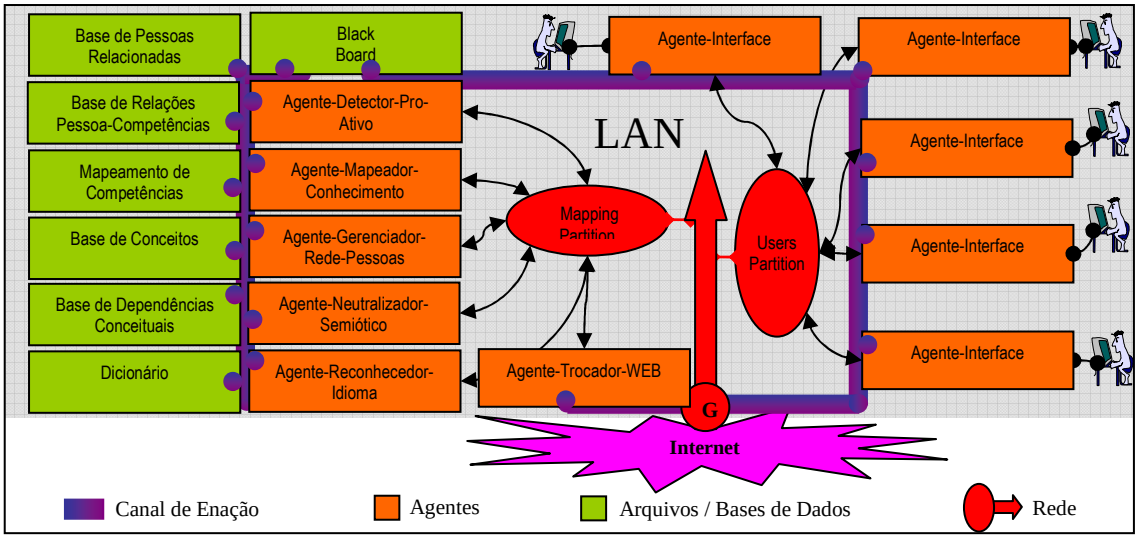


Figura 3.2: Um SMA para o Mapeamento Inteligente de Competências.

[75] Deseja-se dar à solução proposta um caráter geral, no sentido de não orientá-la á uma comunidade particular, como será ilustrado pragmaticamente no capítulo 4.

Para facilitar o entendimento qualitativo[76] do funcionamento do SMA, são descritos, a seguir, dois cenários triviais de auto-organização desta sociedade autopoiética virtual:

- Cenário (i): Mapeamento de Competências. O usuário *Roberto* digita para o usuário *Jean* no Bate-Papo: "I know the C++ language". O agente *Interface*, perturbado por este evento, transforma esta afirmação em uma fórmula LISP do tipo ((ROBERTO JEAN) I KNOW THE C++ LANGUAGE)), que dissipa em direção de *Reconhecedor-Idioma*. *Reconhecedor-Idioma* reconhece, graças a um *Dicionário*, que a frase é escrita em Inglês, e dissipa em direção de *Neutralizador-Semiótico* uma fórmula do tipo ((ROBERTO JEAN ENGLISH) I KNOW THE C++ LANGUAGE)). *Neutralizador-Semiótico*, perturbado por esta fórmula, transforma a semântica expressa em uma Dependência Conceitual[77], usando uma *Base de Conceitos* e uma *Base de Dependências Conceituais*, do tipo ((ROBERTO JEAN ENGLISH) (KPOSS (ACTOR ROBERTO) (OBJECT C++-LANGUAGE))). Tal fórmula é dissipada em direção do agente *Gerenciador-Rede-Pessoas*, que mapeia em uma *Base de Pessoas Relacionadas* às fórmulas (ROBERTO JEAN) e (JEAN ROBERTO), significando "Roberto conhece João" e "João conhece Roberto", e dissipa a mesma fórmula que recebeu originalmente em direção de *Mapeador-Conhecimento*. Este último agente dissipa esta perturbação, registrando a fórmula (KPOSS (OBJECT C++-LANGUAGE)) em um *Mapeamento de Competências*, e o relacionamento entre ROBERTO e esta última fórmula em uma *Base de Relações Pessoas-Competências*.
- Cenário (ii): Proposta Intencional. O usuário *Filipe* digita para o usuário *Dominique* no Bate-Papo: "Do you know the C++ language?". *Interface* é perturbado e dissipa ((FILIPE DOMINIQUE) DO YOU KNOW THE C++ LANGUAGE)) em direção de *Reconhecedor-Idioma*, que dissipa em direção de *Neutralizador-Semiótico* ((FILIPE

[76] A descrição a seguir tem por único objetivo esclarecer o papel de cada agente da macro-sociedade, sem entrar nos detalhes de implementação, que serão abordados a seguir, e nas partes 3.2 e 3.3.
[77] O uso do paradigma das dependências conceituais de Schank ([SCH81], p.9-26), será computacionalmente detalhado na parte 3.2.

> DOMINIQUE ENGLISH) DO YOU KNOW THE C++ LANGUAGE)), que dissipa em direção de *Detector-Pro-Ativo* ((FILIPE DOMINIQUE ENGLISH) (KPOSS (ACTOR DOMINIQUE) (OBJECT C++-LANGUAGE))). *Detector-Pro-Ativo* utiliza o *Mapeamento de Competências*, a *Base de Relações Pessoas-Competências*, e a *Base de Pessoas Relacionadas* para produzir uma fórmula do tipo (I KNOW THE FOLLOWING ABOUT ROBERTO – ROBERTO KNOWS THE C++ LANGUAGE), e dissipar em direção de *Interface*, que dissipa em direção de *Filipe* a frase "I know the following about Roberto – Roberto knows the C++ Language".

Com a simples leitura desta breva descrição qualitativa, o SMA parece exibir uma "inteligência obscura", na medida que consegue sistematicamente auto-organizar-se para resolver o problema de mapeamentos inteligentes de competências. Será mostrado, em seguida, que este efeito emergente, obtido em nível do "AGENTE" de ordem 0, é o resultado não só de enações entre agentes de ordem 1, através do uso de um quadro de avisos (*Black-Board*), mas também entre agentes de ordem 2 que, compondo os agentes de ordem 1, operacionalizam enações de "**Neutralização Semiológica**" e "**Semiotização Natural**".

A Tabela 3.1 detalha a seguir a natureza de cada um dos agentes de nível 1, resumindo o que o perturba, como dissipa esta perturbação, e qual a auto-referência que possui para garantir sua homeostase, e fazer emergir sua intencionalidade.

Agente	Perturbação	Dissipação	Auto-referência (objetivo)
Interface	Frase digitada por usuários ou mensagem no Black-Board.	Frase digitada para usuários ou mensagem no Black-Board.	Transformar frases escritas em linguagem natural em fórmulas LISP e escrever fórmulas LISP em frases de linguagem natural.
Reconhecedor-Idioma	Mensagem no Black-Board.	Mensagem no Black-Board.	Reconhecer se uma frase é escrita, ou não, em um determinado idioma, usando um dicionário.
Neutralizador-Semiótico	Mensagem no Black-Board.	Mensagem no Black-Board.	Transformar frases de linguagem natural em fórmulas semânticas neutras.
Gerenciador-Rede-Pessoas	Mensagem no Black-Board.	Mensagem no Black-Board.	Mapear relacionamentos do tipo "A conhece B" em uma rede de relacionamentos entre agentes (humanos ou não).
Mapeador-Conhecimento	Mensagem no Black-Board.	Mensagem no Black-Board.	Mapear fórmulas semânticas neutras como competências possuídas por pessoas.
Detector-Pró-Ativo	Mensagem no Black-Board.	Mensagem no Black-Board.	Verificar se existem competências e pessoas vinculadas que possam ser pró-ativamente pospostas.
Trocador-WEB	Mensagem no Black-Board.	Mensagem no Black-Board.	Receber e enviar informações semânticas de e para outros agentes em outras LAN (agente Webservice).

Tabela 3.1: Descrição dos agentes de ordem 1 envolvidos no SMA.

A Tabela 3.1 caracteriza a **autonomia** (auto-referência) e mostra a **intencionalidade** emergente dos agentes de nível 1 do SMA. Para que os agentes descritos segam efetivamente a arquitetura essencial, proposta na Figura 2.12, eles devem também exibir uma **identidade topológica**. Tal identidade é pragmaticamente o fato de cada agente ser um **processo** delimitado no ambiente computacional distribuído (um "executável"), cuja natureza algorítimica constitui um fechamento em *loop* ad infinitum:

```
loop
   se perturbação então dissipar(perturbação, regras-autopoieticas);
fim-loop
```

Do SMA completo proposto em 3.2, todos os agentes, menos o *Trocador-WEB* foram efetivamente implementados em LISP (e Delphi no caso do agente *Interface*), sendo cada agente de ordem 1, constituindo o SMA de ordem 0, uma sociedade de agentes de ordem 2. A Figura 3.3 mostra em detalhe a organização fractal de três níveis do protótipo implementado.

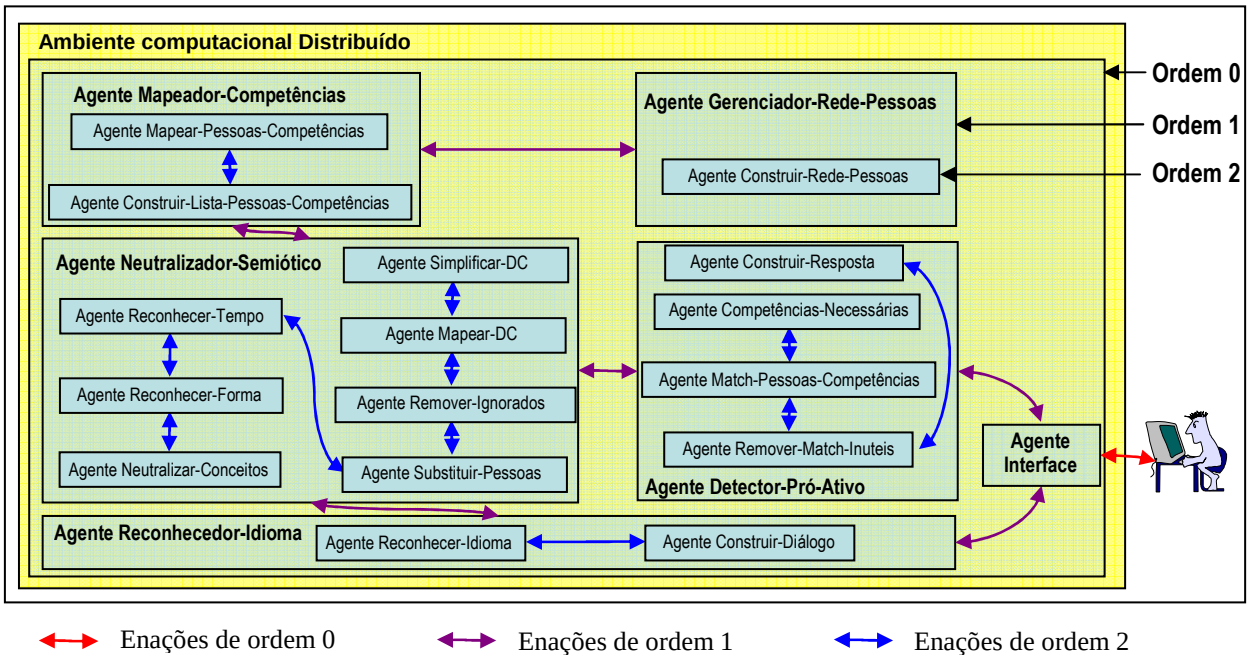


Figura 3.3: Visão Fractal do Protótipo da Arquitetura do SMA.

A Tabela 3.2 detalha a seguir a natureza de cada um dos agentes de ordem 2.

Agente	Perturbação	Dissipação	Auto-referência (objetivo)
Sociedade de ordem 1 **"Reconhecedor-Idioma"**			
Construir-Diálogo	Chamada do Reconhecedor-Idioma	Fórmula LISP contendo um Diálogo	Transformar frases escritas em linguagem natural em diálogos contendo morfemas LISP.
Reconhecer-Idioma	Chamada do Reconhecedor-Idioma	Fórmula LISP com idioma reconhecido	Reconhecer se uma frase é escrita, ou não, em um determinado idioma, usando um dicionário.
Sociedade de ordem 1 **"Neutralizador-Semiótico"**			
Neutralizar-Conceitos	Chamada do Neutralizador-Semiótico	Fórmula LISP com conceitos Neutralizados	Neutralizar os conceitos contidos em listas de morfemas de uma fórmula LISP.
Reconhecer-Forma	Chamada do Neutralizador-Semiótico	Fórmula LISP com forma reconhecida	Localizar a forma (positiva, negativa) empregada nas listas de morfemas de uma fórmula LISP.
Reconhecer-Tempo	Chamada do Neutralizador-Semiótico	Fórmula LISP com tempo reconhecido	Localizar o tempo (passado, presente) empregado nas listas de morfemas de uma fórmula LISP.
Substituir-Pessoas	Chamada do Neutralizador-Semiótico	Fórmula LISP com pessoas substituídas	Localizar pronomes pessoais e substituí-los por conceitos de pessoas em listas de morfemas.
Remover-Ignorados	Chamada do Neutralizador-Semiótico	Fórmula LISP com morfemas não reconhecidos ignorados	Remover de uma fórmula LISP todo morfema que não foi reconhecido conceitualmente.
Mapear-DC	Chamada do Neutralizador-Semiótico	Fórmula LISP com séries de morfemas transformadas em DCs	Transformar listas de morfemas de uma fórmula LISP em DCs, conforme algoritmo *PARSE* de Birnbaum e Selfridge.
Simplificar-DC	Chamada do Neutralizador-Semiótico	Fórmula LISP com DCs simplificadas	Simplificar DCs expressas em LISP, a partir de regras de simplificação.
Sociedade de ordem 1 **"Gerenciador-Rede-Pessoas"**			
Construir-Rede-Pessoas	Chamada do Gerenciador-Rede-Pessoas	Fórmula LISP de uma Rede de Pessoas atualizada.	Atualizar uma Rede de Pessoas a partir de uma fórmula LISP representando um diálogo.
Sociedade de ordem 1 **"Mapeador-Competências"**			
Construir-Lista-Pessoas-Competências	Chamada do Mapeador-Competências	Fórmula LISP contendo uma lista de pessoas e competências a serem mapeadas	Construir uma fórmula LISP contendo binômios Pessoa-Competência a partir de uma fórmula LISP contendo morfemas transformados em DCs.
Mapear-Pessoas-Competências	Chamada do Mapeador-Competências	Atualização de um Mapeamento de Competências e Base Pessoas-Competências	Atualizar fórmulas LISP representando um Mapeamento de Competências e uma Base Pessoas-Competências a partir de uma fórmula LISP contendo binômios Pessoa-Competência.
Sociedade de ordem 1 **"Detector-Pró-Ativo"**			
Competências-Necessárias	Chamada do Detector-Pró-Ativo	Fórmula LISP contendo uma lista de competências requeridas	Construir fórmulas LISP contendo uma lista de competências requeridas a partir de uma fórmula LISP contendo morfemas transformados em DCs.
Match-Pessoas-Competências	Chamada do Detector-Pró-Ativo	Fórmula LISP contendo lista de pessoas e competências requeridas detectadas	Construir listas LISP de pessoas e competências detectadas, a partir de competências requeridas, Mapeamento de Competências e Base Pessoas-Competências.
Remover-Match-Inúteis	Chamada do Detector-Pró-Ativo	Fórmula LISP contendo a lista a mais pertinente no contexto de pessoas e competências requeridas.	Remover de uma fórmula LISP contendo uma lista de pessoas e competências requeridas detectadas as pessoas e competências não relevantes no contexto.
Construir-Resposta	Chamada do Detector-Pró-Ativo	Fórmula LISP contendo uma lista de morfemas realizando uma intervenção lingüística	Construir uma lista de morfemas realizando uma intervenção lingüística, a partir de uma fórmula LISP contendo uma lista de pessoas e competências requeridas detectadas.

Tabela 3.2: Descrição dos agentes de ordem 2 envolvidos no SMA.

Para completar o entendimento da arquitetura de SMA proposta, é apresentada a seguir, de forma sucinta[78], a natureza de cada elemento computacional manipulado pelos agentes de ordem 1, e, por delegação, pelos agentes de ordem 2:

- O ***Black-Board***[79] é uma lista de recados postados e lidos pelos agentes, sendo cada recado uma lista (Emissor, Destinatário, Objetivo, Conteúdo). Exemplo de estado do ***Black-Board***:

```
((Detector-Pro-Ativo Interface Speak "FILIPE KNOWS THE LISP LANGUAGE")
 (Neutralizador-Semiotico Mapeador-Conhecimento Map (KPOSS (ACTOR FILIPE) (OBJECT DANI))))
```

Observação: neste exemplo, o Black-Board contém duas mensagens, um pedido do Detector-Pro-Ativo para o Interface de falar "FILIPE KNOWS THE LISP LANGUAGE", e um pedido do Neutralizador-Semiotico para o Mapeador-Conhecimento de mapear as competências (KPOSS (ACTOR FILIPE) (OBJECT DANI))))

- O **Dicionário** é uma lista de definições de um determinado idioma, sendo cada definição uma dupla (palavra, definição-conceitual). Exemplo de **Dicionário**:

```
((ALGORITHM LIST-OF-STEPS) (AN POINTER-A) (FILIPE PERSON-NAMED-FILIPE) (KNOWS TO-KNOW))
```

Observação: neste exemplo, o **Dicionário** define quatro palavras: Algorithm, An, Filipe, Knows.

- A **Base de Conceitos** é uma lista de conceitos neutros (independentes de idioma), sendo cada conceito uma dupla (nome-conceito, tipo-conceito). O tipo-conceito pertence à lista (ACTION DIRECTION INTERROGATION PERSON POINTER POSITION QUALIFICATION OBJECT). Exemplo de **Base de Conceitos**:

```
((LIST-OF-STEPS OBJECT) (POINTER-A POINTER) (PERSON-NAMED-FILIPE PERSON) (TO-KNOW ACTION))
```

Observação: neste exemplo, a **Base de Conceitos** possui quatro conceitos: LIST-OF-STEPS, POINTER-A, PERSON-NAMED-FILIPE, TO-KNOW.

[78] Um maior detalhamento, bem como exemplos prototípicos, serão apresentados nas partes 3.2 e 3.3.

[79] O uso de um Black-Board (quadro de aviso), é uma técnica clássica de comunicação em sistemas multiagentes. Tal técnica é descrita de forma detalhada em [CRA95].

- A **Base de Dependências Conceituais** é uma lista de traduções de conceitos em termos de formação de dependências conceituais, conforme definido por Schank[80]. Exemplo de Base de Dependências Conceituais:

```
((defword PERSON-NAMED-FILIPE
  ((assign *part-of-speech* 'person *cd-form* *WORD*))))
```

Observação: neste exemplo, a Base de Conceito possui uma única DC definida, PERSON-NAMED-FILIPE. Exemplos mais complexos serão apresentados no detalhamento da natureza da Neutralização Semiológica.

- O **Mapeamento de Competências** é uma lista de dependências conceituais mapeadas, representando competências. Exemplo de **Mapeamento de Competências**:

```
((KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED LIST-OF-STEPS))))
(KPOSS (OBJECT  (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DANI)))))
```

Observação: neste exemplo, o **Mapeamento de Competências** possui duas DCs: (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED LIST-OF-STEPS)))), significando "conhecer um programa", e (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DANI)))), significando "conhecer uma pessoa chamada Dani".

- A **Base de Relações Pessoas-Competências** é uma lista de relações, definidas por uma dupla (Pessoa, Índice-Competência-possuída-no-Mapeamento-Competências). Exemplo de **Base de Relações Pessoas-Competências**:

```
((PERSON-NAMED-FILIPE 1) (PERSON-NAMED-JACK 2) (PERSON-NAMED-DANI 2))
```

Observação: neste exemplo, a **Base de Relações Pessoas-Competências** possui três relações: (PERSON-NAMED-FILIPE 1), significando que o Agente *Filipe* possui a competência indexada em 1 na **Base de Competências**, (PERSON-NAMED-JACK 2), significando que o Agente *Jack* possui a competência indexada em 2, (PERSON-NAMED-DANI 2), significando que o Agente *Dani* possui a competência indexada em 2.

[80] Uma dependência conceitual é uma estrutura recursiva, relacionando um ator (ACTOR), um objeto (OBJECT), uma origem (FROM) e um destino (TO), a partir de uma primitiva conceitual pertencendo à lista (KPOSS, ATRANS, PTRANS, PROPEL, MOVE, GRASP, INGEST, EXPEL, MTRANS, MBUILD, SPEAK, ATTEND), conforme definido em [SCH81], p.9-26

- A **Base de Pessoas Relacionadas** é uma lista de relacionamentos, sendo cada relacionamento uma dupla (Pessoa-Conhecedora, Pessoa-Conhecida). Exemplo de **Base de Pessoas Relacionadas**:

```
((PERSON-NAMED-JOHN PERSON-NAMED-JACK) (PERSON-NAMED-JACK PERSON-NAMED-JOHN))
```

Observação: neste exemplo, a **Base de Pessoas Relacionadas** possui duas relações: (PERSON-NAMED-JOHN PERSON-NAMED-JACK), significando que o Agente *Filipe* conhece o Agente *Jack*, (PERSON-NAMED-JACK PERSON-NAMED-JOHN), significando que os Agentes *Jack* e *John* se conhecem.

O que faz finalmente emergir auto-organização à ordem 0, e permite a delegação de tarefas à ordem 2, são as regras de homoestase auto-referenciada (autonomia) implementada em cada agente de ordem 1, detalhadas a seguir de forma simplificada[81]:

- Agente Interface:

```
loop
Se  (recado_recebido(BlackBoard))
então falar(extrair_conteudo(meu_recado(BlackBoard)));
Se mensagem_enviada(Bate-Papo)  então
  Inicio
  M:= mensagem_enviada(Bate-Papo);
  enviar_recado(Interface Reconhecedor-Idioma Consultar ultima_frase(M));
  enviar_recado(Interface Reconhecedor-Idioma Mapear todas_frases(M));
  fim
fim-loop
```

- Agente Reconhecedor-Idioma:

```
loop
Se  recado_recebido(BlackBoard) então
  Inicio
  formula:=extrair_conteudo(meu_recado(BlackBoard));
  perturbar(Construir-Diálogo, formula, dissipação-construir-diálogo);
  perturbar(Reconhecer-Idioma, dissipação-construir-diálogo , dissipação-reconhecer-idioma);
  formula:= dissipação-reconhecer-idioma;
  Se  recado_recebido(BlackBoard) = “Consultar”
  então enviar_recado(Reconhecedor-Idioma Neutralizador-Semiótico Consultar formula);
  Se  recado_recebido(BlackBoard) = “Mapear”
  então enviar_recado (Reconhecedor-Idioma Neutralizador-Semiótico Mapear formula);
  Fim
fim-loop
```

[81] Para o leitore interessado, o código completo dos agentes pode ser consultado no apêndice B. Nas regras autopoiéticas simplificadas propostas, utiliza-se a primitiva perturbar(Agente-Perturbado, Perturbação-Enviada, Dissipação-Emitida-Pelo-Agente), para provocar a perturbação de um agente.

- Agente Neutralizador-Semiótico:

```
loop
Se  recado-recebido(BlackBoard) então
  Inicio
  formula:=extrair_conteudo(meu_recado(BlackBoard));
  perturbar(Neutralizar-Conceitos, formula, dissipação-neutralizar-conceitos);
  perturbar(Reconhecer-Forma, dissipação-neutralizar-conceitos, dissipação-reconhecer-forma);
  perturbar(Reconhecer-Tempo, dissipação-reconhecer-forma, dissipação-reconhecer-tempo);
  perturbar(Substituir-Pessoas, dissipação-reconhecer-tempo, dissipação-substituir-pessoas);
  perturbar(Remover-Ignorados, dissipação-substituir-pessoas, dissipação-remover-ignorados);
  perturbar(Mapear-DC, dissipação-remover-ignorados, dissipação-mapear-DC);
  perturbar(Simplificar-DC, dissipação-mapear-DC, dissipação-simplificar-DC);
  formula:= dissipação-simplificar-DC;
  Se  recado_recebido(BlackBoard) = "Consultar"
  então enviar_recado (Neutralizador-Semiótico Detector-Pro-Ativo Consultar formula);
  Se  recado_recebido(BlackBoard) = "Mapear"
  então enviar_recado (Neutralizador-Semiótico Mapeador-Conhecimento Mapear formula);
  Fim
fim-loop
```

- Agente Mapeador-Competências:

```
loop
Se  recado_recebido(BlackBoard) então
  Inicio
  formula:=extrair_conteudo(meu_recado(BlackBoard));
  perturbar(Construir-Lista-Pessoas-Competências, formula, dissipação-construir-lista-p-c);
  perturbar(Mapear-Pessoas-Competências, dissipação-construir-lista-p-c, Base-Competências, Base-
  Relações-Pessoas-Competências);
  enviar_recado(Mapeador-Conhecimento Gerenciador-Rede-Pessoas Mapear formula);
  Fim
fim-loop
```

- Agente Gerenciador-Rede-Pessoas:

```
loop
Se  recado_recebido(BlackBoard) então
 Inicio
  fórmula:=extrair_conteudo(meu_recado(BlackBoard));
  perturbar(Construir-Rede-Pessoas, fórmula, Base-Pessoas-Relacionadas);
  Fim
fim-loop
```

- Agente Detector-Pró-Ativo:

```
loop
Se  recado_recebido(BlackBoard) então
 Inicio
  formula:=extrair_conteudo(meu_recado(BlackBoard));
  perturbar(Competências-Necessárias, formula, dissipação-competências-n);
  perturbar(Match-Pessoas-Competências, dissipação-competências-n, dissipação-match-p-c);
  perturbar(Remover-Match-Inúteis, dissipação-match-p-c, dissipação-remover-match-inuteis);
  perturbar(Construir-Resposta, dissipação-remover-match-inuteis, dissipação-construir-resposta);
  formula:= dissipação-construir-resposta;
  enviar_recado(Detector-Pro-ativo Interface Consultar formula);
  Fim
fim-loop
```

3.1.2. Enações no Mundo Real

Sendo a conceituação adotada para resolver o problema do mapeamento inteligente, uma visão formal da Autopoiésis, pode-se realizar neste momento uma discussão pragmática da proposta de Luhmann, contida na afirmação que "sistemas sociais são sistemas auto-referenciados [...] [que] usam comunicações para constituir e interconectar os eventos que constroem o sistema, [sendo], neste sentido, [...] autopoiéticos"[82]. Analisando a solução proposta na Figura 3.2, podemos verificar, em primeiro lugar, a existência de duas sociedades autopoiéticas: **uma composta por pessoas** (a comunidade de profissionais inter-relacionados por tecnologias de colaboração), e **uma composta por artefatos virtuais** (o SMA). A junção destas sociedades, através da ação do agente *Interface*, faz emergir uma terceira sociedade autopoiética **composta por pessoas e tecnologias**. Finalmente, através das enações externas de um agente humano (que pode pertencer a várias sociedades de homens) e do Trocador-WEB, este modelo pode ser infimamente complexificado, dando nascimento a uma visão **sócio-fractal da constituição da cognição**, como ilustrado na Figura 3.4

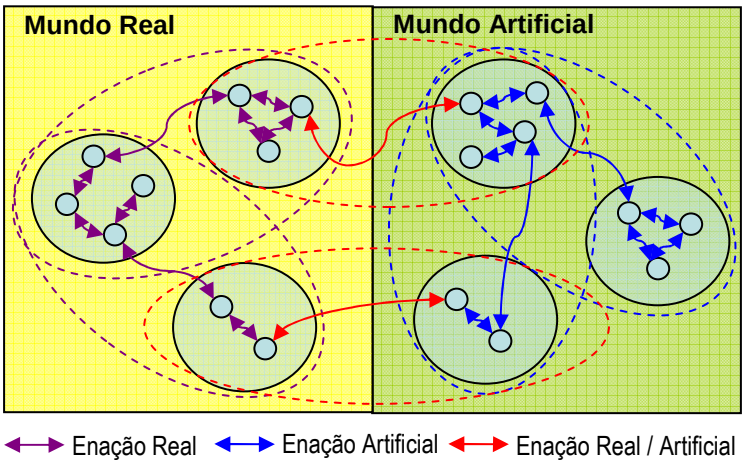


Figura 3.4: Estrutura Sócio-Fractal da Constituição da Cognição.

Sendo esta análise, o trabalho experimental realizado não poderia ser considerado completo se não levasse em consideração, não apenas enações do mundo virtual (entre agentes de software), e enações entre o real e o virtual (entre homens e máquinas), mas também enações entre agentes do mundo real. Por este motivo, foi realizado, em Delphi, um simulador de Bate-Papo, no qual um remetente (*From*), pode enviar mensagens (*Content*)

[82] Luhmann Citado p. 65.

para um destinatário (*To*), conforme ilustrado na Figura 3.5. O simulador permite, portanto, acompanhar as enações semióticas constitutivas da coesão autopoiética da sociedade de pessoas. Cabe observar que o modo de uso deste simulador é detalhado no apêndice A2.

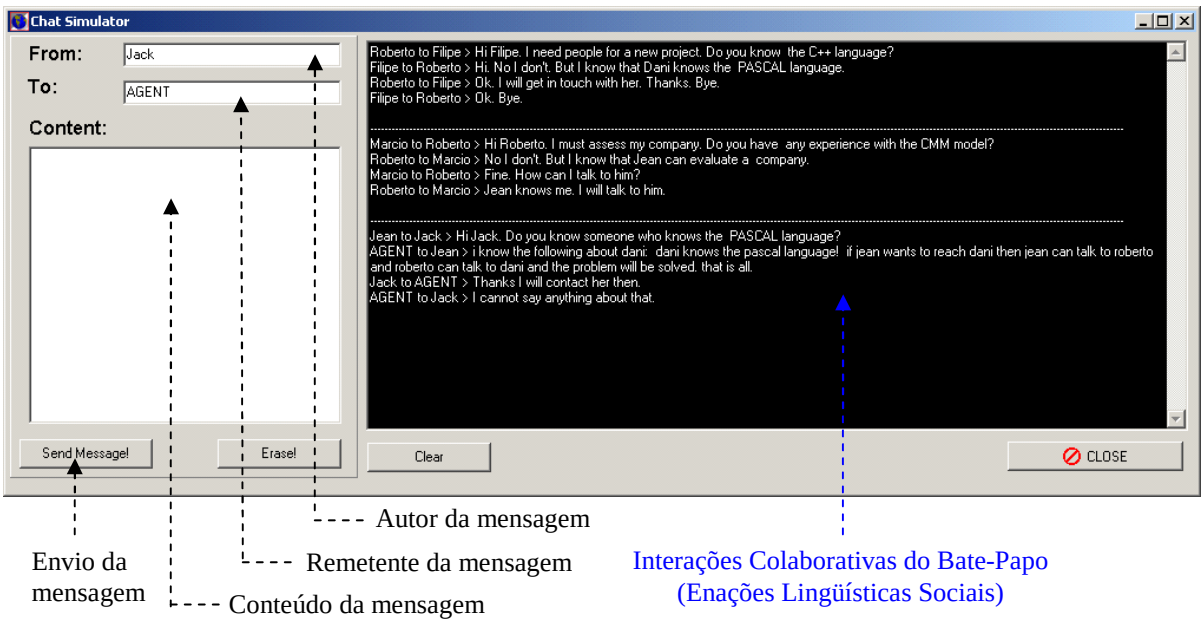


Figura 3.5: Simulador de Bate-Papo para Visualização das Enações no Mundo Real.

Como será detalhado na parte 3.1.4, este simulador constitui também o Agente *Interface*, possuindo o código permitindo o desempenho das enações entre o Real e o Virtual.

3.1.3. Enações no Mundo Artificial

Da mesma forma que os agentes humanos podem colaborar através de um Bate-Papo, garantindo através de enações semióticas a coerência autopoiética da sociedade que constituem, os agentes virtuais devem também poder colaborar em um “Bate-Papo-Virtual”, para também manter a coerência da sua sociedade (o SMA). Para os agentes de ordem 1, tal Bate-Papo é constituído por um quadro de avisos (***Black-Board***).

A idéia do quadro de avisos é muito simples: trata-se de uma lista de mensagens, emitidas e lidas no formato (Agente-Emissor, Agente-Destinatário, Objetivo-Mensagem, Conteúdo-Mensagem). Esta lista é constituída na medida que os agentes, na dissipação das perturbações que recebem, escrevem no quadro mensagens uns para os outros. No protótipo realizado, as variáveis Agente-Emissor e Agente-Destinatário podem receber os valores 'Interface' (designando o Agente *Interface*), 'Language-Recognizer' (designando o

Reconhecedor-Idioma), 'Neutralizer' (designando o Neutralizador-Semiótico), 'Pro-Ac' (designando o Detector-Pró-Ativo), 'People-Net-Manager' (designando o Gerenciador-Rede-Pessoas), e 'Skill-Mapper' (designando o Mapeador-Conhecimento). A variável Objetivo-Mensagem pode receber os valores 'Consult' (se o agente deseja auto-organizar um acesso consultivo ao mapeamento inteligente) ou 'Map' (se o agente deseja auto-organizar uma alteração no mapeamento inteligente).

Pode-se agora verificar que a auto-organização inteligente obtida nos cenários (i) e (ii) da parte 3.1.1 não tem nada de "obscura"[83]. A Tabela 3.3 detalha a seqüência típica de mensagens trocadas pelos agente de ordem 1 no cenário (i):

BLACK-BOARD			
Emissor	Destinatário	Objetivo	Conteúdo
Interface	Language-Recognizer	Map	@BLOCK@ @ACTOR@ Roberto @ACTOR@ @ACTOR@ Jean @ACTOR@ @TEXT@ I Know the C++ Language. @TEXT@ @BLOCK@
Language-Recognizer	Neutralizer	Map	((ROBERTO) (JEAN) ((ENGLISH UNKNOWN NORMAL UNKNOWN) (I KNOW THE C++ LANGUAGE)))
Neutralizer	People-Net-Manager	Map	((PERSON-NAMED-ROBERTO) (PERSON-NAMED-JEAN) ((ENGLISH PRESENT NORMAL POSITIVE) (KPOSS (ACTOR (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO))) (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-C++) (QUALIFIED NORM-OF-EXPRESSION))))))
People-Net-Manager	Skill-Mapper	Map	((PERSON-NAMED-ROBERTO) (PERSON-NAMED-JEAN) ((ENGLISH PRESENT NORMAL POSITIVE) (KPOSS (ACTOR (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO))) (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-C++) (QUALIFIED NORM-OF-EXPRESSION))))))

Tabela 3.3: Detalhes das Enações Artificiais no Cenário (i)

Da mesma forma, a Tabela 3.4 detalha a seqüência típica de mensagens trocadas pelos agentes de ordem 1 no cenário (ii):

BLACK-BOARD			
Emissor	Destinatário	Objetivo	Conteúdo
Interface	Language-Recognizer	Consult	@BLOCK@ @ACTOR@ Filipe @ACTOR@ @ACTOR@ Dominique @ACTOR@ @TEXT@ Do you know the C++ language? @TEXT@ @BLOCK@
Language-Recognizer	Neutralizer	Consult	((FILIPE) (DOMINIQUE) ((ENGLISH UNKNOWN INTERROGATION UNKNOWN) (DO YOU KNOW THE C++ LANGUAGE)))
Neutralizer	Pro-Ac	Consult	((PERSON-NAMED-FILIPE) (PERSON-NAMED-DOMINIQUE) ((ENGLISH PRESENT INTERROGATION POSITIVE) (KPOSS (ACTOR (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DOMINIQUE))) (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-C++) (QUALIFIED NORM-OF-EXPRESSION))))))
Pro-Ac	Interface	Consult	(I KNOW THE FOLLOWING ABOUT ROBERTO - ROBERTO KNOWS THE C++ LANGUAGE ! THERE IS NO WAY THAT FILIPE CAN REACH ROBERTO WITH HIS RELATIONSHIPS. THAT IS ALL.)

Tabela 3.4: Detalhes das Enações Artificiais no Cenário (ii)

[83] Ver página 73.

O detalhamento da criação das fórmulas contidas no campo *Conteúdo* do *Black-Board* será feito nas próximas partes deste capítulo, onde serão explicitadas computacionalmente as operações de Neutralização Semiológica e Semiotização Natural.

Uma outra interface foi realizada em Delphi para permitir acompanhar o desempenho dos agentes de ordem 1, e a troca de mensagens realizada no *Black-Board* (Figura 3.6). Cabe observar que o modo de uso desta interface é detalhado no apêndice A2.

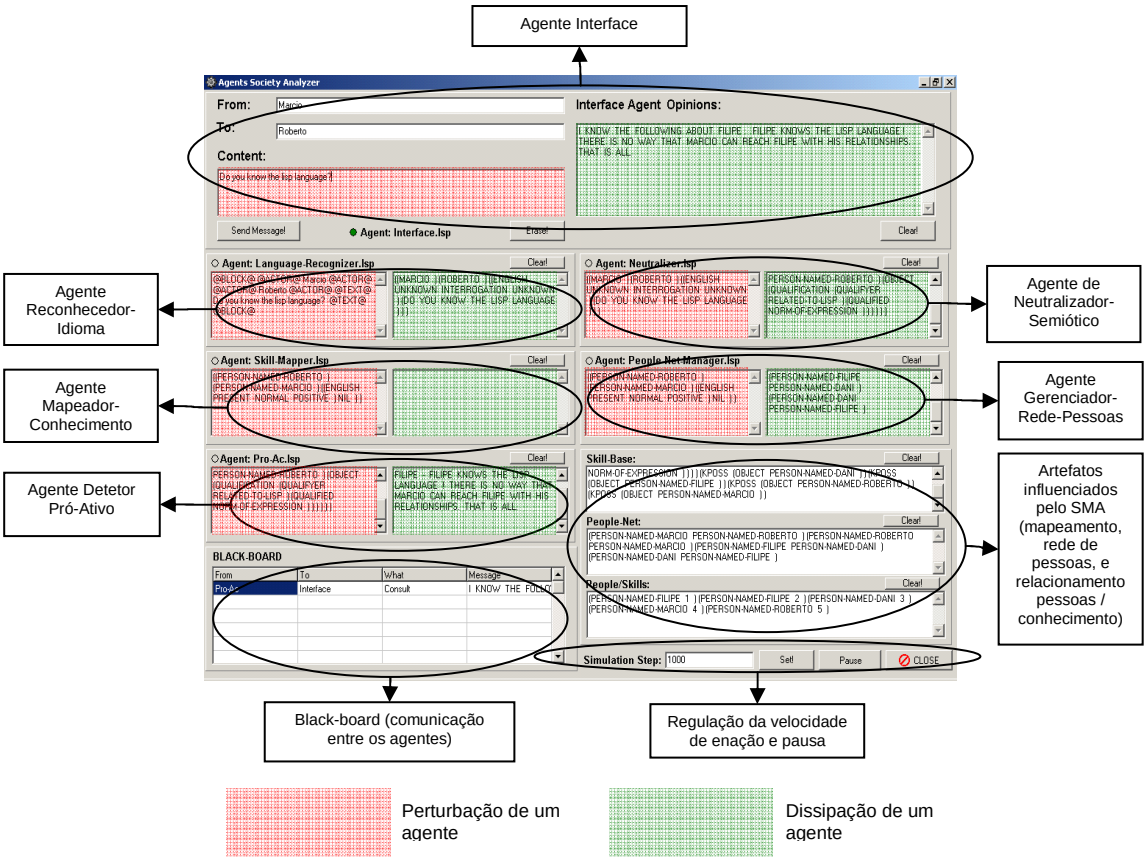


Figura 3.6: Interface de Acompanhamento das Enações de Ordem 1 do SMA.

Finalmente, as enações entre os agentes de ordem 2, para os quais são delegadas operacionalmente a Neutralização Semiológica e a Semiotização Natural, são dadas através de dois tipos arquivos ‘.txt’: um chamado “[Iniciais-Agente-Mestre]-Feel.txt” (que perturba o agente com uma seqüência de *strings*) e um outro chamado “[Iniciais-Agente-Mestre]-Act.txt” (no qual o agente descarrega sua dissipação), conforme detalhado na Tabela 3.5. Todo o trabalho autopoiético dos agentes de ordem 1 consiste desta forma em colocar fórmulas a serem transformadas nos arquivos “Feel” dos agentes de ordem 2 (para os quais

literalmente “terceirizam” tarefas), e recuperar os resultados das transformações nos arquivos “Act” correspondente, conforme detalhado nos algoritmos da parte 3.1.1, nos quais a função *perturbar(X,Y,Z)* implementa uma chamada perturbadora ao agente X, colocando Y em seu arquivo “Feel”, e forçando a utilização de Z na dissipação como arquivo “Act”.

Agente de Ordem 1	Agente de Ordem 2	Arquivo Perturbação	Arquivo Dissipação
Reconhecedor-Idioma	Construir-Diálogo	/objects/LR_Feel.txt	/objects/LR_Act.txt
	Reconhecer-Idioma	/objects/LR_Feel.txt	/objects/LR_Act.txt
Neutralizador-Semiótico	Neutralizar-Conceitos	/objects/N_Feel.txt	/objects/N_Act.txt
	Reconhecer-Forma	/objects/N_Feel.txt	/objects/N_Act.txt
	Reconhecer-Tempo	/objects/N_Feel.txt	/objects/N_Act.txt
	Substituir-Pessoas	/objects/N_Feel.txt	/objects/N_Act.txt
	Remover-Ignorados	/objects/N_Feel.txt	/objects/N_Act.txt
	Mapear-DC	/objects/N_Feel.txt	/objects/N_Act.txt
	Simplificar-DC	/objects/N_Feel.txt	/objects/N_Act.txt
Gerenciador-Rede-Pessoas	Construir-Rede-Pessoas	/objects/PNM_Feel.txt	/objects/PNM_Act.txt
Mapeador-Competências	Construir-Lista-Pessoas-Competências	/objects/SM_Feel.txt	/objects/SM_Act.txt
	Mapear-Pessoas-Competências	/objects/SM_Feel.txt	/objects/SM_Act.txt
Detector-Pró-Ativo	Competências-Necessárias	/objects/PA_Feel.txt	/objects/PA_Act.txt
	Match-Pessoas-Competências	/objects/PA_Feel.txt	/objects/PA_Act.txt
	Remover-Match-Inuteis	/objects/PA_Feel.txt	/objects/PA_Act.txt
	Construir-Resposta	/objects/PA_Feel.txt	/objects/PA_Act.txt

Tabela 3.5: Arquivos de Enação dos Agentes de Ordem 2.

3.1.4. Enações entre o Mundo Real e Artificial

Como já comentado, o simulador de Bate-Papo realizado em Delphi, que permite acompanhar as enações semióticas realizadas no mundo real, constitui também o agente *Interface* do SMA, na medida que é perturbado pelas mensagens escritas pelos agentes humanos, bem como pelas mensagens recebidas através do Black-Board, e pode ele mesmo perturbar outros agentes virtuais e reais.

Segue o procedimento Pascal que permite ao agente *Interface* encaminhar perturbações para o Reconhecedor-Idioma, quando é perturbado por enações no Bate-Papo:

```
procedure TAgent_Simulator.Button_sendClick(Sender: TObject);          // Procedure will be activated when
var S:string; pmessage:^BBMessage; (...)                               // a message is sent on Chat
begin
S:=compose_last_sentence(Bate-Papo); new(pmessage);
pmessage^.MFrom:='Interface'; pmessage^.MTo:='Language-Recognizer';
pmessage^.MWhat:='Consult'; pmessage^.MMessage:=S;
message_list.Add(pmessage);
S:=compose_dialog; new(pmessage);
pmessage^.MFrom:='Interface'; pmessage^.MTo:='Language-Recognizer';
pmessage^.MWhat:='Map'; pmessage^.MMessage:=S; message_list.Add(pmessage);
end;
```

Segue finalmente o procedimento Pascal que permite ao agente Interface verificar a presença de mensagens presentes aos seus cuidados no *Black-Board*:

```
procedure TAgent_Simulator.Timer1Timer(Sender: TObject);          // Procedure will check each n seconds
var i:integer; pmessage:^BBMessage;                               // BlackBoard, and dissipate if any perturbation
   (…)
begin
(…)
// Read message on BB //
pmessage:=message_list.Items[0];
// Check Message //
if pmessage^.MTo='Interface' then
            begin
            (…)
            interfaceanswer.Lines.add(pmessage^.MMessage);
            end;
(…)
end;
```

Através destes dois procedimentos, o agente *Interface* operacionaliza a junção hermenêutica entre a sociedade de agentes humanos e suas enações (os profissionais inter-relacionados com o Bate-Papo), e a sociedade de agentes virtuais e suas enações (o SMA). Este agente é efetivamente perturbado tanto pelas **enações do mundo real** (quando uma mensagem é enviada no Bate-Papo, o procedimento 'TAgent_Simulator.Button_sendClick' é ativado), como pelas **enações do mundo artificial** (o agente *Interface* verifica regularmente com o procedimento 'TAgent_Simulator.Timer1Timer' se uma mensagem encontra-se disponível para ele no Black-Board). Este agente possui, logo, uma importância chave na implementação do mapeamento inteligente, sendo ele o ponto hermenêutico de junção entre dois mundos, garantindo a emergência de uma macro-sociedade autopoiética composta por agentes humanos e agentes de software.

3.2. Do Real ao Artificial : Neutralização Semiológica

Para que o SMA implementado seja efetivamente capaz de construir automaticamente mapeamentos de competências, a partir da simples observação das enações semióticas desempenhadas em linguagem natural por membros de uma comunidade de profissionais, é necessária a operação, pelos agentes deste sistema, de um processamento de linguagem natural, capaz de transformar sentenças em informações conceituais computacionais. Nesta parte, será detalhado computacionalmente como os agentes Construtor-Diálogo, Reconhecer-Idioma, Neutralizar-Conceitos, Reconhecer-Forma, Reconhecer-Tempo, Substituir-Pessoas,

Remover-Ignorados, Mapear-DC, Simplificar-DC, sincronizados e organizados pelos agentes Reconhecedor-Idioma e Neutralizador-Semiótico, são capazes de realizar tal operação, que será denominada de **Neutralização Semiológica**.

3.2.1. Princípios na Neutralização Semiológica

Como argumentado no capítulo 1, pensar que a linguagem caraterisa a cognição consiste em um erro epistemológico (que se trata de uma visão puramente lingüística Aristoteliana do conhecimento, ou de uma visão funcionalista Fodoriana ou Chomskiana). Desta forma, a única propriedade que deve ser extraída de um paradigma de PLN é apenas sua **capacidade espacializante** (transformação do falado, puramente temporal, em algo espácio-temporal, ou seja, representado). Tal capacidade deve, para satisfazer as necessidades de um mapeamento inteligente de competências baseado em tecnologias de colaboração, permitir passar, de forma neutra (qualquer que seja a enação escrita envolvida), da experiência fenomenológica (a competência expressa) para uma representação computacional da experiência (o mapeamento). Em outras palavras, o sistema de PLN deve realizar uma **semiose** (no sentido de Peirce) que possa **neutralizar** o contexto hermeneûtico do expresso através de palavras: deve-se realizar uma **Neutralização Semiológica** (conforme Figura 3.7).

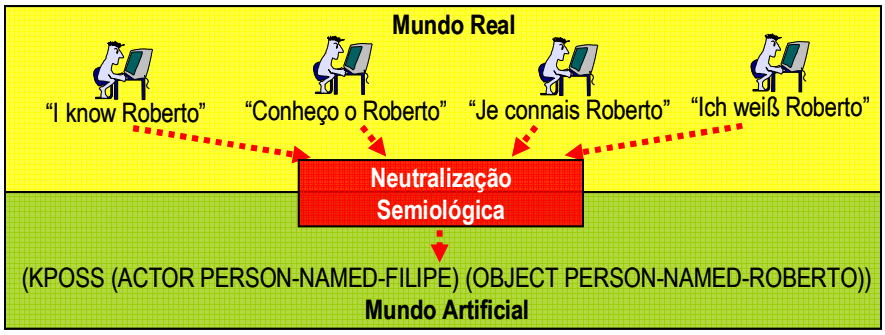


Figura 3.7: Conceito de Neutralização Semiológica.

A implementação da Neutralização Semiológica necessita, portanto, o uso de um paradigma de PLN que visa à transformação de frases escritas, em representações dos conceitos expressos[84]. Este é exatamente o trabalho de pesquisa que foi realizado por Birnbaum e Selfridge: a análise conceitual da linguagem natural[85]. Por este motivo, o coração

[84] Tipicamente, o mapeamento deverá armazenar representações de conceitos como "saber que", "saber fazer", "realizar uma determinada tarefa", "falar um determinado idioma", etc.
[85] O detalhe dos resultados desta pesquisa pode ser consultado em [SCH81], p.318 a 353.

do princípio de Neutralização Semiológica é o próprio algoritmo ***PARSE*** do programa **ELI** (*English Language Interpreter*), resultado desta pesquisa (Figura 3.8).

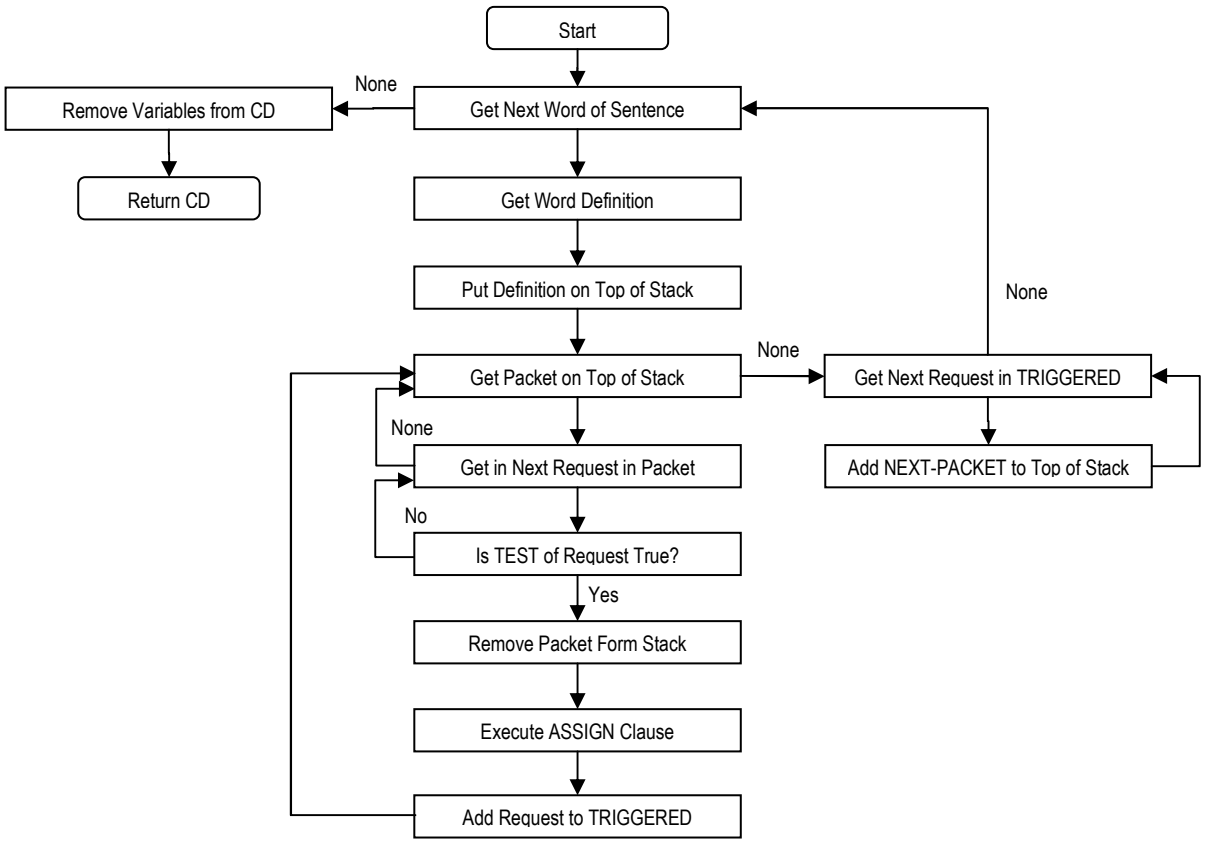


Figura 3.8: Algoritmo *PARSE* do Programa ELI [SCH81], p. 364

O uso deste algoritmo, cujo código pode ser facilmente encontrado na Internet[86], necessita a criação de uma base de tradução de palavras em DCs, conforme modelo:

```
(defword WORD    (          (test A-TEST)
                            (assign SOME-ASSIGNMENTS)
                                       (next-packet
                                       (          (test A-TEST)
                                                  (assign SOME-ASSIGNMENTS)
                                                  (next-packet        [...]        )))
                       [...] ))
 (          (test A-TEST)       [...]       )
  [...]
)
```

[86] Para a implementação do protótipo, foi utilizado o código implementado por Bill Andersen (waander@cs.umd.edu), Department of Computer Science, University of Maryland, College Park, MD 20742, que pode ser encontrado no URL http://www.sims.berkeley.edu/courses/is290-1/s02/MicroProgs/.

A partir destas definições, o algoritmo funciona da seguinte forma: para cada palavra de uma frase F, colocar sua definição (*defword*) em uma pilha *STACK*, pegar a última definição no topo de *STACK*, pegar o próximo pacote *PACKET* desta definição; se a cláusula *TEST* deste *PACKET* é verdadeira, remover o *PAKCET* de *STACK*, realizar as instanciações de variáveis contidas na cláusula *ASSIGN*, colocar os *NEXT-PACKET* em *STACK*, e pegar o próximo *PACKET*; isto até não ter mais nada na pilha *STACK*, e não ter mais palavras de F não tratadas. Desta forma, para que algoritmo *PARSE* funcione, as palavras definidas com a *macro* ***defword*** de ELI devem conter toda a lógica do que pode acontecer no resto da frase, a partir do momento que foram encontradas. Segue o exemplo da definição da palavra especial *START-SENTENCE*, que sempre deve ser processada no início de uma nova frase:

```
(defword *START-SENTENCE*
        (       (assign  *part-of-speech* 'affirmation  *cd-form* nil       *subject* nil       *concept* nil
                         *object-complement* nil        *receptor-complement* nil              *predicates* nil)
                (next-packet
                        (       (test (equal *part-of-speech* 'person))
                                (assign *subject* *cd-form*)
                                (next-packet
                                        (       (test (equal *part-of-speech* 'action))
                                                (assign *concept* *cd-form*))))
                        (
                                (test (or (equal *part-of-speech* 'interrogation) (equal *part-of-speech* 'affirmation)))
                                (assign *concept* *cd-form*)))))
```

Esta definição instancia todas as variáveis globais que serão usadas na análise sintática de *PARSE*:

- **part-of-speech** recebe a parte do discurso analisada (afirmação, pessoa, objeto, direção, interrogação, etc.);
- **subject** recebe o sujeito da frase;
- **object-complement** recebe o complemento de objeto direto da frase;
- **receptor-complement** recebe o complemento de objeto indireto da frase;
- **predicates** recebe os predicados (negação, conjunções, etc.) expressos na frase;
- **concept** recebe o conceito expresso em uma frase, ou um pedaço de frase;
- **cd-form** recebe a dependência conceitual resultado do processamento;

O resultado do processamento efetuado por *PARSE* é um conceito expresso no paradigma das dependências conceituais (DCs) de Schank[87]. Uma DC é, resumidamente, uma estrutura recursiva relacionando uma primitiva conceitual a um ator (*ACTOR*), um objeto (*OBJECT*), uma origem (*FROM*) e um destino (*TO*), conforme Figura 3.9.

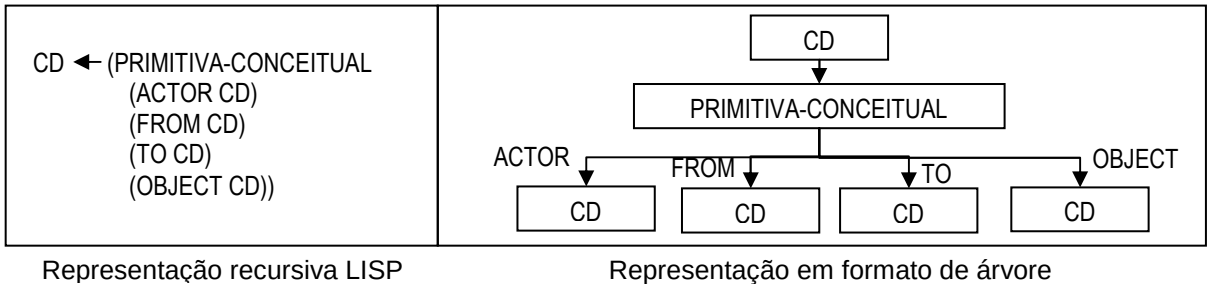


Figura 3.9: Princípio e Representação de uma DC.

Analisando o programa *ELI*, deve-se reconhecer que, para o objetivo de realizar efetivamente uma Neutralização Semiológica, no sentido da Figura 3.7, o algoritmo *PARSE* demonstra várias insuficiências: (i) ele foi projetado para analisar frases escritas com um vocabulário Inglês, e uma ordenação das palavras conforme gramática Inglesa, (ii) ele não leva em consideração nem reconhece o tempo usado na frase, (iii) ele não leva em consideração nem reconhece a forma da frase (interrogativa, negativa, etc.), (iv) ele não reconheçe referências contextuais a pessoas (*Me* se referindo ao autor da frase, por exemplo), ele não possui mecanismos de desambiguação. Por estes motivos, o algoritmo *PARSE* deve ser completado por uma série de outras tarefas, para que seja formado um algoritmo de **Neutralização Semiológica**, conforme descrito na Figura 3.10.

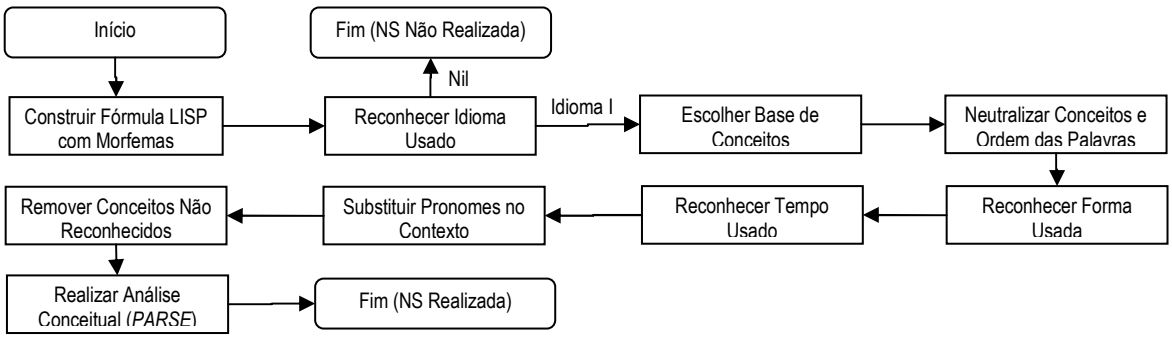


Figura 3.10: Algoritmo Simplificado de Neutralização Semiológica.

[87] Para os detalhes deste paradigma, consultar [SCH81], p.9-26.

3.2.2. Operacionalização Experimental da Neutralização Semiológica

É exatamente a seqüência de tarefas da Figura 3.10 que é executada pelos agentes Reconhecedor-Idioma e Neutralizador-Semiótico. O agente Reconhecedor-Idioma é responsável por executar as tarefas "construir fórmula LISP" e "Reconhecer Idioma" (separadas neste agente, por que podem ser usadas em outros contextos), e o agente Neutralizador-Semiótico é responsável por executar as outras tarefas da Neutralização Semiológica (que são *ad-hoc*, para a construção de mapeamentos inteligentes). Cada agente de ordem 2, pertencendo aos agentes Reconhecedor-Idioma e Neutralizador-Semiótico, é finalmente responsável pela execução pragmática de uma tarefa do algoritmo de Neutralização Semiológica, conforme detalhado na Figura 3.11.

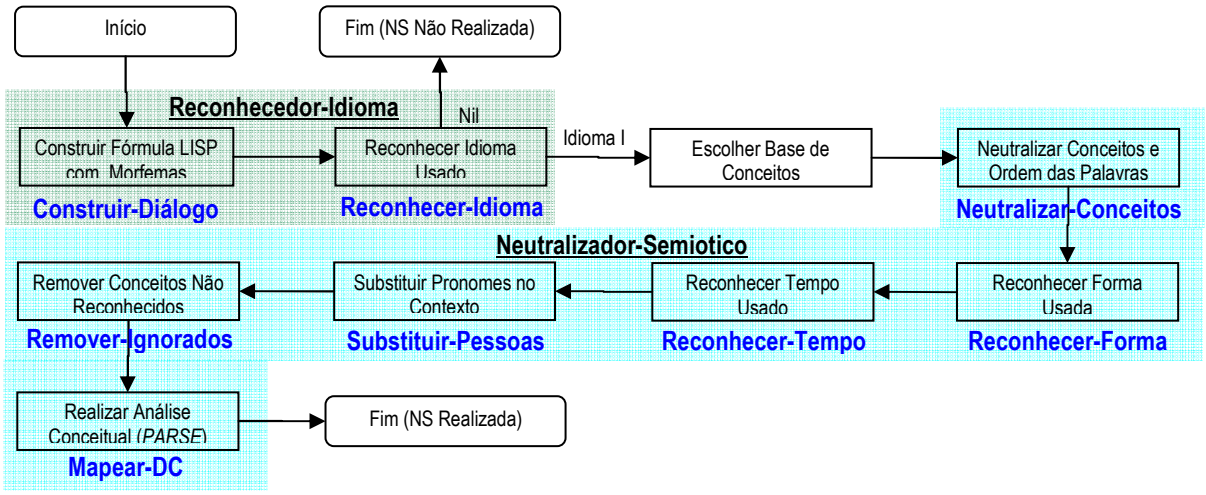


Figura 3.11: Operacionalização Experimental da Neutralização Semiológica.

Deve-se observar que a operacionalização realizada leva em consideração apenas frases escritas em Inglês, na medida que apenas um dicionário Inglês, uma Base de Conceitos Inglesa e uma Base de DC Inglesa foram implementados a fins experimentais. Portanto, a implementação da tarefa de escolha de base de conceitos **não foi necessária** (sendo que apenas uma está disponível), e o agente Neutralizar-Conceitos realiza apenas uma neutralização conceitual sem operar nenhuma reorganização da ordem das palavras (sendo o experimento conduzido apenas em Inglês). Ressaltamos aqui, como foi adiantado no capítulo 1 deste trabalho, que optamos pragmaticamente por realizar uma arquitetura completa e operacional, envolvendo todos agentes requeridos, sem, porém, aprofundar a implementação específica de cada agente.

Todos os agentes LISP de ordem 2 foram construídos a partir do seguinte modelo[88]:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; PATTERN AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;

(defvar *XXXXX*)   (setq *XXXXX* ())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *XXXXX* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *XXXXX* (Think-Function *XXXXX*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *XXXXX* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
```

São apresentadas a seguir, as partes-chave do código LISP implementado, que usa cada agente protótipo de ordem 2 implicado na Neutralização Semiológica:

- Construir-Diálogo (*dialog-construction.lsp*):

```
(defun make-dialog (DIALOG-LIST) "make dialog style from list extracted from text file"
  (let ((RESULT NIL) (ALIST NIL))
        (dolist  (ABLOCK (make-block '@BLOCK@ DIALOG-LIST))
              (push (append (make-block '@ACTOR@ ABLOCK)
                            (make-content (car (make-block '@TEXT@ ABLOCK)))) ALIST))
        (reverse ALIST)))

(defun make-content (TEXT)  ;transform text in content of type (sentence_1 ... sentence)n)
  (let ((RES 'VAR-RESULT) (TYPE 'VAR-TYPE) (LIST-WORDS))
        (cond  ((NULL TEXT) NIL)
              (T (setq LIST-WORD (extract-sentence-w TEXT RES TYPE))
                 (cons (cons (list 'UNKNOWN 'UNKNOWN (eval TYPE) 'UNKNOWN)
                             (list LIST-WORD)) (make-content (eval RES)))))))
```

Observações:

1. O agente *Interface* emite em direção do Construtor-Diálogo uma enação semiótica captada do Bate-Papo sempre formatada da seguinte maneira:

   DIALOG-LIST: = (BLOCK_1 BLOCK_2 ... BLOCK_n)

   BLOCK:= @BLOCK@ @ACTOR@ Autor_1 ... Autor_n @ACTOR@ @ACTOR@ Remetente_1 ... Remetente_n @ACTOR@ @TEXT@ Conteúdo_Mensagem @TEXT@ @BLOCK@

[88] Este modelo é uma implementação do Agente Essencial descrito na Figura 2.12.

2. O resultado do algoritmo *make-dialog* é uma fórmula LISP onde:

   Dialogue := (Block_1 Block_2 ... Block_n)

   Block := (From To Sentence_1 ... Sentence _n)

   From := (Name_Person_1 … Name_Person_n)

   To := (Name_Person_1 … Name_Person_n)

   Sentence := (Characteristics Words)

   Characteristics := (Language Tense Punctuation Form)

   Language ∈ (PORTUGUESE ENGLISH ... FRENCH)

   Tense ∈ (PAST PRESENT)

   Punctuation ∈ (NORMAL INTERROGATION EXCLAMATION)

   Form ∈ (POSITIVE NEGATIVE)

- Reconhecer-Idioma (*language-recognition.lsp*):

```
(defvar *LIMIT*) (setq *LIMIT* 0.75)
(defun count-recognized-words (WORDS DICTIONARY)    “counts number of recognized words of “
        (cond ((NULL WORDS) 0)                      “WORDS of reference DICTIONARY”
              ((NULL DICTIONARY) 0)
              (T (+ (count (caar DICTIONARY) WORDS)
                    (count-recognized-words WORDS (cdr DICTIONARY))))))
(defun recognize-language (WORDS DICTIONARY)        "returns the proportion of words of list”
          (if (not (NULL DICTIONARY))               “WORDS which exist in DICTIONARY"
          (/ (count-recognized-words WORDS DICTIONARY) (length WORDS))))
```

Observações:

1. Experimentalmente, um idioma I de dicionário D é considerado como *reconhecido* em uma frase F, se mais de 75% (LIMIT) das palavras de F pertencem a D.
2. O dicionário usado experimentalmente encontra-se no arquivo ‘*English-Full-Lexic.lxc*’. Ele consiste em uma lista de palavras definidas da seguinte forma:

   WORD := (MORPHEME TENSE PERSON GENDER NUMBER RELATED-CONCEPT)

   TENSE ∈ (PAST PRESENT)

   PERSON ∈ (ME OTHER)

   GENDER ∈ (MASC FEM NEUTRAL)

   NUMBER ∈ (SING PLURAL)

   RELATED-CONCEPT é um conceito da **Base de Conceitos**

- Neutralizar-Conceitos (*concept-neutralization.lsp*):

```
(defun make-predictions (CURRENT-CONCEPT PREDICTION)        "Predict which kind should be
        (cond  ((NULL PREDICTION) '(AFFIRMATION))           "next concept"
               ((equal CURRENT-CONCEPT (caar PREDICTION)) (cadar PREDICTION))
               (T (make-predictions CURRENT-CONCEPT (cdr PREDICTION)))))

(defun replace-word (WORD LEXIC L-PREDICTIONS CONCEPT-BASE) "replace words by concepts"
   (cond  ((NULL LEXIC) '(!UNKNOWN! TINDEF PINDEF GINDEF NINDEF *UNKNOWN-WORD*))
          ((and (equal (caar LEXIC) WORD)
                (member (find-concept-type (car (last (car LEXIC))) CONCEPT-BASE)
                        L-PREDICTIONS))       (car LEXIC))
          (T (replace-word WORD (cdr LEXIC) L-PREDICTIONS CONCEPT-BASE))))
```

Observações:

1. Cada Conceito é definido pela lista: (NAME FORM TYPE), onde:

   NAME é um morfema;

   FORM ∈ (POSITIVE NEGATIVE)

   TYPE ∈ (ACTION AFFIRMATION CONSEQUENCE COORDINATION DIRECTION INTERROGATION PERSON POINTER POSITION QUALIFICATION RELATIVE WOBJECT)

2. Para permitir evitar a confusão de conceitos (o morfema PROGRAM, por exemplo, pode ser o substantivo *programa* ou o verbo *programar*), a neutralização de conceitos usa uma lista de predições de quais conceitos podem logicamente suceder um determinado conceito. Tal lista encontra-se no arquivo '*English-NLP-Predictions.prd'*.

- Reconhecer-Forma (*form-recognition.lsp*):

```
(defun negative-p (WORD CONCEPTS)            "Check if a WORD points to"
        (cond     ((NULL CONCEPTS) NIL)      "a negative Form of Concept"
                  ((NULL WORD) NIL)
                  ((and     (eq (caar CONCEPTS) WORD)
                            (eq (cadar CONCEPTS) 'NEGATIVE)) T)
                  (T (negative-p WORD (cdr CONCEPTS)))))
(defun exist-negative-form (WORDS CONCEPTS)          "Check if a list of word contains at least one"
        (cond     ((NULL WORDS) NIL)                 " Negative Form"
                  ((negative-p (car (last (car WORDS))) CONCEPTS) T)
                  (T (exist-negative-form (cdr WORDS) CONCEPTS))))
```

Observação: o algoritmo principal consiste apenas em verificar se existe pelo menos um conceito negativo na lista de conceitos.

- Reconhecer-Tempo (*tense-recognition.lsp*):

```
(defun exist-past-tense (WORDS)
        (cond   ((NULL WORDS) NIL)
                ((eq (nth 1 (car WORDS)) 'PAST) T)
                (T (exist-past-tense (cdr WORDS)))))
(defun set-tense-sentence (SENTENCE)
        (cond   ((exist-past-tense (extract-words SENTENCE)) (set-tense SENTENCE 'PAST))
                (T (set-tense SENTENCE 'PRESENT))))
```

Observação: o algoritmo principal consiste apenas em verificar se pelo menos uma palavra expressa na lista de morfemas é um verbo conjugado no passado. Se nenhum verbo conjugado no passado é encontrado, a frase é experimentalmente considerada como "presente".

- Substituir-Pessoas (*person-concept-substitution.lsp*):

```
(defun make-word-substitution (MWORD TO-SUBSTITUTE ACTORS-FROM ACTORS-TO)
   (cond ((NULL TO-SUBSTITUTE) MWORD) "substitutes concepts to person-concepts"
         (T (cons (substitute-actors TO-SUBSTITUTE ACTORS-FROM ACTORS-TO) (cdr MWORD)))))
(defun find-word-in-substitution (CONCEPT-BASE SUBSTITUTION)  "finds words which need to be"
    (cond ((NULL SUBSTITUTION) NIL)                                "substituted"
          ((equal CONCEPT-BASE (caar SUBSTITUTION)) (cadar SUBSTITUTION))
          (T (find-word-in-substitution CONCEPT-BASE (cdr SUBSTITUTION)))))
(defun substitute-concept-word (MWORD ACTORS-FROM ACTORS-TO SUBSTITUTION)
     (cond ((find-word-in-substitution (car (last MWORD)) SUBSTITUTION)      "Realize Substitutions"
            (make-word-substitution MWORD
                                    (find-word-in-substitution (car (last MWORD)) SUBSTITUTION)
                                    ACTORS-FROM ACTORS-TO))
           (T MWORD)))
```

Observações:

1. O objetivo do algoritmo de substituição consiste em encontrar, em uma lista de conceitos, os que pertencem à lista SUBSTITUTION, e realizar as substituições recomendadas por esta mesma lista.
2. Experimentalmente, a lista SUBSTITUTION encontra-se no arquivo '*NLP-Substitutions.sub*', e vale: ((MY-SELF (ACTOR-FROM)) (YOUR-SELF (ACTOR-TO)) (OUR-SELVES (ACTOR-FROM)))

- Remover-Ignorados (*unknown-concept-remotion.lsp*):

```
(defun remove-unknown-concept-words (WORDS) "Remove all unknown concepts"
   (cond  ((NULL WORDS) NIL)
          ((equal (caar WORDS) '!UNKNOWN!) (remove-unknown-concept-words (cdr WORDS)))
          (T (cons (car WORDS) (remove-unknown-concept-words (cdr WORDS))))))
```

Observação: o algoritmo principal consiste em remover toda palavra que não foi reconhecida como conceito por *replace-word* de *concept-neutralization.lsp*.

- Mapear-DC (*CD-parser-traced.lsp*):

```
(defun parse (sentence) "Takes a sentence in list form and returns the conceptual analysis for it."
  (setq *concept* nil)
  (init-stack)
  (do ((*abs-concept* nil) (*word* nil)
       (*sentence* (cons '(*START* TINDEF PINDEF GINDEF NINDEF *START-SENTENCE*)
        sentence)))
       ((null (and (setq *word*     (pop *sentence*))
                   (setq *abs-concept* (car (last *word*))))))    (remove-variables *concept*))
       (load-def)
       (run-stack)))
```

Observação: este agente encapsula o algoritmo PARSE de Birnbaum e Selfridge.

- Simplificar-DC (*CD-simplification.lsp*):

```
(defun simplify-CD-words (WORDS SIMPLIFICATIONS)    "Realize a CD simplification based on a"
  (cond   ((NULL WORDS) NIL)                        "SIMPLIFICATIONS list"
          ((ATOM WORDS) WORDS)
          ((word-conceptp WORDS) (car (last WORDS)))
          ((member (header-cd WORDS) (car SIMPLIFICATIONS))
                   (simplify-CD-words (car (roles-CD (cadr (roles-CD WORDS)))) SIMPLIFICATIONS))
          (T (cons (header-cd WORDS)
                   (delete '() (mapcar #'(lambda (X) (simplify-CD-words X SIMPLIFICATIONS)) (roles-cd
                    WORDS)))))))
(defun simplify-CD-sentence (SENTENCE SIMPLIFICATIONS)
   (cond ((NULL SENTENCE) NIL)
         (T  (cons (extract-attributes SENTENCE)
             (cons (qualify (simplify-CD-words (extract-words SENTENCE) SIMPLIFICATIONS))  NIL)))))
```

Observações:

1. O algoritmo de simplificação, implementado principalmente em *simplify-CD-words*, encontra DCs cuja primitiva conceitual pertence à lista SIMPLIFICATION, e realiza a substituição desta DC pelo seu *cadr*[89]. Experimentalmente, a lista SIMPLIFICATION encontra-se no arquivo '*CDs-Simplifications.smp'*, e vale: ((DIRECTION DETERMINATION POSITION RELATIVE ACTION-DETAILS))
2. Na cláusula *simplify-CD-sentence*, a função *qualify* introduz um (QUALIFICATION (QUALIFYER NIL) (QUALIFIED X)) em todos os OBJECT, FROM, TO, e AUTHOR não qualificados.
3. A tarefa de "simplificação de DC" foi criada para aumentar a eficiência do algoritmo *MACTH* [90]utilizado na Semiotização Natural.

[89] *Cadr* é uma função LISP, para maiores detalhes, consultar o ANSI COMMON LISP [GRA96].
[90] Ver descrição da Semiotização Natural na parte 3.3.

Cabe ressaltar que um detalhamento maior dos algoritmos implementados pode ser obtido consultando o código LISP completo disponível no apêndice B.

3.2.3. Exemplo de Neutralização Semiológica

Para poder finalmente clarificar a operacionalização pragmática de uma Neutralização Semiológica, serão analisados a seguir os tratamentos efetuados pelos agentes Reconhecedor-Idioma e Neutralizador-Semiótico ao serem originalmente perturbados pela mensagem:

From: FILIPE	To: JEAN	Content: "Well I know the lisp language."

A Neutralização Semiológica desta semiose necessita da existência das seguintes entidades, conforme descritas:

- Dicionário:

```
(ENGLISH        (FILIPE TINDEF OTHER MASC SING PERSON-NAMED-FILIPE)
                (I TINDEF ME GINDEF SING MY-SELF)
                (JEAN TINDEF OTHER MASC SING PERSON-NAMED-JEAN)
                (KNOW PRESENT PINDEF GINDEF NINDEF TO-KNOW)
                (LANGUAGE TINDEF PINDEF NEUTRAL SING NORM-OF-EXPRESSION))
                (LISP TINDEF PINDEF NEUTRAL NINDEF RELATED-TO-LISP)
                (THE TINDEF PINDEF GINDEF NINDEF POINTER-THE))
```

- Base de Conceitos:

```
(       (*START-SENTENCE* POSITIVE AFFIRMATION)
        (*UNKNOWN-WORD* POSITIVE AFFIRMATION)
        (MY-SELF POSITIVE PERSON)
        (NORM-OF-EXPRESSION POSITIVE WOBJECT)
        (PERSON-NAMED-FILIPE POSITIVE PERSON)
        (PERSON-NAMED-JEAN POSITIVE PERSON)
        (POINTER-THE POSITIVE POINTER)
        (RELATED-TO-LISP POSITIVE QUALIFICATION)
        (TO-KNOW POSITIVE ACTION))
```

- Base de DCs[91]:

```
((defword *START-SENTENCE* ((assign  *part-of-speech* 'affirmation […])
(defword *UNKNOWN-WORD* ((assign  *part-of-speech* 'wobject   *cd-form* *WORD*)))
(defword MY-SELF ((assign   *part-of-speech* 'person *cd-form* *WORD*)))
(defword NORM-OF-EXPRESSION ((assign *part-of-speech* 'wobject *cd-form* *WORD*)))
(defword PERSON-NAMED-FILIPE ((assign *part-of-speech* 'person *cd-form* *WORD*)))
(defword PERSON-NAMED-JEAN ((assign *part-of-speech* 'person  *cd-form* *WORD*)))
(defword POINTER-THE ((assign *part-of-speech* 'pointer […])))
(defword RELATED-TO-LISP ((assign *part-of-speech* 'qualification […])))
(defword TO-KNOW ((test (or (eq *part-of-speech* 'person) […]))))
```

[91] Para mais detalhes, consultar a totalidade do código do protótipo, disponível no apêndice B.

A Figura 3.12 detalha a seqüência das transformações operadas pelos vários agentes de ordem 2 responsáveis pela Neutralização Semiológica da semiose considerada em exemplo.

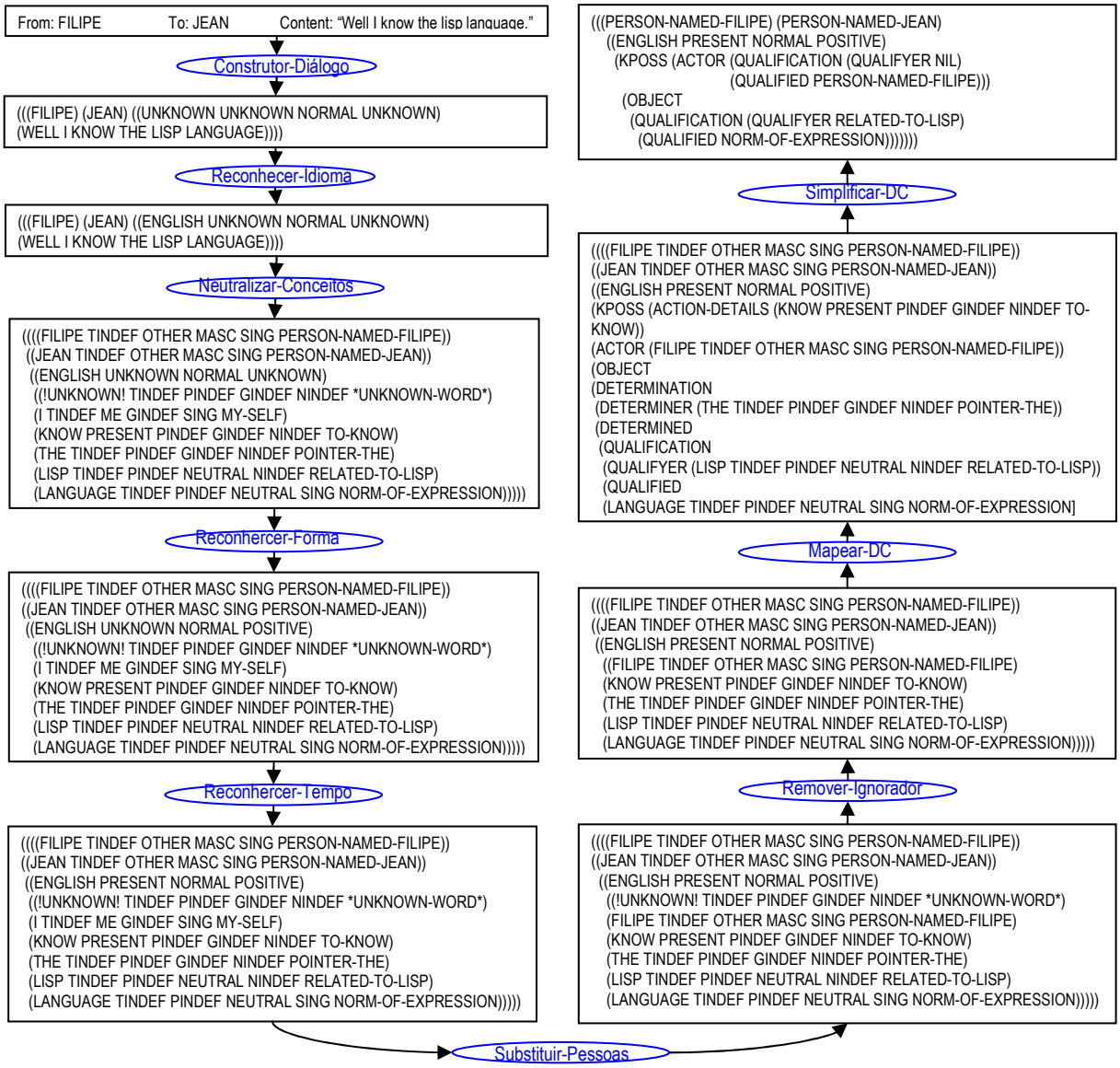


Figura 3.12: Detalhamento da Neutralização Semiológica Através de um Exemplo.

Após ter sido operada a Neutralização Semiológica da semiose ((FILIPE JEAN) (I KNOW THE LISP LANGUAGE)), o Neutralizador-Semiótico passa um recado para o Gerenciador-Rede-Pessoas, para que mapeia na Base de Relacionamentos as fórmulas (PERSON-NAMED-FILIPE PERSON-NAMED-JEAN) e (PERSON-NAMED-JEAN PERSON-NAMED-FILIPE), significando *Filipe* conhece *João* e *João* conhece *Filip*e, e o Gerenciador-Rede-Pessoas re-encaminha o resultado da Neutralização Semiológica para o

Mapeador-Conhecimento, que apenas mapeará o relacionamento entre *Filipe* e as competências (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-JEAN)))) e (KPOSS (OBJECT (QUALIFICATION (QUALIFIER RELATED-TO-LISP) (QUALIFIED NORM-OF-EXPRESSION)))) e entre *João* e a competência (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFYED PERSON-NAMED-FILIPE)))). Assim, a operação de mapeamento inteligente de competência é pró-ativamente finalizada, deixando o Mapeamento de Competências, a Base de Relações Pessoas-Competências e a Base de Pessoas Relacionadas nos seguintes estados:

- Mapeamento de Competências:

```
((KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFYED PERSON-NAMED-FILIPE))))
 (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFYED PERSON-NAMED-JEAN))))
 (KPOSS (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-LISP)
                (QUALIFIED NORM-OF-EXPRESSION)))))
```

- Base de Relações Pessoas-Competências:

```
((PERSON-NAMED-JEAN 1) (PERSON-NAMED-FILIPE 2) (PERSON-NAMED-FILIPE 3))
```

- Base de Pessoas Relacionadas:

```
((PERSON-NAMED-FILIPE PERSON-NAMED-JEAN)
(PERSON-NAMED-JEAN PERSON-NAMED-FILIPE))
```

3.2.4. Hermenêutica do Agente Humano

Sendo que o paradigma da Autopoiésis foi adotado para operacionalizar mapeamentos inteligentes, os artefatos computacionais implementados (os agentes de Software) exibem propriedades de **autonomia**, **intencionalidade,** e **identidade topológica**. Desta forma, os agentes responsáveis pela Neutralização Semiológica adquirem uma dimensão hermenêutica recíproca, uns em relacionamento aos outros: eles se tornam inscrições contextualizadas de saber, sendo o saber a coerência emergente que mantém a sociedade virtual de ordem 0 logicamente estável a um determinado momento.

Se a maioria dos agentes de software implicados na Neutralização Semiológica não possui nenhum contato direto com a sociedade de pessoas envolvidas no seu uso, existe, porém, um agente particular, o agente *Interface*, que possui a capacidade de ser perturbado por enações semióticas naturais **humanas**. A existência deste agente permite propor, da mesma forma que Havelange descreve a inserção hermenêutica do artefato computacional nas

sociedades de homens[92], a noção de **inserção hermenêutica do agente humano nas sociedades virtuais**. Sendo tal inserção *hermenêutica*, é importante ressaltar que ela pode apenas emergir se existir uma superposição enativa semântica entre a sociedade de agentes humanos e a sociedade de agentes virtuais. Em outras palavras, se os agentes humanos usam palavras que não são repertoriadas no Dicionário e na Base de Conceitos usados pelos agentes virtuais ou constroem frases que não se encaixam nas estruturas da Base de Dependências Conceituais, eles deixarão, no mesmo momento, de "fazer sentido" para o agente *Interface*, e, logo, para todos os agentes envolvidos na Neutralização Semiológica. Pragmaticamente falando, sem inserção hermenêutica do agente humano na sociedade virtual, não existe possibilidade de mapear automaticamente qualquer tipo de informação.

Desta forma, a programação orientada-agente, a partir do momento que foi contextualizada no paradigma da Autopoiésis, traz com ela uma nova dificuldade na tarefa de programação: não se necessita mais apenas construir algoritmos corretos (sem erros de programação), mas também construir núcleos autopoiéticos **hermeneuticamente inseridos** na sociedade de usuários onde vão operar, e capaz de **inserir hermeneuticamente** esta mesma sociedade de usuários na sua coerência autônoma.

3.3. Do Artificial ao Real: Semiotização Natural

Uma vez que o SMA realizado é capaz de neutralizar de forma automática frases escritas em linguagem natural para operacionalizar a criação automática de mapeamentos de competências, basta finalmente ele ser capaz de desempenhar também a operação inversa (a restituição automática contextualizada de informações conceituais em linguagem natural), para que o problema proposto neste trabalho seja completamente resolvido. Nesta parte, será portanto detalhado computacionalmente como os agentes Competências-Necessárias, Match-Pessoas-Competências, Remoção-Match-Inúteis e Construir-Resposta, sincronizados e organizados pelo agente Detector-Pró-Ativo, são capazes de realizar tal operação inversa, que será denominada de "**Semiotização Natural**".

[92] "Isto nos permite verificar uma gênese tecnológica do sentido, [...] e reconhecer a dimensão hermenêutica do objeto técnico: a apropriação hermenêutica de um passado não vivenciado é tornada possível através da

3.3.1. Princípios da Semiotização Natural

A operação de Semiotização Natural, descrita no cenário (ii) da Tabela 3.4, é ela mesma desencadeada, no contexto da organização dos agentes de ordem 1, por uma Neutralização Semiológica. Em outras palavras, na fonte de toda Semiotização Natural, o agente *Interface* começa por ser perturbado por uma frase interrogativa, para solicitar um reconhecimento de idioma ao Reconhecedor-Idioma, **no modo da consulta**[93], o que leva à perturbação do Neutralizador-Semiológico no mesmo modo, que perturba finalmente o **Detector-Pro-Ativo**, em vez do Gerenciador-Rede-Pessoas. São finalmente os agentes de ordem 2 coordenados pelo Detector-Pro-Ativo que desempenharão:

- A verificação da existência, no Mapeamento de Competências, de competências *parecidas* com as DCs obtidas por Neutralização Semiológica;
- O encontro, na Base de Pessoas Relacionadas, do caminho mais curto para ir do solicitador de competência aos possuidores das competências *parecidas* encontradas;
- A *realização de uma asserção*, em linguagem natural, a respeito das descobertas que foram feitas no Mapeamento de Competências, na Base de Relações Pessoas-Competências, e na Base de Pessoas Relacionadas;

Sendo que a implementação de um algoritmo de encontro do caminho mais curto em um grafo é extremamente trivial (uma simples busca em *width-first*[94], sem heurística, já permite resolver o problema), os dois principais desafios da Semiotização Natural consistem em realizar um algoritmo que permite ***reconhecer duas DCs parecidas***, e um algoritmo que **traduza DCs** em frases de uma linguagem natural. Um outro bom argumento que levou a escolher o paradigma das Dependências Conceituais de Schank para as tarefas de PLN, no contexto desta pesquisa, foi o fato destes dois problemas já terem sido resolvidos por Schank mesmo, no que diz respeito ao *reconhecimento de DCs parecidas*[95], e por James Meehan, no que diz respeito à tradução de DCs em linguagem natural[96].

---

mediação de um objeto técnico externo", [HAV99c], p96.

[93] O campo "objetivo" do recado emitido no *Black-Board* é instanciado com "*Consult*" , em vez de "Map", utilizado no mapeamento automático de competência, consistindo em uma simples Neutralização Semiológica.

[94] Uma busca *Width-First* (Largura-Primeiro) em um grafe, consiste em expandir a busca, a partir de um nó origem, em primeiro para todos os seus filhos, e assim recursivamente, até encontrar o nó final desejado, registrando todos os caminhos percorridos em paralelo, e guardando finalmente o primeiro caminho que permite encontrar uma solução, sempre eliminando os circuitos gerados, para evitar recursividades infinitas.

[95] O algoritmo criado por Schank se chama "*Match*" e permite verificar se uma DC pode ser transformada em um sub-conjunto de uma outra DC por instanciação de variáveis.

[96] O programa criado por Meehan se chama *Micro-Mumble*, e traduz DCs em frases escritas em Inglês.

A Figura 3.13, a seguir, descreve o algoritmo *Match*, escrito por Schank, que permite verificar se duas DCs "casam", no sentido de "uma das DCs pode ser transformada em um sub-conjunto da outra por instanciação de variáveis".

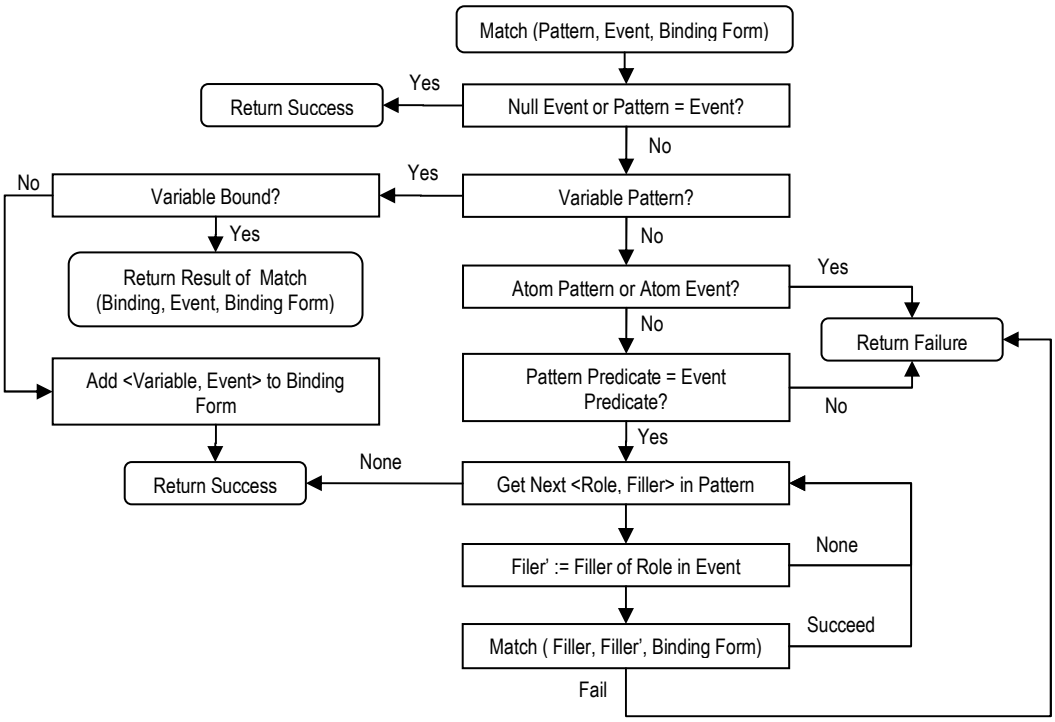


Figura 3.13: Algoritmo *MATCH* de Schank [SCH81], p. 71

O algoritmo ***Match*** recebe como argumentos *Pattern*, consistindo em uma DC podendo conter variáveis anunciadas pelo símbolo *?* (por exemplo, em (PTRANS (ACTOR (?SHOPPER)) (TO (?STORE))), SHOPPER e STORE são variáveis), *Constant*, consistindo em uma DC que não pode conter variáveis (por exemplo, a DC (PTRANS (ACTOR (PERSON)) (TO (STORE))) não contém variáveis), e *Binding*, que consiste em uma lista de associações entre variáveis e constantes, pré-determinadas na execução de *Match,* e sempre começando pela constante T[97] (por exemplo, em (T (SHOPPER (PERSON)) (STORE (STORE))), a variável SHOPPER é pre-instanciada com '(PERSON) e a variável STORE com '(STRORE)). Se existe uma instanciação das variáveis de *Pattern* que permite transformar *Pattern* em um sub-conjunto de *Constant*, respeitando os pré-requisitos de *Binding*, então *Match* retorna **T**, no caso que não existe instanciação de variáveis necessárias ao casamento de *Pattern* e *Constant*, ou **lista das instanciações**, no caso que sejam

[97] T é o átomo reservado *True* em Lisp. Para mais detalhes, consultar o ANSI COMMON LISP [GRA96].

necessárias. Se não existir possibilidade de tornar *Pattern* um sub-conjunto de *Constant*, então *Match* retorna **NIL**[98]. Seguem alguns exemplos exemplificando o conceito de *Match*:

```
> (match '(A (B C) (D E)) '(A (B)) '(T))
> T
> (match '(A (B C) (D E)) '(A (B K)) '(T))
> NIL
> (match '(A (B ?X) (D E)) '(A (B C) (D E)) '(T (X K)))
> NIL
> (match '(A (B ?X) (D E)) '(A (B C) (D E)) '(T))
> (T (X C))
```

O programa ***Micro-Mumble***, escrito por Meehan na sua implementação de *TALE-SPIN*[99], consiste resumidamente em uma série de cláusulas LISP que transformam DCs em frases escritas em Inglês, a partir de uma série de definições das primitivas conceituais das DCs que deverão ser traduzidas. Como o código de *Mirco-Mumble* consiste apenas em cláusulas triviais do tipo "(defun say-ATRANS [...])", não se necessita de detalhamentos complementares destas para o fim de definir o conceito de Semiotização Natural. O leitor interessado em mais detalhes sobre a geração de linguagem natural a partir de DCs poderá consultar [SCH81], p. 225, e p.249 a 256.

A partir do uso de *Match* e *Micro-Mumble*, o algoritmo de Semiotização Natural consiste simplesmente em construir **uma lista de pessoas com competências desejadas** (o que é, na realidade, equivalente a construir relacionamentos Pessoas-Competências, como é feito no momento do Mapeamento Automático Inteligente de Competências, com a diferença do que os relacionamentos construídos são apenas "competências desejadas", e não "competências possuídas"), construir, na seqüência, **uma lista de pessoas e competências encontradas** (o que consiste em aplicar *Match* entre cada uma das competências desejadas e cada uma das competências mapeadas no Mapeamento de Competências), remover desta última lista **as pessoas e suas competências que se tornam inúteis**, em função do contexto (o que consiste em remover pares (Pessoa Competência-Encontrada), se a Pessoa encontra-se presente no bate-papo no momento da operação de Semiotização Natural[100]), e finalmente

[98] NIL ou () é o átomo reservado *Empty-List* (Lista vazia) em Lisp. Para mais detalhes, consultar o ANSI COMMON LISP [GRA96].

[99] TELL-SPIN [SCH81], p. 197-226 é um programa capaz de inventar e contar histórias em linguagem natural, após ter interagido com um usuário.

[100] Esta remoção torna-se necessária para não prejudicar a credibilidade do SMA, e logo não enfraquecer a sua inserção hermenêutica na sociedade autopoiética de profissionais humanos. Efetivamente, se o SMA afirma algo

preparar uma inserção semiótica escrita **em linguagem natural** (através da aplicação de *Micro-Mumble*), a partir da escolha do dicionário correto de tradução DCs-Idioma, que deriva da informação sobre o idioma usado nas enações semióticas reais, contida na fórmula inicial passada pelo agente Neutralizador-Semiótico. Esta seqüência de tarefa é resumida na Figura 3.14.

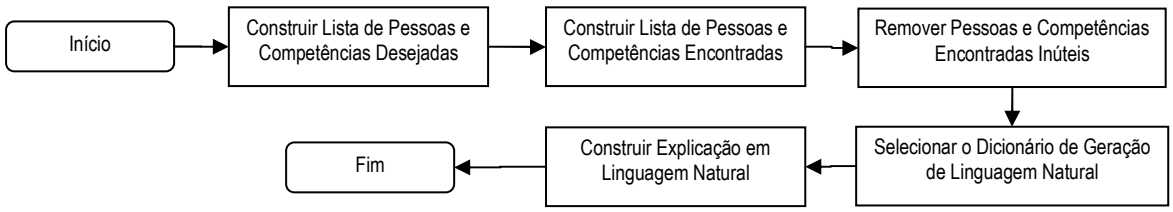


Figura 3.14: Algoritmo Simplificado de Semiotização Natural.

3.3.2. Operacionalização Experimental da Semiotização Natural

Obviamente, é exatamente a seqüência de tarefas da Figura 3.14 que é executada pelo agente **Detector-Pro-Ativo**. Cada agente de ordem 2 coordenado pelo Detector-Pro-Ativo é responsável pela execução pragmática de uma tarefa do algoritmo de Semiotização Natural, conforme detalhado na Figura 3.15.

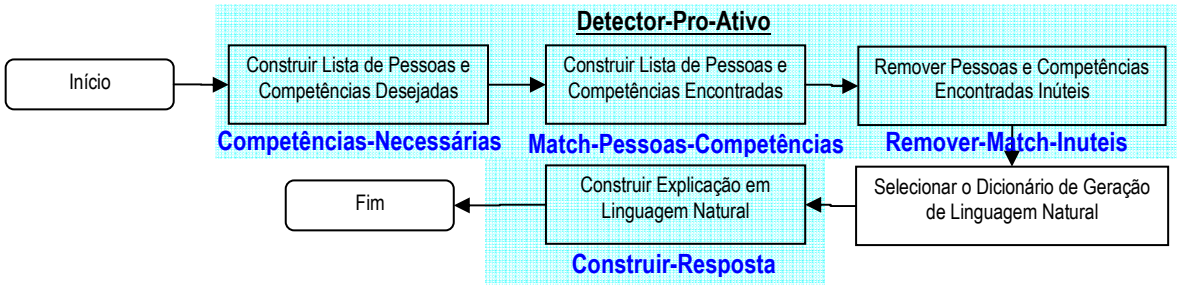


Figura 3.15: Operacionalização Experimental da Semiotização Natural.

Deve-se observar que, como já tinha sido notado na descrição da implementação experimental da Neutralização Semiológica, a operacionalização da Semiotização Natural leva em consideração **apenas frases escritas em Inglês**, na medida que apenas um dicionário de tradução DCs-Linguagem-Natural foi realizado a fins experimentais. Portanto, a

---

trivial, devido ao fato que a pessoa citada na sua afirmação encontra-se envolvida na conversa entre usuários humanos, que está sendo observada, o SMA parecerá estar dando “dicas inúteis”, e logo, perderá o seu crédito como agente virtual inserido em uma sociedade real.

implementação da tarefa de escolha do Dicionário de Geração de Linguagem Natural **não foi necessária** (sendo que apenas um "está disponível"). Mais uma vez, ressaltamos que o objetivo da implementação de um protótipo foi mostrar a realizabilidade das teorias expostas, sem aprofundar detalhes específicos de implementação.

São apresentadas a seguir, as partes chave do código LISP implementado, que usa cada agente protótipo de ordem 2 implicado na Semiotização Natural[101]:

- Competências-Necessárias (*list-people-required-skill-construction.lsp*):

```
(defun set-required-skill-words (WORDS)          "Construct a (PERSON ASSOCIATED-SKILL) list"
  (remove nil (cond                              "from the recursive tracking of a CD"
     ((atom (car WORDS)) NIL) ((NULL (header-cd (car WORDS))) NIL)
     ((and (member (header-cd (car WORDS)) *CD-ACTS*)
           (not (eq (header-cd (car WORDS)) 'QUALIFICATION)))
      (append  (cons  (cons  (cdpath '(QUALIFIED) (filler-role 'actor (car WORDS)))
                             (remove-all-pairs-with (filler-role 'actor (car WORDS)) (car WORDS)))
                      (set-required-skill-words (roles-cd (car WORDS))))
               (set-required-skill-words (cdr WORDS))))
     (T (append  (set-required-skill-words (roles-cd (car WORDS)))
                 (set-required-skill-words (cdr WORDS)))))))
```

Observação: este algoritmo realiza o percurso recursivo de uma DC geral, construindo, a medida do percurso, uma lista do tipo (DUPLET_1 ... DUPLET_n), onde DUPLET := (PERSON ASSOCIATED-SKILL). Um exemplo da lista obtido através da aplicação de *set-required-skill-words* seria: ((PERSON-NAMED-ROBERTO KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED NORM-OF-EXPRESSION))))), possuindo apenas um *DUPLET*, significando que Roberto está associado ao conhecimento de uma linguagem.

- Match-Pessoas-Competências (*List- people-skill-matched-construction.lsp*):

```
(defun list-skilled-people (CD PEOPLE-SKILL-BASE SKILL-BASE)   "Gives back list of Skilled-People"
  (cond ((eq (find-s CD SKILL-BASE) -1) NIL)                   "from BASES"
        (T (list (remove nil (mapcar #'(lambda (X) (if (member (find-s CD SKILL-BASE) X) (car X)))
                                     PEOPLE-SKILL-BASE))))))
(defun list-matched (CD SKILL-BASE)  "Returns list of matched CDs of CD inside SKILL-BASE"
  (cond  ((NULL SKILL-BASE) NIL)     "Applying recursively Match between a CD and a BASE"
         ((or (match CD (car SKILL-BASE) nil) (match (car SKILL-BASE) CD nil))
              (cons (car SKILL-BASE) (list-matched CD (cdr SKILL-BASE))))
         (T (list-matched CD (cdr SKILL-BASE)))))
(defun find-matched (PEOPLE-SKILL SKILL-BASE PEOPLE-SKILL-BASE) "Constructs Duplets"
  (mapcar #'(lambda (X) (cons X (list-skilled-people X PEOPLE-SKILL-BASE SKILL-BASE)))
          (list-matched (cdr PEOPLE-SKILL) SKILL-BASE)))
(defun make-list-matched (PEOPLE-SKILL-LIST SKILL-BASE PEOPLE-SKILL-BASE)
  (cond  ((NULL PEOPLE-SKILL-LIST) NIL) "Constructs List of Duplets (Skill List-Skilled-People)"
         (T (append (find-matched (car PEOPLE-SKILL-LIST) SKILL-BASE PEOPLE-SKILL-BASE)
         (make-list-matched (cdr PEOPLE-SKILL-LIST) SKILL-BASE PEOPLE-SKILL-BASE)))))
```

[101] Como todos os agentes de ordem 2, os agentes coordenados por Detector-Pro-Ativo foram construídos a partir do modelo já exposto p. 91, sendo tal modelo uma implementação do Agente Essencial da Figura 2.12.

1. A função *make-list-matched* constrói, a partir de uma lista de (PERSON ASSOCIATED-SKILL), do Mapeamento de Competências SKILL-BASE, e da base de Relacionamentos Pessoas-Competências PEOPLE-SKILL-BASE, uma lista de MATCHED-DUPLET, onde:
   MATCHED-DUPLET := (CD LIST-SKILLED-PEOPLE)
   LIST-SKILLED-PEOPLE := (PERSON_1 … PERSON_n)
2. Tal construção exige a aplicação recursiva do algoritmo MATCH, entre os ASSOCIATED-SKILL e a SKILL-BASE, realizada pelas funções *find-matched* e *list-matched*.
3. A função *list-skilled-people* simplesmente retorna uma lista de pessoas associadas a uma competência CD, a partir das informações contidas em SKILL-BASE e PEOPLE-SKILL-BASE.

- Remover-Match-Inúteis (*Agent-useless-matched-remotion.lsp*):

```
(defun remove-identic-people (PEOPLE-LIST1 PEOPLE-LIST2) "Removes from PEOPLE-LIST1 all"
   (cond  ((NULL PEOPLE-LIST1) NIL)                          "atoms which can also be found in"
          ((member (car PEOPLE-LIST1) PEOPLE-LIST2)          "PEOPLE-LIST2"
                   (remove-identic-people (cdr PEOPLE-LIST1) PEOPLE-LIST2))
          (t (cons (car PEOPLE-LIST1)
                   (remove-identic-people (cdr PEOPLE-LIST1) PEOPLE-LIST2)))))
(defun remove-useless (SKILL-PEOPLELIST-LIST PEOPLE-LIST) "Remove from SKILL-PEOPLELIST"
   (cond  ((NULL SKILL-PEOPLELIST-LIST) NIL)                 "-LIST all useless matched-duplets"
          ((NULL (remove-identic-people (cadar SKILL-PEOPLELIST-LIST) PEOPLE-LIST))
                   (remove-useless (cdr SKILL-PEOPLELIST-LIST) PEOPLE-LIST))
          (T (cons
                   (cons     (caar SKILL-PEOPLELIST-LIST)
                             (cons (remove-identic-people (cadar SKILL-PEOPLELIST-LIST)
                                                          PEOPLE-LIST) NIL))
                   (remove-useless (cdr SKILL-PEOPLELIST-LIST) PEOPLE-LIST)))))
(defun remove-useless-matched (SKILL-PEOPLELIST-LIST DIALOG) "Removes all useless matched"
          (remove-useless SKILL-PEOPLELIST-LIST (extract-all-actors DIALOG)))
```

Observação: a função *remove-useless* simplesmente apaga na lista SKILL-PEOPLELIST-LIST todos os PERSON da LIST-SKILLED-PEOPLE dos MATCHED-DUPLET cujo nome é também contido na lista das pessoas envolvidas na enação semiótica real (obtida pela cláusula "(extract-all-actors DIALOG)" da função *remove-useless-matched*). Para tal, a função *remove-identic-people* é usada para remover de PEOPLE-LIST1, todos os átomos também contidos em PEOPLE-LIST2.

- Construir-Resposta (*Agent-explanation-construction.lsp*):

```
(defun list-people (LISTEP LEXIC)
   (cond  ((NULL LISTEP) '(ABOUT FOLLOWING THE KNOW I))
          (T (append (list (lexic-value (car LISTEP) LEXIC))
                     (if (not (null (cadr LISTEP))) '(AND)) (list-people (cdr LISTEP) LEXIC)))))
(defun tell-all-pathes (LISTEP LIST-SKILL-PEOPLELIST PENET LEXIC CONCEPT-BASE DIALOG)
   (cond  ((NULL LISTEP) NIL)
          (t (append (tell-path (cons (car (extract-actors-from (car (last DIALOG)))) (list (car LISTEP)))
                                PENET LEXIC)
                     (tell-all-pathes (cdr LISTEP) LIST-SKILL-PEOPLELIST PENET LEXIC
                                      CONCEPT-BASE DIALOG)))))
(defun tell-found-skills (LIST-SKILL-PEOPLELIST PENET LEXIC CONCEPT-BASE DIALOG)
   (cond  ((NULL LIST-SKILL-PEOPLELIST) '(THAT IS ALL.))
          (t (append (reverse (list-people (cadar LIST-SKILL-PEOPLELIST) LEXIC))
                     '( - )
                     (say-cd-english
                       (cons (caaar LIST-SKILL-PEOPLELIST)
                             (cons (cons 'ACTOR (list (caadar LIST-SKILL-PEOPLELIST)))
                                   (cdaar LIST-SKILL-PEOPLELIST))) LEXIC CONCEPT-BASE)
                     '(!)
                     (tell-all-pathes (cadar LIST-SKILL-PEOPLELIST) LIST-SKILL-PEOPLELIST
                                      PENET LEXIC CONCEPT-BASE DIALOG)
                     (tell-found-skills (cdr LIST-SKILL-PEOPLELIST) PENET LEXIC CONCEPT-BASE
                                        DIALOG)))))
```

Observações:

1. A construção da intervenção semiótica em linguagem natural do SMA é realizada de forma recursiva a partir de LIST-SKILL-PEOPLELIST (obtida graças ao agente Remover-Match-Inúteis), pela função *tell-found-skills*, por meio da concatenação da lista resultado da aplicação da função *list-people*, com a lista resultado da aplicação da função *say-cd-english* (que é uma versão simplificada de *Micro-Mumble*), e a lista resultado da aplicação da função *tell-all-pathes*.
2. A função *list-people* constrói uma lista do tipo (I KNOW THE FOLLOWING ABOUT PERSON_1 AND … AND PERSON_n) a partir de um Dicionário LEXIC e uma lista LISTEP do tipo (CONCEPT_PERSON_1 ... CONCEPT_PERSON_n).
3. A função *tell-all-pathes* constroi uma lista “caminhos necessários para encontrar pessoas” do tipo (IF INQUIRER_1 WANTS TO MEET PERSON_FOUNDED_1 THEN INQUIRER_1 CAN TALK TO PERSON_1 AND PERSON_1 CAN TALK TO ... AND PERSON_N-1 CAN TALK TO PERSON_N AND THE PROBLEM WILL BE SOLVED). Cada um destes “caminhos necessários” é construído pela

função *tell-path*, que aplica um simples algoritmo *width-first* sobre a Base de Pessoas Relacionadas. A função *tell-path* pode ser encontrada no arquivo '*people-net-construction.lsp*'.

4. A função *say-cd-english* é uma versão simplificada de *Micro-Mumble* que realiza uma transformação de uma DC em uma expressão natural Inglesa, a partir da aplicação sucessiva de condições do tipo "((eq (header-cd CD) 'PRIMITIVE) (say-PRIMITIVE CD))"[102], onde PRIMITIVE é uma primitiva conceitual de DC. Esta função pode ser encontrada no arquivo '*English-Generator.lsp*'.

Cabe ressaltar, mais uma vez, que um detalhamento maior dos algoritmos implementados pode ser obtido consultando o código LISP disponível no apêndice B.

3.3.3. Exemplo de Semiotização Natural

Da mesma forma que foi pragmaticamente ilustrada a Neutralização Semiológica com um exemplo, serão analisados a seguir os tratamentos efetuados pelo agente Detector-Pro-Ativo ao ser originalmente perturbado pela seguinte mensagem no *Black-Board*:

BLACK-BOARD			
Emissor	Destinatário	Objetivo	Conteúdo
Neutralizer	Pro-Ac	Consult	((PERSON-NAMED-JOHN) (PERSON-NAMED-ROBERTO) ((ENGLISH PRESENT INTERROGATION POSITIVE) (KPOSS (ACTOR (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DOMINIQUE))) (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-LISP) (QUALIFIED NORM-OF-EXPRESSION))))))

Tabela 3.6: Exemplo de Perturbação Implicando uma Semiotização Natural.

Tal perturbação é o resultado de uma seqüência de enações realizadas entre os agentes Interface, Reconhecedor-Idioma, e Neutralizador-Semiotico, na modalidade da Neutralização Semiológica, conforme ilustrado previamente na Tabela 3.4. Em outras palavras, a fase de Semiotização Natural aqui descrita pode ter sida originalmente desencadeada pela semiose:

From: JOHN	To: ROBERTO	Content: "Do you know the Lisp language?"

[102] Para mais detalhes sobre as primitivas LISP usadas, consultar o ANSI COMMON LISP [GRA96].

Para produzir a Semiotização Natural adequada, o Dicionário, a Base de Conceitos e a Base de DCs apresentado na parte 3.2.3 precisam ser complementados da seguinte forma:

- Dicionário:

```
(DO PRESENT PINDEF GINDEF NINDEF DO-INTERROGATION)
(YOU TINDEF OTHER GINDEF SING YOUR-SELF)
```

- Base de Conceitos:

```
(DO-INTERROGATION POSITIVE INTERROGATION)
(YOUR-SELF POSITIVE PERSON)
```

- Base de DCs[103]:

```
(defword DO-INTERROGATION ((assign *part-of-speech* 'interrogation […])
(defword YOUR-SELF((assign *part-of-speech* 'person *cd-form* *WORD*)))
```

Deve finalmente ser feita a suposição, para permitir uma ilustração adequada da Semiotização Natural, de que, após uma série de enações entre a sociedade de agentes de software responsável pelo mapeamento inteligente de competências, e a sociedade de profissionais envolvidos no uso de tecnologias de colaboração, o Mapeamento de Competências, a Base de Relações Pessoas-Competências e a Base de Pessoas Relacionadas encontram-se, por exemplo, no seguinte estado:

- Mapeamento de Competências:

```
((KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-FILIPE))))
 (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-JACK))))
 (KPOSS (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-LISP)
                (QUALIFIED NORM-OF-EXPRESSION)))))
```

- Base de Relações Pessoas-Competências:

```
((PERSON-NAMED-JACK 1) (PERSON-NAMED-JOHN 2) (PERSON-NAMED-FILIPE 3))
```

- Base de Pessoas Relacionadas:

```
((PERSON-NAMED-JACK PERSON-NAMED-FILIPE)
(PERSON-NAMED-JOHN PERSON-NAMED-JACK))
```

A Figura 3.16 detalha a seqüência de transformações operadas pelos vários agentes de ordem 2 responsáveis pela Semiotização Natural considerada neste exemplo.

[103] Para mais detalhes, consultar a totalidade do código do protótipo, disponível no apêndice B.

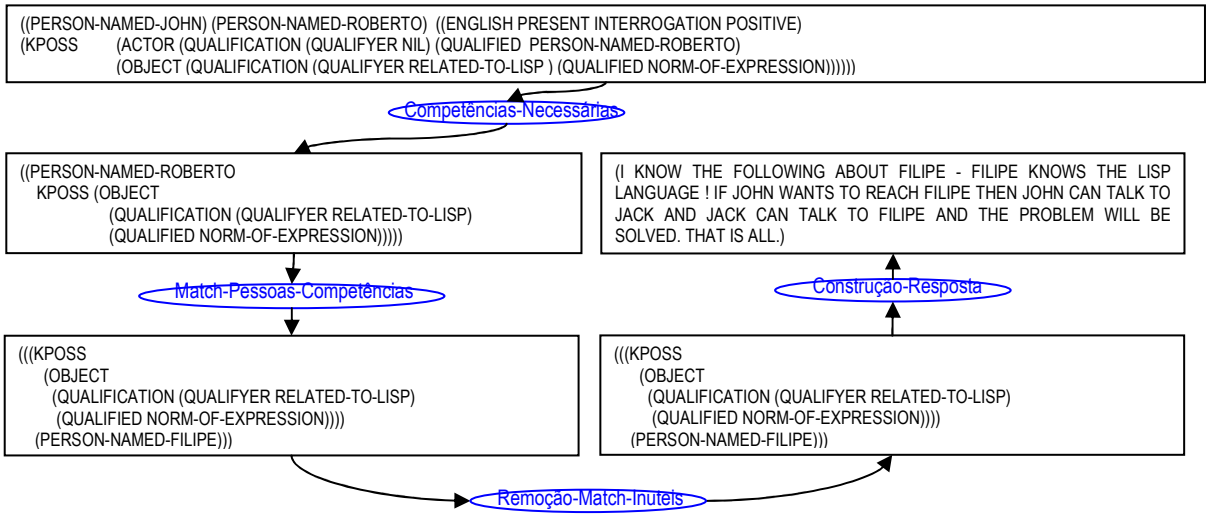


Figura 3.16: Detalhamento da Semiotização Natural Através de um Exemplo.

Após ter sido realizada a Semiotização Natural decorrente da fórmula ((PERSON-NAMED-JOHN PERSON-NAMED-ROBERTO) ((ENGLISH PRESENT INTERROGATION POSITIVE) (KPOSS (ACTOR (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO))) (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-LISP) (QUALIFIED NORM-OF-EXPRESSION))))), o Detector-Pro-Ativo passa um recado no *Black-Board* para o agente Interface, pedindo a geração de uma perturbação semiótica em linguagem natural na direção dos agentes humanos, afirmando, conforme solicitado pelo Detector-Pro-Ativo: “I Know the following about Filipe - Filipe knows the LISP language! If John wants to reach Filipe then John can talk to Jack and Jack can talk to Filipe and the problem will be solved. That is all.”

3.3.4. Hermenêutica do Agente Artificial

A partir da efetiva operacionalização de uma Semiotização Natural realizada por uma sociedade de agentes virtuais, na direção de uma sociedade de agentes humanos, podemos pragmaticamente abordar a noção **inserção hermenêutica do artefato tecnológico**, bem como propor uma revisão do programa de pesquisa da IA. Classicamente, este programa tende modelar sistemas que exibe *inteligência*, ou no sentido de *tomar decisões* ou *agir* de forma *racional*, ou *tomar decisões* ou *agir* como *um ser humano* (como pode ser visto na Tabela 3.7). Nestas abordagens, a cognição é apenas reduzida à **razão** (o que, intuitivamente, parece incompleto), ou apenas abordada de forma **antropomórfica** (o que é também

limitado). Deve também se ressaltar que, nas abordagens racional e antropomórfica, a orientação do pesquisador, tipicamente preso na primeira cibernética, consiste em querer **mecanizar a cognição**, em vez de **criar máquinas cognitivas**. A IA assim abordada trata apenas de governar (*kubernètès*) sistemas para mantê-los, de forma heterônoma, no que achamos, por introspecção, ser o "racional" ou o "humano".

Sistemas que pensam como os humanos	Sistemas que pensam racionalmente
Sistemas que agem como os humanos	Sistemas que agem racionalmente

Tabela 3.7: As quatro categorias de enquadramento da IA.[SHM99], p.5

Porém, a partir da superação de uma visão ontológica objetivista ingênua do acesso ao mundo, pode-se concentrar o esforço de pesquisa sobre o **fenômeno** da inteligência como *fenômeno*. Isto permite verificar que a cognição *emerja* de entidades que *exibem* **autonomia**, **identidade** e **intencionalidade**. Nesta perceptiva, o primeiro esforço da IA deve consistir em criar máquinas topologicamente estáveis, e governadas pelas suas próprias aspirações à auto-conservação. Tais máquinas, não mais condicionadas por entradas, interagem na modalidade da **enação**. É desta interação estável no tempo que, nesta abordagem, *emerja* a inteligência percebida como *intencionalidade*. O que finalmente garante a *minha* percepção da intencionalidade da máquina, é sua estabilização *no meu contexto*. Por este motivo, para efetivamente ser ***cognitiva para mim***, a máquina não apenas deve ser autopoiética, mas também ser hermeneuticamente inserida em uma sociedade da qual participo. Tal sociedade seria, no mínimo, a minha **consciência**: **uma sociedade de eu com eu mesmo**.

Por este motivo, o programa de pesquisa da IA não deveria, na nossa percepção, se perder na busca do "racional" ou do "humano", mas sim investigar o "autônomo" e o "hermeneuticamente inserido". Esta segunda propriedade seria o que Havelange define como "inscrição artefatual do saber" [HAV99c], p.96, e o que definimos aqui como "hermenêutica do artefato tecnológico" [CAS03c], p.12, ou, mais especificamente, "**hermenêutica do agente artificial**". A inserção hermenêutica do agente é tão fundamental para a exibição de inteligência, que, a partir do momento que não existe mais superposição enativa entre eu e o artefato computacional (por simples perda de confiança no agente, por exemplo), tal artefato passa ser naturalmente excluído da minha sociedade, e, logo, não consegue mais gerar inserções no meu contexto, perdendo toda característica de intencionalidade para mim.

3.4. Conclusão

A partir da adoção do paradigma da **Autopoiésis**, para caracterizar a natureza da cognição, e da escolha de uma visão Peirciana do fenômeno semiótico **individual**, e Saussuriana do fenômeno semiótico **coletivo**, foi detalhado neste capítulo como uma sociedade de agentes virtuais exibindo **autonomia**, **identidade topológica** e **intencionalidade emergente**, pode ser efetivamente criada para operacionalizar mapeamentos inteligentes de competências, que sejam (i) **construídos de forma automática**, (ii) **de entendimento comum entre todos os agentes** envolvidos no seu uso, (iii) **completos** e **consistentes**, (iv) **atualizados em "tempo real" em uma "memória corporativa dinâmica"**, e, (v) capazes de fornecer **pró-ativamente, em linguagem natural, informações pertinentes, personalizadas e contextualizadas.**

Foi detalhada, na parte 3.1, uma arquitetura de SMA implementando tal mapeamento, no contexto de uma sociedade de profissionais interconectados por tecnologias de colaboração. Foi mostrado como as enações entre o SMA responsável pelo mapeamento, e a sociedade de profissional pode criar **uma dupla inserção hermenêutica**: uma inserção do real no virtual por **Neutralização Semiológica**, e uma inserção do virtual no real por **Semiotização Natural**. Como resumido na Figura 3.17, a Neutralização Semiológica (detalhada teoricamente e computacionalmente na parte 3.2) consiste em transformar uma semiose qualquer em uma fórmula informacional semioticamente neutra (independente da linguagem e da cultura de quem emitiu a enação responsável pela semiose), e a Semiotização Natural (detalhada teoricamente e computacionalmente na parte 3.3) consiste na operação inversa.

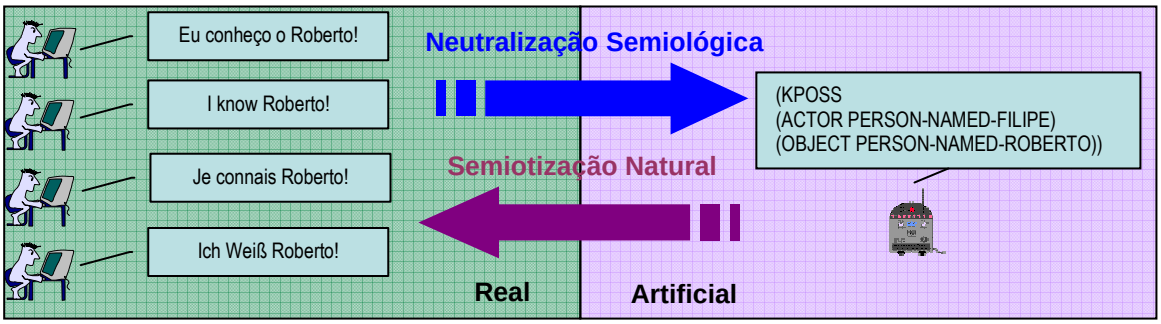


Figura 3.17: Visão Resumida da Neutralização Semiológica e da Semiotização Natural.

Finalmente, a Figura 3.18 resume como as duas operações de Neutralização Semiológica e Semiotização Natural são usadas pela arquitetura de SMA apresentada, para permitir a emergência das propriedades (i), (ii), (iii), (iv) e (v) acima mencionadas.

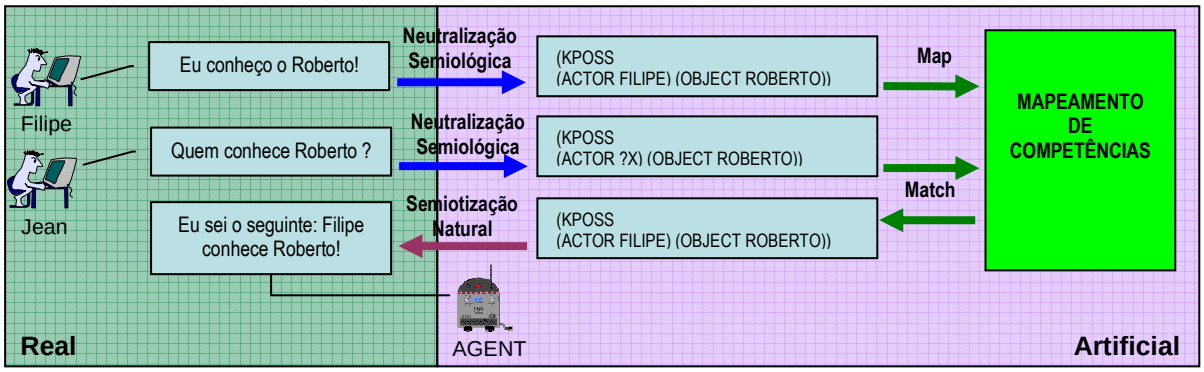


Figura 3.18: Utilização da Neutralização Semiológica e da Semiotização Natural no SMA.

De forma genérica, pode-se afirmar, sendo o reuso feito do paradigma de IAS das DCs de Schanck, que o trabalho experimental realizado consistiu basicamente em **encapsular o código alopoéitico de um paradigma particular de PLN, em um corpo autopoiético**. Em outras palavras, as implementações realizadas consistem em uma encarnação formal do heterônomo, para torná-lo autônomo, e hermeneuticamente inserido.

No próximo capítulo, serão finalmente apresentados e discutidos dois cenários pragmáticos de uso do protótipo proposto neste capítulo, um consistindo no desempenho de um mapeamento inteligente de competência no contexto de uma “incubadora de fábricas de Software”, e o outro no contexto de um sistema de saúde “Home-care”.

Capítulo 4

Pragmática e Discussão

Após ter descrito, no capítulo 3, como o estudo epistemológico realizado no capítulo 2 se aplica computacionalmente à criação de mapeamentos inteligentes, através do uso de arquiteturas multiagente, serão propostas e discutidas neste penúltimo capítulo duas ilustrações pragmáticas[104] do uso do protótipo de sociedade de mapeamento inteligente de competências realizado durante esta pesquisa. A primeira ilustração mostrará, na parte 3.1, como o protótipo pode apoiar uma estratégia de compartilhamento de conhecimento em uma "**Incubadora de Fábricas de Software**", onde várias empresas, com vários colaboradores especializados em tarefas de engenharia, podem colaborar para acelerar o ciclo de desenvolvimento e aumentar a qualidade dos produtos. A segunda ilustração mostrará, na parte 3.2, como usar o mesmo protótipo no contexto de um sistema de saúde "**Home-Care**", onde pacientes e médicos podem ser colocados automaticamente em relação, por ação dos agentes do mapeamento inteligente de competências, o que permite acelerar o tempo de resposta em caso de necessidade, e otimizar a atribuição de tarefas aos médicos.

As duas ilustrações serão discutidas, de forma a poder analisar as limitações do protótipo realizado, o que permitirá encaminhar, na parte 4.3, algumas propostas de extensão do trabalho realizado (expansão da hermenêutica para outras sociedades, para melhores interações lingüísticas, para outros tipos de enações semióticas, e para outros tipos de tarefas).

[104] Os dois cenários abordados foram escolhidos devido à alta intensidade do uso do conhecimento nas atividades vinculadas, conforme argumentado em [TER02], p. 26: "o preço de produtos de saúde e educação [...] cresceu aproxidamante três vezes mais que a inflação [nos últimos 40 anos]"; e p. 28: "As três maiores profissões em termos de crescimento previsto, nos Estados Unidos, para o perdio de 1998-2008, são: Engenheiro de Computação (108%), Especialista de Suporte para computadores (102%), e Analista de Sistemas (94%)".

4.1. Cenário "Incubadora de Fábricas de Software"

Como argumentado na introdução deste trabalho, as organizações do século XXI devem saber gerenciar efetivamente seus ativos intangíveis [SVE98], sendo que, na nova economia, a geração de riquezas passa pela produção e exportação de produtos e serviços ricos em conhecimento [TER02]. Um tipo de produto particularmente intensivo em conhecimento é obviamente o Software, inteiramente intangível por natureza[105]. Exemplos comprovando que um enfoque político dado no aumento do grau médio de escolaridade dos habitantes de um país[106], complementado por uma política industrial voltada a transferência de tecnologia da universidade para o setor produtivo, e proteção do capital intelectual, é capaz de aumentar consideravelmente a capacidade de geração de valor de um país. O exemplo clássico que compara as políticas industriais da Coréia e do Brasil, nos últimos 50 anos, pode ser encontrado em [TER02b]. Tal esforço político passa, entre outras iniciativas, por fomentar a criação de **incubadoras de empresas de alta tecnologia**, como incubadoras de fábricas de software. A partir da existência de tais incubadoras, é obvia a afirmação que a criação de redes de compartilhamento de conhecimento, a partir do uso de tecnologias de colaboração síncronas e assíncronas, colocará as várias empresas incubadas em relação de tipo *ganha-ganha*[107], permitindo otimizar os processos de aprendizagem e produção de conhecimento, aumentando as chances de sucesso no mercado internacional.

A primeira ilustração pragmática proposta mostrará como o SMA descrito neste trabalho pode ser utilizado no contexto de uma incubadora de fábricas de software, **para fomentar o compartilhamento de conhecimento**.

4.1.1. Descrição da Sociedade "Incubadora de Fábricas de Software"

Era uma vez uma incubadora de fábricas de Software, chamada INCFABSOFT, na qual três empresas estavam incubadas: FABSOFTA, FABSOFTB e FABSOFTC. FABSOFTA era uma empresa especializada em de processos de desenvolvimento de Software, possuindo uma grande experiência, não só na avaliação de processos, mas também

[105] Como sabemos, a máquina de Turing (computador digital) é uma máquina conceitual, e o algoritmo que pode seguir (ou software) é independente da realização física de tal máquina.
[106] Terra, em [TER02], p. 28, argumenta que: "um estudo econômico envolvendo 29 paises mostrou que o investimento na educação foi responsável por 25% do crescimento econômico".
[107] Uma relação "ganha-ganha" (*win-win relation*) é um tipo de relacionamento entre duas ou mais partes, onde a ação mútua de uma parte sobre uma outra permite beneficiar todos os envolvidos. Tipicamente, as relações compartilhamento de conhecimento são relações "ganha-ganha".

no apoio a implementação de melhorias. FABSOFTB era uma empresa especializada em desenvolvimento, que já tinha uma grande experiência em projetos envolvendo o uso de várias linguagens de programação. FABSOFTC, finalmente, era uma jovem empresa de desenvolvimento, recém formada e incubada em INCFABSOFT, com pouca experiência.

FABSOFTA era composta por vários colaboradores, entre eles, os analistas de processo Filipe e Roberto. Filipe conhecia as normas ISO14000, ISO9000, ISO9001. Roberto conhecia os modelos CMM, PDCA, TQM. FABSOFTB contava, em seus colaboradores, Márcio, Dani e Dominique. Márcio era um programador experiente que conhecia as linguagens LISP, C++, PASCAL, e VISUALBASIC. Dani tinha acabada de ser contratada por FABSOFTB, e tinha muito a aprender, como muito para oferecer, sendo uma programadora experiente em C++, PASCAL, LISP, e VISUALBASIC também. Dominique era responsável pelo processo de teste de software, e conhecia as técnicas de testes a serem aplicadas para programas escritos em C++, PASCAL, LISP, e VISUALBASIC. Dominique tinha uma segunda atividade fora de FABSOFTB: era professora, e ensinava os modelos CMM, TQM. FABSOFTC possuía vários colaboradores, entre eles, John, Jack e Jean. Jack, ainda pouco experiente, conhecia as linguagens de programação PASCAL, C++. Jean, não muito mais experiente, conhecia apenas a linguagem LISP. John tinha experiência com a linguagem VISUALBASIC, e conhecia também Filipe, da FABSOFTA, por que tinham estudado juntos, no passado.

Recentemente, Roberto, da FABSOFTA, tinha conduzido em FABSOFTB com o apoio de Dominique uma avaliação CMM, a partir da qual tinha recomendado a implementação do modelo TQM. Sendo Dominique competente no uso do modelo TQM, ela tinha iniciado tal implementação em FABSOFTB.

Era regra, em INCFABSOFT, o uso de um Bate-Papo na Intranet da incubadora, para fomentar a colaboração entre as empresas. Os vários atores envolvidos nas sociedades FABSOFTA, FABSOFTB, FABSOFTC, iriam descobrir, a partir da implementação de um SMA na Intranet de INCFABSOFT, a existência de um novo colega, chamado "*AGENT*", sempre disposto a dar boas dicas...

4.1.2. Vocabulário da Sociedade "Incubadora de Fábricas de Software"

Para que o protótipo do SMA realizado nesta pesquisa possa se inserir hermeneuticamente na sociedade "Incubadora de Fábricas de Software" descrita, o seguinte

vocabulário foi definido em termos de Dicionário, Base de Conceitos, Base de Dependências Conceituais, e Dicionário de Tradução DC-Inglês:

A	AN	ABOUT	AGENT	AGENT'S	ALGORITHM	ALGORITHMS
ALL	ASP	AT	ATTRIBUTE	ATTRIBUTED	ATTRIBUTES	AUDIT
AUDITED	AUDITS	BELIEVE	BELIEVED	BELIEVES	BOOK	BOOKS
C++	CAN	CERTIFICATE	CERTIFICATES	CMM	COMPANY	COMPANIES
COMPLETE	COMPLETED	COMPLETES	CORRECT	CORRECTED	CORRECTS	COULD
COURSE	COURSES	CREATE	CREATED	CREATES	DANI	DANI'S
DESCRIBE	DESCRIBED	DESCRIBES	DETECT	DETECTED	DETECTS	DID
DISTRICT	DISTRICTS	DO	DOES	DOMINIQUE	DOMINIQUE'S	ENGLISH
ERROR	ERRORS	EVALUATE	EVALUATED	EVALUATES	EXAMINE	EXAMINED
EXAMINES	EXACT	EXECUTE	EXECUTED	EXECUTES	EXERCISE	EXERCISES
EXPLAIN	EXPLAINED	EXPLAINS	FABSOFTA	FABSOFTB	FABSOFTC	FILIPE
FILIPE'S	FRENCH	FROM	FULFILL	FULFILLED	FULFILS	HAD
HAS	HAVE	HE	HOUSE	HOUSES	HOW	I
IDEA	IDEAS	IDENTIFY	IDENTIFIED	IDENTIFIES	IMPLEMENTATION	IMPLEMENTATIONS
IN	INCFABSOFT	INCUBATOR	INCUBATORS	INEXACT	INITIATE	INITIATED
INITIATES	ISO14000	ISO9000	ISO9001	JACK	JACK'S	JEAN
JEAN'S	JOHN	JOHN'S	KNEW	KNOW	KNOWS	LABORATORY
LANGUAGE	LANGUAGES	LESSON	LESSONS	LISP	LIVE	LIVED
LIVES	MARCIO	MARCIO'S	ME	MISTAKE	MISTAKES	MODEL
MODELS	NO	NONCONFORMITIES	NONCONFORMITY	NOT	NOTICE	NOTICED
NOTICES	OBSERVE	OBSERVED	OBSERVES	ON	OPERATE	OPERATED
OPERATES	PASCAL	PDCA	PEOPLE	PERFORM	PERFORMED	PERFORMS
PORTUGUESE	PROBLEM	PROBLEMS	PROGRAM	PROGRAMMED	PROGRAMS	QUARTER
QUARTERS	READ	READS	RIGHT	ROBERTO	ROBERTO'S	SET
SETS	SHE	SOLVE	SOLVED	SOLVES	SOMEONE	START
STARTED	STARTS	STAY	STAYED	STAYS	TAUGHT	TEACH
TEACHES	TELL	TELLS	TEST	TESTED	TESTS	TEXT
TEXTS	TRACK	TRACKED	TRACKS	THAT	THE	THEY
THINK	THINKS	THIS	THOUGHT	TO	TOLD	TQM
UNDERSTAND	UNDERSTANDS	UNDERSTOOD	US	VALUE	VALUES	VARIABLE
VARIABLES	VISUALBASIC	WE	WHAT	WHERE	WHICH	WHO
WITH	WORK	WORKED	WORKS	WRONG	YOU	

Tabela 4.1: Vocabulário Definido para o Cenário "Incubadora de Fábricas de Software"

Deve-se observar que toda a experimentação foi conduzida em Inglês. A mesma experimentação poderia ser realizada em outros idiomas, a partir da simples definição do mesmo vocabulário nestes outros idiomas.

4.1.3. Enações na Sociedade "Incubadora de Fábricas de Software"

São explicitadas a seguir, nas Figuras 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, e 4.8, as enações semióticas realizadas no Simulador, descrito na parte 3.1.2, como experimentação do uso de um mapeamento inteligente de competência em um cenário de oito dias de uma "Incubadora

de Fábrica de Software". Para facilitar a compreensão, é detalhada de forma simplificada a conseqüência de cada enação sobre o estado do Mapeamento de Competências.

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-1: FILIPE, ROBERTO, at FABSOFTA FILIPE > ROBERTO: Hi Roberto! How are you? ROBERTO > FILIPE: Quite tired! I audited this FABSOFTB company with the CMM model. FILIPE > ROBERTO: Do you know the CMM model? ROBERTO > FILIPE: Yes I know the CMM model. I also know the TQM model. FILIPE > ROBERTO: I know the iso9000 model. It looks like the CMM model in some aspects I guess. I also know the iso9001 model. ROBERTO > FILIPE: Fine I have to go now: I have to meet Dominique of FABSOFTB. FILIPE > ROBERTO: Do you know Dominique? ROBERTO > FILIPE: Yes I know Dominique. Dominique started a TQM implementation in the FABSOFTB company. Do you know her? FILIPE > ROBERTO: Yes I know Dominique too! ROBERTO > FILIPE: All right! Bye! FILIPE >ROBERTO: Bye!	Filipe knows Roberto, Roberto knows Filipe Roberto audits FABSOFTB with CMM model. Roberto knows CMM model. Roberto knows TQM model. Filipe knows iso9000 model. Filipe knows iso9001 model. Filipe knows Dominique. Dominique starts TQM implementation at FABSOFTB Roberto knows Dominique.

Figura 4.1: Primeiro Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-2: MARCIO, DANI, at FABSOFTB MARCIO > DANI: Hi Dani! Can you help me with a program? DANI > MARCIO: Hi! Sure. How may I help? MARCIO > DANI: Do you know the PASCAL language? DANI > MARCIO: Yes I know the PASCAL language. I know the C++ language. I also know the LISP language. MARCIO > DANI: Really. Fine. I know the LISP language too. My problem is a bug in a PASCAL program. DANI > MARCIO: Who tested your program? MARCIO > DANI: Dominique tested this PASCAL program. DANI > MARCIO: What else can Dominique do? MARCIO > DANI: Dominique can test a LISP program. Dominique can also test a C++ program. Dominique also knows the CMM model. AGENT > MARCIO: I know the following about Roberto: Roberto knows the CMM model! There is no way that Marcio can reach Roberto with his relationships. That is all. MARCIO > AGENT: That should interest Dominique. Where does Roberto work? AGENT > MARCIO: I cannot say anything about that. DANI > MARCIO: Coming back to Dominique. What else does Dominique do? MARCIO > DANI: Actually, Dominique is also a teacher. Dominique teaches the CMM model. Dominique also teaches the TQM model. DANI > MARCIO : All right. Let me look at your BUG, and I will get in touch with you. MARCIO > DANI: Fine. Bye. DANI > MARCIO: Bye.	Marcio knows Dani, Dani knows Marcio. Dani knows pascal language. Dani knows C++ language. Dani knows lisp language. Marcio knows lisp language. Marcio knows Dominique, Dani knows Dominique. Dominique tests pascal programs Dominique tests lisp programs. Dominique tests C++ programs. Dominique knows CMM model Marcio knows Roberto. Dominique teaches CMM model. Dominique teaches TQM model.

Figura 4.2: Segundo Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-3: DANI, MARCIO, at FABSOFTB. DANI > MARCIO: Hi Marcio. I corrected the PASCAL program. MARCIO > DANI: Hi! Really? So you detected the error! DANI > MARCIO: Yes. I know how to correct a program. MARCIO > DANI: Well, thanks! Bye. DANI > MARCIO: Bye.	Dani corrects pascal programs. Dani detects erros. Dani knows how to correct programs.

Figura 4.3: Terceiro Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-4: FILIPE, JOHN, in a CHAT JOHN > FILIPE: Hi Filipe! Guess what? FILIPE > JOHN: Hi! What? JOHN > FILIPE: I work in the FABSOFTC company! FILIPE > JOHN: Really? I work in the FABSOFTA company! It is in the same area! JOHN > FILIPE: It will easy to meet then! FILIPE > JOHN: Yes. See you there then. Bye. JOHN > FILIPE: Yes. See you. Bye.	Filipe knows John. John knows Filipe. John works at FABSOFTC. Filipe works at FABSOFTA.

Figura 4.4: Quarto Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-5: JOHN, JACK, JEAN, at FABSOFTC JOHN > JACK: Hi Jack! I am new here. JACK > JOHN: Hi. Nice to meet you. JOHN > JACK: Nice to meet you too. JOHN > JEAN: Hi Jean. Pleasure to meet you. JEAN > JOHN: Welcome. JOHN > JEAN: Thanks. Well, see you. JEAN > JOHN: Bye. JACK > JOHN: Bye.	John knows Jack. Jack knows John. John knows Jean. Jean knows John.

Figura 4.5: Quinto Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-6: DOMINIQUE, ROBERTO in a CHAT DOMINIQUE > ROBERTO: Hi Roberto! I need to meet you to talk about the assessment. ROBERTO > DOMINIQUE: Hi. Ok. Let's meet at 10 a.m. at my office then. Bye. DOMINIQUE > ROBERTO: Ok. See you tomorrow. Bye.	Dominique knows Roberto. Roberto knows Dominique.

Figura 4.6: Sexto Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-7: DOMINIQUE, MARCIO at FABSOFTB DOMINIQUE > MARCIO: Hi Marcio! Are you available for a training on TQM tomorrow? MARCIO > DOMINIQUE: Sure. DOMINIQUE > MARCIO: Ok. Let's schedule at 2 p.m. then. See you tomorrow. MARCIO > DOMINIQUE: Ok. Bye.	Dominique knows Marcio. Marcio knows Dominique.

Figura 4.7: Sétimo Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-8: JEAN, JACK, at FABSOFTC JACK > JEAN: Hi Jack. Can you help me with something? JEAN > JACK: Let's see. JACK > JEAN: I need help on a PASCAL program. JEAN > JACK: Well I do not know the pascal language. AGENT > JEAN: I know the following about Dani: Dani knows the PASCAL language! If Jean wants to reach Dani then Jean can talk to John and John can talk to Filipe and Filipe can talk to Dominique and Dominique can talk to Marcio and Marcio can talk to Dani and the problem will be solved. That is all. JEAN > JACK: What a network! I will talk to John then. JACK > AGENT: Where does Filipe work? AGENT > JACK: I know the following about John: John works at the FABSOFTC Company! If Jack wants to reach John then Jack can talk to John and the problem will be solved I know the following about Filipe: Filipe works at the FABSOFTA Company! If Jack wants to reach Filipe then Jack can talk to John and John can talk to Filipe and the problem will be solved. That is all. JACK > AGENT: Can Filipe test a PASCAL program? AGENT > JACK: I know the following about Dominique: Dominique can examine the Pascal Algorithm! If Jack wants to reach Dominique then Jack can talk to John and John can talk to Filipe and Filipe can talk to Dominique and and the problem will be solved. That is all. JACK > JEAN: Interesting. JEAN > JACK: I need help with a LISP program. Can you help me? JACK > JEAN: I do not know the lisp language. AGENT > JACK: I know the following about Marcio and Dani: Dani knows the LISP language! If Jack wants to reach Dani then Jack can talk to John and John can talk to Filipe and Filipe can talk to Dominique and Dominique can talk to Marcio and Marcio can talk to Dani and the problem will be solved. If Jack wants to reach Marcio then Jack can talk to John and John can talk to Filipe and Filipe can talk to Dominique and Dominique can talk to Marcio and the problem will be solved. That is all. JACK > JEAN: Our software development process is weak. Do you know who could evaluate the company? AGENT > JACK: I know the following about Roberto: Roberto can examine with the CMM model the FABSOFTB Company! If Jack wants to reach Roberto then Jack can talk to John and John can talk to Filipe and Filipe can talk to Roberto and the problem will be solved. That is all. JACK > AGENT: Can someone explain me the CMM model? AGENT > JACK: I know the following about Dominique: Dominique can explain the CMM model! If Jack wants to reach Dominique then Jack can talk to John and John can talk to Filipe and Filipe can talk to Dominique and and the problem will be solved. That is all. JACK > AGENT: Ok. Thanks. Bye.	*Jack knows Jean.* *Jean Knows Jack.*

Figura 4.8: Oitavo Dia Simulado no Cenário "Incubadora de Fábricas de Software"

Pode-se observar nesta simulação como o SMA é capaz de se **inserir hermeneuticamente** na sociedade de profissionais em interação. Serão discutidas a seguir tais inserções. O experimento completo, bem como infinitas variantes dele, podem ser realizados a partir do **simulador** disponibilizado com este trabalho. A instalação e o uso do simulador são detalhados no apêndice A1.

4.1.4. Discussões

Pode-se observar pragmaticamente, com este primeiro experimento, que o objetivo de construir mapeamentos inteligentes exibindo as propriedades de serem (i) **construídos de forma automática**, (ii) **de entendimento comum entre todos os agentes** envolvidos no seu uso, (iii) **completos** e **consistentes**, (iv) **atualizados em "tempo real" em uma "memória corporativa dinâmica"**, e, (v) capazes de fornecer **pró-ativamente, em linguagem natural, informações pertinentes, personalizadas e contextualizadas**, foi razoavelmente bem atingido. O atendimento de (i) é uma conseqüência do SMA ser **autopoiético**, e, logo, ser capaz de manter, de forma autônoma, sua identidade através da dissipação de perturbações constituídas por enações semióticas desempenhadas entre agentes humanos. O atendimento de (ii) é a conseqüência da implementação, através das enações artificiais entre agentes do SMA, das operações de **Neutralização Semiológica** e **Semiotização Natural**. O atendimento de (iv) é a conseqüência da **intencionalidade emergente** do SMA, bem como da existência da operação de **Semiotização Natural**. A obtenção da propriedade (iii) pode ser discutida. De um ponto de vista fenomenológico, o mapeamento é completo, sendo que tudo que pode efetivamente ser percebido pelo SMA é mapeado. De um ponto de vista objetivista, o mapeamento nunca poderá ser completo, sendo que **a fenomenologia de uma máquina cognitiva não pode ser objetiva**[108]. No que diz respeito à consistência do mapeamento, esta é efetivamente uma primeira fraqueza do protótipo implementado: não foi implementado nenhum agente capaz de verificar a **consistência ou para-consistência lógica** (qualquer que seja o tipo de lógica usada) do mapeamento. Em outras palavras, o mapeamento pode muito bem conter inconsistências lógicas, e não existe, no SMA, nenhum método permitindo a realização de deduções lógicas dos relacionamentos das competências umas com as outras.

[108] Conforme argumentado no capítulo 2, a linguagem, ou qualquer outra enação, não foi "desenhada" para acessar ao mundo "do jeito que é": ela apenas permite fazer emergir um mundo "do jeito que é percebido".

Além destas primeiras ressalvas, pode-se acrescentar que o protótipo proposto possui uma série de outras limitações:

- Para que as frases escritas em linguagem natural possam ser efetivamente entendidas pelo SMA, estas precisam ser escritas **sem erros ortográficos** ou **gramaticais**, e de forma **completamente explícita**. Tipicamente, o SMA não será capaz de entender a frase "I know LISP", que deverá ser substituída por algo mais explícito do tipo "I know the LISP language".
- Na mesma linha, o SMA é incapaz de entender frases complexas semanticamente, envolvendo figuras de estilo como **anti-frases**, **exagerações**, **metáforas**, **fórmulas irônicas**, etc. Podemos falar, metaforicamente, que o SMA é "**ingênuo**".
- A ingenuidade do SMA é também refletida na sua forma **estereotipada** de se expressar. As inserções semióticas que opera são sempre construídas sobre o mesmo modelo de frase, e são extremamente "explícitas".
- O SMA **não possui nenhuma memória** de qual dica já emitiu em qual contexto, e pode ficar repetindo sempre as mesmas dicas para as mesmas pessoas, se encontrar várias vezes o mesmo contexto hermenêutico.
- O SMA apenas mapeia o que é conhecido, e não decore nenhuma **não-competência**. Tipicamente, se um agente humano afirma "Eu não sei X", o SMA não realizará nenhum mapeamento, enquanto, do ponto de vista de uma estratégia de gestão do capital intelectual, o "não conhecimento de X" possui tanta importância como o "conhecimento de X".
- Informações sobre o tempo empregado nas frases (futuro, presente ou passado) **não são utilizadas no protótipo**. As informações mapeadas são desta forma "atemporais", e uma vez integradas ao mapeamento, nunca mais desaparecem (enquanto o conhecimento, ele, é dinâmico, e uma competência não praticada durante um certo tempo desaparece).
- Do próprio fato de ter usado o algoritmo *PARSE* do programa "Micro-ELI" de Birnbaum e Selfridge, as variáveis lingüísticas usadas nas tarefas de PLN **são globais e não locais**. Isto implica a impossibilidade de usar duas vezes a mesma palavra em uma frase, para não gerar confusão no procedimento de criação de DCs. Tipicamente, a frase "I can evaluate **the** FABSOFTB company with **the** CMM model" causará uma confusão entre "FABSOFT company" e "CMM model" devido ao uso repetido da

palavra “the”. Esta última frase deverá, portanto, ser substituída por algo como “I can evaluate this FABSOFTB company with the CMM model”.

Apesar desta série de limitações, deve-se ressaltar que o SMA proposto implementa efetivamente o conceito de mapeamento inteligente, e ilustra bem as noções de “**inscrição tecnológica do saber**” e “**inserção hermenêutica do objeto tecnológico**”. Sendo o processo implementando um mapeamento inteligente autopoiético, ele permite conferir à máquina na qual se executa novas propriedades. Além de se tornar uma inscrição artefatual do saber (o que, na realidade, não é uma propriedade realmente nova: um simples livro também é uma inscrição artefatual do saber), a máquina possui também a característica original de exibir uma inserção hermenêutica **intencional** na sociedade dos homens: ela exibe ***vontade***. Isto constitui efetivamente o que pode se chamar de “humanização da máquina”, por oposição à “mecanização do humano” proposto no paradigma clássico da IAS. No segundo dia simulado no experimento, por exemplo, ao ser perturbado pela enação “Dominique also knows the CMM model”, o agente exibe pró-ativamente (do ponto de vista da fenomenologia de Marcio e Dani) a vontade de participar do diálogo, realizando a enação “Roberto knows the CMM model”. No oitavo dia simulado, várias outras enações “constitutivas de saber” são realizadas pelo SMA, contribuindo para a criação de uma meta-sociedade composta por homens e máquinas, da qual passa a fazer parte.

4.2. Cenário “Home-care”

Um outro ramo de atividade intensivo em conhecimento que se desenvolverá muito nos próximo anos, e cuja prática será cada vez mais uma alta fonte de geração de riqueza, é a **Saúde**[109]. Por este motivo, o segundo cenário escolhido para validar pragmaticamente o uso do SMA proposto na realização de mapeamentos inteligentes de competências é um cenário voltado ao uso da Rede Internet em sistemas integrados de saúde, como o “***Home-Care***” (Figura 4.9).

Uma solução prática, humana, e economicamente vantajosa de recuperação de pacientes com patologias crônicas ou com seqüelas, é permitir seu tratamento e seu acompanhamento médico em sua própria casa. Este tipo de solução permite efetivamente

diminuir o número de dias de internação, otimizar recursos hospitalares e permitir ao paciente reabilitar-se mais rapidamente, por estar no seu ambiente e no convívio familiar. A operacionalização de um tal empreendimento, denominado de "Tratamento em Casa", ou "*Home-Care*", requer a sincronização e organização de profissionais de saúde de diversas áreas, preparados para o atendimento domiciliar, envolvendo equipes multidisciplinares, supervisão médica, equipes especializadas de enfermagem, fisioterapeutas, fonoaudiólogos, nutricionistas, etc. A diversidade, complexidade e distribuição espacial de recursos e competências das mais diversas naturezas necessárias nas práticas de *Home-care* exigem um sistema automatizado tanto para a otimização de recursos materiais como para a identificação, disseminação, e alocação correta de competências.

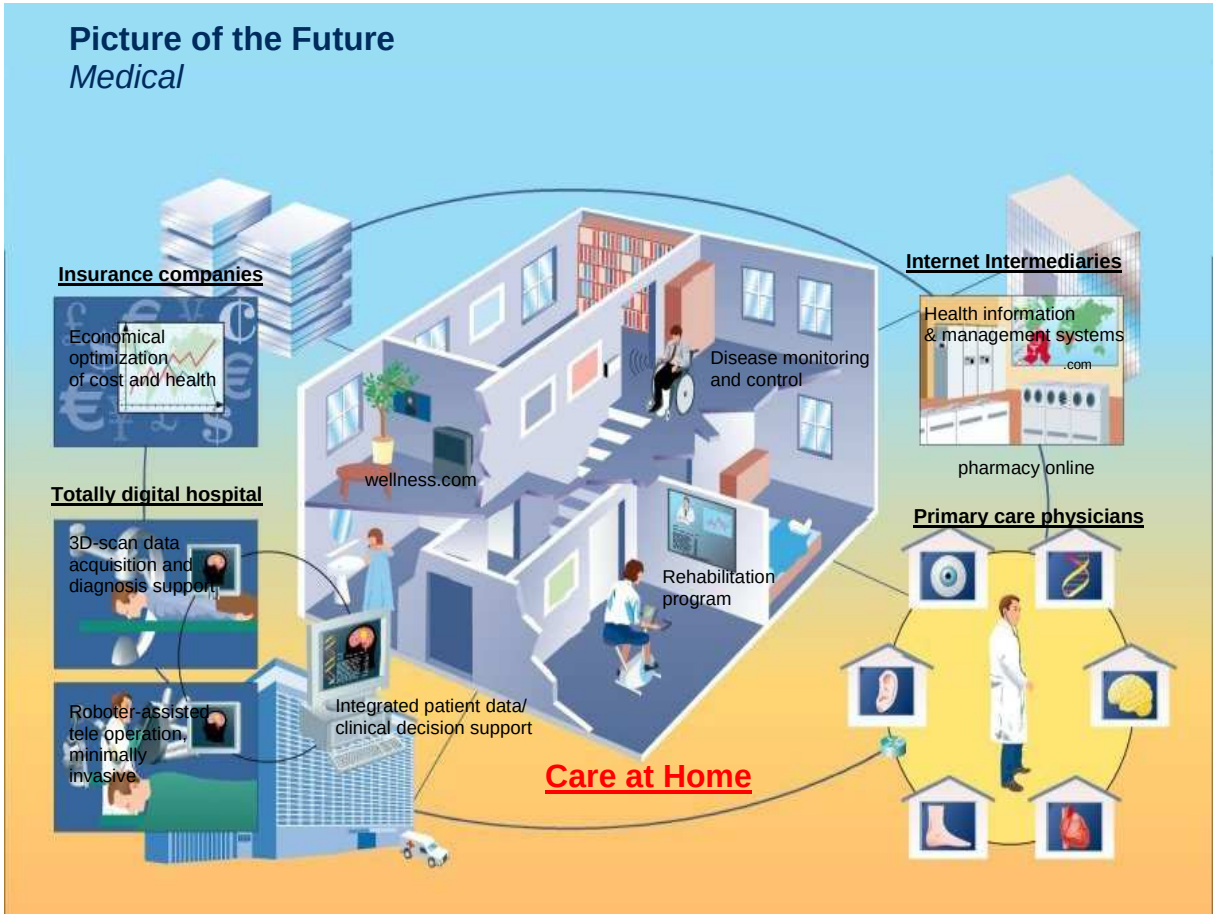


Figura 4.9: Sistema Integrado de Home-Care. [SIE03]

[109] Conforme argumentado por Terra em [TER02], p26: "o poder de compra de bens manufaturados caiu de aproximadamente 75% nos últimos 40 anos, enquanto o preço de produtos de saúde e educação [...] cresceu aproximadamente três vezes mais do que a inflação".

Tomando como principal foco as competências, estas podem ser identificadas, mapeadas, e pró-ativamente propostas, a partir de um processo de **análise automática das interações semióticas** desempenhadas pelas diversas pessoas envolvidas em uma prática de *Home-Care*. Portanto, a segunda ilustração pragmática proposta mostrará como o mesmo SMA capaz de se integrar semioticamente em uma sociedade do tipo "Incubadora de Fábricas de Software", pode também ser utilizado no contexto de uma sociedade "*Home-Care*", fomentando, da mesma forma, a criação de fluxos de conhecimento.

4.2.1. Descrição da Sociedade "Home-care"

Era uma vez um hospital, chamado de "Hospital das Clínicas", no qual o cirurgião Filipe tinha recém ingressado. Seu amigo de infância, Jack, era enfermeiro neste mesmo hospital, onde praticava injeções. Jean era anestesista no Hospital das Clinicas, e John trabalhava no pronto socorro. John era acostumado a praticar reanimações, através de massagens cardíacas, durante suas missões de resgate.

O Hospital das Clínicas trabalhava freqüentemente em conjunto com o laboratório de análise ANALAB, onde Dani realizava análises sangüíneas, diagnósticos de doenças, bem como injeções. Incentivando práticas de "*Home-Care*", o Hospital das Clínicas trabalhava também com a fisioterapeuta Dominique, que, morando no bairro de Batel, praticava massagens para reabilitação de pacientes em suas próprias casas.

Enquanto Filipe iria realizar sua primeira intervenção sobre o Roberto, um homem de idade, obeso e com pressão alta, que teria sofrido um grave acidente de carro, Flávio iria ser contratado pelo hospital para cuidar do atendimento telefônico nas emergências.

No hospital das Clínicas, era clássico o uso de um **Bate-Papo** na Intranet, aberto na Internet para atendimento de pacientes. Os atores envolvidos neste cenário iriam descobrir, a partir da implementação de um SMA na Intranet do hospital, a existência de um novo colega, chamado "*AGENT*", sempre disposto a ajudar com seus conhecimentos. Em particular, Flávio iria se relacionar diretamente com "AGENT" para responder as chamadas telefônicas recebidas no setor de emergências.

4.2.2. Vocabulário da Sociedade "Home-care"

Para que o protótipo de SMA realizado nesta pesquisa possa se inserir hermeneuticamente na sociedade "Home-Care" descrita, o seguinte vocabulário foi definido

em termos de Dicionário, Base de Conceitos, Base de Dependências Conceituais, e Dicionário de Tradução DC-Inglês:

A	ABOUT	ACUTE	AGENT	AGENT'S	AN	ANALAB
ANALYSIS	ANESTHESIA	AGGRAVATION	AGGRAVATIONS	AIDS'	ALL	ARM
ARMS	ARM'S	ARMS'	AT	ATTACK	ATTACKS	BACK
BACKS	BACK'S	BACKS'	BATEL'S	BELIEVE	BELIEVES	BELIEVED
BLOOD	CAN	CHEST	CHESTS	CHEST'S	CHESTS'	CHIRURGICAL
CHRONIC	CLINICAS	COMPLETE	COMPLETES	COMPLETED	COMPLICATION	COMPLICATIONS
CONSTANT	CONTINUAL	COULD	DANI	DANI'S	DESCRIBE	DESCRIBED
DESCRIBES	DETECT	DETECTED	DETECTS	DIABETIC	DIAGNOSE	DIAGNOSED
DIAGNOSES	DID	DIFFICULTY	DIFFICULTIES	DISEASE	DISEASES	DISORDER
DISORDERS	DISTRICT	DISTRICTS	DO	DOES	DOMINIQUE	DOMINIQUE'S
ELBOW	ELBOWS	ELBOW'S	ELBOWS'	ENGLISH	EXAMINE	EXAMINED
EXAMINES	EXECUTE	EXECUTED	EXECUTES	EXHIBIT	EXHIBITED	EXHIBITS
EXHIBITED	EXHIBITS	FAZENDINHA'S	FILIPE	FILIPE'S	FLAVIO	FLAVIO'S
FOLLOW	FOLLOWED	FOLLOWS	FOR	FRACTURE	FRACTURES	FRENCH
FROM	FULFILL	FULFILLED	FULFILS	HAD	HAS	HAVE
HE	HEART	HEARTS	HEART'S	HEARTS'	HEMORRHAGE	HIGH
HIP	HIPS	HIP'S	HIPS'	HOSPITAL	HOSPITALS	HOW
I	IDENTIFY	IDENTIFIED	IDENTIFIES	IN	INDICATOR	INDICATORS
INFECTION	INFECTIONS	INJECTION	INJECTIONS	INSULIN	INSULIN'S	INTERVENTION
INTERVENTIONS	JACK	JACK'S	JEAN	JEAN'S	JOHN	JOHN'S
KNEE	KNEES	KNEE'S	KNEES'	KNEW	KNOW	KNOWS
LABORATORY	LABORATORIES	LANGUAGE	LANGUAGES	LEG	LEGS	LEG'S
LEGS'	LIVE	LIVED	LIVES	LOW	LUNG	LUNGS
LUNG'S	LUNGS'	MARCIO	MARCIO'S	MASSAGE	MASSAGES	ME
NECK	NECKS	NECK'S	NECKS'	NO	NOT	NOTICE
NOTICED	NOTICES	OBSERVE	OBSERVED	OBSERVES	ON	OPERATE
OPERATED	OPERATES	PARALYSIS	PEOPLE	PERFORATION	PERFORATIONS	PERFORM
PERFORMED	PERFORMS	PERSISTENT	PORTUGUESE	PRESCRIBE	PRESCRIBED	PRESCRIBES
PRESSURE	PRESSURES	PROPOSE	PROPOSED	PROPOSES	QUARTER	QUARTERS
RECOMMEND	RECOMMENDED	RECOMMENDS	REVEAL	REVEALED	REVEALS	ROBERTO
ROBERTO'S	RUPTURE	RUPTURES	SEVERE	SHE	SHOULDER	SHOULDERS
SHOULDER'S	SHOULDERS'	SHOW	SHOWED	SHOWS	SIGN	SIGNS
SOMEONE	STAY	STAYED	STAYS	STOMACH	STOMACHS	STOMACH'S
STOMACHS'	SUGGEST	SUGGESTED	SUGGESTS	SYMPTOM	SYMPTOMS	SYNDROME
SYNDROMES	TELL	THAT	THE	THEY	THINK	THINKS
THIS	THOUGHT	TO	TOLD	TRACK	TRACKED	TRACKS
TRANSFUSION	TRANSFUSIONS	TRUNK	TRUNKS	TRUNK'S	TRUNKS'	UNCEASING
UNENDING	US	VIRUS	VIRUSES	WE	WHAT	WHERE
WHICH	WHO	WITH	WORK	WORKED	WORKS	WRIST
WRISTS	WRIST'S	WRISTS'	YOU			

Tabela 4.2: Vocabulário Definido para o Cenário "*Home-Care*"

Deve-se observar, mais uma vez, que toda a experimentação foi conduzida em Inglês. A mesma experimentação pode ser realizada em outros idiomas, a partir do momento que é definido o mesmo vocabulário nestes outros idiomas.

4.2.3. Enações na Sociedade "Home-care"

São explicitadas a seguir, nas Figuras 4.10, 4.11, 4.12, 4.13, e 4.14, as enações semióticas realizadas no Simulador, como experimentação do uso de um mapeamento inteligente de competência em um cenário de cinco dias de um sistema de saúde "*Home-Care*". Para facilitar a compreensão, é detalhada de forma simplificada a conseqüência de cada enação sobre o estado do Mapeamento de Competências.

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-1: FILIPE, JACK, at the CLINICAS Hospital FILIPE > JACK: Hi Jack! Do you remember me? We studied together! JACK > FILIPE: Hi Filipe! Sure! What are you doing here? FILIPE > JACK: I work at the Clinicas Hospital now! JACK > FILIPE: Really? FILIPE > JACK: Yes. Monday was my first intervention. JACK > FILIPE: Tell me about that! FILIPE > JACK: There was a car accident. Roberto was the victim. Roberto showed a lung's perforation. I operated a chirurgical intervention on Roberto's lung. JACK > FILIPE: Who rescued him? FILIPE > JACK: John was sent. Roberto has a high pressure. Roberto had a heart's attack. Therefore John performed a heart's massage on Roberto. JACK > FILIPE: Yes John works at the Clinicas Hospital too. FILIPE > JACK: The intervention was complicated. Roberto had a blood's hemorrhage. I performed a blood's transfusion on Roberto. JACK > FILIPE: Who executed the blood's analysis? FILIPE > JACK: Dani executed the blood's analysis. JACK > FILIPE: Dani works at the ANALAB laboratory. Dani also performs the injections. I perform the injections too. FILIPE > JACK: Can Dani diagnose diseases? JACK > FILIPE: Yes Dani can identify the viruses. Who did perform the anesthesia on Roberto? FILIPE > JACK: Jean executed the anesthesia on Roberto. JACK > FILIPE: Ok. I need to go. Bye! FILIPE > JACK: Bye!	Filipe knows Jack. Jack knows Filipe Filipe works at Clinicas. Filipe knows Roberto. Jack knows Roberto. Roberto has a lung's perforation. Filipe operated a chirurgical intervention on Roberto. Filipe knows John. Jack knows John. Roberto has high pressure. Roberto had a heart's attack. John performed a heart's massage on Roberto. John works at Clinicas. Roberto had a blood's hemorrhage. Filipe performed a blood's transfusion on Roberto. Filipe knows Dani. Jack knows Dani. Dani executes blood's analysis. Dani works at ANALAB. Dani performs injections. Jack performs injections. Dani can identify viruses. Filipe knows Jean. Jack knows Jean. Jean executed the anesthesia on Roberto.

Figura 4.10: Primeiro Dia Simulado no Cenário "Home-Care"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-2: FILIPE, JEAN, at the CLINICAS Hospital FILIPE > JEAN: We put Roberto back home under home-care treatment. JEAN > FILIPE: Roberto had an arm's fracture. FILIPE > JEAN: Yes I prescribed an arms' massage to Roberto. Dominique executes a massage on Roberto's arms. JEAN > FILIPE : Right. Dominique lives at the Batel's quarter. FILIPE > JEAN: Right. I need to go. Bye! JEAN > FILIPE: Bye!	Jean knows Filipe. Jean knows Roberto. Roberto had an arm's fracture. Filipe prescribed an arm's massage to Roberto. Filipe knows Dominique. Jean knows Dominique. Dominique executes a massage on Roberto's arm. Dominique lives at Batel.

Figura 4.11: Segundo Dia Simulado no Cenário "Home-Care"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-3 : FLAVIO, JEAN, at the CLINICAS Hospital FLAVIO > JEAN: Hi. I am new here. JEAN > FLAVIO: Hi. Welcome. What's your job? FLAVIO > JEAN: I answer emergency call. JEAN > FLAVIO: Well good-luck! FLAVIO > JEAN: Thanks. Bye. JEAN > FLAVIO: Bye.	Flavio knows Jean. Jean knows Flavio.

Figura 4.12: Terceiro Dia Simulado no Cenário "Home-Care"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-4: FLAVIO answering the phone, at the CLINICAS Hospital FLAVIO > AGENT: Who can perform a heart's massage? AGENT > FLAVIO: I know the following about John: John can perform on a Roberto the heart's massage! If Flavio wants to reach John then Flavio can talk to Jean and Jean can talk to Filipe and Filipe can talk to John and the problem will be solved. I know the following about Dominique: Dominique can perform on the Roberto's arm a massage! If Flavio wants to reach Dominique then Flavio can talk to Jean and Jean can talk to Dominique and the problem will be solved. That is all. FLAVIO > AGENT: Who can execute an intervention? AGENT > FLAVIO: I know the following about Filipe: Filipe can perform on the Roberto's lung the chirurgical intervention! If Flavio wants to reach Filipe then Flavio can talk to Jean and Jean can talk to Filipe and the problem will be solved. That is all. FLAVIO > AGENT: Who can perform an anesthesia? AGENT > FLAVIO: I know the following about Jean: Jean can perform on a Roberto a anesthesia. If Flavio wants to reach Jean then Flavio can talk to Jean and the problem will be solved. That is all. FLAVIO > AGENT: Who can fulfill a blood's analysis? AGENT > FLAVIO: I know the following about Dani: Dani can perform the blood's analysis! If Flavio wants to reach Dani then Flavio can talk to Jean and Jean can talk to Filipe and Filipe can talk to Dani and the problem will be solved. That is all. FLAVIO > AGENT: Does Dani work at ANALAB? AGENT > FLAVIO: I know the following about Dani: Dani stays at the Analab District! If Flavio wants to reach Dani then Flavio can talk to Jean and Jean can talk to Filipe and Filipe can talk to Dani and the problem will be solved. That is all. FLAVIO > AGENT: Who can execute a transfusion? AGENT > FLAVIO: I know the following about Filipe: Filipe can perform on a Roberto the blood's transfusion! If Flavio wants to reach Filipe then Flavio can talk to Jean and Jean can talk to Filipe and the problem will be solved. That is all.	

Figura 4.13: Quarto Dia Simulado no Cenário "Home-Care"

Enações Semióticas Simuladas	Dissipação no Mapeamento
Day-5: Flacio, Marcio, in a Chat MARCIO > FLAVIO: Good afternoon. FLAVIO > MARCIO: Good afternoon Sir. How may I help you? MARCIO > FLAVIO: I had a leg's fracture. Who can perform a massage? AGENT > MARCIO: I know the following about John: John can perform on a Roberto the heart's massage! If Marcio wants to reach John then Marcio can talk to Flavio and Flavio can talk to Jean and Jean can talk to Filipe and Filipe can talk to John and the problem will be solved. I know the following about Dominique: Dominique can perform on the Roberto's arm a massage! If Marcio wants to reach Dominique then Marcio can talk to Flavio and Flavio can talk to Jean and Jean can talk to Dominique and the problem will be solved. That is all. FLAVIO > MARCIO: We will contact Dominique then. MARCIO > FLAVIO: Does Dominique work at Batel's district? AGENT > MARCIO: I know the following about Dominique: Dominique stays at the Batel's district! If Marcio wants to reach Dominique then Marcio can talk to Flavio and Flavio can talk to Jean and Jean can talk to Dominique and the problem will be solved. That is all. MARCIO > FLAVIO: Fine. She will then realize my treatment at home. I am a diabetic. Who can execute an insulin's injection on me? AGENT > MARCIO: I know the following about Jack and Dani: Dani can perform a injection! If Marcio wants to reach Dani then Marcio can talk to Flavio and Flavio can talk to Jean and Jean can talk to Filipe and Filipe can talk to Dani and the problem will be solved. If Marcio wants to reach Jack then Marcio can talk to Flavio and Flavio can talk to Jean and Jean can talk to Filipe and Filipe can talk to Jack and the problem will be solved. That is all. FLAVIO > MARCIO: We will contact Dani then. MARCIO > FLAVIO: Thank you Sir. Bye. FLAVIO > MARCIO: You are welcome. Bye.	Marcio knows Flavio. Flavio knows Marcio. Marcio had a leg's fracture. Marcio knows Dominique.

Figura 4.14: Quinto Dia Simulado no Cenário "Home-Care"

Pode-se observar novamente, neste experimento, como o SMA é capaz de se **inserir hermeneuticamente** de forma adequada nesta nova sociedade. Como no caso do primeiro experimento descrito, esta simulação, bem como infinitas variantes dela, podem ser realizadas a partir do simulador disponibilizado com este trabalho (a instalação e o uso do simulador são detalhados no apêndice A1).

4.2.4. Discussões

Através deste segundo experimento, pode-se verificar a tentativa de dar um caráter "universal" ao mapeamento inteligente, através do desempenho das operações de Neutralização Semiológica e Semiotização Natural. Apesar do contexto do segundo cenário ser muito diferente do contexto do primeiro, pode efetivamente se verificar que a atuação do "*AGENT*" exibe a mesma propriedade de **intencionalidade**, garantindo novamente sua inserção hermenêutica na sociedade de usuários humanos. Em particular, no quarto dia da

simulação, pode-se verificar como um agente virtual e um agente humano podem entrar em interação por meio de um diálogo direto, sendo as duas entidades autopoiéticas, neste exato momento**, a materialização da ponte enativa** criada entre a sociedade virtual e a sociedade real, conforme teoricamente proposto por Luhmann[110], no caso particular de sociedades apenas humanas.

Deve-se ressaltar, porém, que o desempenho do SMA neste segundo cenário exibe as mesmas fraquezas identificadas no experimento "Incubadora de Fábricas de Software". O mapeamento não é "completo", sendo construído por percepção fenomenológica. Ele também não é consistente, sendo a inexistência de um agente capaz de verificar tal consistência ou para-consistência. As operações de PLN, implementadas de forma simples, implicam a mesma ingenuidade do SMA, que percebe apenas frases perfeitamente corretas e extremamente explícita, oferecendo respostas estereotipadas. Como já notado, o SMA, não possuindo memória das dicas já emitida, pode ficar emitindo sempre as mesmas afirmações para as mesmas pessoas.

4.3. Expansão da Hermenêutica

A partir das discussões realizadas nas duas partes anteriores, será agora proposto uma série de considerações sobre a continuação que pode ser dada a este trabalho. Tais expansões da inserção hermenêutica do objeto tecnológico serão analisadas segundo quatro focos principais: expansão da hermenêutica para **outras sociedades autopoiéticas**, expansão das **capacidades de PLN** dos agentes, expansão das enações semióticas para **outras semioses**, e expansão do SMA para **outras tarefas**.

4.3.1. Expansão para outras Sociedades

Como pragmaticamente ilustrado com a escolha de tratar dois domínios cognitivos diferentes nos experimentos conduzidos para validar empiricamente esta pesquisa, o princípio de um mapeamento inteligente operacionalizado por uma sociedade multiagente de software pode ser expandido à uma infinidade de **outras sociedades autopoiéticas**. Para tal, é suficiente ser capaz de determinar a identidade topológica de tal nova organização, a partir da operação de **distinção** de Maturana e Varela, baseada na *minha* percepção fenomenológica do

[110] Ver Luhmann [PRO00], p. 23, citado p. 65

mundo. Isto significa também que o mapeamento inteligente apenas poderá ser expandido aos domínios cognitivos que ***posso reconhecer como tal***.

Alguns exemplos economicamente interessantes, no contexto da "era do conhecimento", seriam as sociedades: "Bolsa de Valores" (BOVESPA[111]), "Engenharia Genética", "Engenharia Química", "Agência Imobiliária", "Serviço de Atendimento a Clientes", entre muitos outros. A adaptação do protótipo construído nesta pesquisa para outras sociedades necessita apenas a criação, a partir do vocabulário-chave destas sociedades, de um Dicionário, uma Base de Conceitos, uma Base de Dependências Conceituais, e um Dicionário de Tradução DCs-Inglês.

Finalmente, a implementação do agente "**Trocador-WEB**", que não foi realizada no contexto do trabalho experimental desta pesquisa, permitiria ampliar a noção de enações entre sociedades de agentes, através do compartilhamento de informações semânticas entre várias sociedades, e vários SMAs, interagindo em várias LANs.

4.3.2. Expansão da Percepção Lingüística e Ontologias

Como discutido nas partes 4.1.4 e 4.2.4, os mecanismos de PLN implementados no protótipo são extremamente simples, e incompletos. Por este motivo, os agentes "Mapear-DCs" e "Simplificar-DCs" poderiam ser substituídos por outros agentes de ordem 2, encapsulando uma teoria mais complexa de PLN, que pudesse lidar com erros ortográficos e gramaticais, semelhanças fonéticas, bem como figuras de estilo (anti-frases, exagerações, metáforas, fórmulas irônicas, etc). Da mesma forma, o agente "Construir-Resposta" poderia, baseando-se nesta mesma teoria mais elaborada de PLN, demonstrar menos ingenuidade nas suas inserções semióticas. Os vários paradigmas de representação computacional da linguagem natural, abordados na parte 2.2.4 poderiam, por exemplo, ser objetos de um encapsulamento em novos agentes. Pode inclusive imaginar-se, seguindo a idéia de "**Sociedade da Mente**"[112], proposta por Minksy, que uma mesma expressão lingüística poderia ser o alvo de várias representações computacionais efetuadas por vários agentes, que poderiam finalmente entrar em **negociação**, para que emerja a interpretação a mais pragmaticamente interessante no contexto, do ponto de vista do SMA. Estas negociações internas do SMA, se aproximariam, segundo Minsky, da natureza mesmo da mente: "A tese

[111] Bolsa de Valores de São Paulo.
[112] Para mais detalhes sobre a natureza do paradigma das "Sociedades da Mente", ver [MIN91], e [MIN87].

da Sociedade da Mente suporta que a maioria dos *agentes* que crescem em nossos cérebros precisam apenas operar em uma escala tão pequena, que cada um, em si, não parece mais do que um brinquedo. Mas ao combinar alguns deles [...] é possível levá-los a realizar praticamente qualquer coisa". [MIN91], p. 13.

Além disto, a dimensão temporal das expressões lingüísticas percebidas pelo SMA poderia também ser levada sistematicamente em consideração (não só nos verbos conjugados, mas também nas datas as quais as expressões são formuladas), para tentar relativizar a credibilidade que pode ser dada a uma determinada competência mapeada. Tipicamente, se várias pessoas, em várias datas próximas do dia de hoje, afirmaram que uma pessoa possui uma determinada competência, a probabilidade para isto ser verdade é bem superior que se apenas uma pessoa fez esta mesma afirmação em uma época mais remota.

Finalmente, as competências mapeadas poderiam ser inter-relacionadas a partir da implementação das **ontologias** dos domínios cognitivos nos quais o SMA interage. Tais ontologias deveriam, neste caso, serem mantidas por agentes especializados nesta tarefa, completando o SMA proposto neste trabalho. Thomas Gruber, em [GRU91], ilustra como ontologias podem ser utilizadas para permitir o compartilhamento e o reuso do que qualifica de "Bases de Conhecimento". Gruber define portanto ontologias como "conjuntos coerentes de termos representacionais acoplados à definições textuais e formais, que incorporam um conjunto de escolhas representacionais." Deve-se ressaltar, conforme estudo epistemológico conduzido no capítulo 2, que uma "ontologia", sendo por essência Aristotélica, sempre constituíra uma fenomenologia particular, e, logo, nunca poderá afirmar-se a existência de **uma ontologia objetiva lingüística** capaz de resolver o problema da percepção. De um outro lado, desde a adoção de uma perspectiva fenomenológica transcendental da natureza do conhecimento, falar em "base de conhecimento" não poderá fazer sentido, até com a adoção da noção de ontologia do domínio: a ontologia continuará informação subjetiva mutuamente construída pelas interações de uma sociedade e seu meio.

4.3.3. Expansão para Outros Tipos de Interações Semióticas

Com os avanços tecnológicos da pesquisa em interações humano-computador, as sociedades de agentes virtuais, de forma geral, podem e, poderão cada vez mais, ampliar suas capacidades de enação com sociedades biológicas. Como ilustrado na Figura 4.15, se hoje, a maioria das interações que se realizam com as máquinas são de natureza inconfortáveis e

complexas (teclados, mouses, *touch-screens*, etc.), ferramentas de PLN, e reconhecimento de diálogo já estão fazendo sua aparição, transformando o modo de relacionamento que se pode ter com o digital. Aplicações de reconhecimento de texto manuscrito começam a ser também utilizadas comercialmente[113]. Em empresas de forte base tecnológica, onde a pesquisa realizada sobre interações humano-computador é transferida[114], experimentos conclusivos já estão sendo conduzidos nos campos do reconhecimento de movimentos e da expressão facial, dando nascimento à noção de "**Computação Cognitiva**". Estudos prospectivos, como se pode encontrar em [SIE03], anunciam também interações baseadas em atividade cerebral e emoções.

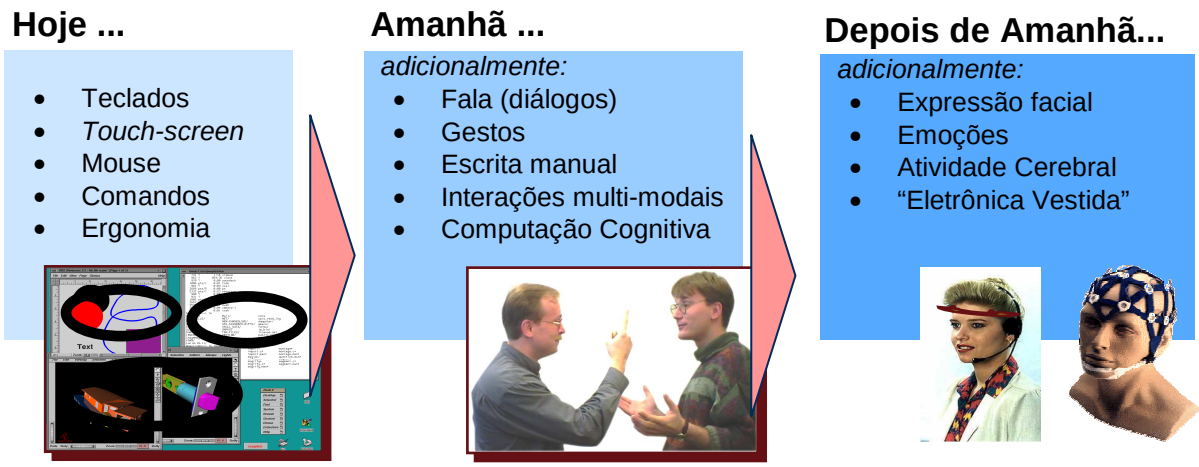


Figura 4.15: Imagens do Futuro – Interfaces Homem/Máquina. [SIE03]

Todos estes avanços contribuem à expansão das possibilidades de enações semióticas entre sociedades biológicas e sociedades digitais, diminuindo cada vez mais a diferença que se pode fazer entre estes dois tipos de sociedade. Além do avanço científico, tal perspectiva levanta obviamente vários questionamentos **éticos**, **morais** e **legais**.

Para o caso específico do protótipo implementado nesta pesquisa, é simples imaginar a inserção de um agente encarregado de transformar a fala em frases escritas em inglês (como existe já na maioria dos Softwares comerciais de tratamento de texto), para tornar o SMA proposto um "secretário eletrônico", encarregado de mapear informações a partir da sua participação em reuniões, em uma determinada organização. Além disto, o mesmo SMA

[113] Como é o caso para o reconhecimento de números escritos em cheques, ou CEPs para correspondências.
[114] IBM e Toshiba são exemplos clássicos.

ampliado desta forma poderia se inserir hermeneuticamente por meio de uma voz gerada sinteticamente.

4.3.4. Expansão para Outros Tipos de Tarefas

A partir das discussões dos experimentos descritos neste capítulo, pode também se propor uma expansão do trabalho experimental desta pesquisa para **outras tarefas complementares**, ou até objetivos diferentes do simples mapeamentos de competências.

Em primeiro lugar, a propriedade de **consistência** do mapeamento de informações realizado pelo SMA pode ser tomada como foco, com a criação de agentes encapsulando regras de verificação e descoberta de informações baseadas em **lógicas**, sejam elas de predicado, temporal, ou até mesmo paraconsistente. Mais uma vez, tal expansão torna-se trivial em um SMA, sendo que necessita apenas a incorporação de código computacional alopoiético em um processo autopoiético, como foi feito especificamente para as funções LISP de Schank, Birnbaum, Selfridge e Meehan no trabalho experimental detalhado neste capítulo. Outros agentes poderiam também, baseando-se em **redes neurais**, ter o objetivo de descobrir padrões nas informações mapeadas que não sejam necessariamente vinculados a um determinado tipo de lógica (tratar-se-ia, tipicamente, de encapsular código de *data-mining*[115] em um agente).

O protótipo de SMA objeto deste trabalho pode também facilmente ser ampliado a outras tarefas conexas como a classificação automática de textos ou a criação automática de taxonomias genéricas e personalizadas de informações. Poucas alterações seriam necessárias para tal: o agente "Construir-Resposta" deveria apenas ser substituído por um agente "**Construir-Taxonomia**", e um agente "**Classificar-Texto**" deveria ser acrescentado na sociedade. A partir destas alterações, a intencionalidade globalmente emergente do SMA não consistiria mais na geração de inserções semióticas contextualizadas, mas na **criação taxonomias e na classificação de documentos nestas taxonomias**.

De forma geral, a literatura atual mostra que sistemas multiagente estão cada vez mais confirmando sua presença no cenário da pesquisa em tecnologias de Gestão do Conhecimento e da Aprendizagem. Alguns exemplos recentes são [BAR02], onde Barthès e Tacla mostram como sociedades de agentes podem suportar o funcionamento de **portais de GC** em projetos complexos de Pesquisa & Desenvolvimento; [NAB02], onde Nabeth, Angehrn e Roda

propõem a noção de **sistemas ativos de GC**, capazes de providenciar socialmente informações personalizadas através do uso de agentes; [ROD03], onde os mesmos autores propõem o uso de sociedades de agentes para **fomentar o compartilhamento de conhecimento em comunidades de prática**; e [TAC02], onde Barthès e Tacla descrevem **uma arquitetura multiagente genérica** para a criação de sistemas de GC. Estes são apenas alguns dos inúmeros exemplos que podem ser encontrados na literatura voltada ao tema Tecnologia de IA para apoio à Gestão do Conhecimento.

Finalmente, um ramo de pesquisa extremamente interessante, que poderia ser aberto, consistiria na utilização de **algoritmos genéticos**[116] para construção automática de agentes via reprodução virtual, e seleção natural por *feed-back* da sociedade onde os agentes criados automaticamente são inseridos. Tipicamente, tais *feed-back* poderiam consistir na medição do tempo de interação com outros agentes (humanos ou não): se este tempo de interação é alto, significa que o agente é hermeneuticamente bem inserido, e logo, que ele deve transmitir seus genes virtuais para uma próxima geração; se, ao contrário, este tempo é baixo, pode-se inferir que a inserção do agente não é adequada, e que, logo, ele deve desaparecer sem transmitir sua sabedoria genética. Por seleção natural, durante gerações virtuais e reais, poder-se-ia imaginar que as sociedades de agentes evoluiriam em conjunto com as sociedades biológicas, tornando efetivamente a computação **autônoma e hermenêutica**, no sentido que um agente sempre deveria ser pensado no seu contexto histórico-cultural.

4.4. Conclusão

Este último capítulo ilustrou pragmaticamente, a partir de dois experimentos, as noções de **sociedade autopoiéticas** de agentes humanos e virtuais, e **inserção hermenêutica do objeto tecnológico** em uma meta-sociedade mista composta por pessoas e máquinas.

Os dois exemplos experimentais apresentados (Sociedade "Incubadora de Fábrica de Software", na parte 4.1, e "*Home-Care*", na parte 4.2) foram escolhidos devido à alta

---

[115] O *Data-mining*, ou Mineração de Dados, é uma técnica de IA que consiste em descobrir padrões escondidos em amplas bases de dados, a partir da comparação sistemática de eventos.

[116] Algoritmos genéticos, também chamados de algoritmos evolucionários, são o resultado do trabalho de pesquisa de John Koza (M.I.T.), que fez, em 1992, a proposta de aplicar a teoria Darwinista da seleção natural e as leis de Mendel, à programas de computador possuindo informações sobre o seu desempenho codificadas em uma molécula virtual de DNA. Os resultados desta simples idéia são extremamente impressionantes: com uma função de seleção (*fitness*) bem adaptada a um determinado problema, poucas gerações são suficientes para conseguir a descoberta automática de soluções contidas em espaços de busca gigantescos, sem necessidade de recorrer a muito conhecimento sobre o domínio, codificado, por exemplo, com heurísticas.

intensidade do uso do conhecimento nos cenários vinculados. As limitações constatadas nos experimentos, diretamente decorrentes das escolhas pragmáticas que foram feitas para focar a natureza deste trabalho, foram também apresentadas e discutidas, para levar a uma proposta de expansão desta pesquisa sob quatro perspectivas: expansão da hermenêutica para outras sociedades autopoiéticas, expansão das capacidades de PLN dos agentes, expansão das enações semióticas para outras semioses, e expansão do SMA para outras tarefas.

Desta forma, pode-se concluir que, em função do tipo de continuação que deseja ser dado a este trabalho, diferentes pesquisas podem ser encaminhadas: para tornar o SMA proposto **uma inovação tecnológica**, mecanismos de PLN mais avançados devem ser incorporados nos agentes encarregados de transformar sentenças em fórmulas computacionais; para **ampliar as capacidades de enação do SMA**, resultados de pesquisas em interações homem-máquina devem ser incorporadas ao agente *Interface*; para estender **as capacidades do SMA**, ramos de pesquisas interessantes de ser aproximado são a **lógica**, as **ontologias** e as **redes neurais artificiais**; finalmente, no que diz respeito **a naturalização da cognição**, desde uma abordagem hermenêutica do fenômeno cognitivo, o uso da teoria da **seleção natural**, através de **algoritmos genéticos**, para garantir a evolução automática da inserção histórico-cultural do artefato tecnológico, pode ser pesquisado.

Conclusão

Como foi anunciado na introdução deste trabalho, a pesquisa necessária à realização de mapeamentos inteligentes está inserida na intersecção de múltiplas disciplinas, como a epistemologia, a psicologia, a lingüística, a ciência da computação, a sociologia e a gestão. Por este motivo, as conclusões alcançadas ao termo deste trabalho **transcendem aspectos práticos e tecnológicos da informática**. Além de ter sustentado epistemologicamente a escolha da **Autopoiésis**, teoria construtivista biológica da cognição, como quadro adequado de trabalho para o objetivo de pesquisar o que são "**máquinas inteligentes**", e ter mostrado pragmaticamente, através de uma aplicação computacional, como **sistemas multiagente autopoiéticos** são efetivamente capazes de mapear de forma inteligente informações, para se inserir hermeneuticamente, como artefatos autopoiéticos, em uma determinada sociedade, esta pesquisa levou também a uma série de conclusões sobre a natureza da percepção e da construção da cognição.

Ao inverso da tendência das ciências cognitivas clássicas, consistindo em querer **mecanizar o humano**, conforme ilustrado historicamente de Descartes à Fodor, passando por Pascal, Leibnitz, Boole, Frege, Hilbert, Turing, Wiener, McCullogh & Pitts e Von Neumann, **este trabalho propõe humanizar a máquina**, através do resgate de uma visão hermenêutica da cognição, ultimamente descartada devido ao fato de enfatizar a subjetividade do fenômeno cognitivo, conforme proposto de Kant à Brooks, passando por Hegel, Heidegger, Husserl, Bergson, Searle, Maturana, Varella, Prigogine, Havelange, e Stewart. Pesquisas recentes em neurobiologia estão porém apontando que não existe, para os seres cognitivos, um mundo ontologicamente independente acessado de forma objetiva: **existe um mundo diferente para cada um**[117].

A cognição aparece, logo, como um fenômeno **emergente** que não pode ser separado do fenômeno social: não existe cognição sem sociedade, nem sociedade sem cognição, a cognição sendo o **cimento capaz de manter a coesão autopoiética de uma sociedade**. Sendo que, neste paradigma, o conhecimento é novamente enxergado como transcendental,

não faz mais sentido falar em “base de conhecimento”, ou “conhecimento explícito”. O lado físico do fenômeno cognitivo é a enação, da qual a informação digital é apenas uma expressão particular.

O fenômeno social é, ele, abordado neste paradigma como **fractal**. Uma sociedade se define, da mesma forma que um ser cognitivo, como **uma máquina autopoiética**. Desta forma, o indivíduo e a sociedade constituem o mesmo tipo fenômeno, observado à escalas diferentes. A noção de “**sociedade da mente**” de Minsky, abordando o cérebro como uma sociedade de agentes simples em interação, faz plenamente sentido na abordagem proposta neste trabalho. Uma outra conseqüência desta visão, é o fato que as noções de **cognição organizacional** e **memória organizacional** são plenamente justificadas, desde a natureza cognitiva de uma sociedade, e sendo que a meta-cognição da sociedade supera as cognições individuais dos agentes que a compõem.

Esta pesquisa leva também a uma revisão da noção de **retro-alimentação** (ou *feed-back*) para as máquinas cognitivas. O *feed-back* **corretivo** de Wiener, visando a ajustar o comportamento de um sistema para mantê-lo em uma determinada direção, é condenado ao fracasso, quando aplicado à sistemas autopoiéticos. Efetivamente, estes sistemas não são dirigidos pelos *inputs* que recebem para gerar certos *outputs*. Seu comportamento é, ao contrário, orientado por **processos dissipativos internos**, visando a eliminar toda perturbação indesejável. Por este motivo, uma segunda cibernética, a cibernética da **autonomia**, deve ser aplicada às máquinas autopoiéticas, onde não se usa mais *feed-backs* **corretivos**, mas sim *feed-back* **construtivos**. Um *feed-back* se torna construtivo, a partir do momento que foi incorporado as **regras autopoiéticas** garantindo a **intencionalidade** e a **identidade** de um ser cognitivo. Em outras palavras, é inútil aplicar um *feed-back* corretivo sobre uma pessoa ou uma sociedade: assim que esta ação desaparecer, a sociedade, ou o individuo, voltará ao seu natural autopoiético original. Para influenciar a autonomia de uma organização, deve, ao contrário, aplicar-se *feed-backs* **construtivos** que poderão (ou não) ser incorporados aos processos decisórios internos.

Esta pesquisa leva também à proposta de um novo tipo de atividade científica e tecnológica, visando a naturalizar a cognição: a **engenharia sócio-cognitiva**. A engenharia sócio-cognitiva pode ser definida como um campo de pesquisa multi-disciplinar, visando estudar e reproduzir as condições sociais a partir das quais o conhecimento pode emergir entre

[117] Ver “Um mundo diferente para cada um”, em [CAR02], p. 208 a 270.

agentes (biológicos ou não) de uma sociedade autopoiética [CAS04]. Nesta perspectiva, a Gestão do Conhecimento, visando criar vantagem competitiva nas organizações, beneficia-se muito mais de **redes sociais** do que qualquer forma de **explicitação *ad-hoc***. Um relatório, por si só, não tem valor: seu único valor está na sua capacidade de entrar em rede, e se transformar em um vínculo cognitivo entre um publicador e um leitor. Portanto, se o objetivo é melhorar a competitividade de uma organização, a diretriz-chave da GC deveria ser: **menos necessidade de explicitação, melhor**. Tecnologicamente falando, o programa de pesquisa da engenharia sócio-cognitiva visa em realizar **sociedades mistas** compostas por **pessoas** e **agentes tecnológicos**, onde enações naturais permitem fazer emergir uma semântica. O agente cognitivo de software é finalmente definido, no contexto da engenharia sócio-cognitiva, a partir de três características essenciais: **autonomia** (homeostase auto-referenciada), **intencionalidade emergente** (ações intencionais emergentes observadas sustentando a homeostase), e **identidade** topológica (manutenção emergente observada do espaço no qual a homeostase se realiza).

> «De um ponto de vista objetivista, a cognição é a representação subjetiva de uma realidade objetiva ontologicamente independente. De um ponto de vista construtivista, […] o sujeito e o objeto do conhecimento não são independentes, mas sim, mutuamente constitutivos. » [STE99], p.3.

Referências Bibliográficas

[AND54] Anderson, A. R. Minds and Machines. Prentice-Hall; 1954.

[ANN96] Annells, M. Hermeneutic phenemenology: Philosophical perspectives and current use in nursing research. Journal of Advanced Nursing, 23, 705-713; 1996.

[ARI90] Aristóteles. Leçons de Physique. Agora; 1990.

[BAC96] Bachimond, B. Hermeneutique materielle et Artefacture: des machines qui pensent aux machines qui donnent a penser. Thèse de Doctorat a l'école polytechnique en épistémologie ; 1996.

[BAC97a] Bachimond, B. Fondements des théories cognitives de l'esprit: La philosophie classique. UTC – France; 1997.

[BAC97b] Bachimond, B. Le problème de l'esprit en philosophie et recherches cognitives: une introduction. UTC – France; 1997

[BAR02] Barthès J. P., Tacla, C. Agent Supported Portals and Knowledge Management in Complex R&D Projects. Comnputers in Industry, vol. 48, no 1, pp 3-16; 2002.

[BOO54] Boole, G. Pesquisas sobre as leis do pensamento. 1854.

[CAR02] Carter, R. O Livro de Ouro da Mente. O Funcionamente e os Mistérios do Cérebro Humano. Ediouro Publicações S.A. ISBN 85-00-01296-X; 2002

[CAS03] Cassapo, F. M. O que entendemos exatamente por "conhecimento tácito" e "conhecimento explícito". SGBC. http://www.sbgc.org.br; 2003.

[CAS03b] Cassapo, F. M., Scalabrin, E. E.: Autopoietic Humans-and-Agents' Societies: a Hermeneutic Approach to Artificial Cognition. SCEF 2003 (1st International Conference on Socio-Cognitive Engineering Foundations). Rome, Itália; 2003.

[CAS03c] Cassapo, F., Scalabrin, E. E.: Sociedades de Agentes e Enação Semiótica: Um novo Caminho para as Tecnologias de Gestão do Conhecimento. KM BRASIL 2003. São Paulo; 2003.

[CAS04] Cassapo, F., Scalabrin, E: Autopoietic Societies: A Hermeneutic Approach to Socio-Cognitive Engineering.. 5th OKLC (5th European Conference on Organizacional Knowledge, Learning, and Capabilities). Innsbruck, Austria; 2004.

[CHO 57] Chomsky, N. Syntactic structures. The Hague, Mounton; 1957.

[CHO65] Chomsky, N. Aspects of the theory of syntax. Cambridge, Mass. MIT. Press; 1965.

[CRA95] Craig, I: Blackboard Systems. Ablex Publishing Corporation, New Jersey-USA; 1995.

[DAV98] Davenport, T. Working Knowledge: How Organizations Manage What They Know; 1998.

[DAV02] Davenport, T., Probst, G. Siemens Knowledge Management Case Book. 2002, 2nd edition. ISBN: 3-89578-181-9; 2002.

[DEC63] Descartes R. Œuvres philosophiques de Descartes, volume I : 1618-1637. Paris: Garnier. Edition critique établie par F. Alquié; 1963.

[DUP94] Dupuy J.P. Aux origines des sciences cognitives. Paris, La Découverte, 1994.

[DUR89] Durfee, E.H., Lesser, V.R. and Corkill, D.D. Trends in Cooperative Distributed Problem Solving. In: IEEE Transactions on Knowledge and Data Engineering KDE-1(1), pages 63-83; 1989.

[FER99] Ferber J., Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Pub Co; ISBN: 0201360489; 1999.

[FOD75] Fodor, J. A. The Language of Though. New York: Thomas Y. Crowell Company; 1975.

[FRA96] Franklin, S.; Graesser, A., Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents. Em European Conference on Artificial Intelligence - ECAI' 96, Budapest, Hungary; 1996.

[FRE84] Frege, G. Os fundamentos da aritmética; 1854.

[GLE97] Gleiser, M. A dança do universo: dos mitos da criação ao Big-Bang. 2nda edição, Companhia das Letras. ISBN 85-7164-677-5; 1997.

[GRA96] Graham, P. ANSI Common LISP, Prentice Hall, USA ; 1996.

[GRU91] Gruber, T. R. The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases. White Paper of the Knowledge System Laboratory of the Standford University; 1991.

[HAV95] Havelange, V. Article Critique: Jean-Pierre Dupuy, Aux origines des sciences cognitives. Paris, La Découverte. Intellectica, 1995/1, 20, pp. 247-261, 1994.

[HAV99] Havelange, V. Society and Human Action: Cognition or Interpretation. In Cognition, Biology, Technology: The Science and Philosophy of Emdodied Mind. UTC, IGC ; 1999.

[HAV99b] Havelange, V. Mémoire collective: la constitution technique da la cognition. In Cognition, Biology, Technology: The Science and Philosophy of Emdodied Mind. UTC, IGC; 1999.

[HAV99c] Havelange, V. Phenomenology and the Technological Genesis of Meaning. In Cognition, Biology, Technology: The Science and Philosophy of Emdodied Mind. UTC, IGC ; 1999.

[HAV03] Havelange, V., Lenay, C., e Stewart, J. Les représentations : mémoire externe et objets techniques. Intellectica 35; 2003.

[HAY95] Hayes-Roth B. An Architecture for Adaptive Intelligent Systems. Artificial Intelligence: Special Issue on Agents and Interactivity, 72, 329-365; 1995.

[HEG07] Hegel F. Fenomenologia na mente; 1807.

[HEI62] Heidegger, M. Being and time. New York: Harper; 1962 (Original work published 1927).

[HUS01] Husserl E. Logische Untersuchugen. Erster Band: Prolegomena zur reinen Logik; 1901.

[HUS59] Husserl E. Recherches Logiques. Epiméthée. Paris: Presses Universitaires de France. Logischen Untersuchungen, 1895, trad. Scherer, Elie et Kelkel, 1959.

[IDC99] IDC's third annual Email Usage Forecast and Analysis, 2001-2005. IDC #W25335; 2001.

[JEN98] Jennings, N., Sycara, K., Wooldrige, M. A roadmap of agent research and development. In Autonomous Agents and Multi-Agents systems vol. 1, n. 1 Kluwer Acaddemic Publishers; 1998.

[JOR00] Jorge, A. M. Guimarães. O que é Semiótica Peirceana? INTERCOM - GT de Semiótica; 2000.

[JUR00] Jufrasky, D., Martin, J. Speech and Language Processing. New Jersey, Prentice-Hall; 2000.

[KAN81] Kant, A crítica da razão pura, 1781.

[KVA96] Kvale, S. InterViews: An introduction to qualitative research. Thousand Oaks, CA: Sage; 1996.

[MAC43] McCulloch, W. S., Pitts, W. A logical Calculus of the ideas immanent in nervous activity, no jornal “Bulletin of Mathematical Biophysics”, Oxford, Elsevier Sciences, Vol. 5, p. 115–133; 1943.

[MAE95] Maes, P. Artificial Life Meets Entertainments: Life Like Autonomous Agents. In Communication of the ACM, 38, 11, 108-114; 1995.

[MAT76] Maturana H., Varela, F., Uribe, R. Autopoiesis: The Organization of Living Systems, its Characterization and a Model. In “Biosystems”, p. 187-196; 1976.

[MAT80] Maturana H., Varela, F. Autopoiesis & Cognition: the realization of the Living. Reidel, Dordrecht; 1980.

[MAT81] Maturana H. Autopoiesis. In M. Zeleny (Ed.) “Autopoiesis: a Theory of Living Organization”. Amsterdam: Elsevier North Holland ; 1981.

[MIN87] Minsky, M. The Society of Mind. Simon & Schuster, New York; 1987.

[MIN91] Minsky, M. Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy. In Artificial Intelligence at MIT., Expanding Frontiers, Patrick H. Winston (Ed.), Vol 1, MIT Press, 1990. Reprinted in AI Magazine; 1991.

[MOR85] Moreau J. Aristote et son École. Dito. Paris: Presses Universitaires de France; 1985.

[MOR94] Moreno A., Umerez J., e Fernandez J. Definition of Life and the Research Program in Artificial Life. In Ludus Vitalis, Vol II, num. 3; 1994.

[MUL99] Müller, J.P. Van Dyke Parunak, H. Multiagent Systems and Manufacturing. In Cognition, Biology, Technology: The Science and Philosophy of Emdodied Mind. UTC, IGC; 1999.

[NAB02] Nabeth T., Roda C., Angehrn A. Towards Personalized, Socially Aware and Active Knowledge Management System. In Proceeding: E-2002 e-Business and e-Work Annual Conference. Prague; 2002.

[NIJ88] Nijholt, A. Computers and languages – theory and practice. Amsterdam: Elsevier; 1988.

[NON95] Nonaka, I., Takeuchi, H. The Knowledge-Creating Company. New York, N.Y.: Oxford University Press; ISBN 0-19-509269-4; 1995.

[PAS86] Pascal, G. Les Grands Textes de la Philosophie. Bordas; 1986.

[PEI36] Peirce, C. S. Collected Papers of Charles S. Peirce. C. Hartshorne, P. Weiss (eds.), v. 1-6, e W. Burks (ed.), v. 7-8. Cambridge: Harvard University Press; 1931-58.

[PYL84] Pylyshyn Z. W. Computation and cognition: toward a foundation of cognitive science. MIT Press. Cambridge; 1984.

[PRO00] Provost J. R. W. Structure and Change in Complex Systems. Disponível na internet, em 22.05.00, no endereço http://www.geocities.com:0080/ñ4bz/Nonlihm.htm;

[RIV92] Rivelaygue J. Leçons de métaphysique allemande, Tome II: Kant, Heidegger, Habermas. Paris: Grasset, 1992.

[ROD03] Roda C., Angehrn A., Nabeth T., Razmerita, L. Using Converational Agents to Support the Adoption of Knowledge Sharing Practices. In Interacting with Computers, Special issue on Intelligence & Interaction in Community-based Systems; 2003.

[ROD71] Rodis-Lewis G.. L'œuvre de Descartes. A la recherche de la vérité. Paris: Librairie Philosophique Jean Vrin ; 1971.

[RUS95] Russell, S. J., Norvik, P. Artificial Intelligence: A Modern Approach, Prentice Hall, Englewood Cliffs, N.J.; 1995.

[SAN98] Santaella, L. A percepção: uma teoria semiótica. São Paulo: Experimento; 1998.

[SAU72] Saussure F. Cours de linguistique générale. Bibliothèque scientifique Payot. ISBN: 2-228-88165-1; 1972.

[SCA96] Scalabrin, E. E. Conception et réalistion d'environnement de développement de systèmes d'agents. Doctorate Degree Thesis at the "Université de Technologie de Compiègne" – France ; 1996.

[SCA99] Scalabrin, E. E. Da Inteligência Artificial à Inteligência Artificial Distribuída / Edson Emílio Scalabrin. Documento da Internet disponível em julho de 2000: http://www.ppgia.pucpr.br/~scalabrin/ftp/IAD/Aula01 52T.ppt

[SCH81] Schank, R. C., Riesbeck, C. K. Inside Computer Understanding. LEA Publishers; 1981.

[SCH82] Schank R. C. Dynamic Memory: A Theory of Learning in Computers and People, Cambridge University Press, Cambridge; 1982.

[SCH99] Schank R. C. Dynamic Memory Revisited. Cambridge University Press, Cambridge; 1999.

[SCI91] Sciences-illustrées no 9, p10-18: « Le chaos rentre dans l'ordre » ; 1991.

[SEA80] Searle, J. R. Minds, brains, and programs. Behavioral and Brain Sciences 3 (3): 417-457; 1980.

[SEN90] Senge, P. The fifth discipline: The art and the practice of the Learning Organization. New York: Doubleday/Currency; 1990.

[SHM99] Shmeil, M. A. H. Sistemas Multiagente na Modelação da Estrutura e Relações de Contratação de Organizações. Doctorate Degree Thesis at the "Faculdade de Engenharia Universidade do Porto" – Portugal; 1999.

[SIE03] Siemens'Picture of the Future. Documento da Internet disponível em 2002: http://www.siemens.com/index.jsp?sdc_p=t15cs6uo1032859d1031520pn1031318flm&sdc_sid=1847317262&

[SMI94] Smith D.C., Cypher, A., Spohrer, J. Kidsim: Programming Agents without a Programming Language. In Communication of the ACM, 37, 7, 55-57; 1994.

[STE99] Setwart, J. Cognition = Life: Implications for higher-level cognition. In Cognition, Biology, Technology: The Science and Philosophy of Emdodied Mind. UTC, IGC; 1999.

[SVE98] Sveiby, K. E.: The new organizational wealth. Editor Campus; 1998.

[TAC02] Tacla C. A., Barthès J. P. A Multi-agent Architecture for KM Systems. Second IEEE International Symposium on Advanced Distributed Computing Systems ISADS 2002. Guadalajara, Mexico; 2002.

[TOF90] Toffler, A. Powershift: Knowledge, Wealth and Violence at the Edge of the 21st Century, Batam Books, New York; 1990.

[TER02] Terra, J. C. C., Gordon C. Portais Corporativos de. ISBN: 8586014842. Negócio Editora; 2002.

[TER02b] Terra, J. C. C., Weiss, J. M. Guimarães. Rumo à "Sociedade do Conhecimento": As Trajetórias do Brasil e da Coréia do Sul. Em XXII Simpósio de Gestão da Inovação Tecnológica. Salvador. Bahia; 2002.

[TUR50] Turing, A. M. Computer Machinery and Intelligence; 1950.

[TUR95] Turing, A. M., Girard J.Y. La machine de Turing. Seuil, Paris ; 1995.

[VAR79] Varela, F. Principles of Biological Autonomy. New-York. North Holand. Autonomie et connaissance, trad. Paul Dumouchel et Paul Bourgine, Paris, Seuil ; 1989.

[VAR91] Varela, F., Thomson, E., Rosch, E. L'inscription corporelle de l'esprit, trad. Véronique Havelange, Paris, Seuil; 1993.

[VIE00] Vieira, R., Strube V. L. de Lima. Lingüística computacional: princípios e aplicações ; 2000.

[VIG99] Vigotsky, L. S. Pensamento e linguagem. São Paulo: Martins Fontes; 1999.

[WIE48] Winener, N. Cybernetics; 1948.

[WOO95] Wooldridge, M., Jennings, N. R. Agent Theories, Architectures, and Languages: A Survey. In: Intelligent Agents. (Eds: Wolldridge, Michael; Jennings, Nicholas R) Springer-Verlag, Berlin, 1-39; 1995.

Apêndice A

Instalação e Utilização do Protótipo

Este primeiro apêndice descreve como instalar e utilizar em um *desktop* o protótipo realizado no contexto desta pesquisa. A partir deste protótipo, os experimentos descritos nas partes 4.1 e 4.2 podem ser realizados, bem como qualquer outro tipo de experimento utilizando o vocabulário definido para estes mesmos experimentos.

Como detalhado na parte 4.3.1, o protótipo pode facilmente ser expandido para o mapeamento inteligente de competências em outras sociedades, bastando acrescentar o vocabulário-chave destas sociedades no Dicionário, na Base de Conceitos, na Base de Dependências Conceituais, e no Dicionário de Tradução DCs-Inglês. Os agentes deste protótipo podem também ser utilizados de forma modular em outras aplicações, e o SMA em si pode finalmente ser facilmente complementado por outros agentes, conforme detalhado nas partes 4.3.2 e 4.3.4.

A.1. Instalação e Desinstalação do Protótipo

Para instalar o protótipo do SMA implementado nesta pesquisa, basta utilizar o CD em anexo no final deste trabalho. Para que o protótipo funcione corretamente, os seguintes requisitos devem ser respeitados, para a máquina onde será efetuada a instalação:

- Sistema Operacional: Windows 95/98/2000/NT/XP
- Memória viva: >32 MB;
- Espaço disco necessário: 10MB;
- Processador: >500 Mhtz;
- Direitos Read/Write/Execute no Drive C:/';

Após ter introduzido o CD "*MAS Prototype*" no leitor de CD da máquina, executar o programa "*Setup.exe*". Este programa permitirá instalar o protótipo, acessar aos simuladores, acessar ao documento eletrônico contendo este trabalho, e acessar aos artigos que foram publicados para divulgar este trabalho. (Figura A1.1)

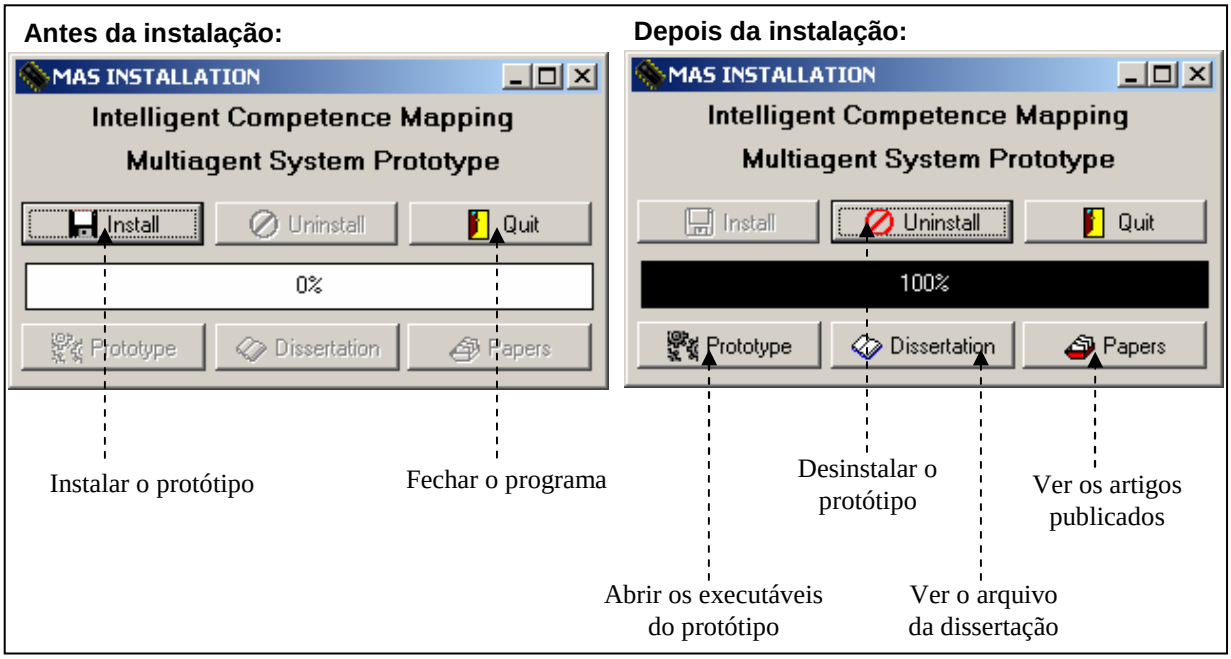


Figura A1.1: Instalação e Desinstalação do Protótipo.

Uma vez o programa "*Setup.exe*" em funcionamento:

- Clicar no botão "*Install*" para instalar o protótipo (todos os arquivos serão instalados na pasta C:/master/vf);
- Clicar no botão "*Uninstall*" para desinstalar o protótipo (todos os arquivos instalados serão removidos);
- Clicar em "*Prototype*" para abrir os executáveis do protótipo;
- Clicar em "*Dissertation*" para ver a versão eletrônica do trabalho de pesquisa;
- Clicar em "*Papers*" para ver os artigos publicados no contexto deste trabalho;
- Clicar em "*Quit*" para fechar o programa "*Setup.exe*";

Cabe observar que, após ter clicado em “Install”, uma janela de interrogação será aberta, pedindo informar o caminho do seu leitor de CD. É necessário digitar nesta janela tal caminho (por exemplo: “D:/”).

A.2 Utilização do Protótipo

O protótipo realizado pode ser utilizado a partir de dois executáveis: o programa “*Chat Simulator.exe*” (localizado na pasta C:/master/vf, após a instalação) é o simulador de bate-papo apresentado na parte 3.1.2, e o programa “*SMA Architecture Analyzer.exe*” é a “Interface de Acompanhamento das Enações de Ordem 1”, apresentada na parte 3.1.3. Para executar estes programas, basta dupla-clicar neles. Observamos que os dois programas não deveriam ser acionados simultaneamente, sendo que acessam aos mesmos arquivos, e poderiam, portanto, gerar conflitos e erros ao realizar estes acessos.

Para usar o “Simulador de Bate-Papo”, basta, conforme Figura 3.5, digitar no campo “*From*” o nome da pessoa emitindo uma mensagem, no campo “*To*”, o nome do remetente desta mensagem, no campo “*Content*”, o conteúdo da mensagem (escrito em Inglês, conforme detalhado no capítulo 4), e clicar finalmente em “*Send Message!*”. O botão “*Erase*” pode ser utilizado para apagar o conteúdo de “*Content*”. As enações lingüísticas realizadas durante o experimento podem finalmente ser apagadas clicando no botão “*Clear*”. O botão “*Close*” permite fechar a aplicação.

Para usar a “Interface de Acompanhamento das Enações de Ordem 1” (com o objetivo de clarificar as operações Neutralização Semiológica e Semiotização Natural, bem como verificar as enações dos agentes de ordem 1 do SMA), basta, conforme Figura 3.6, digitar no campo “*From*” o nome da pessoa emitindo uma mensagem, no campo “*To*”, o nome do remetente desta mensagem, no campo “*Content*”, o conteúdo da mensagem (escrito em Inglês, conforme detalhado no capítulo 4), e clicar finalmente em “*Send Message!*”. O botão “*Erase*” pode ser utilizado para apagar o conteúdo de “*Content*”. As enações lingüísticas geradas pelo agente *Interface* durante o experimento podem finalmente ser apagadas clicando no botão “*Clear*”. O campo “*Simulation Step*” permite regular o tempo de enação dos agentes de ordem 1. Por default, este campo é regulado a 1000 ms. Para acelerar as enações, basta diminuir este valor e clicar em “*Set!*”, para frear as mesmas, basta aumentar este valor, e clicar em “*Set!*”. As enações podem ser temporariamente suspendidas, clicando no botão

"*Pause*", e reiniciadas clicando novamente neste mesmo botão. O botão "*Close*" permite fechar a aplicação.

Um primeiro experimento simples que pode ser realizado, para facilitar o entendimento do protótipo, é o experimento "*I know the LISP language*" (Figura A2.1). Para tal, em "*From*", escrever "*Jack*", em "*To*", escrever "*John*" e em "*Content*", escrever "*I know the lisp language*." Clicar em "*Send Message*": o protótipo realiza uma neutralização semiológica. Pode-se verificar que o Mapeamento de Competências, a Base de Pessoas Relacionadas, e Base de Relações Pessoas-Competências são devidamente alteradas. Após esta alteração, em "*From*", escrever "*Filipe*", em "*To*", escrever "*Dani*" e em "*Content*", escrever "*Do you know a language?*" Clicar em "*Send Message*": o protótipo realiza uma neutralização semiológica, seguida de uma semiotização natural.

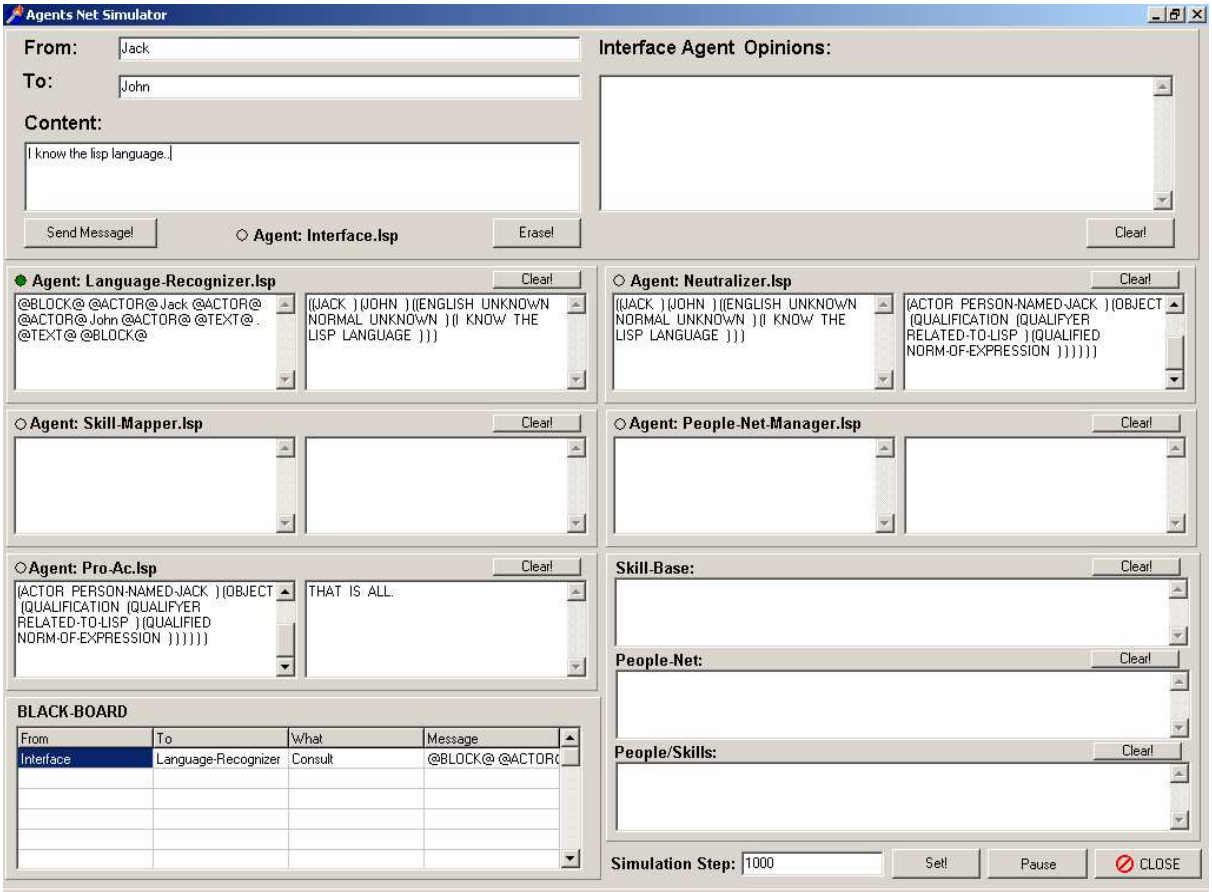


Figura A2.1: Experimento "I know the LISP language."

Cabe finalmente observar que cada uma das duas sociedades descritas nos experimentos do capítulo 4 possui seu próprio dicionário de tradução "DCs-Inglês". Por este

motivo, ao realizar o experimento “Incubadora de Fábricas de Software”, o arquivo “*English-CDs-Translation.lsp*” contido na pasta “C:\master\vf\objects\Software” deve ser copiado na pasta “C:\master\vf\objects”. Da mesma forma, ao realizar o experimento “*Home-Care*”, o arquivo “*English-CDs-Translation.lsp*” contido na pasta “C:\master\vf\objects\Home-care” deve ser copiado na pasta “C:\master\vf\objects”.

Apêndice B

Código Completo dos Agentes Protótipos

Este segundo apêndice possui o código completo dos agentes e dos objetos implementados no contexto desta pesquisa, de forma a completar, para o leitor que o desejar, o entendimento deste trabalho.

O código dos agentes, bem como dos objetos, segue o ANSI COMMON LISP[118], e pode facilmente ser reusado em outros contextos. Parte deste código (estrutura e tratamento das dependências conceituais) foi, inclusive, reaproveitado do resultado das pesquisas do Dr. Bill Andersen[119] do Department of Computer Science da University of Maryland.

[118] Para mais detalhes, consultar o ANSI COMMON LISP [GRA96].

[119] Bill Andersen (waander@cs.umd.edu), Department of Computer Science, University of Maryland, College Park, MD 20742. O código pode ser encontrado no URL http://www.sims.berkeley.edu/courses/is290-1/s02/MicroProgs/.

B.1. Código dos Agentes de Nível 1

Language-Recognizer.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; Language Recognizer ;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(require :file-manipulation)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;

(defvar *LIST-ARGS*)                 (setq *LIST-ARGS*               ())
(defvar *NAME-FILE-PERCEPTION*)  (setq *NAME-FILE-PERCEPTION*   'NOTHING)
(defvar *NAME-FILE-ACTION*)          (setq *NAME-FILE-ACTION*  'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(write-line "") (write-line "*** Iniciating Language Recognizer !!! ***") (write-line "")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;

(setq *LIST-ARGS*                (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*     (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*         (cadr *LIST-ARGS*))

;;;;;;;; Act ;;;;;;;;;;;;;;

(shell
        (concatenate 'string       "lisp.exe -M lispinit.mem //master/vf/agents/Agent-dialog-constructor.lsp "
                                   (string *NAME-FILE-PERCEPTION*)
                                   (string #\space)
                                   (string *NAME-FILE-ACTION*)))

(shell
        (concatenate 'string       "lisp.exe -M lispinit.mem //master/vf/agents/Agent-language-recognition.lsp "
                                   (string *NAME-FILE-ACTION*)
                                   (string #\space)
                                   (string *NAME-FILE-ACTION*)
                                   " //master/vf/objects/English-Full-Lexic.lxc"))

(write-line "") (write-line "*** Finalizing Language Recognizer !!! ***") (write-line "")
;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
```

Neutralizer.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;; Language Neutralizer ;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *LIST-ARGS*)                      (setq *LIST-ARGS*                  ())
(defvar *NAME-FILE-PERCEPTION*)   (setq *NAME-FILE-PERCEPTION*      'NOTHING)
(defvar *NAME-FILE-ACTION*)               (setq *NAME-FILE-ACTION*  'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "") (write-line "*** Iniciating Language Neutralizer!!! ***") (write-line "")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*     (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*  (cadr *LIST-ARGS*))

;;;;;;; Act ;;;;;;;;;;;;;;
(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-concept-neutralization.lsp "
                                       (string *NAME-FILE-PERCEPTION*)    (string #\space)
                                       (string *NAME-FILE-ACTION*)
                                       " //master/vf/objects/English-Mapping-Lexic.lxc"
                                       " //master/vf/objects/Concepts-Base.con"
                                       " //master/vf/objects/English-NLP-predictions.prd"))
(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-form-recognition.lsp "
                                       (string *NAME-FILE-ACTION*)          (string #\space)
                                       (string *NAME-FILE-ACTION*)
                                       " //master/vf/objects/Concepts-Base.con"))
(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-tense-recognition.lsp "
                                       (string *NAME-FILE-ACTION*)          (string #\space)
                                       (string *NAME-FILE-ACTION*)))

(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-person-concept-substitution.lsp "
                                       (string *NAME-FILE-ACTION*)          (string #\space)
                                       (string *NAME-FILE-ACTION*)
                                       " //master/vf/objects/NLP-Substitutions.sub"))
(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-unknown-concept-remotion.lsp "
                                       (string *NAME-FILE-ACTION*)          (string #\space)
                                       (string *NAME-FILE-ACTION*)))
(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-CD-recognition.lsp "
                                       (string *NAME-FILE-ACTION*)          (string #\space)
                                       (string *NAME-FILE-ACTION*)
                                       " //master/vf/objects/English-CDs-Dictionary.lsp"))
(shell    (concatenate 'string         "lisp.exe -M lispinit.mem //master/vf/agents/Agent-CD-simplification.lsp "
                                       (string *NAME-FILE-ACTION*)          (string #\space)
                                       (string *NAME-FILE-ACTION*)
                                       " //master/vf/objects/CDs-Simplifications.smp"))

(write-line "") (write-line "*** Finalizing Language Neutralizer !!! ***") (write-line "")

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
```

Skill-Mapper.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;; Skill Mapper Agent ;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;
(defvar *LIST-ARGS*)                    (setq *LIST-ARGS*             ())
(defvar *NAME-FILE-PERCEPTION*)   (setq *NAME-FILE-PERCEPTION*     'NOTHING)
(defvar *NAME-FILE-ACTION*)             (setq *NAME-FILE-ACTION*  'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "") (write-line "*** Iniciating Skill Mapper !!! ***") (write-line "")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                  (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*       (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*  (cadr *LIST-ARGS*))

;;;;;;;; Act ;;;;;;;;;;;;;;;
(shell    (concatenate 'string        "lisp.exe -M lispinit.mem //master/vf/agents/Agent-list-people-skill-construction.lsp "
                                       (string *NAME-FILE-PERCEPTION*)
                                       (string #\space)
                                       (string *NAME-FILE-ACTION*)))

(shell    (concatenate 'string        "lisp.exe -M lispinit.mem //master/vf/agents/Agent-list-people-skill-mapping.lsp "
                                       (string *NAME-FILE-ACTION*)
                                       " //master/vf/objects/Skill-Base.sb"
                                       " //master/vf/objects/People-Skill-Base.psb"
                                       " //master/vf/objects/PeopleNetwork.net"))

(write-line "") (write-line "*** Finalizing Skill Mapper !!! ***") (write-line "")

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
```

People-Net-Manager.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;; People Manager Agent ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *LIST-ARGS*)                    (setq *LIST-ARGS*                ())
(defvar *NAME-FILE-PERCEPTION*)         (setq *NAME-FILE-PERCEPTION*     'NOTHING)
(defvar *NAME-FILE-ACTION*)             (setq *NAME-FILE-ACTION*  'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "") (write-line "*** Iniciating People Net Manager !!! ***") (write-line "")

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*               (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*    (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*        (cadr *LIST-ARGS*))

;;;;;;;; Act ;;;;;;;;;;;;;;;

(shell
        (concatenate 'string    "lisp.exe -M lispinit.mem //master/vf/agents/Agent-people-net-construction.lsp "
                                (string *NAME-FILE-PERCEPTION*)
                                (string #\space)
                                (string *NAME-FILE-ACTION*)
                                " //master/vf/objects/Concepts-Base.con"
                                " //master/vf/objects/PeopleNetwork.net"))

(write-line "") (write-line "*** Finalizing People Net Manager !!! ***") (write-line "")

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
```

Pro-Ac.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;; Pro-Active Speaking Agent ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *LIST-ARGS*)                    (setq *LIST-ARGS*                ())
(defvar *NAME-FILE-PERCEPTION*)   (setq *NAME-FILE-PERCEPTION*    'NOTHING)
(defvar *NAME-FILE-ACTION*)             (setq *NAME-FILE-ACTION*  'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "") (write-line "*** Iniciating Pro-Active Searcher !!! ***") (write-line "")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                   (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*        (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*  (cadr *LIST-ARGS*))

;;;;;;;; Act ;;;;;;;;;;;;;;
(shell (concatenate 'string     "lisp.exe -M lispinit.mem //master/vf/agents/Agent-list-people-required-skill-construction.lsp "
                                (string *NAME-FILE-PERCEPTION*)
                                (string #\space)
                                (string *NAME-FILE-ACTION*)))
(shell (concatenate 'string     "lisp.exe -M lispinit.mem //master/vf/agents/Agent-list-matched-people-skill-construction.lsp "
                                (string *NAME-FILE-ACTION*)
                                (string #\space)
                                (string *NAME-FILE-ACTION*)
                                " //master/vf/objects/Skill-Base.sb"
                                " //master/vf/objects/People-Skill-Base.psb"))
(shell (concatenate 'string     "lisp.exe -M lispinit.mem //master/vf/agents/Agent-useless-matched-remotion.lsp "
                                (string *NAME-FILE-ACTION*)
                                (string #\space)
                                (string *NAME-FILE-ACTION*)
                                (string #\space)
                                (string *NAME-FILE-PERCEPTION*)))
(shell (concatenate 'string     "lisp.exe -M lispinit.mem //master/vf/agents/Agent-explanation-construction.lsp "
                                (string *NAME-FILE-ACTION*)
                                (string #\space)
                                (string *NAME-FILE-ACTION*)
                                " //master/vf/objects/English-Mapping-Lexic.lxc"
                                " //master/vf/objects/Concepts-Base.con"
                                " //master/vf/objects/PeopleNetwork.net "
                                (string *NAME-FILE-PERCEPTION*)
                                " //master/vf/objects/English-CDs-Translation.lsp"))

(write-line "") (write-line "*** Finalizing Pro-Active Searcher !!! ***") (write-line "")

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
```

B.2. Código dos Agentes de Nível 2

Agent-CD-recognition.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;; CD Recognition Agent ;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :CD-recognition)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-CD-DICTIONARY*)       (setq *NAME-FILE-CD-DICTIONARY* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                        (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*             (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*                 (cadr *LIST-ARGS*))
(setq *NAME-FILE-CD-DICTIONARY*          (caddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent CD Recognition !!! ***")

;;;;;;;;;;;; Read and set Dictionary in Memory ;;;;;;;;;;;;;;;;;
(load *NAME-FILE-CD-DICTIONARY*)

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-CD-dialog *TEXT*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Finalizing Agent CD Recognition !!! ***") (write-line "")
```

Agent-CD-simplification.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; CD Simplifcation Agent ;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :CD-simplification)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *LIST-SIMPLIFICATIONS*)          (setq *LIST-SIMPLIFICATIONS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-SIMPLIFICATION*)      (setq *NAME-FILE-SIMPLIFICATION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                        (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*             (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*                 (cadr *LIST-ARGS*))
(setq *NAME-FILE-SIMPLIFICATION*         (caddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent CD Simplification !!! ***")

;;;;;;;;;;; Read and set Simplifications in Memory ;;;;;;;;;;;;;;;;;
(setq *LIST-SIMPLIFICATIONS* (read-text-file *NAME-FILE-SIMPLIFICATION*))

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (simplify-CD-dialog *TEXT* *LIST-SIMPLIFICATIONS*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
;(write-line "'   *** Finalizing Agent CD Simplification !!! ***") (write-line "")
```

Agent-concept-neutralization.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; CONCEPTs Neutralization Agent ;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :concept-neutralization)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                              (setq *TEXT* ())
(defvar *LEXIC*)                             (setq *LEXIC* ())
(defvar *CONCEPT-BASE*)                      (setq *CONCEPT-BASE* ())
(defvar *PREDICTION*)                        (setq *PREDICTION* ())
(defvar *LIST-ARGS*)                         (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)              (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)                  (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-CONCEPT-BASE*)            (setq *NAME-FILE-CONCEPT-BASE* 'NOTHING)
(defvar *NAME-FILE-LEXIC*)                   (setq *NAME-FILE-LEXIC* 'NOTHING)
(defvar *NAME-FILE-PREDICTION*)              (setq *NAME-FILE-PREDICTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                 (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*      (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*          (cadr *LIST-ARGS*))
(setq *NAME-FILE-LEXIC*           (caddr *LIST-ARGS*))
(setq *NAME-FILE-CONCEPT-BASE*    (cadddr *LIST-ARGS*))
(setq *NAME-FILE-PREDICTION*      (cadddr (cdr *LIST-ARGS*)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent Concept Neutralization !!! ***")

;;;;;;;;;;;; Read and set Dictionary in Memory ;;;;;;;;;;;;;;;;;
(setq *LEXIC* (read-text-file *NAME-FILE-LEXIC*))
(setq *CONCEPT-BASE* (read-text-file *NAME-FILE-CONCEPT-BASE*))
(setq *PREDICTION* (read-text-file *NAME-FILE-PREDICTION*))

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (neutralize-concepts-dialog *TEXT* *LEXIC* *CONCEPT-BASE* *PREDICTION*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
;(write-line "'   *** Finalizing Agent Concepts Neutralization !!! ***") (write-line "")
```

Agent-dialog-constructor.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;; Dialog Constructor Agent ;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :dialog-construction)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                         (setq *TEXT* ())
(defvar *LIST-ARGS*)                    (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)         (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)             (setq *NAME-FILE-ACTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*     (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*         (cadr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "'   *** Iniciating Agent Dialog-Constructor!!! ***")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (make-dialog *TEXT*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
(write-line "'   *** Finalizing Agent Dialog-Constructor!!! ***") (write-line "")
```

Agent-explanation-construction.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Explanation Construction Agent ;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :result-explanation)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;
(defvar *SKILL-PEOPLELIST-LIST*)          (setq *SKILL-PEOPLELIST-LIST* ())
(defvar *DIALOG*)                         (setq *DIALOG* ())
(defvar *LIST-ARGS*)                      (setq *LIST-ARGS* ())
(defvar *LEXIC*)                          (setq *LEXIC* ())
(defvar *CONCEPT-BASE*)                   (setq *CONCEPT-BASE* ())
(defvar *PEOPLE-NET*)                     (setq *PEOPLE-NET* ())
(defvar *NAME-FILE-PERCEPTION*)           (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)               (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-DIALOG*)               (setq *NAME-FILE-DIALOG* 'NOTHING)
(defvar *NAME-FILE-LEXIC*)                (setq *NAME-FILE-LEXIC* 'NOTHING)
(defvar *NAME-FILE-CONCEPT-BASE*)         (setq *NAME-FILE-CONCEPT-BASE* 'NOTHING)
(defvar *NAME-FILE-PEOPLE-NET*)           (setq *NAME-FILE-PEOPLE-NET* 'NOTHING)
(defvar *NAME-FILE-TRANSLATION*)          (setq *NAME-FILE-TRANSLATION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                 (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*      (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*          (cadr *LIST-ARGS*))
(setq *NAME-FILE-LEXIC*           (caddr *LIST-ARGS*))
(setq *NAME-FILE-CONCEPT-BASE*    (cadddr *LIST-ARGS*))
(setq *NAME-FILE-PEOPLE-NET*      (cadddr (cdr *LIST-ARGS*)))
(setq *NAME-FILE-DIALOG*          (cadddr (cddr *LIST-ARGS*)))
(setq *NAME-FILE-TRANSLATION*     (cadddr (cdddr *LIST-ARGS*)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "'   *** Iniciating Agent Eplanation Construction !!! ***")

;;;;;;;;;;; Read and set Translation Dictionary in Memory ;;;;;;;;;;;;;;;;;
(load *NAME-FILE-TRANSLATION*)

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *SKILL-PEOPLELIST-LIST* (read-text-file *NAME-FILE-PERCEPTION*))
(setq *LEXIC* (read-text-file *NAME-FILE-LEXIC*))
(setq *CONCEPT-BASE* (read-text-file *NAME-FILE-CONCEPT-BASE*))
(setq *DIALOG* (read-text-file *NAME-FILE-DIALOG*))
(setq *PEOPLE-NET* (read-text-file *NAME-FILE-PEOPLE-NET*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *SKILL-PEOPLELIST-LIST*
         (tell-found-skills *SKILL-PEOPLELIST-LIST* *PEOPLE-NET* (cdr *LEXIC*) *CONCEPT-BASE* *DIALOG*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *SKILL-PEOPLELIST-LIST* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
(write-line "'   *** Finalizing Agent Explanation Construction !!! ***") (write-line "")
```

Agent-form-recognition.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;; Form Recognition Agent ;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :form-recognition)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *CONCEPT-BASE*)                  (setq *CONCEPT-BASE* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-CONCEPT-BASE*)        (setq *NAME-FILE-CONCEPT-BASE* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*     (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*         (cadr *LIST-ARGS*))
(setq *NAME-FILE-CONCEPT-BASE*   (caddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent Form Recognition !!! ***")

;;;;;;;;;;; Read and set Concept-base in Memory ;;;;;;;;;;;;;;;;;
(setq *CONCEPT-BASE* (read-text-file *NAME-FILE-CONCEPT-BASE*))

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-form-dialog *TEXT* *CONCEPT-BASE*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Finalizing Agent Form Recognition !!! ***") (write-line "")
```

Agent-language-recognition.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;; Language Recognition Agent ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :language-recognition)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *DICTIONARY*)                    (setq *DICTIONARY* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-DICTIONARY*)          (setq *NAME-FILE-DICTIONARY* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*     (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*         (cadr *LIST-ARGS*))
(setq *NAME-FILE-DICTIONARY*     (caddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "'   *** Iniciating Agent Language Recognition!!! ***")

;;;;;;;;;;; Read and set Dictionary in Memory ;;;;;;;;;;;;;;;;
(setq *DICTIONARY* (read-text-file *NAME-FILE-DICTIONARY*))

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-language-dialog *TEXT* *DICTIONARY*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
(write-line "'   *** Finalizing Agent Language Recognition !!! ***") (write-line "")
```

Agent-list-matched-people-skill-construction.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; List Matched People / Skill Construction Agent ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :list-people-skill-matched-construction)

;;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *PEOPLE-SKILL-LIST*)                (setq *PEOPLE-SKILL-LIST* ())
(defvar *SKILL-BASE*)                       (setq *SKILL-BASE* ())
(defvar *PEOPLE-SKILL-BASE*)                (setq *PEOPLE-SKILL-BASE* ())
(defvar *LIST-ARGS*)                        (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)             (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)                 (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-SKILL-BASE*)             (setq *NAME-FILE-SKILL-BASE* 'NOTHING)
(defvar *NAME-FILE-PEOPLE-SKILL-BASE*)      (setq *NAME-FILE-PEOPLE-SKILL-BASE* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                           (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*                (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*                    (cadr *LIST-ARGS*))
(setq *NAME-FILE-SKILL-BASE*                (caddr *LIST-ARGS*))
(setq *NAME-FILE-PEOPLE-SKILL-BASE*         (cadddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "   *** Iniciating Agent Matching People / skill List Conctruction !!! ***")

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *PEOPLE-SKILL-LIST* (read-text-file *NAME-FILE-PERCEPTION*))
(setq *SKILL-BASE* (read-text-file *NAME-FILE-SKILL-BASE*))
(setq *PEOPLE-SKILL-BASE* (read-text-file *NAME-FILE-PEOPLE-SKILL-BASE*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;
(setq *PEOPLE-SKILL-LIST* (make-list-matched *PEOPLE-SKILL-LIST* *SKILL-BASE* *PEOPLE-SKILL-BASE*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;
(write-text-file *PEOPLE-SKILL-LIST* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
(write-line "   *** Finalizing Agent Matching People / skill List Conctruction !!! ***") (write-line "")
```

Agent-list-people-required-skill-construction.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; List People / Required Skill Construction Agent ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :list-people-required-skill-construction)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                        (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*             (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*                 (cadr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "   *** Iniciating Agent People / required skill Mapping !!! ***")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-required-skill-dialog *TEXT*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
(write-line "   *** Finalizing Agent People / required skill Mapping !!! ***") (write-line "")
```

Agent-list-people-skill-construction.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;; List People / Skill Construction Agent ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :list-people-skill-construction)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                         (setq *TEXT* ())
(defvar *LIST-ARGS*)                    (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)         (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)             (setq *NAME-FILE-ACTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                       (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*            (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*                (cadr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "   *** Iniciating Agent People / skill Construction !!! ***")

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-skill-dialog *TEXT*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
(write-line "   *** Finalizing Agent People / skill Construction !!! ***") (write-line "")
```

Agent-list-people-skill-mapping.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; List People / Skill Mapping Agent ;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :people-skill-base-manipulation)
(require :list-people-knowing-people-skill-construction)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *PEOPLE-SKILL-LIST*)                (setq *PEOPLE-SKILL-LIST* ())
(defvar *SKILL-BASE*)                       (setq *SKILL-BASE* ())
(defvar *PEOPLE-SKILL-BASE*)                (setq *PEOPLE-SKILL-BASE* ())
(defvar *PEOPLE-NET*)                       (setq *PEOPLE-NET* ())
(defvar *LIST-ARGS*)                        (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)             (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-SKILL-BASE*)             (setq *NAME-FILE-SKILL-BASE* 'NOTHING)
(defvar *NAME-FILE-PEOPLE-SKILL-BASE*)      (setq *NAME-FILE-PEOPLE-SKILL-BASE* 'NOTHING)
(defvar *NAME-FILE-PEOPLE-NETWORK*)         (setq *NAME-FILE-PEOPLE-SKILL-BASE* 'NOTHING)
(defvar *NAME-FILE-PEOPLE-NET*)             (setq *NAME-FILE-PEOPLE-NET* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                           (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*                (car *LIST-ARGS*))
(setq *NAME-FILE-SKILL-BASE*                (cadr *LIST-ARGS*))
(setq *NAME-FILE-PEOPLE-SKILL-BASE*         (caddr *LIST-ARGS*))
(setq *NAME-FILE-PEOPLE-NET*                (cadddr *LIST-ARGS*))

(defun map-all-skills (L)
  (cond ((NULL L) NIL)
        (t  (setq *SKILL-BASE* (cdr (put-s (cdar L) *SKILL-BASE*)))
                              (setq *PEOPLE-SKILL-BASE* (cdr (put-ps  (cons (caar L)
                                      (cons (car (put-s (cdar L) *SKILL-BASE*)) NIL))    *PEOPLE-SKILL-BASE*)))
                             (map-all-skills (cdr L)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "'   *** Iniciating Agent People / skill Mapping !!! ***")

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *PEOPLE-SKILL-LIST* (read-text-file *NAME-FILE-PERCEPTION*))
(setq *SKILL-BASE* (read-text-file *NAME-FILE-SKILL-BASE*))
(setq *PEOPLE-SKILL-BASE* (read-text-file *NAME-FILE-PEOPLE-SKILL-BASE*))
(setq *PEOPLE-NET* (read-text-file *NAME-FILE-PEOPLE-NET*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(map-all-skills *PEOPLE-SKILL-LIST*)
(map-all-skills (set-knowing-people-skill *PEOPLE-NET*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *SKILL-BASE* *NAME-FILE-SKILL-BASE* 'supersede)
(write-text-file *PEOPLE-SKILL-BASE* *NAME-FILE-PEOPLE-SKILL-BASE* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
(write-line "'   *** Finalizing Agent People / skill Mapping !!! ***") (write-line "")
```

Agent-People-Net-construction.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; People-Net Construction Agent ;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :people-net-construction)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *PEOPLE-NET*)                    (setq *PEOPLE-NET* ())
(defvar *CONCEPT-BASE*)                  (setq *CONCEPT-BASE* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-PEOPLE-NET*)          (setq *NAME-FILE-PEOPLE-NET* 'NOTHING)
(defvar *NAME-FILE-CONCEPT-BASE*)        (setq *NAME-FILE-CONCEPT-BASE* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                 (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*      (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*          (cadr *LIST-ARGS*))
(setq *NAME-FILE-CONCEPT-BASE*    (caddr *LIST-ARGS*))
(setq *NAME-FILE-PEOPLE-NET*      (cadddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent People-Net Construction !!! ***")

;;;;;;;;;;; Read and set Lexic, Concept-base and PeopleNet in Memory ;;;;;;;;;;;;;;;;;
(setq *CONCEPT-BASE* (read-text-file *NAME-FILE-CONCEPT-BASE*))
(setq *PEOPLE-NET* (read-text-file *NAME-FILE-PEOPLE-NET*))

;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-people-dialog *TEXT* *PEOPLE-NET* *CONCEPT-BASE*))

;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;
;(write-line "'   *** Finalizing Agent People-Net-Construction !!! ***") (write-line "")
```

Agent-person-concept-substitution.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Person / Concept Substitution Agent ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :person-concept-substitution)

;;;;;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *SUBSTITUTION*)                  (setq *SUBSTITUTION* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-SUBSTITUTION*)        (setq *NAME-FILE-SUBSTITUTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS* (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*       (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*  (cadr *LIST-ARGS*))
(setq *NAME-FILE-SUBSTITUTION*     (caddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent Person/Concept Substitution !!! ***")

;;;;;;;;;;;;;; Read and set Dictionary in Memory ;;;;;;;;;;;;;;;;;;;;;
(setq *SUBSTITUTION* (read-text-file *NAME-FILE-SUBSTITUTION*))

;;;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;;;;
(setq *TEXT* (substitute-concept-dialog *TEXT* *SUBSTITUTION*))

;;;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;;;;
```

Agent-tense-recognition.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;; Tense Recognition Agent ;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :tense-recognition)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*                (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*     (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*         (cadr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line ""   *** Iniciating Agent Tense Recognition !!! ***")

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (set-tense-dialog *TEXT*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
;(write-line ""   *** Finalizing Agent Tense Recognition !!! ***") (write-line "")
```

Agent-unknown-concept-remotion.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Unknown Concept Remotion Agent ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :unknown-concept-remotion)

;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;
(defvar *TEXT*)                          (setq *TEXT* ())
(defvar *LIST-ARGS*)                     (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)          (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)              (setq *NAME-FILE-ACTION* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS*              (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*   (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*       (cadr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Iniciating Agent Unknown concept Remotion !!! ***")

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (read-text-file *NAME-FILE-PERCEPTION*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;
(setq *TEXT* (remove-unknown-concept-dialog *TEXT*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;
(write-text-file *TEXT* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
;(write-line "'   *** Finalizing Agent Unknown concept Remotion !!! ***") (write-line "")
```

Agent-useless-matched-remotion.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;; People-Net Construction Agent ;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(require :file-manipulation)
(require :useless-matched-remotion)

;;;;;;;;;;;;;;;;; DEFINITION OF GLOBAL VARIABLES ;;;;;;;;;;;;;;;;;;;;;;;;
(defvar *SKILL-PEOPLELIST-LIST*)           (setq *SKILL-PEOPLELIST-LIST* ())
(defvar *DIALOG*)                          (setq *DIALOG* ())
(defvar *LIST-ARGS*)                       (setq *LIST-ARGS* ())
(defvar *NAME-FILE-PERCEPTION*)            (setq *NAME-FILE-PERCEPTION* 'NOTHING)
(defvar *NAME-FILE-ACTION*)                (setq *NAME-FILE-ACTION* 'NOTHING)
(defvar *NAME-FILE-DIALOG*)                (setq *NAME-FILE-DIALOG* 'NOTHING)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; INITIATE AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(setq *LIST-ARGS* (liststring-to-list *ARGS*))
(setq *NAME-FILE-PERCEPTION*      (car *LIST-ARGS*))
(setq *NAME-FILE-ACTION*  (cadr *LIST-ARGS*))
(setq *NAME-FILE-DIALOG*  (caddr *LIST-ARGS*))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; START AGENT ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(write-line "   *** Iniciating Agent Useless Matched Remotion !!! ***")

;;;;;;;;;;;; Feel ;;;;;;;;;;;;;;;;;;
(setq *SKILL-PEOPLELIST-LIST* (read-text-file *NAME-FILE-PERCEPTION*))
(setq *DIALOG* (read-text-file *NAME-FILE-DIALOG*))

;;;;;;;;;;;; Think ;;;;;;;;;;;;;;;;;;
(setq *SKILL-PEOPLELIST-LIST* (remove-useless-matched *SKILL-PEOPLELIST-LIST* *DIALOG*))

;;;;;;;;;;;; Act ;;;;;;;;;;;;;;;;;;
(write-text-file *SKILL-PEOPLELIST-LIST* *NAME-FILE-ACTION* 'supersede)

;;;;;;;;;;;; End ;;;;;;;;;;;;;;;;;;
(write-line "   *** Finalizing Agent Useless Matched Remotion !!! ***") (write-line "")
```

B.3. Código dos Artefatos Manipulados pelos Agentes

Concepts-Base.con

```
(*START-SENTENCE* POSITIVE AFFIRMATION)
(*UNKNOWN-WORD* POSITIVE AFFIRMATION)
(ABOUT-SOMETHING POSITIVE POSITION)
(ANESTHESIA POSITIVE WOBJECT)
(ARM POSITIVE WOBJECT)
(ATTACK POSITIVE WOBJECT)
(BACK POSITIVE WOBJECT)
(BOOK POSITIVE WOBJECT)
(BLOOD POSITIVE WOBJECT)
(BLOOD-ESCAPE POSITIVE WOBJECT)
(BLOOD-TRANSFUSION POSITIVE WOBJECT)
(CAN-INTERROGATION POSITIVE INTERROGATION)
(CHANGABLE-VALUE POSITIVE WOBJECT)
(CHEST POSITIVE WOBJECT)
(COMPANIES-GROUPING-SYSTEM POSITIVE WOBJECT)
(DISORDER-AGGRAVATION POSITIVE WOBJECT)
(DO-INTERROGATION POSITIVE INTERROGATION)
(ELBOW POSITIVE WOBJECT)
(ERROR POSITIVE WOBJECT)
(FOOT POSITIVE WOBJECT)
(FOR-THIS-REASON POSITIVE POSITION)
(FROM-SOMETHING POSITIVE POSITION)
(HABILITY-RECOGNIZING-DOCUMENT POSITIVE WOBJECT)
(HEART POSITIVE WOBJECT)
(HIP POSITIVE WOBJECT)
(HOLE POSITIVE WOBJECT)
(HOW-INTERROGATION POSITIVE INTERROGATION)
(IDEAL-REPRESENTATION-OF-REALITY POSITIVE WOBJECT)
(IN-THIS-DIRECTION POSITIVE DIRECTION)
(INSIDE-SOMETHING POSITIVE POSITION)
(INSULIN POSITIVE WOBJECT)
(KNEE POSITIVE WOBJECT)
(LEG POSITIVE WOBJECT)
(LIST-OF-PROGRAMMED-STEPS POSITIVE WOBJECT)
(LIST-OF-SENTENCE POSITIVE WOBJECT)
(LUNG POSITIVE WOBJECT)
(MASSAGE POSITIVE WOBJECT)
(MEASURABLE-VALUE POSITIVE WOBJECT)
(MOVING-IMPOSSIBILITY POSITIVE WOBJECT)
(MY-SELF POSITIVE PERSON)
(NECK POSITIVE WOBJECT)
(NEGATION NEGATIVE AFFIRMATION)
(NON-CONFORMITY POSITIVE WOBJECT)
(NORM-OF-EXPRESSION POSITIVE WOBJECT)
(ON-SOMETHING POSITIVE POSITION)
(ORGANIZED-GROUP-OF-PEOPLE POSITIVE WOBJECT)
(OUR-SELVES POSITIVE PERSON)
(PERSON-NAMED-DANI POSITIVE PERSON)
(PERSON-NAMED-DOMINIQUE POSITIVE PERSON)
(PERSON-NAMED-FILIPE POSITIVE PERSON)
(PERSON-NAMED-FLAVIO POSITIVE PERSON)
(PERSON-NAMED-JACK POSITIVE PERSON)
```

(PERSON-NAMED-JEAN POSITIVE PERSON)
(PERSON-NAMED-JOHN POSITIVE PERSON)
(PERSON-NAMED-MARCIO POSITIVE PERSON)
(PERSON-NAMED-ROBERTO POSITIVE PERSON)
(PHYSICAL-CRACK POSITIVE WOBJECT)
(PHYSICAL-DIFFICULTY POSITIVE WOBJECT)
(PHYSICAL-DISORDER POSITIVE WOBJECT)
(PHYSICAL-DISORDER-SIGN POSITIVE WOBJECT)
(PHYSICAL-INJECTION POSITIVE WOBJECT)
(PHYSICAL-OPERATION POSITIVE WOBJECT)
(PLACE-TO-CURE POSITIVE WOBJECT)
(PLACE-TO-STORE-THINGS POSITIVE WOBJECT)
(PLACE-TO-LIVE POSITIVE WOBJECT)
(POINTER-A POSITIVE POINTER)
(POINTER-THE POSITIVE POINTER)
(POINTER-THIS POSITIVE POINTER)
(POINTER-THAT POSITIVE POINTER)
(PRESSURE POSITIVE WOBJECT)
(PROCEDURE-FOR-AUDITING POSITIVE WOBJECT)
(PROCEDURE-FOR-TESTING POSITIVE WOBJECT)
(REALIZATION-OF-SOMETHING POSITIVE WOBJECT)
(RELATED-TO-ACUTE POSITIVE QUALIFICATION)
(RELATED-TO-AGENT POSITIVE QUALIFICATION)
(RELATED-TO-AIDS POSITIVE QUALIFICATION)
(RELATED-TO-ANALAB POSITIVE QUALIFICATION)
(RELATED-TO-ARM POSITIVE QUALIFICATION)
(RELATED-TO-ASP POSITIVE QUALIFICATION)
(RELATED-TO-BACK POSITIVE QUALIFICATION)
(RELATED-TO-BATEL POSITIVE QUALIFICATION)
(RELATED-TO-BLOOD POSITIVE QUALIFICATION)
(RELATED-TO-C++ POSITIVE QUALIFICATION)
(RELATED-TO-CHEST POSITIVE QUALIFICATION)
(RELATED-TO-CHIRURGIC POSITIVE QUALIFICATION)
(RELATED-TO-CLINICAS POSITIVE QUALIFICATION)
(RELATED-TO-CMM POSITIVE QUALIFICATION)
(RELATED-TO-DANI POSITIVE QUALIFICATION)
(RELATED-TO-DIABETIC POSITIVE QUALIFICATION)
(RELATED-TO-DOMINIQUE POSITIVE QUALIFICATION)
(RELATED-TO-ELBOW POSITIVE QUALIFICATION)
(RELATED-TO-ENGLISH POSITIVE QUALIFICATION)
(RELATED-TO-EVERYTHING POSITIVE QUALIFICATION)
(RELATED-TO-EXACT POSITIVE QUALIFICATION)
(RELATED-TO-FABSOFTA POSITIVE QUALIFICATION)
(RELATED-TO-FABSOFTB POSITIVE QUALIFICATION)
(RELATED-TO-FABSOFTC POSITIVE QUALIFICATION)
(RELATED-TO-FAZENDINHA POSITIVE QUALIFICATION)
(RELATED-TO-FOOT POSITIVE QUALIFICATION)
(RELATED-TO-FILIPE POSITIVE QUALIFICATION)
(RELATED-TO-FLAVIO POSITIVE QUALIFICATION)
(RELATED-TO-FRENCH POSITIVE QUALIFICATION)
(RELATED-TO-HEART POSITIVE QUALIFICATION)
(RELATED-TO-HIGH POSITIVE QUALIFICATION)
(RELATED-TO-HIP POSITIVE QUALIFICATION)
(RELATED-TO-INCFABSOFT POSITIVE QUALIFICATION)
(RELATED-TO-INSULIN POSITIVE QUALIFICATION)

(RELATED-TO-ISO-14000 POSITIVE QUALIFICATION)
(RELATED-TO-ISO-9000 POSITIVE QUALIFICATION)
(RELATED-TO-ISO-9001 POSITIVE QUALIFICATION)
(RELATED-TO-JACK POSITIVE QUALIFICATION)
(RELATED-TO-JEAN POSITIVE QUALIFICATION)
(RELATED-TO-JOHN POSITIVE QUALIFICATION)
(RELATED-TO-KNEE POSITIVE QUALIFICATION)
(RELATED-TO-LEG POSITIVE QUALIFICATION)
(RELATED-TO-LISP POSITIVE QUALIFICATION)
(RELATED-TO-LOW POSITIVE QUALIFICATION)
(RELATED-TO-LUNG POSITIVE QUALIFICATION)
(RELATED-TO-MARCIO POSITIVE QUALIFICATION)
(RELATED-TO-NECK POSITIVE QUALIFICATION)
(RELATED-TO-PASCAL POSITIVE QUALIFICATION)
(RELATED-TO-PDCA POSITIVE QUALIFICATION)
(RELATED-TO-PERSISTENT POSITIVE QUALIFICATION)
(RELATED-TO-PORTUGUESE POSITIVE QUALIFICATION)
(RELATED-TO-PROGRAMMING POSITIVE QUALIFICATION)
(RELATED-TO-ROBERTO POSITIVE QUALIFICATION)
(RELATED-TO-SANTA-FELICIDADE POSITIVE QUALIFICATION)
(RELATED-TO-SEVERE POSITIVE QUALIFICATION)
(RELATED-TO-SHOULDER POSITIVE QUALIFICATION)
(RELATED-TO-STOMACH POSITIVE QUALIFICATION)
(RELATED-TO-TQM POSITIVE QUALIFICATION)
(RELATED-TO-VISUAL-BASIC POSITIVE QUALIFICATION)
(RELATED-TO-WRIST POSITIVE QUALIFICATION)
(RELATED-TO-WRONG POSITIVE QUALIFICATION)
(RELATIVE-HOW POSITIVE RELATIVE)
(RELATIVE-THAT POSITIVE RELATIVE)
(RELATIVE-WHAT POSITIVE RELATIVE)
(RELATIVE-WHO POSITIVE RELATIVE)
(SHOULDER POSITIVE WOBJECT)
(STEPS-FOR-TEACHING-LEARNING POSITIVE WOBJECT)
(STOMACH POSITIVE WOBJECT)
(THESE-PEOPLE POSITIVE PERSON)
(THIS-MAN POSITIVE PERSON)
(THIS-WOMAN POSITIVE PERSON)
(TO-BE-ABLE-TO POSITIVE ACTION)
(TO-CORRECT POSITIVE ACTION)
(TO-DISCLOSE-KNOWLEDGE POSITIVE ACTION)
(TO-DO POSITIVE ACTION)
(TO-FIND-SOLUTION POSITIVE ACTION)
(TO-GIVE-KNOWLEDGE POSITIVE ACTION)
(TO-HAVE-BELIEF POSITIVE ACTION)
(TO-INITIALIZE POSITIVE ACTION)
(TO-KNOW POSITIVE ACTION)
(TO-MAKE-AUDIT POSITIVE ACTION)
(TO-MAKE-CREATION POSITIVE ACTION)
(TO-MAKE-DESCRIPTION POSITIVE ACTION)
(TO-MAKE-DIAGNOSTIC POSITIVE ACTION)
(TO-MAKE-EVALUATION POSITIVE ACTION)
(TO-MAKE-LIST-OF-PROGRAMMED-STEPS POSITIVE ACTION)
(TO-MAKE-PRESCRIBTION POSITIVE ACTION)
(TO-PERFORM POSITIVE ACTION)
(TO-POSSESS POSITIVE ACTION)

(TO-PRE-VERB POSITIVE ACTION)
(TO-READ POSITIVE ACTION)
(TO-STAY-AT POSITIVE ACTION)
(TO-TELL-CASE POSITIVE ACTION)
(TO-TEST POSITIVE ACTION)
(TO-THINK POSITIVE ACTION)
(TO-TRACK POSITIVE ACTION)
(TO-UNDERSTAND POSITIVE ACTION)
(TRIAL-EXERCISE POSITIVE WOBJECT)
(UNDEFINED-BENCH-OF-PERSON POSITIVE PERSON)
(UNDEFINED-PERSON POSITIVE PERSON)
(VISION-OF-WORLD POSITIVE WOBJECT)
(WHAT-INTERROGATION POSITIVE AFFIRMATION)
(WHO-INTERROGATION POSITIVE INTERROGATION)
(WITH-SOMETHING POSITIVE POSITION)
(WRIST POSITIVE WOBJECT)
(YES POSITIVE AFFIRMATION)
(YOUR-SELF POSITIVE PERSON)

English-CDs-Dictionary.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; CDs Translations of Naural English ;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; Basic CD Structure ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;(defword XXXXXX
;   (
;        (test a-test)
;        (assign some-assigments)
;        (next-packet
;             (
;                  (test a-test)
;                  (assign some-assigments)
;                  (next-packet
;                       (
;                       ...
;                       )
;             )
;             (
;                  (test a-test)
;                  (assign some-assigments)
;                  (next-packet
;                       (...)
;             )
;             (...)
;        )
;   )
;
;  (
;        (test a-test)
;        (assign some-assigments)
;        (next-packet
;             (
;                  (test a-test)
;                  (assign some-assigments)
;                  (next-packet
;                       (...)
;             )
;             (
;                  (test a-test)
;                  (assign some-assigments)
;                  (next-packet
;                       (...)
;             )
;             (...)
;        )
;   )
;
;   (
;   ...
;   )
;
;)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define action concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;; ATRANS ;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;; PTRANS ;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;; EXPEL;;;;;;;;;;;;;;;;;;;;;

(defword TO-POSSESS
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action)
                (eq *part-of-speech* 'relative) (eq *part-of-speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign   *part-of-speech* 'action
                        *cd-form* '(expel    (action-details ?possess-var0)
                                             (actor  ?possess-var1)
                                             (object ?possess-var2)
                                             (from ?possess-var3)
                                             (to ?possess-var4))
                        possess-var0 *word*
                        possess-var1 *subject*
                        possess-var2 *object-complement*
                        possess-var3 nil
                        possess-var4 nil)
        (next-packet
                (
                        (assign possess-var2 *cd-form*)
                )
        )
  )
)

;;;;;;;;;;;;;;;;;; PROPEL ;;;;;;;;;;;;;;;;;;;;

(defword TO-PERFORM
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action)
                (eq *part-of-speech* 'relative) (eq *part-of-speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign   *part-of-speech* 'action
                        *cd-form* '(propel   (action-details ?TO-MAKE-EXECUTION-var0)
                                             (actor  ?TO-MAKE-EXECUTION-var1)
                                             (object ?TO-MAKE-EXECUTION-var2)
                                             (from ?TO-MAKE-EXECUTION-var4)
                                             (to ?TO-MAKE-EXECUTION-var3))
                        TO-MAKE-EXECUTION-var0 *word*
                        TO-MAKE-EXECUTION-var1 *subject*
                        TO-MAKE-EXECUTION-var2 *object-complement*
                        TO-MAKE-EXECUTION-var3 nil
                        TO-MAKE-EXECUTION-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-MAKE-EXECUTION-var2 *cd-form*)
                )
                (
                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                        (assign TO-MAKE-EXECUTION-var2 *cd-form*
                                *object-complement* TO-MAKE-EXECUTION-var2)
                        (next-packet
```

```
                                        (
                                        (test (eq *part-of-speech* 'position))
                                        (assign TO-MAKE-EXECUTION-var3 *cd-form*)
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'pointer))
                                (assign TO-MAKE-EXECUTION-var2 *cd-form*
                                        *object-complement* TO-MAKE-EXECUTION-var2)
                                (next-packet
                                        (
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                (next-packet
                                                        (
                                                        (test (eq *part-of-speech* 'position))
                                                        (assign TO-MAKE-EXECUTION-var3 *cd-form*)
                                                        )
                                                )
                                        )
                                        (
                                                (test (eq *part-of-speech* 'qualification))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                        (test (eq *part-of-speech* 'position))
                                                                        (assign TO-MAKE-EXECUTION-var3 *cd-
form*)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-MAKE-EXECUTION-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-MAKE-EXECUTION-
var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-MAKE-EXECUTION-
var2 *cd-form*
```

```
                                                                                *object-complement* TO-MAKE-
EXECUTION-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
MAKE-EXECUTION-var2 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
MAKE-EXECUTION-var2 *cd-form*
                                                                                                *object-
complement* TO-MAKE-EXECUTION-var2)
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'action))
                                (assign be-able-var2 *cd-form*)
                        )
        )
  )
)


(defword TO-INITIALIZE
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(propel    (action-details ?TO-INITIALIZE-var0)
                                                (actor  ?TO-INITIALIZE-var1)
                                                (object ?TO-INITIALIZE-var2)
                                                (from ?TO-INITIALIZE-var3)
                                                (to ?TO-INITIALIZE-var4))
                        TO-INITIALIZE-var0 *word*
                        TO-INITIALIZE-var1 *subject*
                        TO-INITIALIZE-var2 *object-complement*
                        TO-INITIALIZE-var3 nil
                        TO-INITIALIZE-var4 nil)
        (next-packet
```

```
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-INITIALIZE-var2 *cd-form*)
                )
                (
                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                        (assign TO-INITIALIZE-var2 *cd-form*
                                *object-complement* TO-INITIALIZE-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-INITIALIZE-var3 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-INITIALIZE-var2 *cd-form*
                                *object-complement* TO-INITIALIZE-var2)
                        (next-packet
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                        (next-packet
                                                (
                                                (test (eq *part-of-speech* 'position))
                                                (assign TO-INITIALIZE-var3 *cd-form*)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'position))
                                                                (assign TO-INITIALIZE-var3 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'position))
                        (assign TO-INITIALIZE-var3 *cd-form*)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
```

```
                                                                    (assign TO-INITIALIZE-var2 *cd-
form*)
                                                            )
                                                            (
                                                                    (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                    (assign TO-INITIALIZE-var2 *cd-
form*
                                                                    *object-complement* TO-
INITIALIZE-var2)
                                                            )
                                                    )
                                            )
                                            (
                                                    (test (eq *part-of-speech* 'qualification))
                                                    (next-packet
                                                            (
                                                                    (test (eq *part-of-speech* 'wobject))
                                                                    (next-packet
                                                                            (
                                                                                    (test (eq *part-
of-speech* 'relative))
                                                                                    (assign TO-
INITIALIZE-var2 *cd-form*)
                                                                            )
                                                                            (
                                                                                    (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                    (assign TO-
INITIALIZE-var2 *cd-form*
                                                                                    *object-
complement* TO-INITIALIZE-var2)
                                                                            )
                                                                    )
                                                            )
                                                    )
                                            )
                                    )
                            )
                    )
                    (
                            (test (eq *part-of-speech* 'action))
                            (assign be-able-var2 *cd-form*)
                    )
            )
    )
)

;;;;;;;;;;;;;;;;;;;; MOVE ;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;; GRASP ;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;; INGEST ;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;; SPEAK ;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;; ATTEND ;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword TO-MAKE-EVALUATION
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                    *cd-form* '(attend    (action-details ?TO-MAKE-EVALUATION-var0)
                                          (actor  ?TO-MAKE-EVALUATION-var1)
                                          (object ?TO-MAKE-EVALUATION-var2)
                                          (from ?TO-MAKE-EVALUATION-var3)
                                          (to ?TO-MAKE-EVALUATION-var4))
                    TO-MAKE-EVALUATION-var0 *word*
                    TO-MAKE-EVALUATION-var1 *subject*
                    TO-MAKE-EVALUATION-var2 *subject*
                    TO-MAKE-EVALUATION-var3 nil
                    TO-MAKE-EVALUATION-var4 *object-complement*)
        (next-packet
              (
                    (test (eq *part-of-speech* 'relative))
                    (assign TO-MAKE-EVALUATION-var4 *cd-form*)
              )
              (
                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                    (assign TO-MAKE-EVALUATION-var4 *cd-form*
                          *object-complement* TO-MAKE-EVALUATION-var4)
                    (next-packet
                          (
                          (test (eq *part-of-speech* 'position))
                          (assign TO-MAKE-EVALUATION-var3 *cd-form*)
                          )
                    )
              )
              (
                    (test (eq *part-of-speech* 'pointer))
                    (assign TO-MAKE-EVALUATION-var4 *cd-form*
                          *object-complement* TO-MAKE-EVALUATION-var4)
                    (next-packet
                          (
                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                (next-packet
                                      (
                                      (test (eq *part-of-speech* 'position))
                                      (assign TO-MAKE-EVALUATION-var3 *cd-form*)
                                      )
                                )
                          )
                          (
                                (test (eq *part-of-speech* 'qualification))
                                (next-packet
                                      (
                                            (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))

                                            (next-packet
                                                  (
                                                  (test (eq *part-of-speech* 'position))
```

```
                                                                (assign TO-MAKE-EVALUATION-var3 *cd-
form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-MAKE-EVALUATION-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-MAKE-EVALUATION-
var4 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-MAKE-EVALUATION-
var4 *cd-form*
                                                                                *object-complement* TO-MAKE-
EVALUATION-var4)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
MAKE-EVALUATION-var4 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
MAKE-EVALUATION-var4 *cd-form*
                                                                                                *object-
complement* TO-MAKE-EVALUATION-var4)
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
```

```
                                                )
                                        )
                                )
                        )
                )
        )
  )
)

(defword TO-TEST
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(attend    (action-details ?TO-TEST-var0)
                                                (actor  ?TO-TEST-var1)
                                                (object ?TO-TEST-var2)
                                                (from ?TO-TEST-var3)
                                                (to ?TO-TEST-var4))
                        TO-TEST-var0 *word*
                        TO-TEST-var1 *subject*
                        TO-TEST-var2 *subject*
                        TO-TEST-var3 nil
                        TO-TEST-var4 *object-complement*)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-TEST-var4 *cd-form*)
                )
                (
                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                        (assign TO-TEST-var4 *cd-form*
                                *object-complement* TO-TEST-var4)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-TEST-var3 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-TEST-var4 *cd-form*
                                *object-complement* TO-TEST-var4)
                        (next-packet
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                        (next-packet
                                                (
                                                (test (eq *part-of-speech* 'position))
                                                (assign TO-TEST-var3 *cd-form*)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
```

```
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                        (test (eq *part-of-speech* 'position))
                                                                        (assign TO-TEST-var3 *cd-form*)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-TEST-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-TEST-var4 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-TEST-var4 *cd-form*
                                                                                *object-complement* TO-TEST-
var4)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
TEST-var4 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
TEST-var4 *cd-form*
                                                                                                *object-
complement* TO-TEST-var4)
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
```

```
                                        )
                                    )
                                )
                            )
                        )
                    )
  )
)

(defword TO-MAKE-AUDIT
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                    *cd-form* '(attend    (action-details ?TO-MAKE-AUDIT-var0)
                                        (actor  ?TO-MAKE-AUDIT-var1)
                                        (object ?TO-MAKE-AUDIT-var2)
                                        (from ?TO-MAKE-AUDIT-var3)
                                        (to ?TO-MAKE-AUDIT-var4))
                    TO-MAKE-AUDIT-var0 *word*
                    TO-MAKE-AUDIT-var1 *subject*
                    TO-MAKE-AUDIT-var2 *subject*
                    TO-MAKE-AUDIT-var3 nil
                    TO-MAKE-AUDIT-var4 *object-complement*)
        (next-packet
            (
                    (test (eq *part-of-speech* 'relative))
                    (assign TO-MAKE-AUDIT-var4 *cd-form*)
            )
            (
                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                    (assign TO-MAKE-AUDIT-var4 *cd-form*
                        *object-complement* TO-MAKE-AUDIT-var4)
                    (next-packet
                        (
                        (test (eq *part-of-speech* 'position))
                        (assign TO-MAKE-AUDIT-var3 *cd-form*)
                        )
                    )
            )
            (
                    (test (eq *part-of-speech* 'pointer))
                    (assign TO-MAKE-AUDIT-var4 *cd-form*
                        *object-complement* TO-MAKE-AUDIT-var4)
                    (next-packet
                        (
                            (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                            (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-MAKE-AUDIT-var3 *cd-form*)
                                )
                            )
                        )
                        (
                            (test (eq *part-of-speech* 'qualification))
                            (next-packet
                                (
```

```
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                (next-packet
                                                        (
                                                        (test (eq *part-of-speech* 'position))
                                                        (assign TO-MAKE-AUDIT-var3 *cd-form*)
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'position))
                        (assign TO-MAKE-AUDIT-var3 *cd-form*)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign TO-MAKE-AUDIT-var4
*cd-form*)
                                                                )
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                        (assign TO-MAKE-AUDIT-var4
*cd-form*
                                                                        *object-complement* TO-MAKE-
AUDIT-var4)
                                                                )
                                                        )
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'qualification))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
                                                                        (next-packet
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'relative))
                                                                                        (assign TO-
MAKE-AUDIT-var4 *cd-form*)
                                                                                )
                                                                                (
                                                                                        (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                        (assign TO-
MAKE-AUDIT-var4 *cd-form*
                                                                                        *object-
complement* TO-MAKE-AUDIT-var4)
                                                                                )
```

```
                                                                                  )
                                                                        )
                                                              )
                                                    )
                                          )
                                )
                      )
            )
   )
 )
)

(defword TO-READ
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                    *cd-form* '(attend   (action-details ?TO-READ-var0)
                                         (actor  ?TO-READ-var1)
                                         (object ?TO-READ-var2)
                                         (from ?TO-READ-var3)
                                         (to ?TO-READ-var4))
                    TO-READ-var0 *word*
                    TO-READ-var1 *subject*
                    TO-READ-var2 *subject*
                    TO-READ-var3 nil
                    TO-READ-var4 *object-complement*)
        (next-packet
              (
                    (test (eq *part-of-speech* 'relative))
                    (assign TO-READ-var4 *cd-form*)
              )
              (
                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                    (assign TO-READ-var4 *cd-form*
                          *object-complement* TO-READ-var4)
                    (next-packet
                          (
                          (test (eq *part-of-speech* 'position))
                          (assign TO-READ-var3 *cd-form*)
                          )
                    )
              )
              (
                    (test (eq *part-of-speech* 'pointer))
                    (assign TO-READ-var4 *cd-form*
                          *object-complement* TO-READ-var4)
                    (next-packet
                          (
                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                (next-packet
                                      (
                                      (test (eq *part-of-speech* 'position))
                                      (assign TO-READ-var3 *cd-form*)
                                      )
                                )
                          )
                          (
```

```
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'position))
                                                                (assign TO-READ-var3 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-READ-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-READ-var4 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-READ-var4 *cd-form*
                                                                                *object-complement* TO-READ-
var4)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
READ-var4 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
READ-var4 *cd-form*
                                                                                                *object-
complement* TO-READ-var4)
```

```
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
)

(defword TO-TRACK
    (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(attend    (action-details ?TO-TRACK-var0)
                                                (actor  ?TO-TRACK-var1)
                                                (object ?TO-TRACK-var2)
                                                (from ?TO-TRACK-var3)
                                                (to ?TO-TRACK-var4))
                        TO-TRACK-var0 *word*
                        TO-TRACK-var1 *subject*
                        TO-TRACK-var2 *subject*
                        TO-TRACK-var3 nil
                        TO-TRACK-var4 *object-complement*)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-TRACK-var4 *cd-form*)
                )
                (
                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                        (assign TO-TRACK-var4 *cd-form*
                                *object-complement* TO-TRACK-var4)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-TRACK-var3 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-TRACK-var4 *cd-form*
                                *object-complement* TO-TRACK-var4)
                        (next-packet
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                        (next-packet
                                                (
                                                (test (eq *part-of-speech* 'position))
                                                (assign TO-TRACK-var3 *cd-form*)
                                                )
                                        )
                                )
```

```
                                        (
                                                (test (eq *part-of-speech* 'qualification))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                        (test (eq *part-of-speech* 'position))
                                                                        (assign TO-TRACK-var3 *cd-form*)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-TRACK-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-TRACK-var4 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-TRACK-var4 *cd-form*
                                                                                *object-complement* TO-TRACK-
var4)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
TRACK-var4 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
TRACK-var4 *cd-form*
                                                                                                *object-
```

```
complement* TO-TRACK-var4)
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
)

;;;;;;;;;;;;;;;;;; KPOSS ;;;;;;;;;;;;;;;;;;;;

(defword TO-BE-ABLE-TO
    (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '?be-able-var2
                        be-able-var0 nil
                        be-able-var1 nil
                        be-able-var2 *object-complement*
                        be-able-var3 nil
                        be-able-var4 nil)
        (next-packet
            (
                        (test (eq *part-of-speech* 'action))
                        (assign be-able-var2 *cd-form*)
            )
            (
                        (test (eq *part-of-speech* 'person))
                        (assign *subject* *cd-form*)
                        (next-packet
                            (
                                    (test (eq *part-of-speech* 'action))
                                    (assign be-able-var2 *cd-form*)
                            )
                        )
            )
            (
                        (test (eq *part-of-speech* 'affirmation))  ;can not
                        (assign be-able-var2 *cd-form*)
            )
        )
    )
)

(defword TO-KNOW
    (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(kposs    (action-details ?Know-var0)
                                                (actor  ?know-var1)
                                                (object ?know-var2)
```

```
                                                    (from ?know-var3)
                                                    (to ?know-var4))
                        know-var0 *word*
                        know-var1 *subject*
                        know-var2 *object-complement*
                        know-var3 nil
                        know-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign know-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'person))
                        (assign know-var2 *cd-form*
                                *object-complement* know-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'relative))
                                (assign know-var2 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign know-var2 *cd-form*
                                *object-complement* know-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'relative))
                                (assign know-var2 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign know-var2 *cd-form*
                                *object-complement* know-var2)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'qualification)
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'relative))
                                                                (assign know-var2 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                (next-packet
                                                        (
```

```
                                                                (test (eq *part-of-speech* 'relative))
                                                                (assign know-var2 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )

                )
        )

(defword TO-HAVE-BELIEF
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(kposs    (action-details ?TO-HAVE-BELIEF-var0
                                                (actor  ?TO-HAVE-BELIEF-var1)
                                                (object ?TO-HAVE-BELIEF-var2)
                                                (from ?TO-HAVE-BELIEF-var3)
                                                (to ?TO-HAVE-BELIEF-var4))
                        TO-HAVE-BELIEF-var0 *word*
                        TO-HAVE-BELIEF-var1 *subject*
                        TO-HAVE-BELIEF-var2 *object-complement*
                        TO-HAVE-BELIEF-var3 nil
                        TO-HAVE-BELIEF-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-HAVE-BELIEF-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'person))
                        (assign TO-HAVE-BELIEF-var2 *cd-form*
                                *object-complement* TO-HAVE-BELIEF-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'relative))
                                (assign TO-HAVE-BELIEF-var2 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign TO-HAVE-BELIEF-var2 *cd-form*
                                *object-complement* TO-HAVE-BELIEF-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'relative))
                                (assign TO-HAVE-BELIEF-var2 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-HAVE-BELIEF-var2 *cd-form*
```

```
                                        *object-complement* TO-HAVE-BELIEF-var2)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'qualification)
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign TO-HAVE-BELIEF-var2 *cd-form*)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                        (
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'relative))
                                                                (assign TO-HAVE-BELIEF-var2 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )

                )
        )

(defword TO-THINK
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(kposs    (action-details ?TO-THINK-var0)
                                                (actor  ?TO-THINK-var1)
                                                (object ?TO-THINK-var2)
                                                (from ?TO-THINK-var3)
                                                (to ?TO-THINK-var4))
                        TO-THINK-var0 *word*
                        TO-THINK-var1 *subject*
                        TO-THINK-var2 *object-complement*
                        TO-THINK-var3 nil
                        TO-THINK-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-THINK-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'person))
                        (assign TO-THINK-var2 *cd-form*
                                *object-complement* TO-THINK-var2)
```

```
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'relative))
                                (assign TO-THINK-var2 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign TO-THINK-var2 *cd-form*
                                *object-complement* TO-THINK-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'relative))
                                (assign TO-THINK-var2 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-THINK-var2 *cd-form*
                                *object-complement* TO-THINK-var2)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'qualification)
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'relative))
                                                                (assign TO-THINK-var2 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                (next-packet
                                                        (
                                                        (test (eq *part-of-speech* 'relative))
                                                        (assign TO-THINK-var2 *cd-form*)
                                                        )
                                                )
                                        )
                                )
                        )
                )

        )
   )
)

;;;;;;;;;;;;;;;;;; MTRANS ;;;;;;;;;;;;;;;;;;;;;
```

```
(defword TO-MAKE-DESCRIPTION
 (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(mtrans   (action-details ?TO-MAKE-DESCRIPTION-var0)
                                                (actor  ?TO-MAKE-DESCRIPTION-var1)
                                                (object ?TO-MAKE-DESCRIPTION-var2)
                                                (from ?TO-MAKE-DESCRIPTION-var4)
                                                (to ?TO-MAKE-DESCRIPTION-var3))
                        TO-MAKE-DESCRIPTION-var0 *word*
                        TO-MAKE-DESCRIPTION-var1 *subject*
                        TO-MAKE-DESCRIPTION-var2 *object-complement*
                        TO-MAKE-DESCRIPTION-var3 nil
                        TO-MAKE-DESCRIPTION-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-MAKE-DESCRIPTION-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign TO-MAKE-DESCRIPTION-var2 *cd-form*
                                *object-complement* TO-MAKE-DESCRIPTION-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'direction))
                                (assign TO-MAKE-DESCRIPTION-var3 *cd-form*
                                        *receptor-complement* TO-MAKE-DESCRIPTION-var3)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-MAKE-DESCRIPTION-var2 *cd-form*
                                *object-complement* TO-MAKE-DESCRIPTION-var2)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'wobject))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'direction))
                                                        (assign TO-MAKE-DESCRIPTION-var3 *cd-form*
                                                                *receptor-complement* TO-MAKE-
DESCRIPTION-var3)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech*
'direction))
                                                                        (assign TO-MAKE-DESCRIPTION-
```

```
 var3 *cd-form*
                                                   *receptor-complement*
TO-MAKE-DESCRIPTION-var3)
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
                (

                    (test (eq *part-of-speech* 'person))
                    (assign TO-MAKE-DESCRIPTION-var3 *cd-form*
                        *receptor-complement* TO-MAKE-DESCRIPTION-var3)
                    (next-packet
                        (
                            (test (eq *part-of-speech* 'relative))
                            (assign TO-MAKE-DESCRIPTION-var2 *cd-form*)
                        )
                        (
                            (test (eq *part-of-speech* 'wobject))
                            (assign TO-MAKE-DESCRIPTION-var2 *cd-form*
                                *object-complement* TO-MAKE-DESCRIPTION-var2)
                        )
                        (
                            (test (eq *part-of-speech* 'pointer))
                            (assign TO-MAKE-DESCRIPTION-var2 *cd-form*
                                *object-complement* TO-MAKE-DESCRIPTION-var2)
                        )
                    )
                )
                (

                    (test (eq *part-of-speech* 'direction))
                    (assign TO-MAKE-DESCRIPTION-var3 *cd-form*
                        *receptor-complement* TO-MAKE-DESCRIPTION-var3)
                    (next-packet
                        (
                            (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                            (next-packet
                                (
                                    (test (eq *part-of-speech* 'relative))
                                    (assign TO-MAKE-DESCRIPTION-var2 *cd-form*)
                                )
                                (
                                    (test (eq *part-of-speech* 'wobject))
                                    (assign TO-MAKE-DESCRIPTION-var2 *cd-form*
                                        *object-complement* TO-MAKE-
DESCRIPTION-var2)
                                )
                                (
                                    (test (eq *part-of-speech* 'pointer))
                                    (assign TO-MAKE-DESCRIPTION-var2 *cd-form*
                                        *object-complement* TO-MAKE-
DESCRIPTION-var2)
                                )
```

```
                                        )
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-MAKE-DESCRIPTION-
var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (assign TO-MAKE-DESCRIPTION-
var2 *cd-form*
                                                                                        *object-complement* TO-
MAKE-DESCRIPTION-var2)
                                                                        )
                                                                        (
                                                                                (test (eq *part-of-speech* 'pointer))
                                                                                (assign TO-MAKE-DESCRIPTION-
var2 *cd-form*
                                                                                        *object-complement* TO-
MAKE-DESCRIPTION-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
MAKE-DESCRIPTION-var2 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'wobject))
                                                                                                (assign TO-
MAKE-DESCRIPTION-var2 *cd-form*
        *object-complement* TO-MAKE-DESCRIPTION-var2)
                                                                                        )
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'pointer))
                                                                                                (assign TO-
MAKE-DESCRIPTION-var2 *cd-form*
        *object-complement* TO-MAKE-DESCRIPTION-var2)
```

```
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-MAKE-DESCRIPTION-var4 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-MAKE-DESCRIPTION-
var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                                (assign TO-MAKE-DESCRIPTION-
var2 *cd-form*
                                                                                *object-complement* TO-MAKE-
DESCRIPTION-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
MAKE-DESCRIPTION-var2 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                                                (assign TO-
MAKE-DESCRIPTION-var2 *cd-form*
                                                                                                *object-
complement* TO-MAKE-DESCRIPTION-var2)
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
```

```
                                        )
                                    )
                                )
                            )
                        )
                    )
            )
)

(defword TO-TELL-CASE
    (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                *cd-form* '(mtrans   (action-details ?TO-TELL-CASE-var0)
                                (actor  ?TO-TELL-CASE-var1)
                                (object ?TO-TELL-CASE-var2)
                                (from ?TO-TELL-CASE-var4)
                                (to ?TO-TELL-CASE-var3))
                TO-TELL-CASE-var0 *word*
                TO-TELL-CASE-var1 *subject*
                TO-TELL-CASE-var2 *object-complement*
                TO-TELL-CASE-var3 nil
                TO-TELL-CASE-var4 nil)
        (next-packet
            (
                (test (eq *part-of-speech* 'relative))
                (assign TO-TELL-CASE-var2 *cd-form*)
            )
            (
                (test (eq *part-of-speech* 'wobject))
                (assign TO-TELL-CASE-var2 *cd-form*
                    *object-complement* TO-TELL-CASE-var2)
                (next-packet
                    (
                    (test (eq *part-of-speech* 'direction))
                    (assign TO-TELL-CASE-var3 *cd-form*
                        *receptor-complement* TO-TELL-CASE-var3)
                    )
                )
            )
            (
                (test (eq *part-of-speech* 'pointer))
                (assign TO-TELL-CASE-var2 *cd-form*
                    *object-complement* TO-TELL-CASE-var2)
                (next-packet
                    (
                        (test (eq *part-of-speech* 'wobject))
                        (next-packet
                            (
                                (test (eq *part-of-speech* 'direction))
                                (assign TO-TELL-CASE-var3 *cd-form*
                                    *receptor-complement* TO-TELL-CASE-
var3)
                            )
                        )
                    )
```

```
                                        (
                                                (test (eq *part-of-speech* 'qualification))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech*
'direction))
                                                                                (assign TO-TELL-CASE-var3 *cd-
form*
                                                                                        *receptor-complement*
TO-TELL-CASE-var3)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                                (

                                        (test (eq *part-of-speech* 'person))
                                        (assign TO-TELL-CASE-var3 *cd-form*
                                                *receptor-complement* TO-TELL-CASE-var3)
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'relative))
                                                        (assign TO-TELL-CASE-var2 *cd-form*)
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (assign TO-TELL-CASE-var2 *cd-form*
                                                                *object-complement* TO-TELL-CASE-var2)
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'pointer))
                                                        (assign TO-TELL-CASE-var2 *cd-form*
                                                                *object-complement* TO-TELL-CASE-var2)
                                                )
                                        )
                                )
                                (

                                        (test (eq *part-of-speech* 'direction))
                                        (assign TO-TELL-CASE-var3 *cd-form*
                                                *receptor-complement* TO-TELL-CASE-var3)
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign TO-TELL-CASE-var2 *cd-form*)
                                                                )
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
```

```
                                                (assign TO-TELL-CASE-var2 *cd-form*
                                                        *object-complement* TO-TELL-CASE-var2)
                                        )
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (assign TO-TELL-CASE-var2 *cd-form*
                                                        *object-complement* TO-TELL-CASE-var2)
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'pointer))
                                (next-packet
                                        (
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'relative))
                                                                (assign TO-TELL-CASE-var2 *cd-
form*)
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (assign TO-TELL-CASE-var2 *cd-
form*
                                                                        *object-complement* TO-
TELL-CASE-var2)
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'pointer))
                                                                (assign TO-TELL-CASE-var2 *cd-
form*
                                                                        *object-complement* TO-
TELL-CASE-var2)
                                                        )
                                                )
                                        )
                                        (
                                                (test (eq *part-of-speech* qualification))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                (assign TO-
TELL-CASE-var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (eq *part-
of-speech* 'wobject))
                                                                                (assign TO-
TELL-CASE-var2 *cd-form*
```

```
        *object-complement* TO-TELL-CASE-var2)
                                                                                )
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'pointer))
                                                                                        (assign TO-
TELL-CASE-var2 *cd-form*

        *object-complement* TO-TELL-CASE-var2)
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'position))
                        (assign TO-TELL-CASE-var4 *cd-form*)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign TO-TELL-CASE-var2 *cd-
form*)
                                                                )
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                        (assign TO-TELL-CASE-var2 *cd-
form*
                                                                        *object-complement* TO-TELL-
CASE-var2)
                                                                )
                                                        )
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'qualification))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
                                                                        (next-packet
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'relative))
                                                                                        (assign TO-
TELL-CASE-var2 *cd-form*)
                                                                                )
                                                                                (
                                                                                        (test (or (eq
```

```
                                                                                *part-of-
speech* 'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                                (assign TO-
TELL-CASE-var2 *cd-form*
                                                                                *object-
complement* TO-TELL-CASE-var2)
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
)


(defword TO-GIVE-KNOWLEDGE
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(mtrans   (action-details ?givek-var0)
                                                (actor  ?givek-var1)
                                                (object ?givek-var2)
                                                (from ?givek-var4)
                                                (to ?givek-var3))
                        givek-var0 *word*
                        givek-var1 *subject*
                        givek-var2 *object-complement*
                        givek-var3 nil
                        givek-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign givek-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign givek-var2 *cd-form*
                                *object-complement* givek-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'direction))
                                (assign givek-var3 *cd-form*
                                        *receptor-complement* givek-var3)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign givek-var2 *cd-form*
                                *object-complement* givek-var2)
                        (next-packet
                                (
```

```
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'relative))
                                                        (assign givek-var2 *cd-form*)
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (assign givek-var2 *cd-form*
                                                                *object-complement* givek-var2)
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'pointer))
                                                        (assign givek-var2 *cd-form*
                                                                *object-complement* givek-var2)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign givek-var2 *cd-form*)
                                                                )
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
                                                                        (assign givek-var2 *cd-form*
                                                                                *object-complement*
givek-var2)
                                                                )
                                                                (
                                                                        (test (eq *part-of-speech* 'pointer))
                                                                        (assign givek-var2 *cd-form*
                                                                                *object-complement*
givek-var2)
                                                                )
                                                        )
                                                )
                                                (
                                                        (test (eq *part-of-speech* qualification))
                                                        (next-packet
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                        (next-packet
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'relative))
                                                                                        (assign givek-
var2 *cd-form*)
                                                                                )
                                                                                (
                                                                                        (test (eq *part-
```

```
*part-of-speech* 'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                                                    (assign givek-
var2 *cd-form*
                                                                                                    *object-
complement* givek-var2)
                                                                                                        )
                                                                                                    )
                                                                                                )
                                                                                            )
                                                                                        )
                                                                                    )
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
)

(defword TO-DISCLOSE-KNOWLEDGE
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(mtrans   (action-details ?disclosek-var0)
                                                (actor  ?disclosek-var1)
                                                (object ?disclosek-var2)
                                                (from ?disclosek-var4)
                                                (to ?disclosek-var3))
                        disclosek-var0 *word*
                        disclosek-var1 *subject*
                        disclosek-var2 *object-complement*
                        disclosek-var3 nil
                        disclosek-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign disclosek-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign disclosek-var2 *cd-form*
                                *object-complement* disclosek-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'direction))
                                (assign disclosek-var3 *cd-form*
                                        *receptor-complement* disclosek-var3)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign disclosek-var2 *cd-form*
                                *object-complement* disclosek-var2)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'wobject))
                                        (next-packet
```

```
                                        (
                                                (test (eq *part-of-speech* 'wobject))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'direction))
                                                                (assign disclosek-var3 *cd-form*
                                                                        *receptor-complement* disclosek-var3)
                                                        )
                                                )
                                        )
                                        (
                                                (test (eq *part-of-speech* 'qualification))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech*
'direction))
                                                                                (assign disclosek-var3 *cd-form*
                                                                                        *receptor-complement*
disclosek-var3)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (

                                (test (eq *part-of-speech* 'person))
                                (assign disclosek-var3 *cd-form*
                                        *receptor-complement* disclosek-var3)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'relative))
                                                (assign disclosek-var2 *cd-form*)
                                        )
                                        (
                                                (test (eq *part-of-speech* 'wobject))
                                                (assign disclosek-var2 *cd-form*
                                                        *object-complement* disclosek-var2)
                                        )
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (assign disclosek-var2 *cd-form*
                                                        *object-complement* disclosek-var2)
                                        )
                                )
                        )
                        (

                                (test (eq *part-of-speech* 'direction))
                                (assign disclosek-var3 *cd-form*
                                        *receptor-complement* disclosek-var3)
                                (next-packet
```

```
                                        (
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'relative))
                                                                (assign disclosek-var2 *cd-form*)
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (assign disclosek-var2 *cd-form*
                                                                        *object-complement* disclosek-var2)
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'pointer))
                                                                (assign disclosek-var2 *cd-form*
                                                                        *object-complement* disclosek-var2)
                                                        )
                                                )
                                        )
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign disclosek-var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (assign disclosek-var2 *cd-form*
                                                                                        *object-complement*
disclosek-var2)
                                                                        )
                                                                        (
                                                                                (test (eq *part-of-speech* 'pointer))
                                                                                (assign disclosek-var2 *cd-form*
                                                                                        *object-complement*
disclosek-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign
disclosek-var2 *cd-form*)
                                                                                        )
                                                                                        (
```

```
                                                                                (test (eq *part-
of-speech* 'wobject))
                                                                                (assign
disclosek-var2 *cd-form*
        *object-complement* disclosek-var2)
                                                                        )
                                                                        (
                                                                                (test (eq *part-
of-speech* 'pointer))
                                                                                (assign
disclosek-var2 *cd-form*
        *object-complement* disclosek-var2)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'position))
                        (assign disclosek-var4 *cd-form*)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign disclosek-var2 *cd-form*)
                                                                )
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                        (assign disclosek-var2 *cd-form*
                                                                        *object-complement* disclosek-
var2)
                                                                )
                                                        )
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'qualification))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
                                                                        (next-packet
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'relative))
                                                                                        (assign
disclosek-var2 *cd-form*)
                                                                                )
```

```
                                                                                (
                                                                                        (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                                        (assign
disclosek-var2 *cd-form*
                                                                                        *object-
complement* disclosek-var2)
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
        )
  )
)

(defword TO-MAKE-PRESCRIBTION
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(mtrans   (action-details ?prescribtion-var0)
                                                (actor  ?prescribtion-var1)
                                                (object ?prescribtion-var2)
                                                (from ?prescribtion-var4)
                                                (to ?prescribtion-var3))
                        prescribtion-var0 *word*
                        prescribtion-var1 *subject*
                        prescribtion-var2 *object-complement*
                        prescribtion-var3 nil
                        prescribtion-var4 nil)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign prescribtion-var2 *cd-form*)
                )
                (
                        (test (eq *part-of-speech* 'wobject))
                        (assign prescribtion-var2 *cd-form*
                                *object-complement* prescribtion-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'direction))
                                (assign prescribtion-var3 *cd-form*
                                        *receptor-complement* prescribtion-var3)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign prescribtion-var2 *cd-form*
                                *object-complement* prescribtion-var2)
                        (next-packet
                                (
```

```
                                        (test (eq *part-of-speech* 'wobject))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'direction))
                                                        (assign prescribtion-var3 *cd-form*
                                                                *receptor-complement* prescribtion-var3)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech*
'direction))
                                                                        (assign prescribtion-var3 *cd-form*
                                                                                *receptor-complement*
prescribtion-var3)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (

                        (test (eq *part-of-speech* 'person))
                        (assign prescribtion-var3 *cd-form*
                                *receptor-complement* prescribtion-var3)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'relative))
                                        (assign prescribtion-var2 *cd-form*)
                                )
                                (
                                        (test (eq *part-of-speech* 'wobject))
                                        (assign prescribtion-var2 *cd-form*
                                                *object-complement* prescribtion-var2)
                                )
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (assign prescribtion-var2 *cd-form*
                                                *object-complement* prescribtion-var2)
                                )
                        )
                )
                (

                        (test (eq *part-of-speech* 'direction))
                        (assign prescribtion-var3 *cd-form*
                                *receptor-complement* prescribtion-var3)
                        (next-packet
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
```

```
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'relative))
                                                        (assign prescribtion-var2 *cd-form*)
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (assign prescribtion-var2 *cd-form*
                                                                *object-complement* prescribtion-var2)
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'pointer))
                                                        (assign prescribtion-var2 *cd-form*
                                                                *object-complement* prescribtion-var2)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign prescribtion-var2 *cd-
form*)
                                                                )
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
                                                                        (assign prescribtion-var2 *cd-form*
                                                                                *object-complement*
prescribtion-var2)
                                                                )
                                                                (
                                                                        (test (eq *part-of-speech* 'pointer))
                                                                        (assign prescribtion-var2 *cd-form*
                                                                                *object-complement*
prescribtion-var2)
                                                                )
                                                        )
                                                )
                                                (
                                                        (test (eq *part-of-speech* qualification))
                                                        (next-packet
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                        (next-packet
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'relative))
                                                                                        (assign
prescribtion-var2 *cd-form*)
                                                                                )
                                                                                (
                                                                                        (test (eq *part-
```

```
of-speech* 'wobject))
                                                                                    (assign
prescribtion-var2 *cd-form*

        *object-complement* prescribtion-var2)
                                                                                )
                                                                                (
                                                                                    (test (eq *part-
of-speech* 'pointer))
                                                                                    (assign
prescribtion-var2 *cd-form*

        *object-complement* prescribtion-var2)
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                        (
                                            (test (eq *part-of-speech* 'position))
                                            (assign prescribtion-var4 *cd-form*)
                                            (next-packet
                                                (
                                                    (test (eq *part-of-speech* 'pointer))
                                                    (next-packet
                                                        (
                                                            (test (eq *part-of-speech* 'wobject))
                                                            (next-packet
                                                                (
                                                                    (test (eq *part-of-speech* 'relative))
                                                                    (assign prescribtion-var2 *cd-
form*)
                                                                )
                                                                (
                                                                    (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                    (assign prescribtion-var2 *cd-form*
                                                                    *object-complement* prescribtion-
var2)
                                                                )
                                                            )
                                                        )
                                                        (
                                                            (test (eq *part-of-speech* 'qualification))
                                                            (next-packet
                                                                (
                                                                    (test (eq *part-of-speech* 'wobject))
                                                                    (next-packet
                                                                        (
                                                                            (test (eq *part-
of-speech* 'relative))
                                                                            (assign
prescribtion-var2 *cd-form*)
                                                                        )
```

```
                                                                                                   (
                                                                                                            (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person) (eq *part-of-speech* 'pointer)))
                                                                                                            (assign
prescribtion-var2 *cd-form*
                                                                                                            *object-
complement* prescribtion-var2)
                                                                                                   )
                                                                                          )
                                                                                 )
                                                                        )
                                                               )
                                                      )
                                             )
                                    )
                           )
                  )
   )
)

;;;;;;;;;;;;;;;;; MBUILD ;;;;;;;;;;;;;;;;;;;;;

(defword TO-MAKE-LIST-OF-PROGRAMMED-STEPS
  (
          (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
          (assign    *part-of-speech* 'action
                            *cd-form* '(mbuild   (action-details ?prog-var0)
                                                  (actor  ?prog-var1)
                                                  (object ?prog-var2)
                                                  (from ?prog-var3)
                                                  (to ?prog-var4))
                            prog-var0 *word*
                            prog-var1 *subject*
                            prog-var2 *object-complement*
                            prog-var3 nil
                            prog-var4 *subject*)
          (next-packet
                  (
                            (test (eq *part-of-speech* 'relative))
                            (assign prog-var2 *cd-form*)
                  )
                  (
                            (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                            (assign prog-var2 *cd-form*
                                     *object-complement* prog-var2)
                            (next-packet
                                     (
                                     (test (eq *part-of-speech* 'position))
                                     (assign prog-var3 *cd-form*)
                                     )
                            )
                  )
                  (
                            (test (eq *part-of-speech* 'pointer))
                            (assign prog-var2 *cd-form*
                                     *object-complement* prog-var2)
                            (next-packet
```

```
                                        (
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                (next-packet
                                                        (
                                                        (test (eq *part-of-speech* 'position))
                                                        (assign prog-var3 *cd-form*)
                                                        )
                                                )
                                        )
                                        (
                                                (test (eq *part-of-speech* 'qualification))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                        (test (eq *part-of-speech* 'position))
                                                                        (assign prog-var3 *cd-form*)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign prog-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign prog-var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign prog-var2 *cd-form*
                                                                                *object-complement* prog-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign prog-
```

```
var2 *cd-form*)
                                                                                                        )
                                                                                                        (
                                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                                (assign prog-
var2 *cd-form*
                                                                                                                *object-
complement* prog-var2)
                                                                                                        )
                                                                                                )
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
  )
)

(defword TO-MAKE-CREATION
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(mbuild   (action-details ?TO-MAKE-CREATION-var0)
                                                (actor  ?TO-MAKE-CREATION-var1)
                                                (object ?TO-MAKE-CREATION-var2)
                                                (from ?TO-MAKE-CREATION-var3)
                                                (to ?TO-MAKE-CREATION-var4))
                        TO-MAKE-CREATION-var0 *word*
                        TO-MAKE-CREATION-var1 *subject*
                        TO-MAKE-CREATION-var2 *object-complement*
                        TO-MAKE-CREATION-var3 nil
                        TO-MAKE-CREATION-var4 *subject*)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-MAKE-CREATION-var2 *cd-form*)
                )
                (
                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                        (assign TO-MAKE-CREATION-var2 *cd-form*
                                *object-complement* TO-MAKE-CREATION-var2)
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-MAKE-CREATION-var3 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-MAKE-CREATION-var2 *cd-form*
                                *object-complement* TO-MAKE-CREATION-var2)
                        (next-packet
```

```
                                        (
                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                (next-packet
                                                        (
                                                        (test (eq *part-of-speech* 'position))
                                                        (assign TO-MAKE-CREATION-var3 *cd-form*)
                                                        )
                                                )
                                        )
                                        (
                                                (test (eq *part-of-speech* 'qualification))
                                                (next-packet
                                                        (
                                                                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                                (next-packet
                                                                        (
                                                                        (test (eq *part-of-speech* 'position))
                                                                        (assign TO-MAKE-CREATION-var3 *cd-
form*)
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                        (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-MAKE-CREATION-var3 *cd-form*)
                                (next-packet
                                        (
                                                (test (eq *part-of-speech* 'pointer))
                                                (next-packet
                                                        (
                                                                (test (eq *part-of-speech* 'wobject))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'relative))
                                                                                (assign TO-MAKE-CREATION-
var2 *cd-form*)
                                                                        )
                                                                        (
                                                                                (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-MAKE-CREATION-
var2 *cd-form*
                                                                                *object-complement* TO-MAKE-
CREATION-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
```

```
                                                                                (
                                                                                    (test (eq *part-
of-speech* 'relative))
                                                                                    (assign TO-
MAKE-CREATION-var2 *cd-form*)
                                                                                )
                                                                                (
                                                                                    (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                    (assign TO-
MAKE-CREATION-var2 *cd-form*
                                                                                    *object-
complement* TO-MAKE-CREATION-var2)
                                                                                )
                                                                            )
                                                                        )
                                                                    )
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
)

(defword TO-CORRECT
    (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                    *cd-form* '(MBUILD          (action-details ?TO-CORRECT-var0)
                                                (actor  ?TO-CORRECT-var1)
                                                (object (QUALIFICATION (QUALIFYER (RIGHT TINDEF
PINDEF GINDEF NINDEF RELATED-TO-EXACT))
                                                                        (QUALIFIED ?TO-CORRECT-var2)))
                                                (from ?TO-CORRECT-var3)
                                                (to ?TO-CORRECT-var4))
                    TO-CORRECT-var0 *word*
                    TO-CORRECT-var1 *subject*
                    TO-CORRECT-var2 *object-complement*
                    TO-CORRECT-var3 nil
                    TO-CORRECT-var4 *subject*)
        (next-packet
                (
                    (test (eq *part-of-speech* 'relative))
                    (assign TO-CORRECT-var2 *cd-form*)
                )
                (
                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                    (assign TO-CORRECT-var2 *cd-form*
                            *object-complement* TO-CORRECT-var2)
                    (next-packet
                            (
                            (test (eq *part-of-speech* 'position))
                            (assign TO-CORRECT-var3 *cd-form*)
                            )
                    )
```

```
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-CORRECT-var2 *cd-form*
                                *object-complement* TO-CORRECT-var2)
                        (next-packet
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                        (next-packet
                                                (
                                                (test (eq *part-of-speech* 'position))
                                                (assign TO-CORRECT-var3 *cd-form*)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'position))
                                                                (assign TO-CORRECT-var3 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'position))
                        (assign TO-CORRECT-var3 *cd-form*)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign TO-CORRECT-var2 *cd-
form*)
                                                                )
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                        (assign TO-CORRECT-var2 *cd-
form*
                                                                        *object-complement* TO-
CORRECT-var2)
                                                                )
                                                        )
                                                )
                                                (
```

```
                                                    (test (eq *part-of-speech* 'qualification))
                                                    (next-packet
                                                        (
                                                            (test (eq *part-of-speech* 'wobject))
                                                            (next-packet
                                                                (
                                                                    (test (eq *part-
of-speech* 'relative))
                                                                    (assign TO-
CORRECT-var2 *cd-form*)
                                                                )
                                                                (
                                                                    (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                    (assign TO-
CORRECT-var2 *cd-form*
                                                                    *object-
complement* TO-CORRECT-var2)
                                                                )
                                                            )
                                                        )
                                                    )
                                                )
                                            )
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
)

(defword TO-FIND-SOLUTION
    (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                    *cd-form* '(MBUILD          (action-details ?TO-FIND-SOLUTION-var0)
                                                (actor  ?TO-FIND-SOLUTION-var1)
                                                (object (QUALIFICATION (QUALIFYER (RIGHT TINDEF
PINDEF GINDEF NINDEF RELATED-TO-EXACT))
                                                                        (QUALIFIED ?TO-FIND-SOLUTION-
var2)))
                                                (from ?TO-FIND-SOLUTION-var3)
                                                (to ?TO-FIND-SOLUTION-var4))
                    TO-FIND-SOLUTION-var0 *word*
                    TO-FIND-SOLUTION-var1 *subject*
                    TO-FIND-SOLUTION-var2 *object-complement*
                    TO-FIND-SOLUTION-var3 nil
                    TO-FIND-SOLUTION-var4 *subject*)
        (next-packet
            (
                (test (eq *part-of-speech* 'relative))
                (assign TO-FIND-SOLUTION-var2 *cd-form*)
            )
            (
                (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                (assign TO-FIND-SOLUTION-var2 *cd-form*
                        *object-complement* TO-FIND-SOLUTION-var2)
```

```
                        (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign TO-FIND-SOLUTION-var3 *cd-form*)
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'pointer))
                        (assign TO-FIND-SOLUTION-var2 *cd-form*
                                *object-complement* TO-FIND-SOLUTION-var2)
                        (next-packet
                                (
                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                        (next-packet
                                                (
                                                (test (eq *part-of-speech* 'position))
                                                (assign TO-FIND-SOLUTION-var3 *cd-form*)
                                                )
                                        )
                                )
                                (
                                        (test (eq *part-of-speech* 'qualification))
                                        (next-packet
                                                (
                                                        (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                        (next-packet
                                                                (
                                                                (test (eq *part-of-speech* 'position))
                                                                (assign TO-FIND-SOLUTION-var3 *cd-form*)
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
                (
                        (test (eq *part-of-speech* 'position))
                        (assign TO-FIND-SOLUTION-var3 *cd-form*)
                        (next-packet
                                (
                                        (test (eq *part-of-speech* 'pointer))
                                        (next-packet
                                                (
                                                        (test (eq *part-of-speech* 'wobject))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'relative))
                                                                        (assign TO-FIND-SOLUTION-var2
*cd-form*)
                                                                )
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                        (assign TO-FIND-SOLUTION-var2
*cd-form*
```

```
                                                                                *object-complement* TO-FIND-
SOLUTION-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
FIND-SOLUTION-var2 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
FIND-SOLUTION-var2 *cd-form*
                                                                                                *object-
complement* TO-FIND-SOLUTION-var2)
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
        )
  )
)

(defword TO-UNDERSTAND
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(MBUILD          (action-details ?TO-UNDERSTAND-var0)
                                                    (actor  ?TO-UNDERSTAND-var1)
                                                    (object ?TO-UNDERSTAND-var2)
                                                    (from ?TO-UNDERSTAND-var3)
                                                    (to ?TO-UNDERSTAND-var4))
                        TO-UNDERSTAND-var0 *word*
                        TO-UNDERSTAND-var1 *subject*
                        TO-UNDERSTAND-var2 *object-complement*
                        TO-UNDERSTAND-var3 nil
                        TO-UNDERSTAND-var4 *subject*)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
                        (assign TO-UNDERSTAND-var2 *cd-form*)
                )
                (
```

```
                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                    (assign TO-UNDERSTAND-var2 *cd-form*
                            *object-complement* TO-UNDERSTAND-var2)
                    (next-packet
                            (
                            (test (eq *part-of-speech* 'position))
                            (assign TO-UNDERSTAND-var3 *cd-form*)
                            )
                    )
            )
            (
                    (test (eq *part-of-speech* 'pointer))
                    (assign TO-UNDERSTAND-var2 *cd-form*
                            *object-complement* TO-UNDERSTAND-var2)
                    (next-packet
                            (
                                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                    (next-packet
                                            (
                                            (test (eq *part-of-speech* 'position))
                                            (assign TO-UNDERSTAND-var3 *cd-form*)
                                            )
                                    )
                            )
                            (
                                    (test (eq *part-of-speech* 'qualification))
                                    (next-packet
                                            (
                                                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                                    (next-packet
                                                            (
                                                            (test (eq *part-of-speech* 'position))
                                                            (assign TO-UNDERSTAND-var3 *cd-form*)
                                                            )
                                                    )
                                            )
                                    )
                            )
                    )
            )
            (
                    (test (eq *part-of-speech* 'position))
                    (assign TO-UNDERSTAND-var3 *cd-form*)
                    (next-packet
                            (
                                    (test (eq *part-of-speech* 'pointer))
                                    (next-packet
                                            (
                                                    (test (eq *part-of-speech* 'wobject))
                                                    (next-packet
                                                            (
                                                                    (test (eq *part-of-speech* 'relative))
                                                                    (assign TO-UNDERSTAND-var2
*cd-form*)
                                                            )
                                                            (
                                                                    (test (or (eq *part-of-speech*
```

```
'wobject) (eq *part-of-speech* 'person)))
                                                                                (assign TO-UNDERSTAND-var2
*cd-form*
                                                                                *object-complement* TO-
UNDERSTAND-var2)
                                                                        )
                                                                )
                                                        )
                                                        (
                                                                (test (eq *part-of-speech* 'qualification))
                                                                (next-packet
                                                                        (
                                                                                (test (eq *part-of-speech* 'wobject))
                                                                                (next-packet
                                                                                        (
                                                                                                (test (eq *part-
of-speech* 'relative))
                                                                                                (assign TO-
UNDERSTAND-var2 *cd-form*)
                                                                                        )
                                                                                        (
                                                                                                (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                                (assign TO-
UNDERSTAND-var2 *cd-form*
                                                                                                *object-
complement* TO-UNDERSTAND-var2)
                                                                                        )
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
        )
  )
)

(defword TO-MAKE-DIAGNOSTIC
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(mbuild   (action-details ?diagnostic-var0)
                                                (actor  ?diagnostic-var1)
                                                (object ?diagnostic-var2)
                                                (from ?diagnostic-var3)
                                                (to ?diagnostic-var4))
                        diagnostic-var0 *word*
                        diagnostic-var1 *subject*
                        diagnostic-var2 *object-complement*
                        diagnostic-var3 nil
                        diagnostic-var4 *subject*)
        (next-packet
                (
                        (test (eq *part-of-speech* 'relative))
```

```
                    (assign diagnostic-var2 *cd-form*)
                )
                (
                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                    (assign diagnostic-var2 *cd-form*
                        *object-complement* diagnostic-var2)
                    (next-packet
                        (
                        (test (eq *part-of-speech* 'position))
                        (assign diagnostic-var3 *cd-form*)
                        )
                    )
                )
                (
                    (test (eq *part-of-speech* 'pointer))
                    (assign diagnostic-var2 *cd-form*
                        *object-complement* diagnostic-var2)
                    (next-packet
                        (
                            (test (or (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                            (next-packet
                                (
                                (test (eq *part-of-speech* 'position))
                                (assign diagnostic-var3 *cd-form*)
                                )
                            )
                        )
                        (
                            (test (eq *part-of-speech* 'qualification))
                            (next-packet
                                (
                                    (test (or (eq *part-of-speech* 'wobject) (eq *part-of-
speech* 'person)))
                                    (next-packet
                                        (
                                        (test (eq *part-of-speech* 'position))
                                        (assign diagnostic-var3 *cd-form*)
                                        )
                                    )
                                )
                            )
                        )
                    )
                )
                (
                    (test (eq *part-of-speech* 'position))
                    (assign diagnostic-var3 *cd-form*)
                    (next-packet
                        (
                            (test (eq *part-of-speech* 'pointer))
                            (next-packet
                                (
                                    (test (eq *part-of-speech* 'wobject))
                                    (next-packet
                                        (
                                            (test (eq *part-of-speech* 'relative))
                                            (assign diagnostic-var2 *cd-form*)
                                        )
```

```
                                                                (
                                                                        (test (or (eq *part-of-speech*
'wobject) (eq *part-of-speech* 'person)))
                                                                        (assign diagnostic-var2 *cd-form*
                                                                        *object-complement* diagnostic-
var2)
                                                                )
                                                        )
                                                )
                                                (
                                                        (test (eq *part-of-speech* 'qualification))
                                                        (next-packet
                                                                (
                                                                        (test (eq *part-of-speech* 'wobject))
                                                                        (next-packet
                                                                                (
                                                                                        (test (eq *part-
of-speech* 'relative))
                                                                                        (assign
diagnostic-var2 *cd-form*)
                                                                                )
                                                                                (
                                                                                        (test (or (eq
*part-of-speech* 'wobject) (eq *part-of-speech* 'person)))
                                                                                        (assign
diagnostic-var2 *cd-form*
                                                                                        *object-
complement* diagnostic-var2)
                                                                                )
                                                                        )
                                                                )
                                                        )
                                                )
                                        )
                                )
                        )
                )
        )
  )
)

;;;;;;;;;;;;;;;; POS ;;;;;;;;;;;;;;;;;;;;;;;;

(defword TO-STAY-AT
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'interrogation) (eq *part-of-speech* 'wobject)))
        (assign    *part-of-speech* 'action
                        *cd-form* '(pos       (action-details ?stay-at-var0)
                                                (actor  ?stay-at-var1)
                                                (object ?stay-at-var2)
                                                (from ?stay-at-var3)
                                                (to ?stay-at-var3))
                        stay-at-var0 *word*
                        stay-at-var1 *subject*
                        stay-at-var2 *subject*
                        stay-at-var3 *object-complement*)
        (next-packet
```

```
                        (
                                    (test (eq *part-of-speech* 'position))
                                    (assign stay-at-var3 *cd-form*)
                        )
            )
  )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;; NOTHING ;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword TO-PRE-VERB
  (
            (test (or (eq *part-of-speech* 'relative) (eq *part-of-speech* 'action)))
            (assign    *part-of-speech* 'action
                                    *cd-form* '?to-pre-verb-var0
                                    to-pre-verb-var0 nil)
            (next-packet
                        (
                                    (test (eq *part-of-speech* 'action))
                                    (assign to-pre-verb-var0 *cd-form*)
                        )
            )
  )
)

(defword TO-DO
  (
            (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'action) (eq *part-of-speech* 'relative) (eq *part-of-
speech* 'wobject)))
            (assign    *part-of-speech* 'action
                                    *cd-form* '?do-var0)
            (next-packet
                        (
                                    (assign do-var0 *cd-form*)
                        )
            )
  )
  (
            (test (eq *part-of-speech* 'interrogation))
            (assign *part-of-speech* 'interrogation
                        *cd-form* '?do-var0)
            (next-packet
                        (
                         (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'pointer)))
                         (assign *subject* *cd-form*)
                         (next-packet
                                    (
                                     (test (eq *part-of-speech* 'action))
                                     (assign do-var0 *cd-form*)
                                    )
                         )
                        )
            )
   )
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define affirmation concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword *START-SENTENCE*
	(
		(assign
			*part-of-speech* 'affirmation
			*cd-form* nil
			*subject* nil
			*concept* nil
			*object-complement* nil
			*receptor-complement* nil
			*predicates* nil
		)
		(next-packet
			(
				(test (equal *part-of-speech* 'person))
				(assign *subject* *cd-form*)
				(next-packet
					(
						(test (equal *part-of-speech* 'action))
						(assign *concept* *cd-form*)
					)
				)
			)
			(
				(test (or (equal *part-of-speech* 'interrogation) (equal *part-of-speech* 'affirmation)))
				(assign *concept* *cd-form*)
			)
		)
	)
)

(defword NEGATION
	(
		(assign  *cd-form* '?negation-var1
			negation-var1 nil)
		(next-packet
			(
				(assign negation-var1 *cd-form*)
			)
		)
	)
)

(defword YES
	(
		(assign  *cd-form* '?yes-var1
			yes-var1 nil)
		(next-packet
			(
				(assign yes-var1 *cd-form*)
			)
		)
	)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define consequence concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define coordination concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define direction concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword IN-THIS-DIRECTION
 (
        (assign    *part-of-speech* 'direction
                   *cd-form* '(DIRECTION (DIRECTOR ?direction-var0) (DIRECTION ?direction-var1))
                   direction-var0 *word*
                   direction-var1 nil)
        (next-packet
              (
                   (test (eq *part-of-speech* 'person))
                   (assign direction-var1 *cd-form*)
              )
              (
                   (test (eq *part-of-speech* 'wobject))
                   (assign direction-var1 *cd-form*)
              )
              (
                   (test (eq *part-of-speech* 'pointer))
                   (assign direction-var1 *cd-form*)
              )
        )
 )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define interrogation concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword DO-INTERROGATION
  (
        (assign *part-of-speech* 'interrogation
              *cd-form* '?do-interrogation-var0)
        (next-packet
              (
                (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'pointer)))
                (assign *subject* *cd-form*)
                (next-packet
                   (
                     (test (eq *part-of-speech* 'action))
                     (assign do-interrogation-var0 *cd-form*)
                   )
                )
              )
        )
  )
)

(defword WHAT-INTERROGATION
  (
        (assign *part-of-speech* 'affirmation
              *cd-form* '?what-interrogation-var0)
        (next-packet
              (
                   (test (eq *part-of-speech* 'interrogation))
                   (assign what-interrogation-var0 *cd-form*)
              )
              (
```

```
                        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject)))
                        (assign *object-complement* *cd-form*)
                        (next-packet
                              (
                                    (test (eq *part-of-speech* 'action))
                                    (assign what-interrogation-var0 *cd-form*)
                              )
                        )
                  )
            )
  )
)

(defword WHO-INTERROGATION
  (
            (assign *part-of-speech* 'interrogation
                  *cd-form* '?who-interrogation-var0)
            (next-packet
                  (
                        (test (eq *part-of-speech* 'action))
                        (assign who-interrogation-var0 *cd-form*)
                  )
            )
  )
)

(defword CAN-INTERROGATION
  (
            (assign *part-of-speech* 'interrogation
                  *cd-form* '?can-interrogation-var2
                  can-interrogation-var0 nil
                  can-interrogation-var1 nil
                  can-interrogation-var2 nil
                  can-interrogation-var3 nil
                  can-interrogation-var4 nil)
            (next-packet
                  (
                   (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject) (eq *part-of-speech* 'pointer)))
                   (assign can-interrogation-var1 *cd-form*
                         *subject* *cd-form*)
                   (next-packet
                        (
                         (test (eq *part-of-speech* 'action))
                         (assign can-interrogation-var2 *cd-form*)
                        )
                   )
                  )
            )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define pointer concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword POINTER-A
  (
            (assign *part-of-speech* 'pointer
                  *cd-form* '(DETERMINATION (DETERMINER ?a-pointer-var0) (DETERMINED ?a-pointer-var1))
                  a-pointer-var0 *word*
```

```
                a-pointer-var1 nil)
        (next-packet
                (
                (test (eq *part-of-speech* 'wobject))
                (assign a-pointer-var1 *cd-form*)
                )
                (
                (test (eq *part-of-speech* 'qualification))
                (assign a-pointer-var1 *cd-form*)
                )
        )
  )
)

(defword POINTER-THIS
  (
        (assign *part-of-speech* 'pointer
                *cd-form* '(DETERMINATION (DETERMINER ?this-var0) (DETERMINED ?this-var1))
                this-var0 *word*
                this-var1 nil)
        (next-packet
                (
                (test (eq *part-of-speech* 'wobject))
                (assign this-var1 *cd-form*)
                )
                (
                (test (eq *part-of-speech* 'qualification))
                (assign this-var1 *cd-form*)
                )
        )
  )
)

(defword POINTER-THAT
  (
        (assign *part-of-speech* 'pointer
                *cd-form* '(DETERMINATION (DETERMINER ?THAT-var0) (DETERMINED ?THAT-var1))
                THAT-var0 *word*
                THAT-var1 nil)
        (next-packet
                (
                (test (eq *part-of-speech* 'wobject))
                (assign THAT-var1 *cd-form*)
                )
                (
                (test (eq *part-of-speech* 'qualification))
                (assign THAT-var1 *cd-form*)
                )
        )
  )
)

(defword POINTER-THE
  (
        (assign *part-of-speech* 'pointer
                *cd-form* '(DETERMINATION (DETERMINER ?the-var0) (DETERMINED ?the-var1))
                the-var0 *word*
                the-var1 nil)
```

```
        (next-packet
                (
                (test (eq *part-of-speech* 'wobject))
                (assign the-var1 *cd-form*)
                )
                (
                (test (eq *part-of-speech* 'qualification))
                (assign the-var1 *cd-form*)
                )
        )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define position concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword INSIDE-SOMETHING
  (
        (assign    *part-of-speech* 'position
                        *cd-form* '(POSITION (POSITIONER ?inside-var0) (POSITION ?inside-var1))
                        inside-var0 *word*
                        inside-var1 nil)
        (next-packet
                (
                        (assign inside-var1 *cd-form*)
                )
        )
  )
)

(defword FROM-SOMETHING
  (
        (assign    *part-of-speech* 'position
                        *cd-form* '(POSITION (POSITIONER ?from-var0) (POSITION ?from-var1))
                        from-var0 *word*
                        from-var1 nil)
        (next-packet
                (
                        (assign from-var1 *cd-form*)
                )
        )
  )
)

(defword ABOUT-SOMETHING
  (
        (assign    *part-of-speech* 'position
                        *cd-form* '(POSITION (POSITIONER ?about-var0) (POSITION ?about-var1))
                        about-var0 *word*
                        about-var1 nil)
        (next-packet
                (
                        (assign about-var1 *cd-form*)
                )
        )
  )
)

(defword ON-SOMETHING
```

```
 (
            (assign    *part-of-speech* 'position
                                   *cd-form* '(POSITION (POSITIONER ?on-var0) (POSITION ?on-var1))
                                   on-var0 *word*
                                   on-var1 nil)
            (next-packet
                        (
                                   (assign on-var1 *cd-form*)
                        )
            )
 )
)

(defword WITH-SOMETHING
 (
            (assign    *part-of-speech* 'position
                                   *cd-form* '(POSITION (POSITIONER ?with-var0) (POSITION ?with-var1))
                                   with-var0 *word*
                                   with-var1 nil)
            (next-packet
                        (
                                   (assign with-var1 *cd-form*)
                        )
            )
 )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define qualification concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword RELATED-TO-EVERYTHING
  (
            (assign *part-of-speech* 'qualification
                        *cd-form* '(QUALIFICATION (QUALIFYER ?all-var0) (QUALIFIED ?all-var1))
                        all-var0 *word*
                        all-var1 nil)
            (next-packet
                        (
                        (assign all-var1 *cd-form*)
                        )
            )
  )
)

(defword RELATED-TO-EXACT
  (
            (assign *part-of-speech* 'qualification
                        *cd-form* '(QUALIFICATION (QUALIFYER ?exact-var0) (QUALIFIED ?exact-var1))
                        exact-var0 *word*
                        exact-var1 nil)
            (next-packet
                        (
                        (assign exact-var1 *cd-form*)
                        )
            )
  )
)

(defword RELATED-TO-WRONG
```

```
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?wrong-var0) (QUALIFIED ?wrong-var1))
                wrong-var0 *word*
                wrong-var1 nil)
        (next-packet
                (
                (assign wrong-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-PORTUGUESE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?portuguese-var0) (QUALIFIED ?portuguese-var1))
                portuguese-var0 *word*
                portuguese-var1 nil)
        (next-packet
                (
                (assign portuguese-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ENGLISH
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?english-var0) (QUALIFIED ?english-var1))
                english-var0 *word*
                english-var1 nil)
        (next-packet
                (
                (assign english-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-FRENCH
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FRENCH-var0) (QUALIFIED ?RELATED-
TO-FRENCH-var1))
                RELATED-TO-FRENCH-var0 *word*
                RELATED-TO-FRENCH-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-FRENCH-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-PROGRAMMING
  (
```

```
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?related-programming-var0) (QUALIFIED ?related-
programming-var1))
                related-programming-var0 *word*
                related-programming-var1 nil)
        (next-packet
                (
                (assign related-programming-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-LISP
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?lisp-var0) (QUALIFIED ?lisp-var1))
                lisp-var0 *word*
                lisp-var1 nil)
        (next-packet
                (
                (assign lisp-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-C++
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?c++-var0) (QUALIFIED ?c++-var1))
                c++-var0 *word*
                c++-var1 nil)
        (next-packet
                (
                (assign c++-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ASP
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ASP-var0) (QUALIFIED ?RELATED-TO-
ASP-var1))
                RELATED-TO-ASP-var0 *word*
                RELATED-TO-ASP-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ASP-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-PASCAL
  (
```

```
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-PASCAL-var0) (QUALIFIED ?RELATED-TO-
PASCAL-var1))
                RELATED-TO-PASCAL-var0 *word*
                RELATED-TO-PASCAL-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-PASCAL-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-VISUAL-BASIC
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-VISUAL-BASIC-var0) (QUALIFIED
?RELATED-TO-VISUAL-BASIC-var1))
                RELATED-TO-VISUAL-BASIC-var0 *word*
                RELATED-TO-VISUAL-BASIC-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-VISUAL-BASIC-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-CMM
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-CMM-var0) (QUALIFIED ?RELATED-TO-
CMM-var1))
                RELATED-TO-CMM-var0 *word*
                RELATED-TO-CMM-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-CMM-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ISO-14000
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ISO-14000-var0) (QUALIFIED ?RELATED-
TO-ISO-14000-var1))
                RELATED-TO-ISO-14000-var0 *word*
                RELATED-TO-ISO-14000-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ISO-14000-var1 *cd-form*)
                )
        )
  )
)
```

```
(defword RELATED-TO-ISO-9000
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ISO-9000-var0) (QUALIFIED ?RELATED-
TO-ISO-9000-var1))
                RELATED-TO-ISO-9000-var0 *word*
                RELATED-TO-ISO-9000-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ISO-9000-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ISO-9001
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ISO-9001-var0) (QUALIFIED ?RELATED-
TO-ISO-9001-var1))
                RELATED-TO-ISO-9001-var0 *word*
                RELATED-TO-ISO-9001-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ISO-9001-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-PDCA
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-PDCA-var0) (QUALIFIED ?RELATED-TO-
PDCA-var1))
                RELATED-TO-PDCA-var0 *word*
                RELATED-TO-PDCA-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-PDCA-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-TQM
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-TQM-var0) (QUALIFIED ?RELATED-TO-
TQM-var1))
                RELATED-TO-TQM-var0 *word*
                RELATED-TO-TQM-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-TQM-var1 *cd-form*)
                )
        )
  )
```

```
)

(defword RELATED-TO-CLINICAS
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-CLINICAS-var0) (QUALIFIED ?RELATED-
TO-CLINICAS-var1))
                RELATED-TO-CLINICAS-var0 *word*
                RELATED-TO-CLINICAS-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-CLINICAS-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-NECK
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-NECK-var0) (QUALIFIED ?RELATED-TO-
NECK-var1))
                RELATED-TO-NECK-var0 *word*
                RELATED-TO-NECK-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-NECK-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-HEART
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-HEART-var0) (QUALIFIED ?RELATED-TO-
HEART-var1))
                RELATED-TO-HEART-var0 *word*
                RELATED-TO-HEART-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-HEART-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ARM
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ARM-var0) (QUALIFIED ?RELATED-TO-
ARM-var1))
                RELATED-TO-ARM-var0 *word*
                RELATED-TO-ARM-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ARM-var1 *cd-form*)
                )
```

```
        )
  )
)

(defword RELATED-TO-LEG
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-LEG-var0) (QUALIFIED ?RELATED-TO-
LEG-var1))
                RELATED-TO-LEG-var0 *word*
                RELATED-TO-LEG-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-LEG-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-KNEE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-KNEE-var0) (QUALIFIED ?RELATED-TO-
KNEE-var1))
                RELATED-TO-KNEE-var0 *word*
                RELATED-TO-KNEE-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-KNEE-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ELBOW
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ELBOW-var0) (QUALIFIED ?RELATED-TO-
ELBOW-var1))
                RELATED-TO-ELBOW-var0 *word*
                RELATED-TO-ELBOW-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ELBOW-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-SHOULDER
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-SHOULDER-var0) (QUALIFIED ?RELATED-
TO-SHOULDER-var1))
                RELATED-TO-SHOULDER-var0 *word*
                RELATED-TO-SHOULDER-var1 nil)
        (next-packet
                (
```

```
                    (assign RELATED-TO-SHOULDER-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-CHEST
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-CHEST-var0) (QUALIFIED ?RELATED-TO-
CHEST-var1))
                    RELATED-TO-CHEST-var0 *word*
                    RELATED-TO-CHEST-var1 nil)
          (next-packet
                    (
                    (assign RELATED-TO-CHEST-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-CHIRURGIC
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-CHIRURGIC-var0) (QUALIFIED ?RELATED-
TO-CHIRURGIC-var1))
                    RELATED-TO-CHIRURGIC-var0 *word*
                    RELATED-TO-CHIRURGIC-var1 nil)
          (next-packet
                    (
                    (assign RELATED-TO-CHIRURGIC-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-FOOT
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FOOT-var0) (QUALIFIED ?RELATED-TO-
FOOT-var1))
                    RELATED-TO-FOOT-var0 *word*
                    RELATED-TO-FOOT-var1 nil)
          (next-packet
                    (
                    (assign RELATED-TO-FOOT-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-BACK
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-BACK-var0) (QUALIFIED ?RELATED-TO-
BACK-var1))
                    RELATED-TO-BACK-var0 *word*
                    RELATED-TO-BACK-var1 nil)
```

```
        (next-packet
                (
                (assign RELATED-TO-BACK-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-WRIST
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-WRIST-var0) (QUALIFIED ?RELATED-TO-
WRIST-var1))
                RELATED-TO-WRIST-var0 *word*
                RELATED-TO-WRIST-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-WRIST-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-STOMACH
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-STOMACH-var0) (QUALIFIED ?RELATED-
TO-STOMACH-var1))
                RELATED-TO-STOMACH-var0 *word*
                RELATED-TO-STOMACH-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-STOMACH-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-LUNG
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-LUNG-var0) (QUALIFIED ?RELATED-TO-
LUNG-var1))
                RELATED-TO-LUNG-var0 *word*
                RELATED-TO-LUNG-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-LUNG-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-HIP
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-HIP-var0) (QUALIFIED ?RELATED-TO-HIP-
var1))
```

```
                RELATED-TO-HIP-var0 *word*
                RELATED-TO-HIP-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-HIP-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-SEVERE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-SEVERE-var0) (QUALIFIED ?RELATED-
TO-SEVERE-var1))
                RELATED-TO-SEVERE-var0 *word*
                RELATED-TO-SEVERE-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-SEVERE-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ACUTE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ACUTE-var0) (QUALIFIED ?RELATED-TO-
ACUTE-var1))
                RELATED-TO-ACUTE-var0 *word*
                RELATED-TO-ACUTE-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ACUTE-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-HIGH
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-HIGH-var0) (QUALIFIED ?RELATED-TO-
HIGH-var1))
                RELATED-TO-HIGH-var0 *word*
                RELATED-TO-HIGH-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-HIGH-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-LOW
  (
        (assign *part-of-speech* 'qualification
```

```
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-LOW-var0) (QUALIFIED ?RELATED-TO-
LOW-var1))
                RELATED-TO-LOW-var0 *word*
                RELATED-TO-LOW-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-LOW-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-PERSISTENT
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-PERSISTENT-var0) (QUALIFIED
?RELATED-TO-PERSISTENT-var1))
                RELATED-TO-PERSISTENT-var0 *word*
                RELATED-TO-PERSISTENT-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-PERSISTENT-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-FILIPE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FILIPE-var0) (QUALIFIED ?RELATED-TO-
FILIPE-var1))
                RELATED-TO-FILIPE-var0 *word*
                RELATED-TO-FILIPE-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-FILIPE-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-FLAVIO
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FLAVIO-var0) (QUALIFIED ?RELATED-TO-
FLAVIO-var1))
                RELATED-TO-FLAVIO-var0 *word*
                RELATED-TO-FLAVIO-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-FLAVIO-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-AGENT
```

```
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-AGENT-var0) (QUALIFIED ?RELATED-TO-
AGENT-var1))
                RELATED-TO-AGENT-var0 *word*
                RELATED-TO-AGENT-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-AGENT-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-DANI
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-DANI-var0) (QUALIFIED ?RELATED-TO-
DANI-var1))
                RELATED-TO-DANI-var0 *word*
                RELATED-TO-DANI-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-DANI-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ROBERTO
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ROBERTO-var0) (QUALIFIED ?RELATED-
TO-ROBERTO-var1))
                RELATED-TO-ROBERTO-var0 *word*
                RELATED-TO-ROBERTO-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ROBERTO-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-MARCIO
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-MARCIO-var0) (QUALIFIED ?RELATED-TO-
MARCIO-var1))
                RELATED-TO-MARCIO-var0 *word*
                RELATED-TO-MARCIO-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-MARCIO-var1 *cd-form*)
                )
        )
  )
)
```

```
(defword RELATED-TO-DOMINIQUE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-DOMINIQUE-var0) (QUALIFIED
?RELATED-TO-DOMINIQUE-var1))
                RELATED-TO-DOMINIQUE-var0 *word*
                RELATED-TO-DOMINIQUE-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-DOMINIQUE-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-JEAN
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-JEAN-var0) (QUALIFIED ?RELATED-TO-
JEAN-var1))
                RELATED-TO-JEAN-var0 *word*
                RELATED-TO-JEAN-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-JEAN-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-JOHN
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-JOHN-var0) (QUALIFIED ?RELATED-TO-
JOHN-var1))
                RELATED-TO-JOHN-var0 *word*
                RELATED-TO-JOHN-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-JOHN-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-JACK
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-JACK-var0) (QUALIFIED ?RELATED-TO-
JACK-var1))
                RELATED-TO-JACK-var0 *word*
                RELATED-TO-JACK-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-JACK-var1 *cd-form*)
                )
        )
```

```
  )
)

(defword RELATED-TO-DIABETIC
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-DIABETIC-var0) (QUALIFIED ?RELATED-
TO-DIABETIC-var1))
                RELATED-TO-DIABETIC-var0 *word*
                RELATED-TO-DIABETIC-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-DIABETIC-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-INSULIN
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-INSULIN-var0) (QUALIFIED ?RELATED-TO-
INSULIN-var1))
                RELATED-TO-INSULIN-var0 *word*
                RELATED-TO-INSULIN-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-INSULIN-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-AIDS
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-AIDS-var0) (QUALIFIED ?RELATED-TO-
AIDS-var1))
                RELATED-TO-AIDS-var0 *word*
                RELATED-TO-AIDS-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-AIDS-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-ANALAB
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-ANALAB-var0) (QUALIFIED ?RELATED-TO-
ANALAB-var1))
                RELATED-TO-ANALAB-var0 *word*
                RELATED-TO-ANALAB-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-ANALAB-var1 *cd-form*)
```

```
                )
        )
  )
)

(defword RELATED-TO-SANTA-FELICIDADE
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-SANTA-FELICIDADE-var0) (QUALIFIED
?RELATED-TO-SANTA-FELICIDADE-var1))
                RELATED-TO-SANTA-FELICIDADE-var0 *word*
                RELATED-TO-SANTA-FELICIDADE-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-SANTA-FELICIDADE-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-INCFABSOFT
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-INCFABSOFT-var0) (QUALIFIED
?RELATED-TO-INCFABSOFT-var1))
                RELATED-TO-INCFABSOFT-var0 *word*
                RELATED-TO-INCFABSOFT-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-INCFABSOFT-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-FABSOFTA
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FABSOFTA-var0) (QUALIFIED ?RELATED-
TO-FABSOFTA-var1))
                RELATED-TO-FABSOFTA-var0 *word*
                RELATED-TO-FABSOFTA-var1 nil)
        (next-packet
                (
                (assign RELATED-TO-FABSOFTA-var1 *cd-form*)
                )
        )
  )
)

(defword RELATED-TO-FABSOFTB
  (
        (assign *part-of-speech* 'qualification
                *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FABSOFTB-var0) (QUALIFIED ?RELATED-
TO-FABSOFTB-var1))
                RELATED-TO-FABSOFTB-var0 *word*
                RELATED-TO-FABSOFTB-var1 nil)
        (next-packet
```

```
                    (
                    (assign RELATED-TO-FABSOFTB-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-FABSOFTC
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FABSOFTC-var0) (QUALIFIED ?RELATED-
TO-FABSOFTC-var1))
                    RELATED-TO-FABSOFTC-var0 *word*
                    RELATED-TO-FABSOFTC-var1 nil)
          (next-packet
                    (
                    (assign RELATED-TO-FABSOFTC-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-FAZENDINHA
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-FAZENDINHA-var0) (QUALIFIED
?RELATED-TO-FAZENDINHA-var1))
                    RELATED-TO-FAZENDINHA-var0 *word*
                    RELATED-TO-FAZENDINHA-var1 nil)
          (next-packet
                    (
                    (assign RELATED-TO-FAZENDINHA-var1 *cd-form*)
                    )
          )
  )
)

(defword RELATED-TO-BATEL
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-BATEL-var0) (QUALIFIED ?RELATED-TO-
BATEL-var1))
                    RELATED-TO-BATEL-var0 *word*
                    RELATED-TO-BATEL-var1 nil)
          (next-packet
                    (
                    (assign RELATED-TO-BATEL-var1 *cd-form*)
                    )
          )
  )
)


(defword RELATED-TO-BLOOD
  (
          (assign *part-of-speech* 'qualification
                    *cd-form* '(QUALIFICATION (QUALIFYER ?RELATED-TO-BLOOD-var0) (QUALIFIED ?RELATED-TO-
BLOOD-var1))
```

```
                        RELATED-TO-BLOOD-var0 *word*
                        RELATED-TO-BLOOD-var1 nil)
            (next-packet
                        (
                        (assign RELATED-TO-BLOOD-var1 *cd-form*)
                        )
            )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define relative concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword RELATIVE-WHAT
  (
            (assign *part-of-speech* 'relative
                        *cd-form* '(RELATIVE (RELATIVISOR ?what-var0) (RELATIVISED ?what-var1))
                        what-var0 *word*
                        what-var1 nil
                        *subject* *subject*
                        *object-complement* nil
                        *receptor-complement* nil)
            (next-packet
                        (
                                    (test (equal *part-of-speech* 'action))
                                    (assign *subject* *subject* ; assign *subject* nil permet d'avoir un WHAT TO plus "neutre"
                                                what-var1 *cd-form*)
                        )
                        (
                                    (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject) (eq *part-of-speech*
'pointer)))
                                    (assign *subject* *cd-form*)
                                    (next-packet
                                                (
                                                            (test (equal *part-of-speech* 'action))
                                                            (assign what-var1 *cd-form*)
                                                )
                                    )
                        )
            )
  )
)


(defword RELATIVE-HOW
  (
            (assign *part-of-speech* 'relative
                        *cd-form* '(RELATIVE (RELATIVISOR ?how-var0) (RELATIVISED ?how-var1))
                        how-var0 *word*
                        how-var1 nil
                        *object-complement* nil
                        *receptor-complement* nil)
            (next-packet
                        (
                                    (test (equal *part-of-speech* 'action))
                                    (assign *subject* *subject* ; assign *subject* nil permet d'avoir un how to plus "neutre"
                                                how-var1 *cd-form*)
                        )
                        (
```

```
                        (test (equal *part-of-speech* 'person))
                        (assign *subject* *cd-form*)
                        (next-packet
                                (
                                        (test (equal *part-of-speech* 'action))
                                        (assign how-var1 *cd-form*)
                                )
                        )
                )
        )
  )
)

(defword RELATIVE-THAT
  (
        (test (eq *part-of-speech* 'action))
        (assign *part-of-speech* 'relative
                *cd-form* '(RELATIVE (RELATIVISOR ?that-var0) (RELATIVISED ?that-var1))
                that-var0 *word*
                that-var1 nil
                *subject* *subject*
                *object-complement* nil
                *receptor-complement* nil)
        (next-packet
                (
                        (test (equal *part-of-speech* 'action))
                        (assign *subject* *subject* ; assign *subject* nil permet d'avoir un THAT TO plus "neutre"
                                that-var1 *cd-form*)
                )
                (
                        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject) (eq *part-of-speech*
'pointer)))
                        (assign *subject* *cd-form*)
                        (next-packet
                                (
                                        (test (equal *part-of-speech* 'action))
                                        (assign that-var1 *cd-form*)
                                )
                        )
                )
        )
  )
  (
        (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject)))
        (assign *part-of-speech* 'relative
                *cd-form* '(RELATIVE (RELATIVISOR ?that-var0) (RELATIVISED ?that-var1))
                that-var0 *word*
                that-var1 nil
                *subject* *object-complement*
                *object-complement* nil
                *receptor-complement* nil)
        (next-packet
                (
                        (test (equal *part-of-speech* 'action))
                        (assign *subject* *subject* ; assign *subject* nil permet d'avoir un THAT TO plus "neutre"
                                that-var1 *cd-form*)
                )
                (
```

```
                              (test (or (eq *part-of-speech* 'person) (eq *part-of-speech* 'wobject) (eq *part-of-speech*
'pointer)))
                              (assign *subject* *cd-form*)
                              (next-packet
                                    (
                                          (test (equal *part-of-speech* 'action))
                                          (assign that-var1 *cd-form*)
                                    )
                              )
                        )
                  )
  )
)

(defword RELATIVE-WHO
 (
            (test (eq *part-of-speech* 'action))
            (assign *part-of-speech* 'relative
                        *cd-form* '(RELATIVE (RELATIVISOR ?who-var0) (RELATIVISED ?who-var1))
                        who-var0 *word*
                        who-var1 nil
                        *subject* nil
                        *object-complement* nil
                        *receptor-complement* nil)
            (next-packet
                        (
                        (test (equal *part-of-speech* 'action))
                        (assign *subject* *subject* ; assign *subject* nil permet d'avoir un WHO TO plus "neutre"
                              who-var1 *cd-form*)
                        )
            )
  )
 (
            (test (eq *part-of-speech* 'person))
            (assign *part-of-speech* 'relative
                        *cd-form* '(RELATIVE (RELATIVISOR ?who-var0) (RELATIVISED ?who-var1))
                        who-var0 *word*
                        who-var1 nil
                        *subject* *object-complement*
                        *object-complement* nil
                        *receptor-complement* nil)
            (next-packet
                        (
                        (test (equal *part-of-speech* 'action))
                        (assign who-var1 *cd-form*)
                        )
            )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define wobject concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword *UNKNOWN-WORD*
  (
            (assign    *part-of-speech* 'wobject
                              *cd-form* *WORD* )
  )
)
```

```
(defword MEASURABLE-VALUE
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)


(defword CHANGABLE-VALUE
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)

(defword LIST-OF-PROGRAMMED-STEPS
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)

(defword LIST-OF-SENTENCE
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)

(defword NORM-OF-EXPRESSION
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)

(defword PLACE-TO-STORE-THINGS
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)

(defword UNDERSTANDING-OF-WORLD
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)

(defword BOOK
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD*  )
  )
)
```

```
(defword ERROR
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword HABILITY-RECOGNIZING-DOCUMENT
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword IDEAL-REPRESENTATION-OF-REALITY
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword NON-CONFORMITY
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
(defword ORGANIZED-GROUP-OF-PEOPLE
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
(defword PROCEDURE-FOR-AUDITING
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
(defword PROCEDURE-FOR-TESTING
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
(defword STEPS-FOR-TEACHING-LEARNING
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
(defword TRIAL-EXERCISE
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
(defword VISION-OF-WORLD
```

```
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword ANESTHESIA
  (
        (assign *part-of-speech* 'wobject
                *cd-form* *WORD*)
  )
)


(defword PHYSICAL-DISORDER
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PHYSICAL-DISORDER-SIGN
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PHYSICAL-DIFFICULTY
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PHYSICAL-OPERATION
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword MOVING-IMPOSSIBILITY
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PHYSICAL-CRACK
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PRESSURE
  (
```

```
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword ATTACK
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword BLOOD
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword BLOOD-ESCAPE
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword BLOOD-TRANSFUSION
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword DISORDER-AGGRAVATION
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword NECK
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword HEART
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword ARM
  (
        (assign    *part-of-speech* 'wobject
                        *cd-form* *WORD* )
```

```
  )
)

(defword LEG
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword KNEE
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword ELBOW
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword SHOULDER
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword CHEST
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword FOOT
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword BACK
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword WRIST
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)
```

```
(defword STOMACH
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword LUNG
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword HIP
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword HOLE
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword MASSAGE
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PHYSICAL-INJECTION
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword INSULIN
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PLACE-TO-LIVE
  (
        (assign   *part-of-speech* 'wobject
                        *cd-form* *WORD* )
  )
)

(defword PLACE-TO-CURE
```

```
  (
            (assign    *part-of-speech* 'wobject
                                  *cd-form* *WORD* )
  )
)


(defword COMPANIES-GROUPING-SYSTEM
  (
            (assign    *part-of-speech* 'wobject
                                  *cd-form* *WORD* )
  )
)

(defword REALIZATION-OF-SOMETHING
  (
            (assign    *part-of-speech* 'wobject
                                  *cd-form* *WORD* )
  )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; define person concepts ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defword PERSON-NAMED-AGENT
  (
            (assign    *part-of-speech* 'person
                                  *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-DOMINIQUE
  (
            (assign    *part-of-speech* 'person
                                  *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-JEAN
  (
            (assign    *part-of-speech* 'person
                                  *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-MARCIO
  (
            (assign    *part-of-speech* 'person
                                  *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-ROBERTO
  (
            (assign    *part-of-speech* 'person
                                  *cd-form* *WORD* )
  )
)
```

```
(defword PERSON-NAMED-DANI
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-FILIPE
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-FLAVIO
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-JOHN
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword PERSON-NAMED-JACK
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword MY-SELF
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword YOUR-SELF
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword OUR-SELVES
  (
        (assign    *part-of-speech* 'person
                        *cd-form* *WORD* )
  )
)

(defword THIS-MAN
```

```
  (
          (assign    *part-of-speech* 'person
                         *cd-form* *WORD* )
  )
)

(defword THIS-WOMAN
  (
          (assign    *part-of-speech* 'person
                         *cd-form* *WORD* )
  )
)

(defword THESE-PEOPLE
  (
          (assign    *part-of-speech* 'person
                         *cd-form* *WORD* )
  )
)

(defword BENCH-OF-PERSON
  (
          (assign    *part-of-speech* 'person
                         *cd-form* *WORD* )
  )
)

(defword UNDEFINED-PERSON
  (
          (assign    *part-of-speech* 'person
                         *cd-form* *WORD* )
  )
)

(defword UNDEFINED-BENCH-OF-PERSON
  (
          (assign    *part-of-speech* 'person
                         *cd-form* *WORD* )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

English-CDs-Translation.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; Dictionary of CDs Translation into Natual English ;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 (require :cd-functions)
(require :english-generator)

;;;;;;;;;;;;;;;;;;;;;; Dictionary ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;  atrans
;  ptrans
;  grasp
;  ingest
;  pos goes "stays at ..."
(defun say-pos (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (not (NULL (cdpath '(actor) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEONE) RESULT)))
                (setq RESULT (add '(STAYS AT) RESULT))
                (if (and (not (NULL (cdpath '(to) cd))) (equal (cdpath '(to) cd) (cdpath '(from) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(to)   cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEWHERE) RESULT)))
        )
)

;  expel goes "exhibits ..."
(defun say-expel (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (not (NULL (cdpath '(actor) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEONE) RESULT)))
                (setq RESULT (add '(HAS EXHIBITED) RESULT))
                (if (not (NULL (cdpath '(object) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(object)   cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMETHING) RESULT)))
        )
)


;  mtrans goes "prescribe..."

(defun say-mtrans (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (not (NULL (cdpath '(actor) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEONE) RESULT)))
                (setq RESULT (add '(CAN PRESCRIBE) RESULT))
                (if (not (NULL (cdpath '(to) cd)))
                        (setq RESULT (add (append '(TO)   (say-cd-english (cdpath '(to)   cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(from) cd)))
                        (setq RESULT (add (append '(FROM) (say-cd-english (cdpath '(from) cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(object) cd)))
                        (if (atom (cdpath '(object) cd))
```

```
                                        (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC CONCEPT-BASE)
RESULT))
                                        (if (and (member (car (cdpath '(object) cd)) *cd-acts*) (not (eq (car (cdpath '(object)
cd)) 'QUALIFICATION)))
                                                (setq RESULT (add (append '(THAT) (say-cd-english (cdpath '(object) cd)
LEXIC CONCEPT-BASE)) RESULT))
                                                (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC
CONCEPT-BASE) RESULT))))
                                (setq RESULT (add '(SOMETHING) RESULT)))
        )
)

;  mbuild goes "diagnose..."

(defun say-mbuild (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (not (NULL (cdpath '(actor) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEONE) RESULT)))
                (setq RESULT (add '(CAN DIAGNOSE) RESULT))
                (if (and (not (NULL (cdpath '(to) cd))) (not (equal (cdpath '(to) cd) (cdpath '(actor) cd))))
                        (setq RESULT (add (append '(TO)   (say-cd-english (cdpath '(to)   cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(from) cd)))
                        (setq RESULT (add (append '(IN) (say-cd-english (cdpath '(from) cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(object) cd)))
                        (if (atom (cdpath '(object) cd))
                                (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC CONCEPT-BASE)
RESULT))
                                (if (and (member (car (cdpath '(object) cd)) *cd-acts*) (not (eq (car (cdpath '(object)
cd)) 'QUALIFICATION)))
                                        (setq RESULT (add (append '(THAT) (say-cd-english (cdpath '(object) cd)
LEXIC CONCEPT-BASE)) RESULT))
                                        (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC
CONCEPT-BASE) RESULT))))
                        (setq RESULT (add '(SOMETHING) RESULT)))
        )
)

;  kposs goes "knows..."

(defun say-kposs (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (not (NULL (cdpath '(actor) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEONE) RESULT)))
                (setq RESULT (add '(KNOWS) RESULT))
                (if (and (not (NULL (cdpath '(to) cd))) (not (equal (cdpath '(to) cd) (cdpath '(actor) cd))))
                        (setq RESULT (add (append '(TO)   (say-cd-english (cdpath '(to)   cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(from) cd)))
                        (setq RESULT (add (append '(FROM) (say-cd-english (cdpath '(from) cd) LEXIC CONCEPT-
BASE)) RESULT)))
```

```
                (if (not (NULL (cdpath '(object) cd)))
                        (if (atom (cdpath '(object) cd))
                                (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC CONCEPT-BASE)
RESULT))
                                (if (and (member (car (cdpath '(object) cd)) *cd-acts*) (not (eq (car (cdpath '(object)
cd)) 'QUALIFICATION)))
                                        (setq RESULT (add (append '(THAT) (say-cd-english (cdpath '(object) cd)
LEXIC CONCEPT-BASE)) RESULT))
                                        (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC
CONCEPT-BASE) RESULT))))
                        (setq RESULT (add '(SOMETHING) RESULT)))
        )
)

;  propel goes "perform..."

(defun say-propel (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (not (NULL (cdpath '(actor) cd)))
                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-BASE) RESULT))
                        (setq RESULT (add '(SOMEONE) RESULT)))
                (setq RESULT (add '(CAN PERFORM) RESULT))
                (if (and (not (NULL (cdpath '(to) cd))) (not (equal (cdpath '(to) cd) (cdpath '(actor) cd))))
                        (setq RESULT (add (append '(ON)   (say-cd-english (cdpath '(to)   cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(from) cd)))
                        (setq RESULT (add (append '(FROM) (say-cd-english (cdpath '(from) cd) LEXIC CONCEPT-
BASE)) RESULT)))
                (if (not (NULL (cdpath '(object) cd)))
                        (if (atom (cdpath '(object) cd))
                                (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC CONCEPT-BASE)
RESULT))
                                (if (and (member (car (cdpath '(object) cd)) *cd-acts*) (not (eq (car (cdpath '(object)
cd)) 'QUALIFICATION)))
                                        (setq RESULT (add (append '(THAT) (say-cd-english (cdpath '(object) cd)
LEXIC CONCEPT-BASE)) RESULT))
                                        (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC
CONCEPT-BASE) RESULT))))
                        (setq RESULT (add '(SOMETHING) RESULT)))
        )
)

;  attend goes "examinate..."

(defun say-attend (cd LEXIC CONCEPT-BASE)
        (let ((RESULT NIL))
                (if (equal (cdpath '(actor) cd) (cdpath '(object) cd))
                        (progn
                                (if (not (NULL (cdpath '(actor) cd)))
                                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-
BASE) RESULT))
                                        (setq RESULT (add '(SOMEONE) RESULT)))
                                (setq RESULT (add '(CAN EXAMINE) RESULT))
                                (if (not (NULL (cdpath '(from) cd)))
```

```
                                        (setq RESULT (add (append '(WITH) (say-cd-english (cdpath '(from) cd)
LEXIC CONCEPT-BASE)) RESULT)))
                                (if (not (NULL (cdpath '(to) cd)))
                                        (if (atom (cdpath '(to) cd))
                                                (setq RESULT (add (say-cd-english (cdpath '(to) cd) LEXIC
CONCEPT-BASE) RESULT))
                                                (if (and (member (car (cdpath '(to) cd)) *cd-acts*) (not (eq (car
(cdpath '(to) cd)) 'QUALIFICATION)))
                                                        (setq RESULT (add (append '(THAT) (say-cd-english
(cdpath '(to) cd) LEXIC CONCEPT-BASE)) RESULT))
                                                        (setq RESULT (add (say-cd-english (cdpath '(to) cd)
LEXIC CONCEPT-BASE) RESULT))))
                                        (setq RESULT (add '(SOMETHING) RESULT)))
                        )
                        (progn
                                (if (not (NULL (cdpath '(actor) cd)))
                                        (setq RESULT (add (say-cd-english (cdpath '(actor) cd) LEXIC CONCEPT-
BASE) RESULT))
                                        (setq RESULT (add '(SOMEONE) RESULT)))
                                (setq RESULT (add '(CAN EXAMINE) RESULT))
                                (if (and (not (NULL (cdpath '(to) cd))) (not (equal (cdpath '(to) cd) (cdpath '(actor)
cd))))
                                        (setq RESULT (add (append '(TO)   (say-cd-english (cdpath '(to)   cd)
LEXIC CONCEPT-BASE)) RESULT)))
                                (if (not (NULL (cdpath '(from) cd)))
                                        (setq RESULT (add (append '(FROM) (say-cd-english (cdpath '(from) cd)
LEXIC CONCEPT-BASE)) RESULT)))
                                (if (not (NULL (cdpath '(object) cd)))
                                        (if (atom (cdpath '(object) cd))
                                                (setq RESULT (add (say-cd-english (cdpath '(object) cd) LEXIC
CONCEPT-BASE) RESULT))
                                                (if (and (member (car (cdpath '(object) cd)) *cd-acts*) (not (eq
(car (cdpath '(object) cd)) 'QUALIFICATION)))
                                                        (setq RESULT (add (append '(THAT) (say-cd-english
(cdpath '(object) cd) LEXIC CONCEPT-BASE)) RESULT))
                                                        (setq RESULT (add (say-cd-english (cdpath '(object)
cd) LEXIC CONCEPT-BASE) RESULT))))
                                        (setq RESULT (add '(SOMETHING) RESULT)))
                        )
                )
        )
)

;  qualification

(defun say-qualification (cd LEXIC CONCEPT-BASE)
        (cond   ((NOT (NULL (cdpath '(qualifyer) cd)))
                (append '(THE)
                        (list (lexic-value (cdpath '(qualifyer) cd) LEXIC))
                        (list (lexic-value (cdpath '(qualified) cd) LEXIC))))
                (T (append '(A) (list (lexic-value (cdpath '(qualified) cd) LEXIC))))
        )
)
```

English-Mapping-Lexic.lxc

```
ENGLISH
(A TINDEF PINDEF GINDEF SING POINTER-A)
(AN TINDEF PINDEF GINDEF SING POINTER-A)
(ABOUT TINDEF PINDEF GINDEF NINDEF ABOUT-SOMETHING)
(ACUTE TINDEF OTHER NEUTRAL SING RELATED-TO-ACUTE)
(AGGRAVATION TINDEF PINDEF NEUTRAL SING DISORDER-AGGRAVATION)
(AGGRAVATIONS TINDEF PINDEF NEUTRAL PLURAL DISORDER-AGGRAVATION)
(AIDS^ TINDEF PINDEF NEUTRAL NINDEF RELATED-TO-AIDS)
(ALGORITHM TINDEF PINDEF NEUTRAL SING LIST-OF-PROGRAMMED-STEPS)
(ALGORITHMS TINDEF PINDEF NEUTRAL PLURAL LIST-OF-PROGRAMMED-STEPS)
(ALL TINDEF PINDEF GINDEF PLURAL RELATED-TO-EVERYTHING)
(ANALAB TINDEF OTHER NEUTRAL SING RELATED-TO-ANALAB)
(ANALYSIS TINDEF PINDEF NEUTRAL SING PROCEDURE-FOR-TESTING)
(ANESTHESIA TINDEF PINDEF NEUTRAL SING ANESTHESIA)
(ARM TINDEF PINDEF NEUTRAL SING ARM)
(ARMS TINDEF PINDEF NEUTRAL PLURAL ARM)
(ARM^S TINDEF OTHER NEUTRAL SING RELATED-TO-ARM)
(ARMS^ TINDEF OTHER NEUTRAL SING RELATED-TO-ARM)
(ASP TINDEF PINDEF GINDEF NINDEF RELATED-TO-ASP)
(AT TINDEF PINDEF GINDEF NINDEF INSIDE-SOMETHING)
(ATTACK TINDEF PINDEF NEUTRAL SING ATTACK)
(ATTACKS TINDEF PINDEF NEUTRAL PLURAL ATTACK)
(ATTRIBUTE PRESENT PINDEF GINDEF NINDEF TO-MAKE-ATTRIBUTION)
(ATTRIBUTED PAST PINDEF GINDEF NINDEF TO-MAKE-ATTRIBUTION)
(ATTRIBUTES PRESENT OTHER GINDEF SING TO-MAKE-ATTRIBUTION)
(AUDIT PRESENT PINDEF GINDEF NINDEF TO-MAKE-AUDIT)
(AUDIT TINDEF PINDEF NEUTRAL SING PROCEDURE-FOR-AUDITING)
(AUDITED PAST PINDEF GINDEF NINDEF TO-MAKE-AUDIT)
(AUDITS PRESENT OTHER GINDEF SING TO-MAKE-AUDIT)
(AUDITS TINDEF PINDEF NEUTRAL PLURAL PROCEDURE-FOR-AUDITING)
(BACK TINDEF PINDEF NEUTRAL SING BACK)
(BACKS TINDEF PINDEF NEUTRAL PLURAL BACK)
(BACK^S TINDEF OTHER NEUTRAL SING RELATED-TO-BACK)
(BACKS^ TINDEF OTHER NEUTRAL SING RELATED-TO-BACK)
(BATEL^S TINDEF OTHER NEUTRAL SING RELATED-TO-BATEL)
(BELIEVE PRESENT PINDEF GINDEF NINDEF TO-HAVE-BELIEF)
(BELIEVED PAST PINDEF GINDEF NINDEF TO-HAVE-BELIEF)
(BELIEVES PRESENT OTHER GINDEF SING TO-HAVE-BELIEF)
(BLOOD TINDEF PINDEF NEUTRAL SING BLOOD)
(BLOOD^S TINDEF PINDEF NEUTRAL SING RELATED-TO-BLOOD)
(BOOK TINDEF PINDEF NEUTRAL SING BOOK)
(BOOKS TINDEF PINDEF NEUTRAL PLURAL BOOK)
(C++ TINDEF PINDEF GINDEF NINDEF RELATED-TO-C++)
(CAN PRESENT PINDEF GINDEF NINDEF CAN-INTERROGATION)
(CAN PRESENT PINDEF GINDEF NINDEF TO-BE-ABLE-TO)
(CERTIFICATE TINDEF PINDEF NEUTRAL SING HABILITY-RECOGNIZING-DOCUMENT)
(CERTIFICATES TINDEF PINDEF NEUTRAL PLURAL HABILITY-RECOGNIZING-DOCUMENT)
(CHEST TINDEF PINDEF NEUTRAL SING CHEST)
(CHESTS TINDEF PINDEF NEUTRAL PLURAL CHEST)
(CHEST^S TINDEF OTHER NEUTRAL SING RELATED-TO-CHEST)
(CHESTS^ TINDEF OTHER NEUTRAL SING RELATED-TO-CHEST)
(CHIRURGICAL TINDEF OTHER NEUTRAL SING RELATED-TO-CHIRURGIC)
(CHRONIC TINDEF OTHER NEUTRAL SING RELATED-TO-PERSISTENT)
(CLINICAS TINDEF OTHER NEUTRAL PLURAL RELATED-TO-CLINICAS)
(CMM TINDEF PINDEF GINDEF NINDEF RELATED-TO-CMM)
```

(COMPANY TINDEF PINDEF NEUTRAL SING ORGANIZED-GROUP-OF-PEOPLE)
(COMPANIES TINDEF PINDEF NEUTRAL PLURAL ORGANIZED-GROUP-OF-PEOPLE)
(COMPLETE PRESENT PINDEF GINDEF NINDEF TO-PERFORM)
(COMPLETED PAST PINDEF GINDEF NINDEF TO-PERFORM)
(COMPLETES PRESENT OTHER GINDEF SING TO-PERFORM)
(COMPLICATION TINDEF PINDEF NEUTRAL SING DISORDER-AGGRAVATION)
(COMPLICATIONS TINDEF PINDEF NEUTRAL PLURAL DISORDER-AGGRAVATION)
(CONSTANT TINDEF OTHER NEUTRAL SING RELATED-TO-PERSISTENT)
(CONTINUAL TINDEF OTHER NEUTRAL SING RELATED-TO-PERSISTENT)
(CORRECT PRESENT PINDEF GINDEF NINDEF TO-CORRECT)
(CORRECTED PAST PINDEF GINDEF NINDEF TO-CORRECT)
(CORRECTS PRESENT OTHER GINDEF SING TO-CORRECT)
(COULD PAST PINDEF GINDEF NINDEF TO-BE-ABLE-TO)
(COURSE TINDEF PINDEF NEUTRAL SING STEPS-FOR-TEACHING-LEARNING)
(COURSES TINDEF PINDEF NEUTRAL PLURAL STEPS-FOR-TEACHING-LEARNING)
(CREATE PRESENT PINDEF GINDEF NINDEF TO-MAKE-CREATION)
(CREATED PAST PINDEF GINDEF NINDEF TO-MAKE-CREATION)
(CREATES PRESENT OTHER GINDEF SING TO-MAKE-CREATION)
(DANI TINDEF OTHER FEM SING PERSON-NAMED-DANI)
(DANI^S TINDEF OTHER FEM SING RELATED-TO-DANI)
(DESCRIBE PRESENT PINDEF GINDEF NINDEF TO-MAKE-DESCRIPTION)
(DESCRIBED PAST PINDEF GINDEF NINDEF TO-MAKE-DESCRIPTION)
(DESCRIBES PRESENT OTHER GINDEF SING TO-MAKE-DESCRIPTION)
(DETECT PRESENT PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(DETECTED PAST PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(DETECTS PRESENT OTHER GINDEF SING TO-MAKE-DIAGNOSTIC)
(DIABETIC TINDEF PINDEF NEUTRAL NINDEF RELATED-TO-DIABETIC)
(DIAGNOSE PRESENT PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(DIAGNOSED PAST PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(DIAGNOSES PRESENT OTHER GINDEF SING TO-MAKE-DIAGNOSTIC)
(DID PAST PINDEF GINDEF NINDEF DO-INTERROGATION)
(DID PAST PINDEF GINDEF NINDEF TO-DO)
(DIFFICULTY TINDEF PINDEF NEUTRAL SING PHYSICAL-DIFFICULTY)
(DIFFICULTIES TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DIFFICULTY)
(DISEASE TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER)
(DISEASES TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER)
(DISORDER TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER)
(DISORDERS TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER)
(DISTRICT TINDEF PINDEF NEUTRAL SING PLACE-TO-LIVE)
(DISTRICTS TINDEF PINDEF NEUTRAL PLURAL PLACE-TO-LIVE)
(DO PRESENT PINDEF GINDEF NINDEF DO-INTERROGATION)
(DO PRESENT PINDEF GINDEF NINDEF TO-DO)
(DOES PRESENT OTHER GINDEF SING DO-INTERROGATION)
(DOES PRESENT OTHER GINDEF SING TO-DO)
(DOMINIQUE TINDEF OTHER FEM SING PERSON-NAMED-DOMINIQUE)
(DOMINIQUE^S TINDEF OTHER FEM SING RELATED-TO-DOMINIQUE)
(ELBOW TINDEF PINDEF NEUTRAL SING ELBOW)
(ELBOWS TINDEF PINDEF NEUTRAL PLURAL ELBOW)
(ELBOW^S TINDEF OTHER NEUTRAL SING RELATED-TO-ELBOW)
(ELBOWS^ TINDEF OTHER NEUTRAL SING RELATED-TO-ELBOW)
(ENGLISH TINDEF PINDEF GINDEF NINDEF RELATED-TO-ENGLISH)
(ERROR TINDEF PINDEF NEUTRAL SING ERROR)
(ERRORS TINDEF PINDEF NEUTRAL PLURAL ERROR)
(EVALUATE PRESENT PINDEF GINDEF NINDEF TO-MAKE-EVALUATION)
(EVALUATED PAST PINDEF GINDEF NINDEF TO-MAKE-EVALUATION)

(EVALUATES PRESENT OTHER GINDEF SING TO-MAKE-EVALUATION)
(EXAMINE PRESENT PINDEF GINDEF NINDEF TO-MAKE-AUDIT)
(EXAMINED PAST PINDEF GINDEF NINDEF TO-MAKE-AUDIT)
(EXAMINES PRESENT OTHER GINDEF SING TO-MAKE-AUDIT)
(EXACT TINDEF PINDEF GINDEF NINDEF RELATED-TO-EXACT)
(EXECUTE PRESENT PINDEF GINDEF NINDEF TO-PERFORM)
(EXECUTED PAST PINDEF GINDEF NINDEF TO-PERFORM)
(EXECUTES PRESENT OTHER GINDEF SING TO-PERFORM)
(EXERCISE TINDEF PINDEF NEUTRAL SING TRIAL-EXERCISE)
(EXERCISES TINDEF PINDEF NEUTRAL PLURAL TRIAL-EXERCISE)
(EXHIBIT PRESENT PINDEF GINDEF NINDEF TO-POSSESS)
(EXHIBITED PAST PINDEF GINDEF NINDEF TO-POSSESS)
(EXHIBITS PRESENT OTHER GINDEF SING TO-POSSESS)
(EXPLAIN PRESENT PINDEF GINDEF NINDEF TO-DISCLOSE-KNOWLEDGE)
(EXPLAINED PAST PINDEF GINDEF NINDEF TO-DISCLOSE-KNOWLEDGE)
(EXPLAINS PRESENT OTHER GINDEF SING TO-DISCLOSE-KNOWLEDGE)
(FABSOFTA TINDEF OTHER NEUTRAL SING RELATED-TO-FABSOFTA)
(FABSOFTB TINDEF OTHER NEUTRAL SING RELATED-TO-FABSOFTB)
(FABSOFTC TINDEF OTHER NEUTRAL SING RELATED-TO-FABSOFTC)
(FAZENDINHA^S TINDEF OTHER NEUTRAL SING RELATED-TO-FAZENDINHA)
(FILIPE TINDEF OTHER MASC SING PERSON-NAMED-FILIPE)
(FILIPE^S TINDEF OTHER FEM SING RELATED-TO-FILIPE)
(FLAVIO TINDEF OTHER MASC SING PERSON-NAMED-FLAVIO)
(FLAVIO^S TINDEF OTHER FEM SING RELATED-TO-FLAVIO)
(FOLLOW PRESENT PINDEF GINDEF NINDEF TO-TRACK)
(FOLLOWED PAST PINDEF GINDEF NINDEF TO-TRACK)
(FOLLOWS PRESENT OTHER GINDEF SING TO-TRACK)
(FOOT TINDEF PINDEF NEUTRAL SING FOOT)
(FEET TINDEF PINDEF NEUTRAL PLURAL FOOT)
(FOOT^S TINDEF OTHER NEUTRAL SING RELATED-TO-FOOT)
(FEET^S TINDEF OTHER NEUTRAL SING RELATED-TO-FOOT)
(FOR TINDEF PINDEF GINDEF NINDEF FOR-THIS-REASON)
(FRACTURE TINDEF PINDEF NEUTRAL SING PHYSICAL-CRACK)
(FRACTURES TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-CRACK)
(FRENCH TINDEF PINDEF GINDEF NINDEF RELATED-TO-FRENCH)
(FROM TINDEF PINDEF GINDEF NINDEF FROM-SOMETHING)
(FULFILL PRESENT PINDEF GINDEF NINDEF TO-PERFORM)
(FULFILLED PAST PINDEF GINDEF NINDEF TO-PERFORM)
(FULFILS PRESENT OTHER GINDEF SING TO-PERFORM)
(HAD PAST PINDEF GINDEF NINDEF TO-POSSESS)
(HAS PRESENT OTHER GINDEF SING TO-POSSESS)
(HAVE PRESENT PINDEF GINDEF NINDEF TO-POSSESS)
(HE TINDEF OTHER MASC SING THIS-MAN)
(HEART TINDEF PINDEF NEUTRAL SING HEART)
(HEARTS TINDEF PINDEF NEUTRAL PLURAL HEART)
(HEART^S TINDEF OTHER NEUTRAL SING RELATED-TO-HEART)
(HEARTS^ TINDEF OTHER NEUTRAL SING RELATED-TO-HEART)
(HEMORRHAGE TINDEF PINDEF NEUTRAL SING BLOOD-ESCAPE)
(HIGH TINDEF OTHER NEUTRAL SING RELATED-TO-HIGH)
(HIP TINDEF PINDEF NEUTRAL SING HIP)
(HIPS TINDEF PINDEF NEUTRAL PLURAL HIP)
(HIP^S TINDEF OTHER NEUTRAL SING RELATED-TO-HIP)
(HIPS^ TINDEF OTHER NEUTRAL SING RELATED-TO-HIP)
(HOSPITAL TINDEF PINDEF NEUTRAL SING PLACE-TO-CURE)
(HOSPITALS TINDEF PINDEF NEUTRAL PLURAL PLACE-TO-CURE)

(HOUSE TINDEF PINDEF NEUTRAL SING PLACE-TO-LIVE)
(HOUSES TINDEF PINDEF NEUTRAL PLURAL PLACE-TO-LIVE)
(HOW TINDEF PINDEF GINDEF NINDEF HOW-INTERROGATION)
(HOW TINDEF PINDEF GINDEF NINDEF RELATIVE-HOW)
(I TINDEF ME GINDEF SING MY-SELF)
(IDEA TINDEF PINDEF NEUTRAL SING VISION-OF-WORLD)
(IDEAS TINDEF PINDEF NEUTRAL PLURAL VISION-OF-WORLD)
(IDENTIFY PRESENT PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(IDENTIFIED PAST PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(IDENTIFIES PRESENT OTHER GINDEF SING TO-MAKE-DIAGNOSTIC)
(IMPLEMENTATION TINDEF PINDEF NEUTRAL SING REALIZATION-OF-SOMETHING)
(IMPLEMENTATIONS TINDEF PINDEF NEUTRAL PLURAL REALIZATION-OF-SOMETHING)
(IN TINDEF PINDEF GINDEF NINDEF INSIDE-SOMETHING)
(INCFABSOFT TINDEF OTHER NEUTRAL SING RELATED-TO-INCFABSOFT)
(INCUBATOR TINDEF OTHER NEUTRAL SING COMPANIES-GROUPING-SYSTEM)
(INCUBATORS TINDEF OTHER NEUTRAL SING COMPANIES-GROUPING-SYSTEM)
(INDICATOR TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER-SIGN)
(INDICATORS TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER-SIGN)
(INEXACT TINDEF PINDEF GINDEF NINDEF RELATED-TO-WRONG)
(INFECTION TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER)
(INFECTIONS TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER)
(INITIATE PRESENT PINDEF GINDEF NINDEF TO-INITIALIZE)
(INITIATED PAST PINDEF GINDEF NINDEF TO-INITIALIZE)
(INITIATES PRESENT OTHER GINDEF SING TO-INITIALIZE)
(INJECTION TINDEF PINDEF NEUTRAL SING PHYSICAL-INJECTION)
(INJECTIONS TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-INJECTION)
(INSULIN TINDEF PINDEF NEUTRAL SING INSULIN)
(INSULIN^S TINDEF PINDEF GINDEF NINDEF RELATED-TO-INSULIN)
(INTERVENTION TINDEF PINDEF NEUTRAL SING PHYSICAL-OPERATION)
(INTERVENTIONS TINDEF PINDEF NEUTRAL SING PHYSICAL-OPERATION)
(ISO14000 TINDEF PINDEF GINDEF NINDEF RELATED-TO-ISO-14000)
(ISO9000 TINDEF PINDEF GINDEF NINDEF RELATED-TO-ISO-9000)
(ISO9001 TINDEF PINDEF GINDEF NINDEF RELATED-TO-ISO-9001)
(JACK TINDEF OTHER MASC SING PERSON-NAMED-JACK)
(JACK^S TINDEF OTHER FEM SING RELATED-TO-JACK)
(JEAN TINDEF OTHER MASC SING PERSON-NAMED-JEAN)
(JEAN^S TINDEF OTHER FEM SING RELATED-TO-JEAN)
(JOHN TINDEF OTHER MASC SING PERSON-NAMED-JOHN)
(JOHN^S TINDEF OTHER FEM SING RELATED-TO-JOHN)
(KNEE TINDEF PINDEF NEUTRAL SING KNEE)
(KNEES TINDEF PINDEF NEUTRAL PLURAL KNEE)
(KNEE^S TINDEF OTHER NEUTRAL SING RELATED-TO-KNEE)
(KNEES^ TINDEF OTHER NEUTRAL SING RELATED-TO-KNEE)
(KNEW PAST PINDEF GINDEF NINDEF TO-KNOW)
(KNOW PRESENT PINDEF GINDEF NINDEF TO-KNOW)
(KNOWS PRESENT OTHER GINDEF SING TO-KNOW)
(LABORATORY TINDEF PINDEF NEUTRAL SING PLACE-TO-LIVE)
(LABORATORIES TINDEF PINDEF NEUTRAL PLURAL PLACE-TO-LIVE)
(LANGUAGE TINDEF PINDEF NEUTRAL SING NORM-OF-EXPRESSION)
(LANGUAGES TINDEF PINDEF NEUTRAL PLURAL NORM-OF-EXPRESSION)
(LEG TINDEF PINDEF NEUTRAL SING LEG)
(LEGS TINDEF PINDEF NEUTRAL PLURAL LEG)
(LEG^S TINDEF OTHER NEUTRAL SING RELATED-TO-LEG)
(LEGS^ TINDEF OTHER NEUTRAL SING RELATED-TO-LEG)
(LESSON TINDEF PINDEF NEUTRAL SING STEPS-FOR-TEACHING-LEARNING)

(LESSONS TINDEF PINDEF NEUTRAL PLURAL STEPS-FOR-TEACHING-LEARNING)
(LISP TINDEF PINDEF NEUTRAL NINDEF RELATED-TO-LISP)
(LIVE PRESENT PINDEF GINDEF NINDEF TO-STAY-AT)
(LIVED PAST PINDEF GINDEF NINDEF TO-STAY-AT)
(LIVES PRESENT OTHER GINDEF SING TO-STAY-AT)
(LOW TINDEF OTHER NEUTRAL SING RELATED-TO-LOW)
(LUNG TINDEF PINDEF NEUTRAL SING LUNG)
(LUNGS TINDEF PINDEF NEUTRAL PLURAL LUNG)
(LUNG^S TINDEF OTHER NEUTRAL SING RELATED-TO-LUNG)
(LUNGS^ TINDEF OTHER NEUTRAL SING RELATED-TO-LUNG)
(MARCIO TINDEF OTHER MASC SING PERSON-NAMED-MARCIO)
(MARCIO^S TINDEF OTHER FEM SING RELATED-TO-MARCIO)
(MASSAGE TINDEF PINDEF NEUTRAL SING MASSAGE)
(MASSAGES TINDEF PINDEF NEUTRAL PLURAL MASSAGE)
(ME TINDEF ME GINDEF SING MY-SELF)
(MISTAKE TINDEF PINDEF NEUTRAL SING ERROR)
(MISTAKES TINDEF PINDEF NEUTRAL PLURAL ERROR)
(MODEL TINDEF PINDEF NEUTRAL SING IDEAL-REPRESENTATION-OF-REALITY)
(MODELS TINDEF PINDEF NEUTRAL PLURAL IDEAL-REPRESENTATION-OF-REALITY)
(NECK TINDEF PINDEF NEUTRAL SING NECK)
(NECKS TINDEF PINDEF NEUTRAL PLURAL NECK)
(NECK^S TINDEF OTHER NEUTRAL SING RELATED-TO-NECK)
(NECKS^ TINDEF OTHER NEUTRAL SING RELATED-TO-NECK)
(NO TINDEF PINDEF GINDEF NINDEF NEGATION)
(NON-CONFORMITIES TINDEF PINDEF NEUTRAL PLURAL NON-CONFORMITY)
(NON-CONFORMITY TINDEF PINDEF NEUTRAL SING NON-CONFORMITY)
(NOT TINDEF PINDEF GINDEF NINDEF NEGATION)
(NOTICE PRESENT PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(NOTICED PAST PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(NOTICES PRESENT OTHER GINDEF SING TO-MAKE-DIAGNOSTIC)
(OBSERVE PRESENT PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(OBSERVED PAST PINDEF GINDEF NINDEF TO-MAKE-DIAGNOSTIC)
(OBSERVES PRESENT OTHER GINDEF SING TO-MAKE-DIAGNOSTIC)
(ON TINDEF PINDEF GINDEF NINDEF ON-SOMETHING)
(OPERATE PRESENT PINDEF GINDEF NINDEF TO-PERFORM)
(OPERATED PAST PINDEF GINDEF NINDEF TO-PERFORM)
(OPERATES PRESENT OTHER GINDEF SING TO-PERFORM)
(PARALYSIS TINDEF PINDEF NEUTRAL SING MOVING-IMPOSSIBILITY)
(PASCAL TINDEF PINDEF NEUTRAL NINDEF RELATED-TO-PASCAL)
(PDCA TINDEF PINDEF GINDEF NINDEF RELATED-TO-PDCA)
(PEOPLE TINDEF OTHER GINDEF PLURAL UNDEFINED-BENCH-OF-PERSON)
(PERFORATION TINDEF PINDEF NEUTRAL SING HOLE)
(PERFORATIONS TINDEF PINDEF NEUTRAL PLURAL HOLE)
(PERFORM PRESENT PINDEF GINDEF NINDEF TO-PERFORM)
(PERFORMED PAST PINDEF GINDEF NINDEF TO-PERFORM)
(PERFORMS PRESENT OTHER GINDEF SING TO-PERFORM)
(PERSISTENT TINDEF OTHER NEUTRAL SING RELATED-TO-PERSISTENT)
(PORTUGUESE TINDEF PINDEF GINDEF NINDEF RELATED-TO-PORTUGUESE)
(PRESCRIBE PRESENT PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(PRESCRIBED PAST PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(PRESCRIBES PRESENT OTHER GINDEF SING TO-MAKE-PRESCRIBTION)
(PRESSURE TINDEF PINDEF NEUTRAL SING PRESSURE)
(PRESSURES TINDEF PINDEF NEUTRAL PLURAL PRESSURE)
(PROBLEM TINDEF PINDEF NEUTRAL SING ERROR)
(PROBLEMS TINDEF PINDEF NEUTRAL PLURAL ERROR)

(PROGRAM TINDEF PINDEF NEUTRAL SING LIST-OF-PROGRAMMED-STEPS)
(PROGRAM PRESENT PINDEF GINDEF NINDEF TO-MAKE-LIST-OF-PROGRAMMED-STEPS)
(PROGRAMMED PAST PINDEF GINDEF NINDEF TO-MAKE-LIST-OF-PROGRAMMED-STEPS)
(PROGRAMMING TINDEF PINDEF GINDEF NINDEF RELATED-TO-PROGRAMMING)
(PROGRAMS PRESENT OTHER GINDEF SING TO-MAKE-LIST-OF-PROGRAMMED-STEPS)
(PROGRAMS TINDEF PINDEF NEUTRAL PLURAL LIST-OF-PROGRAMMED-STEPS)
(PROPOSE PRESENT PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(PROPOSED PAST PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(PROPOSES PRESENT OTHER GINDEF SING TO-MAKE-PRESCRIBTION)
(QUARTER TINDEF PINDEF NEUTRAL SING PLACE-TO-LIVE)
(QUARTERS TINDEF PINDEF NEUTRAL PLURAL PLACE-TO-LIVE)
(READ TINDEF PINDEF GINDEF NINDEF TO-READ)
(READS PRESENT OTHER GINDEF SING TO-READ)
(RECOMMEND PRESENT PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(RECOMMENDED PAST PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(RECOMMENDS PRESENT OTHER GINDEF SING TO-MAKE-PRESCRIBTION)
(REVEAL PRESENT PINDEF GINDEF NINDEF TO-POSSESS)
(REVEALED PAST PINDEF GINDEF NINDEF TO-POSSESS)
(REVEALS PRESENT OTHER GINDEF SING TO-POSSESS)
(RIGHT TINDEF PINDEF GINDEF NINDEF RELATED-TO-EXACT)
(ROBERTO TINDEF OTHER MASC SING PERSON-NAMED-ROBERTO)
(ROBERTO^S TINDEF OTHER FEM SING RELATED-TO-ROBERTO)
(RUPTURE TINDEF PINDEF NEUTRAL SING PHYSICAL-CRACK)
(RUPTURES TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-CRACK)
(SANTA-FELICIDADE^S TINDEF OTHER NEUTRAL SING RELATED-TO-SANTA-FELICIDADE)
(SET TINDEF PINDEF GINDEF NINDEF TO-INITIALIZE)
(SETS PRESENT OTHER GINDEF SING TO-INITIALIZE)
(SEVERE TINDEF OTHER NEUTRAL SING RELATED-TO-SEVERE)
(SHE TINDEF OTHER FEM SING THIS-WOMAN)
(SHOULDER TINDEF PINDEF NEUTRAL SING SHOULDER)
(SHOULDERS TINDEF PINDEF NEUTRAL PLURAL SHOULDER)
(SHOULDER^S TINDEF OTHER NEUTRAL SING RELATED-TO-SHOULDER)
(SHOULDERS^ TINDEF OTHER NEUTRAL SING RELATED-TO-SHOULDER)
(SHOW PRESENT PINDEF GINDEF NINDEF TO-POSSESS)
(SHOWED PAST PINDEF GINDEF NINDEF TO-POSSESS)
(SHOWS PRESENT OTHER GINDEF SING TO-POSSESS)
(SIGN TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER-SIGN)
(SIGNS TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER-SIGN)
(SOLVE PRESENT PINDEF GINDEF NINDEF TO-FIND-SOLUTION)
(SOLVED PAST PINDEF GINDEF NINDEF TO-FIND-SOLUTION)
(SOLVES PRESENT OTHER GINDEF SING TO-FIND-SOLUTION)
(SOMEONE TINDEF OTHER GINDEF SING UNDEFINED-PERSON)
(START PRESENT PINDEF GINDEF NINDEF TO-INITIALIZE)
(STARTED PAST PINDEF GINDEF NINDEF TO-INITIALIZE)
(STARTS PRESENT OTHER GINDEF SING TO-INITIALIZE)
(STAY PRESENT PINDEF GINDEF NINDEF TO-STAY-AT)
(STAYED PAST PINDEF GINDEF NINDEF TO-STAY-AT)
(STAYS PRESENT OTHER GINDEF SING TO-STAY-AT)
(STOMACH TINDEF PINDEF NEUTRAL SING STOMACH)
(STOMACHS TINDEF PINDEF NEUTRAL PLURAL STOMACH)
(STOMACH^S TINDEF OTHER NEUTRAL SING RELATED-TO-STOMACH)
(STOMACHS^ TINDEF OTHER NEUTRAL SING RELATED-TO-STOMACH)
(STORE PRESENT PINDEF GINDEF NINDEF TO-STORE)
(STORE TINDEF PINDEF NEUTRAL SING PLACE-TO-STORE-THINGS)
(STORED PAST PINDEF NEUTRAL NINDEF TO-STORE)

(STORES PRESENT OTHER GINDEF SING TO-STORE)
(STORES TINDEF PINDEF NEUTRAL PLURAL PLACE-TO-STORE-THINGS)
(SUGGEST PRESENT PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(SUGGESTED PAST PINDEF GINDEF NINDEF TO-MAKE-PRESCRIBTION)
(SUGGESTS PRESENT OTHER GINDEF SING TO-MAKE-PRESCRIBTION)
(SYMPTOM TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER-SIGN)
(SYMPTOMS TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER-SIGN)
(SYNDROME TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER)
(SYNDROMES TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER)
(TAUGHT PAST PINDEF GINDEF NINDEF TO-GIVE-KNOWLEDGE)
(TEACH PRESENT PINDEF GINDEF NINDEF TO-GIVE-KNOWLEDGE)
(TEACHES PRESENT OTHER GINDEF SING TO-GIVE-KNOWLEDGE)
(TELL PRESENT PINDEF GINDEF NINDEF TO-TELL-CASE)
(TELLS PRESENT OTHER GINDEF SING TO-TELL-CASE)
(TEST PRESENT PINDEF GINDEF SING TO-TEST)
(TEST TINDEF PINDEF NEUTRAL SING PROCEDURE-FOR-TESTING)
(TESTED PAST PINDEF GINDEF SING TO-TEST)
(TESTS PRESENT OTHER GINDEF SING TO-TEST)
(TESTS TINDEF PINDEF NEUTRAL PLURAL PROCEDURE-FOR-TESTING)
(TEXT TINDEF PINDEF NEUTRAL SING LIST-OF-SENTENCE)
(TEXTS TINDEF PINDEF NEUTRAL PLURAL LIST-OF-SENTENCE)
(THAT TINDEF PINDEF GINDEF SING RELATIVE-THAT)
(THAT TINDEF PINDEF GINDEF SING POINTER-THAT)
(THE TINDEF PINDEF GINDEF NINDEF POINTER-THE)
(THEY TINDEF OTHER GINDEF PLURAL THESE-PEOPLE)
(THINK PRESENT PINDEF GINDEF NINDEF TO-THINK)
(THINKS PRESENT OTHER GINDEF SING TO-THINK)
(THIS TINDEF PINDEF GINDEF SING POINTER-THIS)
(THOUGHT PAST PINDEF GINDEF NINDEF TO-THINK)
(TO TINDEF PINDEF GINDEF NINDEF IN-THIS-DIRECTION)
(TO TINDEF PINDEF GINDEF NINDEF TO-PRE-VERB)
(TOLD PAST PINDEF GINDEF NINDEF TO-TELL-CASE)
(TQM TINDEF PINDEF GINDEF NINDEF RELATED-TO-TQM)
(TRACK PRESENT PINDEF GINDEF NINDEF TO-TRACK)
(TRACKED PAST PINDEF GINDEF NINDEF TO-TRACK)
(TRACKS PRESENT OTHER GINDEF SING TO-TRACK)
(TRANSFUSION TINDEF PINDEF NEUTRAL SING BLOOD-TRANSFUSION)
(TRANSFUSIONS TINDEF PINDEF NEUTRAL SING BLOOD-TRANSFUSION)
(TRUNK TINDEF PINDEF NEUTRAL SING CHEST)
(TRUNKS TINDEF PINDEF NEUTRAL PLURAL CHEST)
(TRUNK^S TINDEF PINDEF NEUTRAL SING RELATED-TO-CHEST)
(TRUNKS^ TINDEF PINDEF NEUTRAL SING RELATED-TO-CHEST)
(UNCEASING TINDEF OTHER NEUTRAL SING RELATED-TO-PERSISTENT)
(UNDERSTAND PRESENT PINDEF GINDEF NINDEF TO-UNDERSTAND)
(UNDERSTANDS PRESENT OTHER GINDEF SING TO-UNDERSTAND)
(UNDERSTOOD PAST PINDEF GINDEF NINDEF TO-UNDERSTAND)
(UNENDING TINDEF OTHER NEUTRAL SING RELATED-TO-PERSISTENT)
(US TINDEF ME GINDEF PLURAL OUR-SELVES)
(VALUE TINDEF PINDEF NEUTRAL SING MEASURABLE-VALUE)
(VALUES TINDEF PINDEF NEUTRAL PLURAL MEASURABLE-VALUE)
(VARIABLE TINDEF PINDEF NEUTRAL SING CHANGABLE-VALUE)
(VARIABLES TINDEF PINDEF NEUTRAL PLURAL CHANGABLE-VALUE)
(VIRUS TINDEF PINDEF NEUTRAL SING PHYSICAL-DISORDER)
(VIRUSES TINDEF PINDEF NEUTRAL PLURAL PHYSICAL-DISORDER)
(VISUAL-BASIC TINDEF PINDEF NEUTRAL NINDEF RELATED-TO-VISUAL-BASIC)

(WE TINDEF ME GINDEF PLURAL OUR-SELVES)
(WHAT TINDEF PINDEF GINDEF NINDEF WHAT-INTERROGATION)
(WHAT TINDEF PINDEF GINDEF NINDEF RELATIVE-WHAT)
(WHERE TINDEF PINDEF GINDEF NINDEF WHO-INTERROGATION)
(WHERE TINDEF PINDEF GINDEF NINDEF RELATIVE-WHO)
(WHICH TINDEF PINDEF GINDEF NINDEF WHAT-INTERROGATION)
(WHICH TINDEF PINDEF GINDEF NINDEF RELATIVE-WHAT)
(WHO TINDEF PINDEF GINDEF NINDEF WHO-INTERROGATION)
(WHO TINDEF PINDEF GINDEF NINDEF RELATIVE-WHO)
(WITH TINDEF PINDEF GINDEF NINDEF WITH-SOMETHING)
(WORK PRESENT PINDEF GINDEF NINDEF TO-STAY-AT)
(WORKED PAST PINDEF GINDEF NINDEF TO-STAY-AT)
(WORKS PRESENT OTHER GINDEF SING TO-STAY-AT)
(WRIST TINDEF PINDEF NEUTRAL SING WRIST)
(WRISTS TINDEF PINDEF NEUTRAL PLURAL WRIST)
(WRIST^S TINDEF OTHER NEUTRAL SING RELATED-TO-WRIST)
(WRISTS^ TINDEF OTHER NEUTRAL SING RELATED-TO-WRIST)
(WRONG TINDEF PINDEF GINDEF NINDEF RELATED-TO-WRONG)
(YOU TINDEF OTHER GINDEF SING YOUR-SELF)

PeopleNetwork.net (após quinto dia de simulação no cenário "*Home-Care*")

(PERSON-NAMED-MARCIO PERSON-NAMED-FLAVIO) (PERSON-NAMED-FLAVIO PERSON-NAMED-MARCIO) (PERSON-NAMED-FLAVIO PERSON-NAMED-JEAN) (PERSON-NAMED-JEAN PERSON-NAMED-FLAVIO) (PERSON-NAMED-JEAN PERSON-NAMED-DOMINIQUE) (PERSON-NAMED-FILIPE PERSON-NAMED-DOMINIQUE) (PERSON-NAMED-JEAN PERSON-NAMED-ROBERTO) (PERSON-NAMED-JEAN PERSON-NAMED-FILIPE) (PERSON-NAMED-FILIPE PERSON-NAMED-JEAN) (PERSON-NAMED-JACK PERSON-NAMED-JEAN) (PERSON-NAMED-JACK PERSON-NAMED-DANI) (PERSON-NAMED-FILIPE PERSON-NAMED-DANI) (PERSON-NAMED-JACK PERSON-NAMED-JOHN) (PERSON-NAMED-FILIPE PERSON-NAMED-JOHN) (PERSON-NAMED-JACK PERSON-NAMED-ROBERTO) (PERSON-NAMED-FILIPE PERSON-NAMED-ROBERTO) (PERSON-NAMED-FILIPE PERSON-NAMED-JACK) (PERSON-NAMED-JACK PERSON-NAMED-FILIPE)

English-NLP-Predictions.prd

(ACTION (AFFIRMATION ACTION DIRECTION PERSON POINTER POSITION QUALIFICATION RELATIVE))
(AFFIRMATION (AFFIRMATION ACTION INTERROGATION PERSON POINTER WOBJECT))
(CONSEQUENCE (AFFIRMATION PERSON POINTER POSITION WOBJECT))
(COORDINATION (ACTION DIRECTION INTERROGATION PERSON POINTER POSITION QUALIFICATION))
(DIRECTION (PERSON POINTER QUALIFICATION RELATIVE))
(INTERROGATION (ACTION PERSON POINTER))
(PERSON (ACTION COORDINATION DIRECTION PERSON POINTER POSITION RELATIVE))
(POINTER (QUALIFICATION WOBJECT))
(POSITION (PERSON POINTER QUALIFICATION))
(QUALIFICATION (COORDINATION PERSON QUALIFICATION WOBJECT))
(RELATIVE (ACTION PERSON POINTER))
(WOBJECT (ACTION COORDINATION DIRECTION POINTER POSITION RELATIVE))

People-Skill-Base.psb (após quinto dia de simulação no cenário "*Home-Care*")

(PERSON-NAMED-FILIPE 1) (PERSON-NAMED-JACK 2) (PERSON-NAMED-FILIPE 3) (PERSON-NAMED-ROBERTO 4) (PERSON-NAMED-FILIPE 5) (PERSON-NAMED-JACK 6) (PERSON-NAMED-FILIPE 6) (PERSON-NAMED-ROBERTO 7) (PERSON-NAMED-ROBERTO 8) (PERSON-NAMED-JOHN 9) (PERSON-NAMED-JACK 10) (PERSON-NAMED-FILIPE 10) (PERSON-NAMED-JOHN 11) (PERSON-NAMED-ROBERTO 12) (PERSON-NAMED-FILIPE 13) (PERSON-NAMED-DANI 14) (PERSON-NAMED-JACK 15) (PERSON-NAMED-FILIPE 15) (PERSON-NAMED-DANI 16) (PERSON-NAMED-DANI 17) (PERSON-NAMED-JACK 17) (PERSON-NAMED-DANI 18) (PERSON-NAMED-JEAN 19) (PERSON-NAMED-FILIPE 20) (PERSON-NAMED-JACK 20) (PERSON-NAMED-JEAN 2) (PERSON-NAMED-ROBERTO 21) (PERSON-NAMED-JEAN 6) (PERSON-NAMED-FILIPE 22) (PERSON-NAMED-DOMINIQUE 23) (PERSON-NAMED-JEAN 24) (PERSON-NAMED-FILIPE 24) (PERSON-NAMED-DOMINIQUE 25) (PERSON-NAMED-FLAVIO 20) (PERSON-NAMED-JEAN 26) (PERSON-NAMED-MARCIO 26) (PERSON-NAMED-FLAVIO 27) (PERSON-NAMED-MARCIO 28)

Skill-Base.sb (após quinto dia de simulação no cenário "*Home-Care*")

(KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-JACK)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-FILIPE)))) (POS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-FILIPE))) (FROM (QUALIFICATION (QUALIFYER RELATED-TO-CLINICAS) (QUALIFIED PLACE-TO-CURE))) (TO (QUALIFICATION (QUALIFYER RELATED-TO-CLINICAS) (QUALIFIED PLACE-TO-CURE)))) (EXPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-LUNG) (QUALIFIED HOLE)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-CHIRURGIC) (QUALIFIED PHYSICAL-OPERATION))) (TO (QUALIFICATION (QUALIFYER RELATED-TO-ROBERTO) (QUALIFIED LUNG)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO)))) (EXPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-HIGH) (QUALIFIED PRESSURE)))) (EXPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-HEART) (QUALIFIED ATTACK)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-HEART) (QUALIFIED MASSAGE))) (TO (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-JOHN)))) (POS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-JOHN))) (FROM (QUALIFICATION (QUALIFYER RELATED-TO-CLINICAS) (QUALIFIED PLACE-TO-CURE))) (TO (QUALIFICATION (QUALIFYER RELATED-TO-CLINICAS) (QUALIFIED PLACE-TO-CURE)))) (EXPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-BLOOD) (QUALIFIED BLOOD-ESCAPE)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-BLOOD) (QUALIFIED BLOOD-TRANSFUSION))) (TO (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-BLOOD) (QUALIFIED PROCEDURE-FOR-TESTING)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DANI)))) (POS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DANI))) (FROM (QUALIFICATION (QUALIFYER RELATED-TO-ANALAB) (QUALIFIED PLACE-TO-LIVE))) (TO (QUALIFICATION (QUALIFYER RELATED-TO-ANALAB) (QUALIFIED PLACE-TO-LIVE)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PHYSICAL-INJECTION)))) (MBUILD (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PHYSICAL-DISORDER))) (TO (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DANI)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED ANESTHESIA))) (TO (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-JEAN)))) (EXPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-ARM) (QUALIFIED PHYSICAL-CRACK)))) (MTRANS (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-ARM) (QUALIFIED MASSAGE))) (TO (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-ROBERTO)))) (PROPEL (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED MASSAGE))) (TO (QUALIFICATION (QUALIFYER RELATED-TO-ROBERTO) (QUALIFIED ARM)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DOMINIQUE)))) (POS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-DOMINIQUE))) (FROM (QUALIFICATION (QUALIFYER RELATED-TO-BATEL) (QUALIFIED PLACE-TO-LIVE))) (TO (QUALIFICATION (QUALIFYER RELATED-TO-BATEL) (QUALIFIED PLACE-TO-LIVE)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-FLAVIO)))) (KPOSS (OBJECT (QUALIFICATION (QUALIFYER NIL) (QUALIFIED PERSON-NAMED-MARCIO)))) (EXPEL (OBJECT (QUALIFICATION (QUALIFYER RELATED-TO-LEG) (QUALIFIED PHYSICAL-CRACK))))

CDs-Simplifications.smp

(DIRECTION DETERMINATION POSITION RELATIVE ACTION-DETAILS)

NLP-Substitutions.sub

(MY-SELF (ACTOR-FROM)) (YOUR-SELF (ACTOR-TO)) (OUR-SELVES (ACTOR-FROM))

B.4. Código das Funções Utilizadas pelos Agentes de Nível 2

CD-functions.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;  CD Functions  Common Lisp implementation by:
;;;;  Bill Andersen (waander@cs.umd.edu), Department of Computer Science University of Maryland College Park

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Global Variables

(defconstant *cd-acts*
  '(atrans ptrans propel mpropel move grasp ingest
    expel mexpel mtrans conc mbuild attend speak Kposs pos qualification)
  "List of valid CD ACTS predicates")

(defparameter *user-trace* t
  "Controls output from USER-TRACE function.  If T,
message to USER-TRACE is printed.  If NIL, no output
occurs.")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Data Structures

(defun header-cd (x)
   "Returns predicate of a CD form"
   (car x))

(defun roles-cd (x)
   "Returns list of (role value) pairs for a CD form."
   (cdr x))

(defun role-pair (x)
   "Returns the role of a (role filler) pair."
   (car x))

(defun filler-pair (x)
   "Returns the filler of a (role filler) pair."
   (cadr x))

(defun remove-all-pairs-with (W CD)
   "Removes all pairs (role filler) which possess W as filler form CD"
            (cons      (header-cd cd)
                       (remove NIL
                                 (mapcar
                                          #'(lambda (X) (if (equal (filler-pair X) W) NIL X))
                                          (roles-cd CD)))))

(defun remove-role (role cd)
    (cons (header-cd cd) (remove role (roles-cd cd) :key #'role-pair)))

(defun filler-role (role cd)
  (let ((pair (assoc role (roles-cd cd))))
    (and pair (filler-pair pair))))
```

```
(defun is-cd-p (x)
 (and (listp x) (atom (header-cd x))  (list-of-role-filler-pairs-p (roles-cd x))))

(defun list-of-role-filler-pairs-p (x)
 (or (null x) (and (listp x)  (listp (car x))  (atom (role-pair (car x))  (list-of-role-filler-pairs-p (cdr x)))))

(defun setrole (role filler cd)
 (cons (header-cd cd)
      (cons (list role filler) (remove role (roles-cd cd) :key #'role-pair))))

(defun cdpath (rolelist cd)
 (if (null rolelist)
   cd (cdpath (cdr rolelist) (filler-role (car rolelist) cd))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  CD Pattern Matcher

(defstruct (cd-var (:print-function print-cd-var)
                (:predicate is-var))  name)

(defun print-cd-var (struct stream depth)
 (declare (ignore depth)) (format stream "?~s" (cd-var-name struct)))

(defun name-var (x)
 (assert (is-var x) () "~s is not a CD variable." x) (cd-var-name x))

(set-macro-character #\?
 #'(lambda (stream char) (make-cd-var :name (read stream t nil t))) t)

(defun match (pattern constant bindings)
 (let ((binding-form (or bindings (list t))))
   (cond ((or (null constant) (equal pattern constant)) binding-form)
        ((is-var pattern) (match-var pattern constant binding-form))
        ((or (atom constant) (atom pattern)) nil)
        ((eq (header-cd pattern) (header-cd constant)) (match-args (roles-cd pattern) constant binding-form)))))

(defun match-args (pattern-args constant bindings)
 (dolist (pattern-arg pattern-args)
   (let ((const (filler-role (role-pair pattern-arg) constant))
        (var (filler-pair pattern-arg)))
      (setq bindings (match var const bindings))
      (if (null bindings)
        (return nil)))) bindings)

(defun match-var (pattern constant bindings)
 (let ((var-value (filler-role (name-var pattern) bindings)))
   (if var-value (match var-value constant bindings)
     (cons-end bindings (list (name-var pattern) constant)))))

(defun instantiate (cd-form bindings)
 (cond ((symbolp cd-form) cd-form)
      ((is-var cd-form) (instantiate (filler-role (name-var cd-form) bindings) bindings))
      (t (cons (header-cd cd-form)
             (mapcar #'(lambda (pair)  (list (role-pair pair) (instantiate (filler-pair pair) bindings))) (roles-cd cd-form))))))
```

CD-functions.lsp

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  Utility Functions

(defun cons-end (l x)
  "Adds x to the end of list l"
  (append l (list x)))

(defun user-trace (str &rest args)
  (let ((*print-pretty* t))
    (when *user-trace*
      (apply #'format t str args))))

(provide :cd-functions)
```

form-recognition.lsp

```
;;;;;;;;;;;;;;;;;;;; Set of functions for Form recognition ;;;;;;;;;;;;;;;;

(require :dialog-manipulation)

(defun negative-p (WORD CONCEPTS)
        (cond   ((NULL CONCEPTS) NIL)
                ((NULL WORD) NIL)
                ((and   (eq (caar CONCEPTS) WORD)
                        (eq (cadar CONCEPTS) 'NEGATIVE)) T)
                (T (negative-p WORD (cdr CONCEPTS)))))

(defun exist-negative-form (WORDS CONCEPTS)
        (cond   ((NULL WORDS) NIL)
                ((negative-p (car (last (car WORDS))) CONCEPTS) T)
                (T (exist-negative-form (cdr WORDS) CONCEPTS))))


(defun set-form-sentence (SENTENCE CONCEPTS)
        (cond   ((exist-negative-form (extract-words SENTENCE) CONCEPTS) (set-form SENTENCE 'NEGATIVE))
                (T (set-form SENTENCE 'POSITIVE))))

(defun set-form-content (CONTENT CONCEPTS)
        (cond   ((NULL CONTENT) NIL)
                (T (cons   (set-form-sentence (extract-sentence 0 CONTENT) CONCEPTS)
                                (set-form-content (cdr CONTENT) CONCEPTS)))))

(defun set-form-block (ABLOCK CONCEPTS)
        (cons (extract-actors-from ABLOCK)
          (cons (extract-actors-to ABLOCK)
                (set-form-content (extract-content ABLOCK) CONCEPTS))))

(defun set-form-dialog (DIALOG CONCEPTS)
        (cond   ((NULL DIALOG) NIL)
                (T (cons   (set-form-block (extract-block 0 DIALOG) CONCEPTS)
                                (set-form-dialog (cdr DIALOG) CONCEPTS)))))

(provide :form-recognition)
```

CD-parser-traced.lsp

```
;;;;;; Natural Language Analyser - Natural Language -> CDs ;;;;;;;;;;;;;;;;

(require :cd-functions)
(require :concept-neutralization) ; rajouter a cause de la fonction word-conceptp dans remove-variable

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  Global Variables ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *stack* nil)                ; request packet stack
(defvar *concept*)                  ; globals set by request assignments
(defvar *sentence*)
(defvar *cd-form*)                  ; stack de la CD-form
(defvar *abs-concept*)                      ; concept abstrait du mot en train d'être traiter
(defvar *word*)                             ; mot en train d'être traiter
(defvar *part-of-speech*)       ; partie du discours reconnue
(defvar *predicates*)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  Data Structures ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun top-stack ()
            ;(user-trace "~1%  Top-stack: ~s" (car *stack*))
            (car *stack*))
(defun add-stack (packet)(and packet (push packet *stack*))
            ;(user-trace "~1%  Add-stack: ~s" packet)
            packet)
(defun pop-stack () (pop *stack*)
            ;(user-trace "~1%  Pop-stack!")
            )
(defun init-stack () (setq *stack* nil))
(defun empty-stack-p () (null *stack*))

(defun load-def ()
  "Adds a word's request packet to the stack. Word definitions are stored under the property DEFINITION."
  ;(user-trace "~1%  load-def of ~s" *abs-concept*)

  ; prend ce qu'il  y a definit apres defword xx
  (let ((packet (get *abs-concept* 'definition)))

  ; test si la def existe
  (cond (packet (add-stack packet)
                        ;(user-trace "~1%  the def is ~s" packet)
                        )
            (t
                                    ;(user-trace " - not in dictionary~%")
                                    nil))))

(defun req-clause (key l)
  "Extracts clauses from requests of the form:((test ...) (assign ...) (next-packet ...))"

  ; recherhe dans la liste de pair l (du type ( (a b) (a c) (e f)) ) la premiere paire qui commence par key
  (let ((x (assoc key l)))
    (and x (cdr x))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  Top Level Functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun process-text (text)
  "Process a list of sentences, parsing each one, and printing the result."
  (dolist (sentence text)
    ;(user-trace "~2%Input is ~s~%" sentence)
    (let ((cd (parse sentence)))
      ;(user-trace "~2%CD form is ~s" cd)
                          ))
  (values))

(defun parse (sentence)
  "Takes a sentence in list form and returns the conceptual analysis for it."
  (setq *concept* nil)
  (init-stack)
  (do

      ;assign les variables
      ((*abs-concept* nil) (*word* nil) (*sentence* (cons '(*START* TINDEF PINDEF GINDEF NINDEF *START-SENTENCE*)
sentence)))                                    ;Assign les variables

      ;Update form (l'astuce est que l'update de trouve dans la condition de finalization (le pop!)

      ;Condition de finalization (sentence = NIL)                    + action terminaison loop
      ; la modification faite (car (last (pop *SENTENCE*))) vient du fait que sentence est
      ; maintenand une liste de liste.
      ; Chaque liste contenue dans sentence est en fait word neuralizaer ontologiquement

      ((null (and (setq *word*      (pop *sentence*))
                  (setq *abs-concept* (car (last *word*)))))           (remove-variables *concept*))


      ; code a effectuer
;      (user-trace "~2%**************************** Processing word ~s" *abs-concept*)
      (load-def)   ; load-def met dans le top de la stack la def de *abs-concept* si word est defini
;      (user-trace "~1%  State of stack: ~s" *stack*)
      (run-stack)  ; fait tourner la stack
      ; fin
  )
)

(defun run-stack ()

  "If some request in the packet on the top of the stack
can be triggered, that packet is removed from the stack,
and the request is saved and executed.  When the top packet
contains no triggerable requests, the packets in the
requests which were triggered and saved are added to the
stack."
```

```
  ;(user-trace "~1%  Run-stack!")

  (do

    ;assign les variables + Update form
    ((request (check-top) (check-top)) (triggered nil))

   ;Condition de finalization + action terminaison loop
   ((null request) (add-packets triggered))

    (pop-stack)
;    (user-trace "~1%  State of stack: ~s" *stack*)
    (do-assigns request)
    (push request triggered)
;   (user-trace "~1% Push inside trigerred: ~s" request)
))

(defun check-top ()
  "Returns the first request with a true test from the top packet in the stack."
  ;(user-trace "~1%  Check-top!")
  (unless  (empty-stack-p)
                          (dolist (request (top-stack))
                          (when (or (null request) (is-triggered request))
;                                                 (user-trace "~1%  Is triggered on top: ~s" request)
                                                  (return request)))))

(defun is-triggered (request)
  "Returns T if a request has no test or if the test evaluates to T."
  (let ((test (req-clause 'test request))) (or (null test) (eval (car test)))))

(defun do-assigns (request)
  "Sets the global variables given in the ASSIGN clause of a request."
  ;(user-trace "~1%  do assignments:")
  (do ((assignments (req-clause 'assign request)
                    (cddr assignments)))
      ((null assignments))
    (reassign (first assignments)
              (second assignments))))

(defun reassign (var val)
  "Reassigns var to val and prints a message."
  (set var (eval val))
 ; (user-trace "~&  ~s := ~s~%" var (eval var))
)

(defun add-packets (requests)
  "Takes a list of requests and add their NEXT-PACKETs to the stack."
;  (user-trace "~1%  add-packets (next-packet):")
  (dolist (request requests)
    (add-stack (req-clause 'next-packet request))))
```

```
;; NOTE:
;;   This function has been modified to handle a list of CD forms
;; instead of a single form.  The reason for this is that some
;; concepts cannot be expressed with a single CD.  For example,
;; "buying" is an ATRANS of something to the actor *and* and
;; ATRANS of money from the actor to the seller.
;;   The parser is not affected since all it does is control the
;; request firing which in turn makes variable assignments.

(defun remove-variables (cd-form)
  "Takes a parsed CD from Micro-ELI and returns a copy
of the pattern with the variables replaced by values.
Roles with NIL fillers are left out of the result.  This
works like INSTANTIATE in Micro-SAM and Micro-POLITICS
except that the values are derived from global variables
rather than binding lists."
 ; (user-trace "~1%  Remove variables in CD-form: ~s~%" cd-form)
  (cond    ((symbolp cd-form) cd-form)
          ((word-conceptp cd-form) cd-form) ; ligne rajouter par moi pour detecter les word-concpets (qui s'appelaient avt
ontology...)
          ((is-var cd-form) (remove-variables (eval (name-var cd-form))))
          (t (replace-list cd-form))))

;; REPLACE-LIST is just an auxiliary function for
;; REMOVE-VARIABLES.
(defun replace-list (cd-form)
  ;(user-trace "~1%  Replace List in CD-form: ~s~%" cd-form)
  (cond ((null cd-form) nil)
        ((atom (header-cd cd-form))
               (cons (header-cd cd-form)
              (let (result)
                (dolist (pair (roles-cd cd-form))
                  (let ((val (remove-variables (filler-pair pair))))
                    (when val (push `(,(role-pair pair) ,val)
                      result))))
                (nreverse result))))
        (t (cons (replace-list (car cd-form))
                 (replace-list (cdr cd-form))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  Dictionary Functions ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmacro defword (&body def)
  `(progn (setf (get ',(car def) 'definition) ',(cdr def))
          ',(car def)))


(provide :CD-parser)
```

CD-recognition.lsp

```
;;;;;;;;;;;;;;;;;;; Set of functions for CD recognition ;;;;;;;;;;;;;;;;;

(require :CD-parser-traced)
(require :dialog-manipulation)

(defun set-CD-sentence (SENTENCE)
        (cons (extract-attributes SENTENCE) (list (parse (extract-words SENTENCE)))))

(defun set-CD-content (CONTENT)
        (cond   ((NULL CONTENT) NIL)
                (T (cons  (set-CD-sentence (extract-sentence 0 CONTENT))  (set-CD-content (cdr CONTENT))))))

(defun set-CD-block (ABLOCK)
        (cons (extract-actors-from ABLOCK)
          (cons (extract-actors-to ABLOCK)
                (set-CD-content (extract-content ABLOCK)))))

(defun set-CD-dialog (DIALOG)
        (cond   ((NULL DIALOG) NIL)
                (T (cons  (set-CD-block (extract-block 0 DIALOG))  (set-CD-dialog (cdr DIALOG))))))

(provide :CD-recognition)
```

English-Generator.lsp

```
;;;;;;;;;;;;;;;; Traduction of CDs Translations into Naural English ;;;;;;;;;;;;;;;;;;;;;;

(require :concept-neutralization) ;because of lexic-value
(require :cd-functions)

(defun add (S1 S2) ; adds sentence S1 to sentence S2, obtaining Sentence S2 S1
        (reverse (append (reverse S1) (reverse S2))))

;;;;;;;;;;;;;;;;;;;;;;;; General English Generator ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun say-cd-english (CD LEXIC CONCEPT-BASE)
        (cond
                ((NULL CD) NIL)
                ((atom CD)          (if         (eq (find-concept-type CD CONCEPT-BASE) 'PERSON)
                                                (list (lexic-value CD LEXIC))
                                                (add (list (lexic-value CD LEXIC)) '(A))))
                ((eq (header-cd CD) 'ATRANS)          (say-atrans cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'PTRANS)          (say-ptrans cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'GRASP)           (say-grasp cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'INGEST)          (say-ingest cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'MTRANS)          (say-mtrans cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'MBUILD)          (say-mbuild cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'KPOSS)           (say-kposs cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'PROPEL)          (say-propel cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'ATTEND)          (say-attend cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'POS)    (say-pos cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'EXPEL)           (say-expel cd LEXIC CONCEPT-BASE))
                ((eq (header-cd CD) 'QUALIFICATION) (say-qualification cd LEXIC CONCEPT-BASE))))

(provide :english-generator)
```

CD-simplification.lsp

```
;;;;;;;;;;;;;;;;;; Set of functions for CD simplification ;;;;;;;;;;;;;;;;;

(require :cd-functions)
(require :dialog-manipulation)
(require :concept-neutralization) ; rajouter a cause de la fonction word-conceptp dans remove-variable

(defun qualify (CD) ;add a (QUALIFICATION (QUALIFYER NIL) (QUALIFIED X)) to all objects / actor / from / to which are
not qualified
        (cond   ((NULL CD) NIL)
                ((ATOM CD) (car (list CD)))
                ((AND   (OR (EQ (CAR CD) 'OBJECT) (EQ (CAR CD) 'ACTOR) (EQ (CAR CD) 'FROM) (EQ (CAR CD)
'TO))
                        (ATOM (cadr CD)))
                        (append (cons   (car CD)
                                        (cons
                                                (cons   'QUALIFICATION
                                                        (cons
                                                                (cons   'QUALIFYER (cons NIL NIL))
                                                                (cons   (cons   'QUALIFIED (cons (cadr
CD) NIL)) NIL)))   NIL))))
                (T (cons (qualify (car CD)) (qualify (cdr CD))))))

(defun simplify-CD-words (WORDS SIMPLIFICATIONS)
        (cond
                ((NULL WORDS) NIL)
                ((ATOM WORDS) WORDS)
                ((word-conceptp WORDS) (car (last WORDS)))
                ((member (header-cd WORDS) (car SIMPLIFICATIONS))
                        (simplify-CD-words (car (roles-CD (cadr (roles-CD WORDS)))) SIMPLIFICATIONS))
                (T (cons (header-cd WORDS)
                         (delete '() (mapcar #'(lambda (X) (simplify-CD-words X SIMPLIFICATIONS)) (roles-cd
WORDS)))))))

(defun simplify-CD-sentence (SENTENCE SIMPLIFICATIONS)
        (cond   ((NULL SENTENCE) NIL)
                (T (cons (extract-attributes SENTENCE)
           (cons (qualify (simplify-CD-words (extract-words SENTENCE) SIMPLIFICATIONS))  NIL)))))

(defun simplify-CD-content (CONTENT SIMPLIFICATIONS)
        (cond   ((NULL CONTENT) NIL)
                (T (cons  (simplify-CD-sentence (extract-sentence 0 CONTENT) SIMPLIFICATIONS)
                                (simplify-CD-content (cdr CONTENT) SIMPLIFICATIONS)))))

(defun simplify-CD-block (ABLOCK SIMPLIFICATIONS)
        (cons (last (car (extract-actors-from ABLOCK)))
          (cons (last (car (extract-actors-to ABLOCK)))
                (simplify-CD-content (extract-content ABLOCK) SIMPLIFICATIONS))))

(defun simplify-CD-dialog (DIALOG SIMPLIFICATIONS)
        (cond   ((NULL DIALOG) NIL)
                (T (cons  (simplify-CD-block (extract-block 0 DIALOG) SIMPLIFICATIONS)
                                (simplify-CD-dialog (cdr DIALOG) SIMPLIFICATIONS)))))

(provide :CD-simplification)
```

concept-neutralization.lsp

```
;;;;;;;;;;;;;;; Set of functions for concepts neutralization ;;;;;;;;;;;;;;;

(require :dialog-manipulation)

(defun lexic-value (N-CONCEPT LEXIC) ;Gives back the value of lexic which has N-CONCEPT as concept
        (cond   ((NULL LEXIC) '!UNKNOWN-CONCEPT!)
                ((eq (car (last (car LEXIC))) N-CONCEPT) (caar LEXIC))
                (T (lexic-value N-CONCEPT (cdr LEXIC)))))

(defun check-if-person (WORD CONCEPT-BASE)
        (cond   ((NULL CONCEPT-BASE) NIL)
                ((and (eq  WORD (caar CONCEPT-BASE)) (eq (car (last (car CONCEPT-BASE))) 'PERSON)) T)
                (T (check-if-person WORD (cdr CONCEPT-BASE)))))

(defun neutralize-list-person (L LEXIC CONCEPT-BASE)
        (cond   ((NULL L) NIL)
                (T (cons (neutralize-person (car L) LEXIC CONCEPT-BASE) (neutralize-list-person (cdr L) LEXIC
CONCEPT-BASE)))))

(defun neutralize-person (WORD LEXIC CONCEPT-BASE)
        (cond   ((eq (car (replace-word WORD LEXIC '(PERSON) CONCEPT-BASE)) '!UNKNOWN!) NIL)
                (T (replace-word WORD LEXIC '(PERSON) CONCEPT-BASE))))

(defun count-atoms (L) ;count number of atoms in list L
        (cond   ((NULL L) 0)
                ((atom L) 0)
                ((atom (car L)) (+ 1 (count-atoms (cdr L))))
                (t (+ 0  (count-atoms (cdr L))))))

(defun word-conceptp (CONCEPT-WORD)         (= (count-atoms CONCEPT-WORD) 6))

(defun find-concept-type (CONCEPT-WORD CONCEPT-BASE)    ; returns type of Concept CONCEPT-WORD according to
CONCEPT-BASE dictionary CONCEPT-BASE
        (cond   ((NULL CONCEPT-BASE) '!UNDEFINED-CONCEPT!)
                ((equal CONCEPT-WORD (caar CONCEPT-BASE)) (car (last (car CONCEPT-BASE))))
                (T (find-concept-type CONCEPT-WORD (cdr CONCEPT-BASE)))))

(defun make-predictions (CURRENT-CONCEPT PREDICTION)
        (cond   ((NULL PREDICTION) '(AFFIRMATION))
                ((equal CURRENT-CONCEPT (caar PREDICTION)) (cadar PREDICTION))
                (T (make-predictions CURRENT-CONCEPT (cdr PREDICTION)))))

(defun replace-word (WORD LEXIC L-PREDICTIONS CONCEPT-BASE)
        (cond   ((NULL LEXIC) '(!UNKNOWN! TINDEF PINDEF GINDEF NINDEF *UNKNOWN-WORD*))
                ((and (equal (caar LEXIC) WORD) (member (find-concept-type (car (last (car LEXIC))) CONCEPT-
BASE) L-PREDICTIONS))
                                (car LEXIC))
                (T (replace-word WORD (cdr LEXIC) L-PREDICTIONS CONCEPT-BASE))))

(defun neutralize-concepts-words (WORDS CURRENT-CONCEPT LEXIC CONCEPT-BASE PREDICTION)
        (cond   ((NULL WORDS) NIL)
                (T (cons (replace-word (car WORDS) LEXIC (make-predictions CURRENT-CONCEPT PREDICTION)
CONCEPT-BASE)
                                (neutralize-concepts-words (cdr WORDS)
```

```
                                        (find-concept-type (car (last (replace-word (car WORDS) LEXIC (make-predictions
CURRENT-CONCEPT PREDICTION) CONCEPT-BASE))) CONCEPT-BASE)
                                        LEXIC CONCEPT-BASE PREDICTION)))))

(defun neutralize-concepts-sentence (SENTENCE LEXIC CONCEPT-BASE PREDICTION)
        (cond   ((NULL SENTENCE) NIL)
                (T (cons (extract-attributes SENTENCE)
            (cons (neutralize-concepts-words (extract-words SENTENCE) 'AFFIRMATION LEXIC CONCEPT-BASE
PREDICTION)  NIL)))))

(defun neutralize-concepts-content (CONTENT LEXIC CONCEPT-BASE PREDICTION)
        (cond   ((NULL CONTENT) NIL)
                (T (cons   (neutralize-concepts-sentence (extract-sentence 0 CONTENT) (cdr LEXIC) CONCEPT-BASE
PREDICTION)
                                        (neutralize-concepts-content (cdr CONTENT) LEXIC CONCEPT-BASE
PREDICTION)))))

(defun neutralize-concepts-block (ABLOCK LEXIC CONCEPT-BASE PREDICTION)
        (cons (neutralize-list-person (extract-actors-from ABLOCK) (cdr LEXIC) CONCEPT-BASE)
          (cons (neutralize-list-person (extract-actors-to ABLOCK) (cdr LEXIC) CONCEPT-BASE)
                (neutralize-concepts-content (extract-content ABLOCK) LEXIC CONCEPT-BASE PREDICTION))))

(defun neutralize-concepts-dialog (DIALOG LEXIC CONCEPT-BASE PREDICTION)
        (cond   ((NULL DIALOG) NIL)
                (T (cons   (neutralize-concepts-block (extract-block 0 DIALOG) LEXIC CONCEPT-BASE PREDICTION)
                                        (neutralize-concepts-dialog (cdr DIALOG) LEXIC CONCEPT-BASE
PREDICTION)))))

(provide :concept-neutralization)
```

dialog-construction.lsp

```
;;;;;;; Set of functions for dialog data structure construction ;;;;;;;;;;;

 (require :string-manipulation)

(defun begin-block (A L) (cdr (member A L)))

(defun end-block (A L R) "search in L block ending with A and returns this block plus rest of L in R"
        (cond   ((NULL L) (progn (set R ()) ()) )
                ((eq (car L) A) (progn (set R (cdr L)) () ))
                (T (cons (car L) (end-block A (cdr L) R)))))

(defun make-block (ATOME LIST) "make blocks of type ATOME block ATOME inside list LIST"
        (let ((R 'RESULT))
          (cond   ((NULL LIST) () )
                ((begin-block ATOME LIST) (cons (end-block ATOME (begin-block ATOME LIST) R) (make-block
ATOME (eval R)))))))

(defun analyze-word (W) ;analyze word W and return (!) if W ends with ! , etc...;
        (let    ((L ()))
                (setq L (explode W))
                (cond   ((member '! L) 'EXCLAMATION)
                        ((member '|?| L) 'INTERROGATION)
                        ((member '|.| L) 'NORMAL)
                        (T 'CONTINUE))))

(defun extract-sentence-w (TEXT REST TYPE)    ;extract from text first sentence (without punctuation) and put rest in rest
                                                                                ;type is set to the type of punctuation
(exclamatio, interrogation, normal)
        (cond
                (       (NULL TEXT)
                        (set REST NIL)
                        (set TYPE 'NOTHING)
                        ())

                (       (eq (analyze-word (car TEXT)) 'EXCLAMATION)
                        (set TYPE 'EXCLAMATION)
                        (set REST (cdr TEXT))
                        (List (remove-punctuation-from-atom (car TEXT))))

                (       (eq (analyze-word (car TEXT)) 'INTERROGATION)
                        (set TYPE 'INTERROGATION)
                        (set REST (cdr TEXT))
                        (List (remove-punctuation-from-atom (car TEXT))))

                (       (eq (analyze-word (car TEXT)) 'NORMAL) (set TYPE 'NORMAL)
                        (set REST (cdr TEXT))
                        (List (remove-punctuation-from-atom (car TEXT))))

                (       T
                        (cons (car TEXT) (extract-sentence-w (cdr TEXT) REST TYPE) ) ) ) )

(defun make-content (TEXT)  ;transform text in content
                                ;Content is of type (sentence1 ... sentencen)
                                ;where sentence is ( (LANGUAGE TENSE TYPE FORM) (W1 ... Wn) )
```

```
                                        ;where LANGUAGE is the language os the sentence
                                        ;where TENSE is the tense os the sentence
                                        ;where TYPE is the type of the sentence (?!.)
                                        ;where FORM is the form of the sentence (afirmative, negative, ...)
                                        ;W are words
  (let ((RES 'VAR-RESULT) (TYPE 'VAR-TYPE) (LIST-WORDS))
          (cond
                  ((NULL TEXT) NIL)
                  (T          (setq LIST-WORD (extract-sentence-w TEXT RES TYPE))
                              (cons (cons (list 'UNKNOWN 'UNKNOWN (eval TYPE) 'UNKNOWN) (list LIST-WORD)) (make-
content (eval RES)))))))

(defun make-dialog (DIALOG-LIST) "make dialog style from list extracted from textfile"
  (let ((RESULT NIL) (ALIST NIL))
          (dolist     (ABLOCK (make-block '@BLOCK@ DIALOG-LIST))
                              (push (append       (make-block '@ACTOR@ ABLOCK)
                                                          (make-content (car (make-block '@TEXT@ ABLOCK)))) ALIST))

          (reverse ALIST)))

(provide :dialog-construction)
```

dialog-manipulation.lsp

```
;;;;;;; Set of functions for dialog data structure manipulation ;;;;;;;;;;;

(defun extract-block (N DIALOG)
        (nth N DIALOG))

(defun extract-actors-from (ABLOCK)
        (car ABLOCK))

(defun extract-actors-to (ABLOCK)
        (cadr ABLOCK))

(defun extract-all-actors (DIALOG)
        (flatten (mapcar #'(lambda (X) (append (extract-actors-from X) (extract-actors-to X)))
                        DIALOG)))

(defun extract-content (ABLOCK)
        (cddr ABLOCK))

(defun extract-sentence (N CONTENT)
        (nth N CONTENT))

(defun extract-attributes (SENTENCE)
        (car SENTENCE))

(defun extract-words (SENTENCE)
        (cadr SENTENCE))

(defun set-language (SENTENCE VALUE)
        (cons (cons VALUE (cdr (extract-attributes SENTENCE))) (cdr SENTENCE)))

(defun set-tense (SENTENCE VALUE)
  (cons
        (list (car (extract-attributes SENTENCE)) VALUE (caddr (extract-attributes SENTENCE)) (cadddr (extract-
attributes SENTENCE)))
        (cdr SENTENCE)))

(defun set-punctuation (SENTENCE VALUE)
  (cons
        (list (car (extract-attributes SENTENCE)) (cadr (extract-attributes SENTENCE)) VALUE (cadddr (extract-attributes
SENTENCE)))
        (cdr SENTENCE)))

(defun set-form (SENTENCE VALUE)
  (cons
        (list (car (extract-attributes SENTENCE)) (cadr (extract-attributes SENTENCE)) (caddr (extract-attributes
SENTENCE)) VALUE)
        (cdr SENTENCE)))

(provide :dialog-manipulation)
```

file-manipulation.lsp

```
;;;;;;;;;;;;;;;;;;;; Set of functions for file manipulation ;;;;;;;;;;;;;;

(require :string-manipulation)

(defun make-reserve-name (S) "make reserve name of file named S: S.xxx => S.res"
        (let  ((name) (l () ))
                (setq name (read-from-string S))
                (setq l (cdddr (reverse (explode name))))
                (push 'r l) (push 'e l)(push 's l)(reverse l)
                (list-to-string2 (reverse l))))

(defun can-reserve-file? (NAME) "if file NAME is not reserved, reserve file, if not, return false"
        (let ((NAME-RES) (FILE NIL) (FILE2 NIL))
                (setq NAME-RES (make-reserve-name (string NAME)))
                (setq FILE (open (read-from-string NAME-RES) :if-does-not-exist nil))
                (if (not FILE)
                        (setq FILE2 (open (read-from-string NAME-RES) :if-does-not-exist :create))
                        (close FILE))
                (if FILE2 (close FILE2))
                (not FILE)))

(defun can-free-file? (NAME) "if file NAME is reserved, free file, if not, return false"
        (let ((NAME-RES) (FILE) (FILE2))
                (setq NAME-RES (make-reserve-name (string NAME)))
                (setq FILE (open (read-from-string NAME-RES) :if-does-not-exist nil))
                (if (not FILE) () (delete-file FILE))
                (not (not FILE))))

(defun read-text-file (NAME-FILE)  "read text file, and return content of file with list of symbols"
        (let  ( (FILE) (TEXT) (LINE "") )
                (setq FILE (open NAME-FILE :direction :input))
                (loop
                        (setq LINE (read-line FILE nil 'eof))
                        (if     (not (eq LINE 'eof))
                                (setq TEXT (append TEXT (string-to-list (remove-punctuation LINE))))
                                (return) ))
                (close FILE)
        (car (list TEXT))))

(defun read-symbol-file (NAME-FILE)  "read text file, and return content of file with list of symbols"
        (let  ( (FILE) (TEXT NIL) (SYMB) )
                (setq FILE (open NAME-FILE :direction :input))
                (loop
                        (setq SYMB (read FILE nil 'eof))
                        (if     (not (eq SYMB 'eof))
                                (push SYMB TEXT)
                                (return) ))
                (close FILE)
        (car TEXT)))

(defun write-symbol-file (VAR-LIST NAME-FILE ACTION)  "write list of symbols in text file"
                ;if ACTION=append, append elements of VAR-LIST to file NAME-FILE
                ;if ACTION=supersede, supersede elements of VAR-LIST to file NAME-FILE
                ;if ACTION=overwrite, overwrite elements of VAR-LIST to file NAME-FILE
                ;default is supersede
        (let  ( (FILE) (LINE "") )
```

```
                    (cond
                        ((eq ACTION 'append)        (setq FILE (open NAME-FILE :direction :output :if-exists :append
:if-does-not-exist :create)))
                        ((eq ACTION 'overwrite) (setq FILE (open NAME-FILE :direction :output :if-exists :overwrite :if-
does-not-exist :create)))
                        (T                                  (setq FILE (open NAME-FILE :direction :output :if-
exists :supersede :if-does-not-exist :create))))
                    (if (not (null VAR-LIST)) (write VAR-LIST :stream FILE))
                    (close FILE)))

(defun write-text-file (VAR-LIST NAME-FILE ACTION)  "write list of symbols in text file"
                    ;if ACTION=append, append elements of VAR-LIST to file NAME-FILE
                    ;if ACTION=supersede, supersede elements of VAR-LIST to file NAME-FILE
                    ;if ACTION=overwrite, overwrite elements of VAR-LIST to file NAME-FILE
                    ;default is supersede
          (let  ( (FILE) (LINE "") )
                    (cond
                        ((eq ACTION 'append)        (setq FILE (open NAME-FILE :direction :output :if-exists :append
:if-does-not-exist :create)))
                        ((eq ACTION 'overwrite) (setq FILE (open NAME-FILE :direction :output :if-exists :overwrite :if-
does-not-exist :create)))
                        (T                                  (setq FILE (open NAME-FILE :direction :output :if-
exists :supersede :if-does-not-exist :create))))
                    (if (not (null VAR-LIST)) (write-line (list-to-liststring VAR-LIST) FILE))
                    (close FILE)))

(provide :file-manipulation)
```

language-recognition.lsp

```
;;;;;;;;;;;;;;;; Set of functions for language recognition ;;;;;;;;;;;;;;;;

(require :dialog-manipulation)

(defvar *LIMIT*) (setq *LIMIT* 0.74)

(defun count-recognized-words (WORDS DICTIONARY)
        (cond   ((NULL WORDS) 0)
                ((NULL DICTIONARY) 0)
                (T (+ (count (caar DICTIONARY) WORDS) (count-recognized-words WORDS (cdr DICTIONARY))))))

(defun recognize-language (WORDS DICTIONARY) "returns the probability of the bench of WORDS being writen in
language of DICTIONARY"
        (if (not (NULL DICTIONARY))
        (/ (count-recognized-words WORDS DICTIONARY) (length WORDS))))

(defun set-language-sentence (SENTENCE DICTIONARY)
        (if     (> (recognize-language (extract-words SENTENCE) (cdr DICTIONARY)) *LIMIT*)
                (set-language SENTENCE (car DICTIONARY))
                (car (list SENTENCE))))

(defun set-language-content (CONTENT DICTIONARY)
        (cond   ((NULL CONTENT) NIL)
                (T (cons  (set-language-sentence (extract-sentence 0 CONTENT) DICTIONARY)
                                (set-language-content (cdr CONTENT) DICTIONARY)))))

(defun set-language-block (ABLOCK DICTIONARY)
        (cons (extract-actors-from ABLOCK)
          (cons (extract-actors-to ABLOCK)
                (set-language-content (extract-content ABLOCK) DICTIONARY))))

(defun set-language-dialog (DIALOG DICTIONARY)
        (cond   ((NULL DIALOG) NIL)
                (T (cons  (set-language-block (extract-block 0 DIALOG) DICTIONARY)
                                (set-language-dialog (cdr DIALOG) DICTIONARY)))))

(provide :language-recognition)
```

list-people-knowing-people-skill-construction.lsp

```
;; Set of functions for List people/skill construction ;;;;;;;;
; creates a list of all knowing-people skills

(defun set-knowing-people-skill (PEPNET)
        (cond ((NULL PEPNET) nil)
                (T  (cons (cons     (caar PEPNET)  (cons         'KPOSS
                                                (list (cons         'OBJECT
                                                        (cons (cons 'QUALIFICATION (cons (cons 'QUALIFYER (cons
NIL NIL)) (cons (cons 'QUALIFIED (cons (cadar PEPNET) NIL)) NIL))) NIL)))))
                        (set-knowing-people-skill (cdr PEPNET))))))

(provide :list-people-knowing-people-skill-construction)
```

list-people-required-skill-construction.lsp

```
;; Set of functions for List people/ require skill construction ;;;;;;;;

(require :dialog-manipulation)
(require :cd-functions)

(defun set-required-skill-words (WORDS)
  (remove nil
          (cond     ((atom (car WORDS))
                          NIL)
                    ((NULL (header-cd (car WORDS)))
                          NIL)
                    ((and (member (header-cd (car WORDS)) *CD-ACTS*) (not (eq (header-cd (car WORDS))
'QUALIFICATION)))

                                    (append
                                              (cons
                                                        (cons (cdpath '(QUALIFIED) (filler-role 'actor (car WORDS))) (remove-all-
pairs-with (filler-role 'actor (car WORDS)) (car WORDS)))
                                                        (set-required-skill-words (roles-cd (car WORDS))))
                                              (set-required-skill-words (cdr WORDS))))
                    (T
                              (append
                                        (set-required-skill-words (roles-cd (car WORDS)))
                                        (set-required-skill-words (cdr WORDS)))))))

(defun set-required-skill-sentence (SENTENCE)
                    (set-required-skill-words (cons (extract-words SENTENCE) NIL)))

(defun set-required-skill-content (CONTENT)
          (cond     ((NULL CONTENT) NIL)
                    (T (append          (set-required-skill-sentence (extract-sentence 0 CONTENT))
                                        (set-required-skill-content (cdr CONTENT))))))

(defun set-required-skill-block (ABLOCK)          (set-required-skill-content (extract-content ABLOCK)))

(defun set-required-skill-dialog (DIALOG)
          (cond     ((NULL DIALOG) NIL)
                    (T (append (set-required-skill-block (extract-block 0 DIALOG))
                              (set-required-skill-dialog (cdr DIALOG))))))

(provide :list-people-required-required-skill-construction)
```

list-people-skill-construction.lsp

```
;; Set of functions for List people/skill construction ;;;;;;;;

(require :dialog-manipulation)
(require :cd-functions)

(defun set-skill-words (WORDS)
  (remove nil       (cond
                    ((atom (car WORDS))
                              ;(user-trace "~1%  0---")
                              NIL)
                    ((NULL (header-cd (car WORDS)))
                              ;(user-trace "~1%  1---")
                              NIL)
                    ((and (member (header-cd (car WORDS)) *CD-ACTS*) (not (eq (header-cd (car WORDS))
'QUALIFICATION)))
                              ;(user-trace "~1%  2---")
                              ;(user-trace "~1%  on a: ~s" (cons (filler-role 'actor (car WORDS)) (remove-role 'actor (car
WORDS))))
                              (append
                                        (cons
                                                  (cons (cdpath '(QUALIFIED) (filler-role 'actor (car WORDS))) (remove-role
'actor (car WORDS)))
                                                  (set-skill-words (roles-cd (car WORDS))))
                                        (set-skill-words (cdr WORDS))))
                    (T
                              ;(user-trace "~1%  3---")
                              (append
                                        (set-skill-words (roles-cd (car WORDS)))
                                        (set-skill-words (cdr WORDS)))))))

(defun set-skill-sentence (SENTENCE) (set-skill-words (cons (extract-words SENTENCE) NIL)))

(defun set-skill-content (CONTENT)
          (cond     ((NULL CONTENT) NIL)
                    ((eq (caddr (extract-attributes (extract-sentence 0 CONTENT))) 'interrogation)
                              (set-skill-content (cdr CONTENT)))
                    ((eq (cadddr (extract-attributes (extract-sentence 0 CONTENT))) 'negative)
                              (set-skill-content (cdr CONTENT)))
                    (T (append          (set-skill-sentence (extract-sentence 0 CONTENT))
                                        (set-skill-content (cdr CONTENT))))))

(defun set-skill-block (ABLOCK)         (set-skill-content (extract-content ABLOCK)))

(defun set-skill-dialog (DIALOG)
          (cond     ((NULL DIALOG) NIL)
                    (T (append (set-skill-block (extract-block 0 DIALOG))
                              (set-skill-dialog (cdr DIALOG))))))

(provide :list-people-skill-construction)
```

list-people-skill-matched-construction.lsp

```
;; Set of functions for List people/skill matched construction ;;;;;;;;

(require :cd-functions)
(require :people-skill-base-manipulation)

(defun list-skilled-people (CD PEOPLE-SKILL-BASE SKILL-BASE)
        (cond   ((eq (find-s CD SKILL-BASE) -1) NIL)
                (T      (list (remove nil (mapcar   #'(lambda (X) (if (member (find-s CD SKILL-BASE) X) (car X)))
                                                    PEOPLE-SKILL-BASE))))))

(defun list-matched (CD SKILL-BASE) ;returns list of matched cd of cd inside SKILL-BASE
        (cond   ((NULL SKILL-BASE) NIL)
                ((or (match CD (car SKILL-BASE) nil) (match (car SKILL-BASE) CD nil)) (cons (car SKILL-BASE) (list-
matched CD (cdr SKILL-BASE))))
                (T (list-matched CD (cdr SKILL-BASE)))))


(defun find-matched (PEOPLE-SKILL SKILL-BASE PEOPLE-SKILL-BASE)
; will return:
; ( (MATCHED-skill1 (P1 ...Pn) ... (matched-skillm (P1 ... Pn)))

        (mapcar #'(lambda (X) (cons X (list-skilled-people X PEOPLE-SKILL-BASE SKILL-BASE)))
                (list-matched (cdr PEOPLE-SKILL) SKILL-BASE)))

(defun make-list-matched (PEOPLE-SKILL-LIST SKILL-BASE PEOPLE-SKILL-BASE)
        (cond   ((NULL PEOPLE-SKILL-LIST) NIL)
                (T (append      (find-matched (car PEOPLE-SKILL-LIST) SKILL-BASE PEOPLE-SKILL-BASE)
                                (make-list-matched (cdr PEOPLE-SKILL-LIST) SKILL-BASE PEOPLE-SKILL-
BASE)))))

(provide :list-people-skill-matched-construction)
```

people-net-construction.lsp

```
;; Set of functions for People Net construction ;;;;;;;;

(require :dialog-manipulation)
(require :people-net-fct)
(require :concept-neutralization) ; required because of neutralize-person, lexic-value and others
(require :cd-functions)

; 1 - ACTORS-FROM and ACTORS-TO KNOW EACH OTHER
; 2 - ACTORS-FROM and ACTORS-TO KNOW EVERY PERSON EVOLVED IN CDs
; 3 - A PERSON ENVOLVED AS ACTOR IN A CD KNOWS EVERY PERSON EVOLVED IN THE OBJECT OF THIS CD

(defun set-people-words (WORDS PENET L-ACTORS CONCEPTS)
        (cond   ((NULL (header-cd WORDS))           PENET)
                ((check-if-person (header-cd WORDS) CONCEPTS) (make-all-r L-ACTORS (header-cd WORDS)
PENET))
                ((atom (header-cd WORDS)) PENET)
                ((and (eq (header-cd (car WORDS)) 'ACTOR) (check-if-person (car (roles-cd (car WORDS)))
CONCEPTS))      (set-people-words  (cdr WORDS)      (make-all-r L-ACTORS (car (roles-cd (car WORDS))) PENET)
                                   (cons (car (roles-cd (car WORDS))) L-ACTORS)  CONCEPTS))
                (T        (set-people-words  (cdr WORDS)
                                             (set-people-words (roles-cd (car WORDS)) PENET L-ACTORS
CONCEPTS)       L-ACTORS          CONCEPTS))))

(defun set-people-sentence (SENTENCE PENET ACTORS-FROM ACTORS-TO CONCEPTS)
                (set-people-words (cons (extract-words SENTENCE) NIL) PENET (cons ACTORS-FROM (cons
ACTORS-TO NIL)) CONCEPTS))

(defun set-people-content (CONTENT PENET ACTORS-FROM ACTORS-TO CONCEPTS)
        (cond   ((NULL CONTENT) (remove-all-nil-r
                                        (put-r
                                                (make-r ACTORS-FROM ACTORS-TO)
                                                (put-r (make-r ACTORS-TO ACTORS-FROM) PENET))))
                ((eq (caddr (extract-attributes (extract-sentence 0 CONTENT))) 'interrogation)
                        (set-people-content (cdr CONTENT) PENET ACTORS-FROM ACTORS-TO CONCEPTS))
                ((eq (cadddr (extract-attributes (extract-sentence 0 CONTENT))) 'negative)
                        (set-people-content (cdr CONTENT) PENET ACTORS-FROM ACTORS-TO CONCEPTS))
                (T (set-people-content          (cdr CONTENT)
                                                (set-people-sentence (extract-sentence 0 CONTENT) PENET ACTORS-
FROM ACTORS-TO CONCEPTS)       ACTORS-FROM    ACTORS-TO         CONCEPTS))))

(defun set-people-block (ABLOCK PENET CONCEPTS)
        (set-people-content  (extract-content ABLOCK) PENET (car (extract-actors-from ABLOCK)) (car (extract-actors-to
ABLOCK)) CONCEPTS))


(defun set-people-dialog (DIALOG PENET CONCEPTS)
        (cond   ((NULL DIALOG) PENET)
                (T (set-people-dialog (cdr DIALOG) (set-people-block (extract-block 0 DIALOG) PENET CONCEPTS)
                        CONCEPTS))))

(defun tell-path (R NET LEXIC)
        (cond   ((NULL (best-path R NET)) (append     '(There is no way that)  (cons (lexic-value (car R) LEXIC) nil)
'(can reach)        (cons (lexic-value (cadr R) LEXIC) nil)'(with his relationships.)))
                (T (append          '(IF)
                                    (cons (lexic-value (car R) LEXIC) nil)
```

```
                                    '(wants to reach)     (cons (lexic-value (cadr R) LEXIC) nil)
                                    '(THEN)   (tell-found-path (best-path R NET) LEXIC)))))

(defun tell-found-path (FOUND-PATH LEXIC)
        (cond     ((NULL FOUND-PATH) '(THE PROBLEM WILL BE SOLVED.))
                  (T (append        (cons (lexic-value (caar FOUND-PATH) LEXIC) nil)        '(CAN TALK TO)
                                    (cons (lexic-value (cadar FOUND-PATH) LEXIC) nil)       '(AND)
                                    (tell-found-path (cdr FOUND-PATH) LEXIC)))))

(provide :people-net-construction)
```

tense-recognition.lsp

```
;;;;;;;;;;;;;;;; Set of functions for Tense recognition ;;;;;;;;;;;;;;;;;;;

(require :dialog-manipulation)

(defun exist-past-tense (WORDS)
        (cond     ((NULL WORDS) NIL)
                  ((eq (nth 1 (car WORDS)) 'PAST) T)
                  (T (exist-past-tense (cdr WORDS)))))

(defun set-tense-sentence (SENTENCE)
        (cond     ((exist-past-tense (extract-words SENTENCE)) (set-tense SENTENCE 'PAST))
                  (T (set-tense SENTENCE 'PRESENT))))

(defun set-tense-content (CONTENT)
        (cond     ((NULL CONTENT) NIL)
                  (T (cons  (set-tense-sentence (extract-sentence 0 CONTENT))
                            (set-tense-content (cdr CONTENT))))))

(defun set-tense-block (ABLOCK)
        (cons (extract-actors-from ABLOCK)
          (cons (extract-actors-to ABLOCK)
                  (set-tense-content (extract-content ABLOCK)))))

(defun set-tense-dialog (DIALOG)
        (cond     ((NULL DIALOG) NIL)
                  (T (cons  (set-tense-block (extract-block 0 DIALOG))  (set-tense-dialog (cdr DIALOG))))))

(provide :tense-recognition)
```

people-net-fct.lsp

```
;; Set of functions for manipulating People Nets ;;;;;;;

(defun make-all-r (L-ACTORS PERSON NET)
        (cond   ((NULL L-ACTORS) NET)
                (T (make-all-r (cdr L-ACTORS) PERSON (put-r (make-r (car L-ACTORS) PERSON) NET)))))

(defun remove-all-nil-r (NET)
        (cond   ((NULL NET) NIL)
                ((or (NULL (caar NET)) (NULL (cadar NET)) (eq (caar NET) (cadar NET))) (remove-all-nil-r (cdr NET)))
                (T (cons (car NET) (remove-all-nil-r (cdr NET))))))

(defun make-r (A B)             (cons A (cons B NIL)))

(defun member-r (R NET)
        (cond   ((NULL NET) NIL)
                ((NULL R) NIL)
                ((equal R (car NET)) (cons R (cdr NET)))
                (T (member-r R (cdr NET)))))

(defun put-r (R NET)
        (cond   ((member-r R NET) NET)      (T (cons R NET))))

(defun remove-r (R NET)
        (cond   ((NULL NET) NIL)
                ((NULL R) NIL)
                ((equal R (car NET)) (remove-r R (cdr NET)))
                (T (cons (car NET) (remove-r R (cdr NET))))))

(defun all-start-with (P1 NET)
        (cond   ((NULL NET) NIL)
                ((NULL P1) NIL)
                ((eq P1 (caar NET)) (cons (car NET) (all-start-with P1 (cdr NET))))
                (T (all-start-with P1 (cdr NET)))))

(defun found-path (R RES)
        (cond   ((NULL RES) NIL)
                ((NULL R) NIL)
                ((eq (cadr R) (cadr (car (last (car RES))))) (car RES))
                (T (found-path R (cdr RES)))))


(defun best-path1 (R NET RES)
        (cond   ((NULL NET) NIL)
                ((NULL R) NIL)
                ((member-r R NET) (cons R NIL))
                (T (best-path2 R NET (mapcar  #'(lambda (X) (cons X NIL))  (all-start-with (car R) NET)) ))))

(defun flat-1-level (L)
        (cond   ((NULL L) NIL)
                ((LISTP (car L)) (append (car L) (flat-1-level (cdr L))))
                (T (cons (car L) (flat-1-level (cdr L))))))

(defun all-pathes-with-1+step (A NET)
        (mapcar #'(lambda (X) (reverse (cons X (reverse A))))
                        (all-start-with (cadr (car (last A))) NET)))
```

```
(defun is-dead (PATH)
        (cond   ((NULL PATH) NIL)
                ((member-r (car PATH) (cdr PATH)) T)
                (T (is-dead (cdr PATH)))))

(defun kill-dead-pathes (RES)
        (cond   ((NULL RES) NIL)
                ((is-dead (car RES)) (kill-dead-pathes (cdr RES)))
                (T (cons (car RES) (kill-dead-pathes (cdr RES))))))

(defun best-path2 (R NET RES)
        (cond   ((NULL NET) NIL)
                ((NULL R) NIL)
                ((NULL RES) NIL)
                ((member (cadr R) (mapcar #'(lambda (X) (cadr (car (last X)))) RES)) (found-path R RES))
                (T  (best-path2 R NET
                        (kill-dead-pathes
                                (flat-1-level
                                        (mapcar #'(lambda (X) (all-pathes-with-1+step X NET)) RES)))))))

(defun best-path (R NET)      (best-path1 R NET NIL))

(provide :people-net-fct)
```

people-skill-base-manipulation.lsp

```
;; Set of functions for Skill Base and People Base Manipulation ;;;;;;;

(require :cd-functions)

;;;;;;;;;;;;;; usefull fct ;;;;;;;;;;;;;;;;;;
(defun remove-number (L)
        (cond   ((NULL L) NIL)
                ((numberp (car L)) (cdr L))
                (T (cons (car L) (remove-number (cdr L))))))

(defun find-number (L)
        (cond   ((NULL L) NIL)
                ((numberp (car L)) (car L))
                (T (find-number (cdr L)))))

(defun put-number-first (L)
        (cond   ((NULL (find-number L)) (cons '-1 L))
                (T (cons (find-number L) (remove-number L)))))

(defun remove-s1 (S SB N)
        (cond   ((NULL S) (cons '-1 SB))
                ((NULL SB) (cons '-1 SB))
                ((equal S (car SB)) (cons N (cdr SB)))
                (T (cons (car SB) (remove-s1 S (cdr SB) (+ 1 N))))))

(defun number-s (S SB) ;returns the number of skill S in SB. If this number > lenghth SB, S does not exist in SB
        (cond   ((NULL SB) 1)
                ((NULL S) 1)
                ((equal S (car SB)) 1)
                (T (+ 1 (number-s S (cdr SB))))))

;;;;;;;;;;;;;; Skill Base ;;;;;;;;;;;;;;;;;;
(defun count-s (SB) ; counts number of skills inside SB
        (cond   ((NULL SB) '0)
                (T (+ 1 (count-s (cdr SB))))))

(defun get-s (N SB) ; gets skills number N inside of SB       (nth (- N 1) SB))

(defun member-s (S SB) ; returns T if S exists inside of SB, nil if not
        (cond   ((NULL SB) NIL)
                ((NULL S) NIL)
                ((equal S (car SB)) (cons S (cdr SB)))
                (T (member-s S (cdr SB)))))

(defun put-s (S SB)             ; (Car) of Skill-base returned is the position where S has been included.
                                ; (Cdr) of Skill-base returned is SB modified
        (cond   ((member-s S SB) (cons (find-s S SB) SB))
                (T (cons (+ 1 (count-s SB)) (reverse (cons S (reverse SB)))))))

(defun remove-s (S SB)  ; (Car) of Skill-base returned is the position where S has been deleted.
                                ; If this position = -1, then S did not exist inside SB
                                ; (Cdr) of Skill-base returned is SB modified
        (put-number-first (remove-s1 S SB 1)))
```

```
 (defun remove-s-n (N SB)  ; Remove skill number n inside of SB
          (cond      ((NULL SB) NIL)
                     ((< N 1) SB)
                     ((eq N 1) (cdr SB))
                     (T (cons (car SB) (remove-s-n (- N 1) (cdr SB))))))

(defun find-s (S SB) ; retruns number of S inside SB, returns -1 if S does not exist in SB
          (cond      ((not (member-s S SB)) '-1)
                     (T (number-s S SB))))

;;;;;;;;;;;;;; People / Skill Base ;;;;;;;;;;;;;;;;;;
(defun count-ps (PSB) ; counts number of skills inside SB
          (cond      ((NULL PSB) '0)
                     (T (+ 1 (count-ps (cdr PSB))))))


(defun get-ps (N PSB) ; gets people-skill relation number N inside of PSB (nth (- N 1) PSB))

(defun member-ps (PS PSB) ; returns T if PS exists inside of PSB, nil if not
          (cond      ((NULL PSB) NIL)
                     ((NULL PS) NIL)
                     ((equal PS (car PSB)) (cons PS (cdr PSB)))
                     (T (member-ps PS (cdr PSB)))))

(defun put-ps (PS PSB)         ; (Car) of People-Skill-base returned is the position where PS has been included.
                               ; (Cdr) of People-Skill-base returned is PSB modified
          (cond      ((member-ps PS PSB) (cons (find-ps PS PSB) PSB))
                     (T (cons (+ 1 (count-ps PSB)) (reverse (cons PS (reverse PSB)))))))

(defun remove-ps (PS PSB)  ; (Car) of People-Skill-base returned is the position where PS has been deleted.
                             ; If this position = -1, then PS did not exist inside PSB
                             ; (Cdr) of Skill-base returned is PSB modified
          (put-number-first (remove-s1 PS PSB 1)))


(defun remove-ps-n (N PSB)  ; Remove skill number n inside of SB
          (cond      ((NULL PSB) NIL)
                     ((< N 1) PSB)
                     ((eq N 1) (cdr PSB))
                     (T (cons (car PSB) (remove-ps-n (- N 1) (cdr PSB))))))

(defun find-ps (PS PSB) ; retruns number of PS inside PSB, returns -1 if PS does not exist in PSB
          (cond      ((not (member-ps PS PSB)) '-1)
                     (T (number-s PS PSB))))
(provide :people-skill-base-manipulation)
```

person-concept-substitution.lsp

```
;; Set of functions for person-concept substitution ;;;;;;
;; PERSONAL PRONOM ARE SUBSTITUTED BY ACTOR-FROM AND ACTOR TO, ACCODRING TO VARIABLE
SUBSTITUTION

(require :dialog-manipulation)
(require :concept-neutralization)

(defun substitute-actors (TO-SUBSTITUTE ACTORS-FROM ACTORS-TO)
        (cond   ((NULL TO-SUBSTITUTE) NIL)
                ((equal (car TO-SUBSTITUTE) 'ACTOR-FROM) (append ACTORS-FROM (substitute-actors (cdr TO-
SUBSTITUTE) ACTORS-FROM ACTORS-TO)))
                ((equal (car TO-SUBSTITUTE) 'ACTOR-TO) (append ACTORS-TO (substitute-actors (cdr TO-
SUBSTITUTE) ACTORS-FROM ACTORS-TO)))
                (T (append      (list (car TO-SUBSTITUTE))
                                (substitute-actors (cdr TO-SUBSTITUTE) ACTORS-FROM ACTORS-TO)))))

(defun make-word-substitution (MWORD TO-SUBSTITUTE ACTORS-FROM ACTORS-TO)
        (cond   ((NULL TO-SUBSTITUTE) MWORD)
                (T (cons (substitute-actors TO-SUBSTITUTE ACTORS-FROM ACTORS-TO) (cdr MWORD)))))

(defun find-word-in-substitution (CONCEPT-BASE SUBSTITUTION)
        (cond   ((NULL SUBSTITUTION) NIL)
                ((equal CONCEPT-BASE (caar SUBSTITUTION)) (cadar SUBSTITUTION))
                (T (find-word-in-substitution CONCEPT-BASE (cdr SUBSTITUTION)))))

(defun substitute-concept-word (MWORD ACTORS-FROM ACTORS-TO SUBSTITUTION)
        (cond   ((find-word-in-substitution (car (last MWORD)) SUBSTITUTION)
                        (make-word-substitution MWORD
                          (find-word-in-substitution (car (last MWORD)) SUBSTITUTION)  ACTORS-FROM
                                                                ACTORS-TO))       (T MWORD)))

(defun substitute-concept-words (WORDS ACTORS-FROM ACTORS-TO SUBSTITUTION)
        (cond   ((NULL WORDS) NIL)
                (T (cons  (substitute-concept-word (car WORDS) ACTORS-FROM ACTORS-TO SUBSTITUTION)
                        (substitute-concept-words (cdr WORDS) ACTORS-FROM ACTORS-TO SUBSTITUTION)))))

(defun reconstruct-words (WORDS)
        (cond   ((NULL WORDS) NIL)
                ((listp (caar WORDS))
                        (append (caar WORDS) (reconstruct-words (cdr WORDS))))
                (T (cons (car WORDS) (reconstruct-words (cdr WORDS))))))

(defun substitute-concept-sentence (SENTENCE ACTORS-FROM ACTORS-TO SUBSTITUTION)
        (cond   ((NULL SENTENCE) NIL)
                (T (cons (extract-attributes SENTENCE)  (cons (reconstruct-words
                 (substitute-concept-words (extract-words SENTENCE) ACTORS-FROM ACTORS-TO
SUBSTITUTION)) NIL)))))

(defun substitute-concept-content (CONTENT ACTORS-FROM ACTORS-TO SUBSTITUTION)
        (cond   ((NULL CONTENT) NIL)
                (T (cons  (substitute-concept-sentence (extract-sentence 0 CONTENT) ACTORS-FROM ACTORS-TO
SUBSTITUTION)  (substitute-concept-content (cdr CONTENT) ACTORS-FROM ACTORS-TO SUBSTITUTION)))))
```

```
(defun substitute-concept-block (ABLOCK SUBSTITUTION)
        (cons (extract-actors-from ABLOCK)
          (cons (extract-actors-to ABLOCK)
                (substitute-concept-content (extract-content ABLOCK) (extract-actors-from ABLOCK) (extract-actors-to
ABLOCK) SUBSTITUTION))))

(defun substitute-concept-dialog (DIALOG SUBSTITUTION)
        (cond   ((NULL DIALOG) NIL)
                (T (cons  (substitute-concept-block (extract-block 0 DIALOG) SUBSTITUTION)
                          (substitute-concept-dialog (cdr DIALOG) SUBSTITUTION)))))

(provide :person-concept-substitution)
```

result-explanation.lsp

```
;; Set of functions for results explanation ;;;;;;;;

(require :dialog-manipulation)
(require :concept-neutralization)
(require :people-net-construction)
(require :English-Generator)

(defun list-people (LISTEP LEXIC)
          (cond   ((NULL LISTEP) '(ABOUT FOLLOWING THE KNOW I))
                  (T (append      (list (lexic-value (car LISTEP) LEXIC))
                                  (if (not (null (cadr LISTEP))) '(AND))
                                  (list-people (cdr LISTEP) LEXIC)))))

(defun tell-all-pathes (LISTEP LIST-SKILL-PEOPLELIST PENET LEXIC CONCEPT-BASE DIALOG)
        (cond   ((NULL LISTEP) NIL)
                (t  (append (tell-path  (cons   (car (extract-actors-from (car (last DIALOG)))) (list (car LISTEP)))
                                                PENET LEXIC)
                            (tell-all-pathes (cdr LISTEP) LIST-SKILL-PEOPLELIST PENET LEXIC CONCEPT-
BASE DIALOG)))))

(defun tell-found-skills (LIST-SKILL-PEOPLELIST PENET LEXIC CONCEPT-BASE DIALOG)
        (cond   ((NULL LIST-SKILL-PEOPLELIST) '(THAT IS ALL.))
                (t (append      (reverse (list-people (cadar LIST-SKILL-PEOPLELIST) LEXIC))
                                '( - )
                                (cdr (say-cd-english
                                                (cons   (caaar LIST-SKILL-PEOPLELIST)
                                                        (cons   (cons 'ACTOR (cons (cons 'QUALIFICATION
(cons (cons 'QUALIFYER (cons NIL NIL)) (cons (cons 'QUALIFIED (cons (caadar LIST-SKILL-PEOPLELIST) NIL)) NIL)))
NIL))
                                                                        (cdaar LIST-SKILL-PEOPLELIST)))
                                                LEXIC CONCEPT-BASE))
                                '(!)
                                (tell-all-pathes (cadar LIST-SKILL-PEOPLELIST) LIST-SKILL-PEOPLELIST PENET
LEXIC CONCEPT-BASE DIALOG)
                                (tell-found-skills (cdr LIST-SKILL-PEOPLELIST) PENET LEXIC CONCEPT-BASE
DIALOG)))))

(provide :result-explanation)
```

string-manipulation.lsp

```
;;;;;;;;;;;;;;;;;; Set of functions for string manipulation ;;;;;;;;;;;;;;

(defun flatten (x)
        (labels ((rec (x acc)
        (cond ((null x) acc)
        ((atom x) (cons x acc))
        (t (rec (car x) (rec (cdr x) acc))))))
        (rec x nil)))

(defun wait(n) (let ((i)) (setq i 1)  (loop (setq i (+ 1 i)) (if (eq i n) (return)))))

(defun explode (sym)
        (map 'list #'(lambda (c)(intern (make-string 1 :initial-element c)))
        (symbol-name sym)))

(defun punctuation-p (chars) (find chars ",'")) "punctuation predicate"

(defun punctuation-list-p (L)  "test if ? or . or ! belong to a list" (or (member '! L) (member '? L) (member '|.| L)))

(defun remove-punctuation-from-atom (ATOME) "remove punctuation of punctuation-list-p from atom ATOME"
        (read-from-string (list-to-string2 (remove (car (punctuation-list-p (explode ATOME))) (explode ATOME)))))

(defun remove-punctuation (string) "Replace a strings punctuation characters with spaces."
  (substitute-if #\space #'punctuation-p string))

(defun string-to-list (string) "Convert a string to a list of words."
  (read-from-string (concatenate 'string "(" string ")")))

(defun liststring-to-list (L) "convert a list of string to a list of symbol"
        (cond ((NULL L) ()) (T (cons (read-from-string (car L)) (liststring-to-list (cdr L))))))

(defun list-to-string3 (L)  "convert a list of symbols to a string"
        (cond ((NULL L) "") (T (concatenate 'string (string (car L)) '" " (list-to-string3 (cdr l))))))

(defun list-to-string (L)  "convert a list of symbols to a string"
        (cond ((NULL L) "") (T (concatenate 'string (string (car L)) '"  " (list-to-string (cdr l))))))

(defun list-to-string2 (L)  "convert a list of symbols to a string"
        (cond ((NULL L) "") (T (concatenate 'string (string (car L)) (list-to-string2 (cdr l))))))

(defun list-to-list-in-string (L)  "convert a n-levels list to the string of this list"
        (concatenate 'string '"( " (list-to-liststring L) '")"))

(defun list-to-liststring (L)  "convert a n-levels list to the string of this list"
        (cond   ((NULL L)           "")
                ((numberp (car L)) (concatenate 'string (write-to-string (car L)) '"  " (list-to-liststring (cdr l))))
                ((atom (car L)) (concatenate 'string (string (car L)) '"  " (list-to-liststring (cdr l))))
                (T                      (concatenate 'string '"(" (list-to-liststring (car l)) '") " (list-to-liststring (cdr l))))))

(provide :string-manipulation)
```

unknown-concept-remotion.lsp

```
;; Set of functions for unknown concept remotion ;;;;;;
;; These function also delete all NIL phrases ;;
(require :dialog-manipulation)

(defun remove-unknown-concept-words (WORDS)
        (cond   ((NULL WORDS) NIL)
                ((equal (caar WORDS) '!UNKNOWN!) (remove-unknown-concept-words (cdr WORDS)))
                (T (cons (car WORDS) (remove-unknown-concept-words (cdr WORDS))))))

(defun remove-unknown-concept-sentence (SENTENCE)
        (cond   ((NULL SENTENCE) NIL)
                (T (cons (extract-attributes SENTENCE)  (cons (remove-unknown-concept-words (extract-words
SENTENCE))  NIL)))))

(defun remove-unknown-concept-content (CONTENT)
        (cond   ((NULL CONTENT) NIL)
                ((NULL (cadr (remove-unknown-concept-sentence (extract-sentence 0 CONTENT)))) (remove-unknown-
concept-content (cdr CONTENT)))
                (T (cons   (remove-unknown-concept-sentence (extract-sentence 0 CONTENT))
                                (remove-unknown-concept-content (cdr CONTENT))))))

(defun remove-unknown-concept-block (ABLOCK)
        (cons (extract-actors-from ABLOCK)   (cons (extract-actors-to ABLOCK)
                (remove-unknown-concept-content (extract-content ABLOCK)))))

(defun remove-unknown-concept-dialog (DIALOG)
        (cond   ((NULL DIALOG) NIL)
                (T (cons   (remove-unknown-concept-block (extract-block 0 DIALOG))
                                (remove-unknown-concept-dialog (cdr DIALOG))))))

(provide :unknown-concept-remotion)
```

useless-matched-remotion.lsp

```
;; Set of functions for useless people/skill matched remotion ;;;;;;;;
(require :dialog-manipulation)

(defun remove-identic-people (PEOPLE-LIST1 PEOPLE-LIST2)
        (cond   ((NULL PEOPLE-LIST1) NIL)
                ((member (car PEOPLE-LIST1) PEOPLE-LIST2) (remove-identic-people (cdr PEOPLE-LIST1) PEOPLE-
LIST2))
                (t (cons (car PEOPLE-LIST1) (remove-identic-people (cdr PEOPLE-LIST1) PEOPLE-LIST2)))))

(defun remove-useless (SKILL-PEOPLELIST-LIST PEOPLE-LIST)
        (cond   ((NULL SKILL-PEOPLELIST-LIST) NIL)
                ((NULL (remove-identic-people (cadar SKILL-PEOPLELIST-LIST) PEOPLE-LIST))
                        (remove-useless (cdr SKILL-PEOPLELIST-LIST) PEOPLE-LIST))
                (T (cons (cons      (caar SKILL-PEOPLELIST-LIST)
                                (cons (remove-identic-people (cadar SKILL-PEOPLELIST-LIST) PEOPLE-LIST)
NIL))
                        (remove-useless (cdr SKILL-PEOPLELIST-LIST) PEOPLE-LIST)))))

(defun remove-useless-matched (SKILL-PEOPLELIST-LIST DIALOG)
        (remove-useless SKILL-PEOPLELIST-LIST  (extract-all-actors DIALOG)))

(provide :useless-matched-remotion)
```