UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

IMPLEMENTAÇÃO DOS PREDICADOS DE MANIPULAÇÃO DE TERMOS NA BASE DE DADOS DE UM AMBIENTE DE PROGRAMAÇÃO LÓGICA

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

AORON BEYER

BLUMENAU, JUNHO/1999

1999/1-06

IMPLEMENTAÇÃO DOS PREDICADOS DE MANIPULAÇÃO DE TERMOS NA BASE DE DADOS DE UM AMBIENTE DE PROGRAMAÇÃO LÓGICA

AORON BEYER

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Roberto Heinzle — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Roberto Heinzle

_____

Prof. Maurício C. Lopes

_____

Prof. Marcel Hugo

DEDICATÓRIA

À minha namorada Iara e a minha mãe Ada.

AGRADECIMENTOS

Ao professor Roberto Heinzle, pela orientação no desenvolvimento deste trabalho, pelo incentivo e pelos ensinamentos.

Ao amigo Wendy, pelo companheirismo durante o curso e pelo exemplo de dedicação e empenho no estudo.

À namorada Iara, pela paciência durante o semestre de estudo e pelo incentivo e apoio durante esta fase.

A Deus que tem mostrado seu amor e me sustentado em todas as minhas necessidades.

RESUMO

Este trabalho é o estudo sobre o gerenciamento de uma base de dados para um ambiente de programação lógica. Descreve-se os sistemas especialistas seus conceitos, componentes, ferramentas para construção. São mostradas as principais características, a base de conhecimentos, o mecanismo de aprendizagem e aquisição do conhecimento, a máquina de inferência, o quadro negro, o sistema de justificação e o sistema de consulta. Como ferramentas de construção caracteriza-se as principais linguagens e shells. Incluindo, ainda, um estudo genérico sobre programação de computadores, linguagens procedurais imperativas e linguagens declarativas. Direciona o estudo para a programação lógica, mostrando breve histórico e com maior ênfase no Prolog fazendo um estudo mais aprofundado de seus principais conceitos e características. Adicionalmente, é descrita a implementação de uma ferramenta, em nível de protótipo, que visa experimentar na prática a gerência de uma base de dados em um ambiente de programação lógica.

ABSTRACT

This work is the study on the management of a database for an environment of logical programming. One describes the systems specialists its concepts, components, tools for construction. The main features, the base of knowledge, the mechanism of learning and acquisition of the knowledge are shown, the machine of inference, the black picture, the system of justification and the system of consultation. As construction tools one characterizes the main languages and shells. It includes, still, a generic study on computer programming, imperative procedurais languages and declarative languages. It directs the study for the logical programming soon, showing historical and with bigger emphasis in the Prolog making a study more deepened of its main concepts and features. Additionally, the implementation of a tool, the archetype level is described, that it aims at to try in the practical a management of a database in an environment of logical programming.

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

1.1 ORIGEM DO TRABALHO

No decorrer dos anos tem-se tornado evidente uma busca em acrescentar inteligência aos convencionais sistemas computacionais. Inteligência, como entende-se em computação, sempre está associada a uma larga base de conhecimentos, ou seja, com uma série de informações, as quais o sistema usa como base para tomar qualquer decisão.

Os sistemas que possuem a capacidade de manipularem os conhecimentos armazenados, executando inferências sobre eles e chegando a conclusões inteligentes são os chamados sistemas especialistas. Tem este nome justamente por serem semelhantes a um especialista em determinada área, dominando o conhecimento sobre esta [HAR88].

Uma das ferramentas mais utilizadas para construção de sistemas especialista é a linguagem de programação Prolog. Prolog introduz um paradigma diferente do convencional: o paradigma da programação lógica. Esta baseia-se em fatos e regras, sua base de dados para chegar a determinadas conclusões.

Assim, pelo interesse neste aspecto da programação lógica, a gerência de sua base de dados, tem origem este trabalho.

1.2 ÁREA

A grande área de pesquisa deste trabalho é a programação de computadores. Faz-se uma análise geral sobre os principais aspectos da programação procedural imperativa e programação declarativa, abordando a programação funcional e com um estudo mais profundo da programação lógica e especificamente sobre Prolog. Estuda-se alguns pontos de bancos de dados que venham a embasar o estudo da gerência de bases de dados em programação lógica.

1.3 PROBLEMA

Nas atuais ferramentas de programação lógica deixa-se a gerência da base de dados a cargo de algum gerenciador de bases de dados pelo fato dos gerenciadores específicos serem pouco expressivos. Normalmente cria-se bases de dados estáticas, que não podem ser modificáveis em tempo de execução, ou seja, uma vez os dados armazenados não existe mais a capacidade de acrescer novos dados. Em alguns casos os dados chegam a ficar armazenados no código do programa, totalmente estáticos.

Pretende-se aqui estudar e prototipar uma outra forma de manipulação desses dados. Através de predicados (comandos) que permitam acrescentar novos dados a uma base disponível a um programa em lógica.

1.4 OBJETIVOS DO TRABALHO

O principal objetivos deste trabalho é:

a) Implementar os predicados de manipulação de termos para um ambiente de programação lógica.

E os objetivos secundários, que são base para o objetivo principal, são:

a) Estudar sistemas especialistas, como ferramentas desenvolvidas a partir de ambientes de programação lógica;
b) estudar programação lógica, seus conceitos e principais exemplos;
c) estudar Prolog como ambiente base para a implementação dos predicados de manipulação de termos.

1.5 ESTRUTURA DO TRABALHO

O capítulo 1 faz uma introdução, detalhando origem, área, problema, justificativas, objetivos e estrutura do trabalho.

O capítulo 2 trata de sistemas especialistas. Faz-se um estudo sobre suas características, componentes, ferramentas para construção abordando as shells e as

linguagens de programação úteis para construção destes sistemas. Tem-se aqui o objetivo de localizar o trabalho dentro do seu contexto. Um dos componentes de um sistema especialista é a base de conhecimentos. Sua gerência é objeto de estudo, dentro de uma linguagem de programação, que é o Prolog.

O capítulo 3 aborda programação de computadores. Mostra-se uma visão geral sobre as linguagens procedurais imperativas, abordando seus principais aspectos e características. Faz-se um estudo sobre linguagens declarativas em suas subdivisões funcionais e lógicas. Estuda-se programação lógica, suas características, origens e aplicações.

O capítulo 4 estuda o principal exemplo das linguagens de programação lógica, o Prolog. Apresenta-se os principais conceitos, principais cláusulas e mostra-se alguns exemplos.

O capítulo 5 estuda a alguns aspectos da gerência da base de dados de um ambiente de programação lógica. Fala-se sobre aspectos gerais da gerência de bases de dados e faz-se um estudo sobre suas características em Prolog, que é o objeto de estudo deste trabalho. Mostra-se as estruturas de dados utilizadas pelo Prolog, seus principais predicados usados para manipular a base de dados. Subdivide-se os predicados em quatro grupos: os de manipulação de termos, cláusulas, árvores e tabelas *hash*.

O capítulo 6 mostra o protótipo desenvolvido como resultado do estudo. Especifica-se o protótipo, mostram-se seus objetivos, ambiente de desenvolvimento, metodologia utilizada. Mostram-se os predicados implementados, as estruturas de dados utilizadas, o sistema desenvolvido e os testes efetuados com o mesmo.

O capítulo 7 é a conclusão do trabalho. São apresentadas as considerações finais, abordando as principais vantagens e as limitações, mostrando ainda extensões para futuros trabalhos.

2 SISTEMAS ESPECIALISTAS

Sistemas especialistas são aqueles que solucionam problemas possíveis de resolução apenas por pessoas especialistas em determinada área. Segundo [HAR88] um sistema especialista é um programa inteligente de computador que usa conhecimentos e procedimentos inferenciais, para resolver problemas que são bastante difíceis, de forma a requererem, para sua solução, muita perícia humana. O conhecimento necessário para atuar neste nível, mais os procedimentos inferenciais empregados, pode se considerar como um modelo da perícia parecido com os melhores profissionais do ramo.

Para melhor precisar o que é um sistema especialista, deve-se olhar o que é um especialista. [RAB95] afirma que um especialista é uma pessoa que, devido ao treino e experiência, é capaz de executar coisas que os outros não conseguem: especialistas não são apenas proficientes, mas também exímios e eficientes no que fazem. Especialistas conhecem um grande número de coisas e usam artifícios e cuidados em aplicar o que sabem nos problemas e tarefas: também são bons em explorar a informação irrelevante, tentando atingir a essência, e são bons no reconhecimento de problemas como típicos da área em que têm familiaridade. Atrás do comportamento do especialista está o corpo do conhecimento operativo denominado perito. É razoável supor, então, que os especialistas são aqueles que se deve consultar quando se quer representar a perícia que torna seus comportamentos possíveis.

O conhecimento de um sistema especialista consiste em fatos e heurísticas. Os fatos constituem um corpo de informação que é compartido, publicamente disponível e geralmente aceito pelos especialistas em um campo. As heurísticas são, em sua maioria, privadas, regras pouco discutidas de bom discernimento (regras do raciocínio plausível, regras de boa conjectura), que caracterizam a tomada de decisão a nível de especialista na área. O nível de desempenho de um sistema especialista é função principalmente do tamanho e da qualidade do banco de dados que possui [LUG89].

Durante o processo de raciocínio, o sistema especialista vai verificando qual a importância dos fatos que encontra, comparando-os com as informações já contidas no seu conhecimento acumulado sobre esses fatos e hipóteses. Neste processo, vai formulando

novas hipóteses e verificando novos fatos; e esses novos fatos vão influenciar no processo de raciocínio. Este raciocínio é sempre baseado no conhecimento prévio acumulado. Um sistema especialista com esse processo de raciocínio pode não chegar a uma decisão se os fatos de que dispõe para aplicar o seu conhecimento prévio não forem suficientes. Pode, por este motivo, inclusive chegar a uma conclusão errada; mas este erro é justificado em função dos fatos que encontrou e do seu conhecimento acumulado previamente [KEL97].

2.1 CARACTERÍSTICAS DE UM SISTEMA ESPECIALISTA

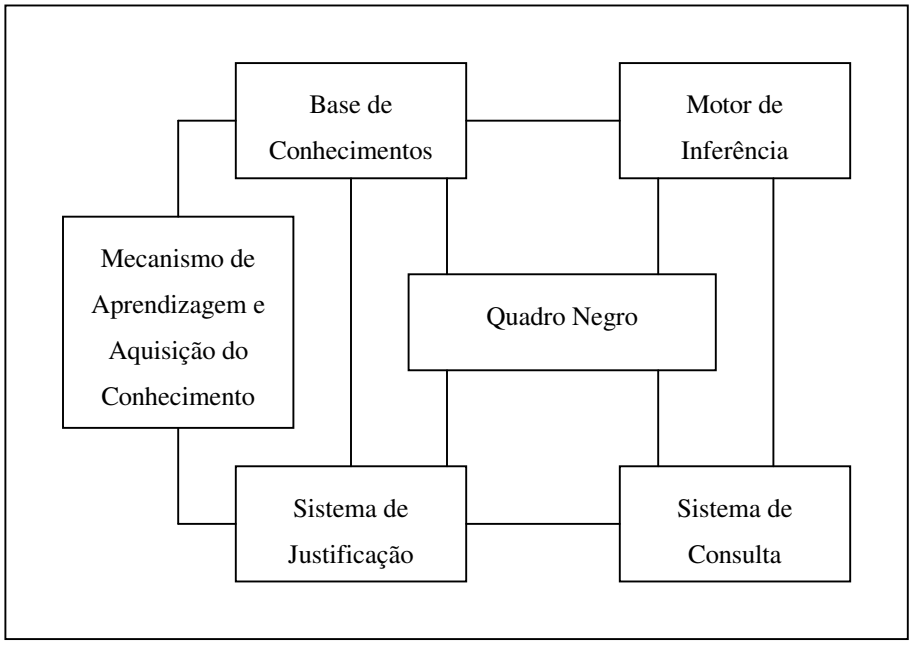
Um sistema tradicional normalmente é estabelecido para resolver um problema bem especificado que pode ser colocado de forma algoritimizada. Um sistema tradicional é projetado para sempre terminar emitindo um resultado final correto. Esse sistema também processa dados, isto é, os algoritmos são projetados para processar volumes de dados de maneira repetitiva [RIB87]. Já um sistema especialista processa conhecimento e não processa dados. O conhecimento é armazenado em uma base de conhecimentos, e os dados são ajustados contra ela. O processamento é feito em cima desse conhecimento e não existe processamento de dados.

Um sistema especialista tem seu funcionamento básico apoiado em heurística, por isso ele é a solução para problemas para os quais, na forma tradicional, não é possível fazer-se um algoritmo ou, se feito, irá obrigar a um processamento muito demorado para a obtenção da solução. Deve-se buscar um processo heurístico para a solução. Um processo heurístico normalmente conduz a soluções de maneira rápida, porém eventualmente pode não conduzir a solução alguma. Assim um sistema especialista pode chegar ou não à solução, e pode ainda chegar a uma solução distorcida, isto é, não existe obrigatoriedade de se chegar a uma solução correta. O sistema pode errar, porém o seu erro ou a não resposta ocorrem dentro de determinadas circunstâncias que são justificadas pelo próprio sistema [RIC93].

2.2 COMPONENTES DE UM SISTEMA ESPECIALISTA

A composição de um sistema especialista depende de fatores como a generalidade pretendida, os objetivos do mesmo, a representação interna do conhecimento e as

ferramentas usadas na implementação [HEI95]. O modelo geral da arquitetura é mostrado na figura1. Na terminologia e arquitetura empregada há diferenças entre os diversos autores mas de uma maneira geral o sistema é constituído de seis elementos básicos que são: base de conhecimentos, mecanismo de aprendizagem e aquisição de conhecimento, máquina ou motor de inferência, sistema de justificação, sistema de consulta e quadro negro.



Fonte: [HEI95], p.13

FIGURA 1 - COMPONENTES DE UM SISTEMA ESPECIALISTA

A seguir são apresentados cada um destes componentes com uma breve descrição dos mesmo.

2.2.1 A BASE DE CONHECIMENTOS

Base de conhecimento é um elemento permanente, mas específico de um sistema especialista. É onde estão armazenadas as informações de um sistema especialista, ou seja os fatos e as regras. A base de conhecimento determina as características de funcionamento do sistema, que terá o conhecimento do que for colocado na sua base de conhecimento, isto é, se ela for projetada para receber informações de uma determinada ciência, o sistema será especialista nesta ciência. Para a confecção das regras aqui armazenadas são usadas as heurísticas que provém do especialista. A base de conhecimentos possui grande

importância. Já em 1977, Feigenbaum apud [RAB95] citou “A potência de um sistema especialista deriva do conhecimento que ele possui e não do formalismo e esquemas específicos que ele emprega”.

2.2.2 MECANISMO DE APRENDIZAGEM E AQUISIÇÃO DO CONHECIMENTO

Os sistemas especialistas devem possuir meios que permitam ampliar, alterar ou atualizar o seu conhecimento. Geralmente existe um módulo no sistema que se utiliza de recursos como editores de textos e classificadores, que permitem adequar ou formatar o conhecimento para ser introduzido na base de conhecimentos [HEI95]. Normalmente o mecanismo de aprendizagem é muito rudimentar. Na maioria dos sistemas, porém, existe a possibilidade de tornar este recurso mais potente, fazendo com que adquira uma capacidade maior, depurando a base de conhecimentos, reordenando prioridades, estabelecendo mecanismos de controle, executando outras ações que melhorem o desempenho do sistema.

2.2.3 A MÁQUINA DE INFERÊNCIA

Máquina de inferência ou motor de inferência, ou ainda mecanismo de inferência é um elemento permanente, que pode inclusive ser reutilizado por vários sistemas especialistas. É a parte responsável pela busca das regras da base de conhecimento para serem avaliadas, direcionando o processo de inferência. A máquina de inferência depende de como se está representando o conhecimento, pois esta deve estar preparada para a interpretação.

Esta máquina não é normalmente um único módulo de programa. É, em geral, entendido como compreendendo o interpretador de regras e o escalonador das regras, quando o sistema especialista envolve regras de produção. Porém em outras formas de representação do conhecimento, pode estar intimamente ligado à estrutura de representação.

Assume-se que o programa está sempre tentando identificar algum objetivo. Ele recupera todas as regras necessárias para a obtenção de um desses objetivos. Para atender a um objetivo, algumas vezes é necessário determinarem-se submetas e, para atender a estas,

novas regras serão buscadas e avaliadas, determinando novas submetas, e assim por diante. Isto irá gerar um processo de avaliação que irá ser dependente dos fatos que encontra.

A busca de regras é feita de maneira automática para que uma meta ou submeta seja atingida. Entretanto, existem casos em que a reposta pode ser obtida de maneira mais imediata e, nesses casos, são estabelecidas estratégias de avaliação imediata, evitando todo o processo natural de busca e avaliação de regras. Algumas dessas estratégias consistem em recuperar todas as regras necessárias para se chegar a uma submeta. Nessas regras, procuram-se determinar todas as cláusulas com certeza de 100% e avaliar imediatamente estas regras, com o que é corretamente conhecido. Essa busca pode ser feita em uma seqüência de regras, e o resultado é chamado de caminho único [RIB87].

Outra estratégia usada consiste no mecanismo de inferência proceder antes à busca das novas regras que foram causadas pela necessidade de se atender a uma meta, e avaliar essas regras a serem pesquisadas. Como os atributos são encontrados em diversas regras, o valor de uma cláusula já pode ter sido estabelecido. Esse valor, sozinho, permite determinar antecipadamente que a premissa da regra é falsa, e que não há razões para novas buscas.

Para a implementação da máquina de inferência, em alguns casos pode-se usar softwares prontos disponíveis, mas em outros é preciso elaborá-los. Nos dois casos existem vantagens e desvantagens. Quando da adoção de um software, adota-se o todo. Isso muitas vezes implica em algumas partes deficientes. A opção pela elaboração do software implicará em custo e tempo adicionais, podendo satisfazer mais em detalhes e permitir melhor adaptação ao problema [RAB95]. Por outro lado, encontram-se softwares no mercado que permitem ao usuário dispor dos recursos fundamentais de inferência para algumas formas de representação do conhecimento e, além disso, fazer certas adaptações.

2.2.4 O QUADRO-NEGRO

O quadro-negro é a área de trabalho do sistema especialista, muitas vezes chamado rascunho. É uma área de memória onde o sistema vai gravando e apagando informações, fatos, estruturas de suporte ao funcionamento do sistema. Para se chegar a uma solução, há necessidade de se avaliarem regras que são recuperadas da base de conhecimentos para

uma área de trabalho da memória. Neste local essas regras são reordenadas periodicamente em uma nova ordem para serem avaliadas. Durante essa avaliação deve-se verificar fatos e hipóteses e também há necessidade de uma área onde são guardados os valores das variáveis para se trabalhar tais fatos e hipóteses. As conclusões dessas regras irão gerar novos fatos e novas hipóteses que precisam ser guardados temporariamente durante o processo de inferência, em algum local. Essa área de memória usada para a execução das operações descritas acima chama-se quadro negro [RIB87].

Alguns autores, entre eles [RAB95] dividem o quadro negro em três partes. O plano de atuação que contém elementos que descrevem a estratégia geral a ser seguida, ligada diretamente ao interpretador de regras e ao justificador . A agenda, que registra ações em espera de execução, está diretamente associada ao escalonador de regras. E ainda a parte da solução, que contém elementos que comporão as soluções potenciais, e está ligado ao módulo verificador de consistência.

2.2.5 SISTEMA DE JUSTIFICAÇÃO

Forsyth apud [RAB95] chama este componente de "A janela humana". O sistema de justificação tem a função de esclarecer o usuário a respeito de uma conclusão apresentada ou ainda de explicar uma pergunta que está sendo feita. A justificação é um requisito importante nos sistemas especialistas. Em muitos dos domínios nos quais os sistemas operam, as pessoas não aceitam resultados se estes não estiverem devidamente justificados. Na medicina, por exemplo, onde o médico tem a responsabilidade final por um diagnóstico, certamente um sistema teria que mostrar os motivos que o levaram a alcançar uma determinada conclusão [HEI95].

2.2.6 SISTEMA DE CONSULTA

Os usuários de sistemas especialistas interagem de forma intensa com o sistema pois além de receberem dele as conclusões alcançadas também participam ativamente do processo de inferência e da construção da base de conhecimentos. Estes sistemas devem, portanto, oferecer bons recursos de comunicação que permitam, até ao usuário sem conhecimentos computacionais, tirar proveito dos mesmos [HEI95].

Segundo [RAB95] o sistema de consulta deve ter módulos explícitos e implícitos voltados ao usuário, de preferência abstendo-se de termos computacionais, podendo adotar uma linguagem parecida com a linguagem nativa ou estabelecer módulos orientados exclusivamente ao problema.

A maioria dos sistemas existentes usa técnicas simples de interação com o usuário, quase sempre utilizando perguntas já pré formatadas e resposta tipo múltipla escolha.

2.3 FERRAMENTAS PARA CONSTRUÇÃO DE UM SISTEMA ESPECIALISTA

Existem uma série de ferramentas próprias para o uso de técnicas de inteligência artificial. Ferramentas para ajudar a fazer, depurar e manter uma base de conhecimento, ou extrair automaticamente o conhecimento de um especialista e os mecanismos para acessar este conhecimento. Segundo [JAC86] as ferramentas para construção de sistemas especialistas podem ser divididas em 3 classes: *shells* para sistemas especialistas, linguagens de programação de alto nível e ambientes de programação mista.

2.3.1 LINGUAGENS

As ferramentas mais tradicionais para o desenvolvimento de sistemas especialistas são as linguagens. Elas possuem recursos internos que facilitam que se façam mecanismos para a inferência, ou que permitam processar listas. As principais linguagens são:

a) Lisp – usada principalmente pela comunidade científica americana de pesquisa em inteligência artificial, há bastante tempo. Por esse motivo é uma linguagem bastante experimentada, e com bastante confiabilidade e eficiência de desempenho. Muitas das ferramentas mais específicas de inteligência artificial são desenvolvidas em LISP. Esta linguagem foi utilizada por John McCarty em 1960, para possibilitar a implementação de seus principais conceitos. Implementa alguns dos principais conceitos defendidos por John Backus, originador do FORTRAN, para as características de uma linguagem funcional, onde deve-se ter um conjunto de funções primitivas poderosas que permitam se escrever novas

funções mais poderosas que as primeiras, que também passarão a pertencer a um conjunto de funções iniciais que irão gerar novas funções. Logo nesta filosofia de funcionamento, não existe dependência de máquina, apenas dependência de conceitos matemáticos. Por isso o LISP é considerado uma linguagem funcional e tem como principal característica manusear listas e símbolos.

b) PROLOG – palavra que expressa uma abreviação de "PROgramming in LOGic", é uma linguagem inicialmente concebida para processamento de linguagem natural, e da mesma forma que LISP, orientada para o processamento simbólico. Esta linguagem é discutida com maiores detalhes mais adiante em capítulo próprio.

2.3.2 SHELLS

Nos primeiros sistemas especialistas desenvolvidos construía-se todos os módulos do sistema antes que fosse criada a base de conhecimentos sobre a qual o sistema iria agir. Posteriormente, observou-se que muitos deles tinham em comum alguns destes módulos, pois eles eram construídos como um conjunto de representações declarativas, especialmente em forma de regras, que eram combinadas com um interpretador dessas representações [HEI95]. Assim, constatou-se que seria possível criar sistemas genéricos que poderiam ser usados para criar novos sistemas especialistas pela simples adição ou criação da base de conhecimentos correspondente ao domínio do problema. Esses sistemas de uso genérico receberam o nome de *shells*.

Shells é o nome dado a uma família de ferramentas, não linguagens de programação, que objetivam apoiar e simplificar o processo de construção de sistemas especialistas. São softwares que contém alguns dos principais elementos de um sistema especialista, tais como, o motor de inferência, o justificador e outros. Estas ferramentas também pré-definem a estruturação do conhecimento a ser utilizada pelo sistema. Ao projetista do sistema especialista usuário de uma *shell* cabe apenas a tarefa de construir uma base de conhecimentos.

3 LINGUAGENS DE PROGRAMAÇÃO

Uma linguagem de programação é uma notação sistemática através da qual se pode descrever processos computacionais. Entende-se processo computacional como uma série de passos que uma máquina deve percorrer para resolver uma tarefa. Para descrever a solução de um problema a um computador, precisa-se saber um conjunto de comandos que a máquina pode entender e executar [HOR84].

3.1 LINGUAGENS PROCEDURAIS IMPERATIVAS

As linguagens procedurais imperativas possuem uma linha clara que as identifica e as torna semelhantes: a arquitetura Von Neumann. A maioria das linguagens atuais são abstrações construídas em cima dessa arquitetura. Para prover essas abstrações, uma linguagem deve atingir um compromisso entre utilidade de mecanismos e eficiência de execução, onde a eficiência de execução é medida pelo desempenho em um computador Von Neumann. Assim a arquitetura Von Neumann tem formado a base para o projeto de linguagens de programação [GHE91].

A arquitetura Von Neumann consiste em uma arquitetura que possui um processador central acoplado a uma área de memória, manipulada por variáveis as quais estão associadas a um ponto desta memória. Juntas, as diversas variáveis descrevem o estado da computação em determinado momento [HOR84].

3.1.1 CARACTERÍSTICAS E PREMISSAS BÁSICAS

As linguagens procedurais imperativas são delineadas por uma característica básica e de muito valor: o papel dominante desempenhado pelos comandos ou instruções imperativas. A unidade de trabalho em um programa escrito nestas linguagens é o comando. Os efeitos de comandos individuais são combinados para a obtenção dos resultados desejados em um programa [GHE91].

3.1.1.1 VARIÁVEIS

Um dos principais componentes da arquitetura é a memória, que é subdividida em uma série de 'compartimentos' que se pode chamar de células. Nestas células os dados ficam armazenados. Para melhor controle e localização da informação, estas células devem ter nomes. Os dados são armazenados em células e pode-se ter acesso a eles através do nome dessas células. Este conceito é representado pelas variáveis, segundo [GHE91] um dos conceitos mais importantes em linguagens de programação.

Uma variável em uma linguagem de programação é essencialmente uma célula de memória, onde se armazenam valores, com um nome. Assim apesar da finalidade do programa ser produzir valores, fala-se não somente de valores, mas também da células onde os valores de interesse residem. Os problemas de efeitos colaterais e sinonímia aparecem por causa da existência de variáveis [GHE91].

Quando se fala em memória e em células, a função de atribuição de valores a estas células se torna uma conseqüência. Justamente porque cada valor computado deve ser atribuído a uma célula. As noções de baixo nível de célula de memória e atribuição permeiam todas as linguagens de programação, forçando ao programador um estilo de pensamento que é moldado pelas arquiteturas Von Neumann [GHE91].

3.1.1.2 REPETIÇÃO

Um programa em uma linguagem imperativa geralmente cumpre sua tarefa executando repetidamente uma seqüência de passos elementares. Isto é uma conseqüência da arquitetura Von Neumann, na qual as instruções são armazenadas na memória. A única maneira de fazer algo complicado é repetindo uma seqüência de instruções [GHE91].

3.1.2 PRINCIPAIS LINGUAGENS

A maioria das linguagens atuais são linguagens procedurais imperativas. Segundo [GHE91] as principais linguagens são FORTRAN, COBOL, PASCAL, C e ADA. Cada uma possui características particulares:

FORTRAN: *FORmula TRANslator* é a linguagem para aplicações científicas e numéricas. O principal objetivo no projeto da linguagem foi a eficiência de execução de programas.

COBOL: *Common Business Oriented Language* é uma linguagem para aplicações comerciais. Programas em COBOL em geral especificam computações simples em grande quantidade de dados. Já foi uma das linguagens mais usadas, mas ela não influenciou significativamente o projeto de linguagens mais modernas.

PASCAL: foi originalmente projetada para o ensino da programação disciplinada. A linguagem foi um enorme sucesso, e existem hoje implementações na maioria das máquinas, incluindo microprocessadores. Pascal influenciou praticamente todas as linguagens mais recentes.

C: foi desenvolvida para a representação de sistemas de programação para o PCP-11. C foi generalizada e implementada em muitos computadores, grandes e pequenos. C é apoiada por um conjunto completo de ferramentas no sistema operacional UNIX. Um subconjunto portátil foi definido.

ADA: foi projetada para apoiar aplicações numéricas, a programação de sistemas e aplicações que envolvem considerações de tempo real e concorrência.

3.2 LINGUAGENS DECLARATIVAS

As linguagens declarativas exigem que regras e fatos a respeito de determinados símbolos sejam declarados, para depois perguntar se uma determinada meta segue, de uma forma lógica, estas regras e fatos. Ao contrário das linguagens imperativas, nas declarativas não é preciso usar uma linguagem para informar ao compilador como procurar uma solução, onde olhar, quando parar. As linguagens declarativas dividem-se em funcionais e lógicas.

3.2.1 PROGRAMAÇÃO FUNCIONAL

Segundo [BIR88] programar em uma linguagem funcional consiste em construir definições e usar o computador para avaliar expressões. O papel principal do programador consiste em construir uma função para resolver um problema dado. Essa função que pode envolver um numero subsidiário de funções, é expressada em uma notação que obedece princípios matemáticos. O principal papel do computador é agir como um avaliador ou calculador: é seu trabalho avaliar as expressões e imprimir os resultados. Nesse aspecto o computador age como uma simples calculadora de bolso. O que distingue uma calculadora funcional das outras é a habilidade do programador de fazer definições para aumentar o poder de cálculo. Expressões que contém ocorrências de nomes de funções definidas pelo programador são avaliadas usando as definições dadas como regras de simplificação (ou redução) para converter expressões em um resultado.

A característica que domina na programação funcional segundo [BIR88], é que o significado de uma expressão é seu valor, e o papel do computador é simplesmente obtê-lo. Outra característica é que uma função em uma linguagem funcional pode ser construída, manipulada e resolvida, como qualquer outro tipo de expressão matemática, usando leis algébricas. O resultado, como se espera justificar, é conciso, simples, flexível e poderoso.

Um dos principais exemplos de linguagens funcionais é o Lisp, cuja estrutura é totalmente voltada a funções.

3.2.2 PROGRAMAÇÃO EM LÓGICA

Um programa em lógica é constituído por um conjunto finito de sentenças lógicas, que expressam o conhecimento relevante para o problema que se pretende solucionar. A formulação de tal conhecimento emprega dois conceitos básicos: a existência de objetos discretos, que serão denominados indivíduos, e a existência de relações entre eles. Os indivíduos, considerados no contexto de um problema particular, constituem o domínio do problema. Por exemplo, se o problema é solucionar uma equação algébrica, então o domínio deve incluir pelo menos os números reais. Ao contrário das linguagens

tradicionais, um programa em lógica não é a descrição de um procedimento para obter as soluções de um problema.

Para que possam ser representados por meio de um sistema simbólico tal como a lógica, tanto os indivíduos quanto as relações devem receber nomes. A atribuição de nomes é, entretanto, apenas uma tarefa preliminar na criação de modelos simbólicos para a representação de conhecimento. A tarefa principal é a construção de sentenças expressando as diversas propriedades lógicas das relações nomeadas. O raciocínio sobre algum problema baseado no domínio representado é obtido através da manipulação de tais sentenças por meio de inferência lógica. Em um ambiente típico de programação em lógica, o programador estabelece sentenças lógicas que, reunidas, formam um programa. O computador então executa as inferências necessárias para a solução dos problemas propostos.

Como principal exemplo das linguagens de programação lógica tem-se o Prolog.

3.2.2.1 CARACTERÍSTICAS BÁSICAS

De acordo com [PAL97] as principais características da programação lógica são as seguintes:

Especificações são Programas: A linguagem de especificação é entendida pela máquina e é, por si só, uma linguagem de programação. Naturalmente, o refinamento de especificações é mais efetivo do que o refinamento de programas. Um número ilimitado de cláusulas diferentes pode ser usado e predicados (procedimentos) com qualquer número de argumentos são possíveis. Não há distinção entre o programa e os dados. As cláusulas podem ser usadas com grande vantagem sobre as construções convencionais para a representação de tipos abstratos de dados. A adequação da lógica para a representação simultânea de programas e suas especificações a torna um instrumento especialmente útil para o desenvolvimento de ambientes e protótipos.

Capacidade Dedutiva: O conceito de computação confunde-se com o de (passo de) inferência. A execução de um programa é a prova do teorema representado pela

consulta formulada, com base nos axiomas representados pelas cláusulas (fatos e regras) do programa.

Não-determinismo: Os procedimentos podem apresentar múltiplas respostas, da mesma forma que podem solucionar múltiplas e aleatoriamente variáveis condições de entrada. Através de um mecanismo especial, denominado "backtracking", uma seqüência de resultados alternativos pode ser obtida.

Reversibilidade das Relações: (Ou "computação bidirecional"). Os argumentos de um procedimento podem alternativamente, em diferentes chamadas representar ora parâmetros de entrada, ora de saída. Os procedimentos podem assim ser projetados para atender a múltiplos propósitos. A execução pode ocorrer em qualquer sentido, dependendo do contexto. Por exemplo, o mesmo procedimento para inserir um elemento no topo de uma pilha qualquer pode ser usado, em sentido contrário, para remover o elemento que se encontrar no topo desta pilha.

Tríplice Interpretação dos Programas em Lógica: Um programa em lógica pode ser semanticamente interpretado de três modos distintos: (1) por meio da semântica declarativa, inerente à lógica, (2) por meio da semântica procedimental, onde as cláusulas dos programas são vistas como entrada para um método de prova e, (3) por meio da semântica operacional, onde as cláusulas são vistas como comandos para um procedimento particular de prova por refutação. Essas três interpretações são intercambiáveis segundo a particular abordagem que se mostrar mais vantajosa ao problema que se tenta solucionar.

Recursão: A recursão, na programação lógica, é a forma natural de ver e representar dados e programas. Entretanto, na sintaxe das linguagens não há laços do tipo "for" ou "while" (apesar de poderem ser facilmente programados), simplesmente porque eles são absolutamente desnecessários. Também são dispensados comandos de atribuição e, evidentemente, o "goto". Uma estrutura de dados contendo variáveis livres pode ser retornada como a saída de um procedimento. Essas variáveis livres podem ser posteriormente instanciadas por outros procedimentos produzindo o efeito de atribuições implícitas a estruturas de dados. Onde for necessário, variáveis livres são automaticamente agrupadas por meio de referências transparentes ao programador. Assim, as variáveis

lógicas um potencial de representação significativamente maior do que oferecido por operações de atribuição e referência nas linguagens convencionais.

3.2.2.2 ORIGEM

Segundo [BEN96] o uso da lógica na representação dos processos de raciocínio remonta aos estudos de Boole (1815-1864) e de De Morgan (1806-1871), sobre o que veio a ser mais tarde chamado "Álgebra de Boole". Como o próprio nome indica, esses trabalhos estavam mais próximos de outras teorias matemáticas do que propriamente da lógica. Deve-se ao matemático alemão Göttlob Frege no seu "Begriffsschrift" (1879) a primeira versão do que hoje denomina-se cálculo de predicados, proposto por ele como uma ferramenta para formalizar princípios lógicos. Esse sistema oferecia uma notação rica e consistente que Frege pretendia adequada para a representação de todos os conceitos matemáticos e para a formalização exata do raciocínio dedutivo sobre tais conceitos, o que, afinal, acabou acontecendo.

Segundo [BRO92] no final do século passado a matemática havia atingido um estágio de desenvolvimento mais do que propício à exploração do novo instrumento proposto por Frege. Os matemáticos estavam abertos a novas áreas de pesquisa que demandavam profundo entendimento lógico assim como procedimentos sistemáticos de prova de teoremas mais poderosos e eficientes do que os até então empregados. O relacionamento entre lógica e matemática foi profundamente investigado por Alfred North Whitehead e Bertrand Russel, que em "Principia Mathematica" (1910) demonstraram ser a lógica um instrumento adequado para a representação formal de grande parte da matemática.

De acordo com [BEN96] no início da Segunda Guerra Mundial, em 1939, toda a fundamentação teórica básica da lógica computacional estava pronta. Faltava apenas um meio prático para realizar o imenso volume de computações necessárias aos procedimentos de prova. Apenas exemplos muito simples podiam ser resolvidos manualmente. O estado de guerra deslocou a maior parte dos recursos destinados à pesquisa teórica, nos EUA, Europa e Japão para as técnicas de assassinato em massa. Foi somente a partir da metade dos anos 50 que o desenvolvimento da então novíssima tecnologia dos computadores

conseguiu oferecer aos pesquisadores o potencial computacional necessário para a realização de experiências mais significativas com o cálculo de predicados.

Em 1958, uma forma simplificada do cálculo de predicados denominada forma clausal começou a despertar o interesse dos estudiosos do assunto [STE86]. Tal forma empregava um tipo particular muito simples de sentença lógica denominada cláusula. Uma cláusula é uma (possivelmente vazia) disjunção de literais [STE86]. Também por essa época, Dag Prawitz (1960) propôs um novo tipo de operação sobre os objetos do cálculo de predicados, que mais tarde veio a ser conhecida por unificação. A unificação se revelou fundamental para o desenvolvimento de sistemas simbólicos e de programação em lógica [BEN96].

A programação em lógica em sistemas computacionais somente se tornou realmente possível a partir da pesquisa sobre prova automática de teoremas, particularmente no desenvolvimento do Princípio da Resolução por J. A. Robinson (1965). Um dos primeiros trabalhos relacionando o Princípio da Resolução com a programação de computadores deve-se a Cordell C. Green (1969) que mostrou como o mecanismo para a extração de respostas em sistemas de resolução poderia ser empregado para sintetizar programas convencionais [PAL97].

A expressão "programação em lógica" (*logic programming*, originalmente em inglês) é devido a Robert Kowalski (1974) e designa o uso da lógica como linguagem de programação de computadores. Kowalski identificou, em um particular procedimento de prova de teoremas, um procedimento computacional, permitindo uma interpretação procedimental da lógica e estabelecendo as condições que nos permitem entendê-la como uma linguagem de programação de uso geral. Este foi um avanço essencial, necessário para adaptar os conceitos relacionados com a prova de teoremas às técnicas computacionais já dominadas pelos programadores. Aperfeiçoamentos realizados nas técnicas de implementação também foram de grande importância para o emprego da lógica como linguagem de programação. Segundo [SET90] o primeiro interpretador experimental foi desenvolvido por um grupo de pesquisadores liderados por Alain Colmerauer na Universidade de Aix-Marseille (1972) com o nome de Prolog, um acrônimo para "Programmation en Logique". Seguindo-se a este primeiro passo, implementações mais

praticas foram desenvolvidas por Battani e Meloni (1973), Bruynooghe (1976) e, principalmente, David H. D. Warren, Luís Moniz Pereira e outros pesquisadores da Universidade de Edimburgo (U.K.) que, em 1977, formalmente definiram o sistema hoje denominado "Prolog de Edimburgo", usado como referência para a maioria das atuais implementações da linguagem Prolog. Deve-se também a Warren a especificação da WAM (Warren Abstract Machine), um modelo formal empregado até hoje na pesquisa de arquiteturas computacionais orientadas à programação em lógica [CLO94].

3.2.2.3 PRINCIPAIS APLICAÇÕES

O campo de aplicações da programação lógica é vasto, porém, as principais aplicações podem ser identificadas em:

Sistemas Baseados em Conhecimento: Ou *knowledge-based systems,* são sistemas que aplicam mecanismos automatizados de raciocínio para a representação e inferência de conhecimento. Tais sistemas costumam ser identificados como simplesmente "de inteligência artificial aplicada" e representam uma abrangente classe de aplicações da qual todas as demais seriam aproximadamente subclasses [MAI88].

Sistemas de Bases de Dados: Uma particularmente bem definida aplicação dos sistemas baseados em conhecimento são bases de dados. Sistemas de bases de dados convencionais tradicionalmente manipulam dados como coleções de relações armazenadas de modo extensional sob a forma de tabelas. O modelo relacional serviu de base à implementação de diversos sistemas fundamentados na álgebra relacional, que oferece operadores tais como junção e projeção. O processador de consultas de uma base de dados convencional deriva, a partir de uma consulta fornecida como entrada, alguma conjunção específica de tais operações algébricas que um programa gerenciador então aplica às tabelas visando a recuperação de conjuntos de dados (n-tuplas) apropriados, se existirem. A recuperação de dados é intrínseca ao mecanismo de inferência dos interpretadores lógicos [MAI88].

Processamento da Linguagem Natural: A implementação de sistemas de processamento de linguagem natural em computadores requer não somente a formalização

sintática, como também a formalização semântica, isto é, o correto significado das palavras, sentenças, frases, expressões, etc. que povoam a comunicação natural humana. Segundo [BRO92] o uso da lógica das cláusulas de Horn são adequadas à representação de qualquer gramática livre-de-contexto e permitem que questões sobre a estrutura de sentenças em linguagem natural sejam formuladas como objetivos ao sistema, e que diferentes procedimentos de prova aplicados a representações lógicas da linguagem natural correspondam a diferentes estratégias de análise.

Educação: A proposta do uso da linguagem natural na educação foi testada em 1978 quando Kowalski introduziu a programação em lógica na Park House Middle School em Wimbledon, na Inglaterra, usando acesso *on-line* aos computadores do Imperial College. Os resultados obtidos desde então tem mostrado que a programação em lógica não somente é assimilada mais facilmente do que as linguagens convencionais, como também pode ser introduzida até mesmo a crianças na faixa dos 10 a 12 anos, as quais ainda se beneficiam do desenvolvimento do pensamento lógico-formal que o uso de programação lógica induz [PAL97].

Arquiteturas Não-Convencionais: Nesta área o uso da programação em lógica vem sendo aplicado na especificação e implementação de máquinas abstratas de processamento paralelo. O paralelismo pode ser modelado pela programação em lógica em variados graus de atividade se implementado em conjunto com o mecanismo de unificação [MAI88].

4 A LINGUAGEM PROLOG

A principal utilização da linguagem Prolog reside no domínio da programação simbólica, não-numérica, sendo especialmente adequada à solução de problemas, envolvendo objetos e relações entre objetos. O advento da linguagem Prolog reforçou a tese de que a lógica é um formalismo conveniente para representar e processar conhecimento. Seu uso evita que o programador descreva os procedimentos necessários para a solução de um problema, permitindo que ele expresse declarativamente apenas a sua estrutura lógica, através de fatos, regras e consultas.

4.1 PRINCIPAIS CARACTERÍSTICAS

De acordo com [PAL97] as principais características do Prolog são:

a) orientada para processamento simbólico;
b) representa uma implementação da lógica como linguagem de programação;
c) apresenta uma semântica declarativa inerente à lógica.
d) permite a obtenção de respostas alternativas;
e) suporta estrutura de dados que permite simular registros ou listas;
f) permite recuperação dedutiva de informação;
g) representa programas e dados através do mesmo formalismo (cláusulas);
h) incorpora facilidades computacionais extra e metalógicas.

4.2 CONCEITOS

Os principais conceitos Prolog são os fatos e as regras, além das questões, variáveis, conjunções, construções recursivas e o mecanismo de *backtraking*.

4.2.1 FATOS

O tipo mais simples de uma declaração é chamado fato. Fatos declaram que existe algum relacionamento entre os objetos. Um exemplo de fato pode ser:

```
pai(adao,caim).
```

Este fato mostra que Adão é pai de Caim, pode-se dizer que existe uma relação de pai entre Adão e Caim. Um fato sempre consiste em objetos, no caso Adão e Caim, e uma relação entre eles. No exemplo acima a relação é pai.

Quando se escreve fatos, deve-se estabelecer uma regra quanto a ordem dos objetos de um determinado fato. O Prolog não exige, mas precisa-se seguir uma ordem para a consistência dos fatos. Por exemplo, não é a mesma coisa dizer que Caim é Pai de Adão. Nos exemplos mostrados usa-se escrever objetos e seus relacionamentos em letras minúsculas.

Alguns outros exemplos de fatos:

```
solido(ferro).          O ferro é sólido.
mulher(maria).          Maria é mulher.
homem(joao).            João é homem.
pai(joao,maria).        João é pai de Maria.
dar(joao,livro,maria).  João dá um livro a Maria.
```

Os objetos usados em um fato, são chamados de argumentos. O nome da relação, que vem exatamente antes do parênteses é chamado de predicado. Assim pode-se dizer que solido é um predicado com um argumento. Já pai é um predicado com dois argumentos e dar com três argumentos. Os predicados podem ter um número qualquer de argumentos, dependendo de qual seu objetivo. O número de argumentos de um predicado é chamado de aridade.

Pode-se declarar fatos que não sejam realidade no mundo real. Pode-se dizer que:

```
rei(pedro, brasil).     Pedro é rei do Brasil.
```

No entanto sabe-se que isto não é realidade. Mas o Prolog não conhece a realidade e ela também não importa. Fatos em Prolog simplesmente permitem expressar relacionamentos entre objetos.

4.2.2 QUESTÕES

A partir do momento que se tem fatos, pode-se formular algumas questões a respeito destes fatos. Uma questão é usada para obter uma resposta de um programa Prolog. Uma questão é feita para verificar se determinada relação existe entre os objetos. Por exemplo:

```
?- capital(brasilia,brasil)
```

Na questão, interpretando-se que brasilia seja uma cidade e brasil um país, tem-se a pergunta Brasília é capital do Brasil? ou É um fato que Brasília é capital do Brasil ?

Quando o Prolog é questionado, uma busca pela base de dados ocorre. O Prolog busca por fatos que satisfaçam a questão. Se um fato é encontrado, Prolog responde *yes*, caso contrário Prolog responde *no*. Considerando a seguinte base de dados:

```
comprar(mario,bola).          Mario comprou uma bola.
comprar(pedro,computador).    Pedro comprou um computador.
comprar(joao,camisa).         Joao comprou uma camisa.
comprar(maria,casa).          Maria comprou uma casa.
comprar(jorge,carro).         Jorge comprou um carro.
comprar(claudio,carro).       Cláudio comprou um carro.
```

Pode-se formular as seguintes perguntas ao Prolog:

```
?- comprar(pedro,bola).
no
?- comprar(maria,carro).
no
?- comprar(jorge,camisa).
no
?- comprar(joao,camisa).
yes
```

Na base de dados descrita acima observa-se que Jorge e Cláudio compraram um mesmo objeto. Pode-se fazer esta observação pois o mesmo nome, carro, aparece nos dois fatos. Através de questões, como aqui apresentadas, não se pode chegar a esta conclusão. Por exemplo, quer-se saber O que Pedro comprou ? ou Quem comprou um carro ? para resposta destas perguntas, precisa-se usar variáveis.

4.2.3 VARIÁVEIS

Para perguntar a um programa Prolog o que Pedro comprou, fica cansativo perguntar Pedro comprou um Carro ? ou Pedro comprou uma Bola ? ou ainda Pedro comprou um Computador ? até finalmente obter a resposta certa. A pergunta mais prática a fazer seria O que Pedro comprou ? em Prolog formula-se esta questão da seguinte forma Pedro comprou X ? Quando se faz esta pergunta, não se sabe com que valor X deveria ser instanciado, quer-se que o Prolog responda com as alternativas corretas. Em Prolog pode-se usar não somente nomes fixos, mas também variáveis que serão instanciadas pelo próprio Prolog.

O Prolog destingue variáveis de nomes pois variáveis iniciam com uma letra maiúscula. Qualquer nome pode ser usado como variável, desde que sua primeira letra seja maiúscula. Quando faz-se uma questão ao Prolog contendo uma variável, ocorre uma busca através de todos os fatos até encontrar um objeto que seja possível de instanciar a variável. Assim quando se pergunta Mario comprou X ?, Prolog faz uma busca por toda a base de dados para encontrar valores possíveis de serem instanciados em X. Exemplos:

```
?- comprar(mario,X).
X=bola
?- comprar(X,carro).
X=jorge;
```

Quando o Prolog recebe uma questão, inicialmente, a variável não é instanciada com nenhum valor. Prolog faz uma busca de um fato, que possua um argumento na mesma posição da variável. No exemplo acima, Prolog busca qualquer fato onde o predicado seja comprar e o segundo argumento seja carro. O primeiro argumento pode ser qualquer coisa, pois ele é uma variável não instanciada. Quando encontrado, o Prolog instancia a variável com o primeiro argumento e mostra-o na tela. Se o usuário continuar, Prolog marca este fato, para controle de Backtraking, discutido mais adiante. E continua a buscar novos fatos que satisfaçam a questão.

4.2.4 CONJUNÇÕES

A conjunção permite a composição de fatos, formando o chamado ***e*** lógico:

```
comprar(mario,bola), comprar(maria,casa).
```

A vírgula (**,**) entre as questões determinam a validade mútua de ambas (ou do conjunto envolvido). O objetivo da conjunção é a realização de uma busca no banco de dados para a verificação da validade da questão ou busca dos objetos relacionados com os argumentos propostos. O mecanismo de busca é uma estratégia de controle do tipo busca em profundidade com retrocesso.

```
?-comprar(jorge,Oque),comprar(claudio,Oque).
Oque = carro
```

A consulta teve como objetivo obter o objeto carro que faltava para validar a conjunção entre os dois objetivos. Objetivo, neste contexto, é cada um dos componentes da

conjunção. Há duas formas de validação de conjunções. A primeira ocorre caso todos os objetivos sejam fatos como, por exemplo:

```
?- comprar(mario,bola),comprar(joao,camisa).
no
```

Todos os objetivos devem ser validados. A segunda forma ocorre caso hajam variáveis e existam correspondentes entre as variáveis dos objetivos.

A construção da conjunção pode ser vista como uma estrutura em árvore, onde os objetos são folhas das relações, as quais estão ligadas à questão, que torna-se a raíz da árvore. No caso da questão cujo objeto carro era a incógnita, tem-se como nodos filhos da raiz os predicados da relação comprar, e de cada relação partem duas folhas com os objetos. Através da varredura desta árvore pode-se deduzir que a variável Oque referia-se ao objeto carro , comum em ambos objetivos, tornando, deste modo, a questão válida.

4.2.5 REGRAS

Um programa de árvore genealógica contém exemplos para poder-se estudar regras:

```
progenitor(maria, josé).
progenitor(joão, josé).
progenitor(joão, ana).
progenitor(josé, júlia).
progenitor(josé, íris).
progenitor(íris, jorge).
masculino(joão).
masculino(josé).
masculino(jorge).
feminino(maria).
feminino(júlia).
feminino(ana).
feminino(íris).
```

Um predicado que possuem um único argumento, normalmente é usado para declarar propriedades simples de determinado objeto.

Além dos predicados acima, pode-se criar relações representando os filhos.

```
filho(josé, joão).
```

Entretanto pode-se definir a relação filho de uma maneira mais elegante, fazendo o uso do fato de que ela é o inverso da relação progenitor e esta já está definida. Tal alternativa pode ser baseada na seguinte declaração lógica:

Para todo X e Y

Y é filho de X se

X é progenitor de Y.

Essa formulação já se encontra bastante próxima do formalismo adotado em Prolog. A cláusula correspondente, com a mesma leitura acima, é:

```
filho(Y, X) ¬ progenitor(X, Y).
```

que também pode ser lida como: Para todo X e Y, se X é progenitor de Y, então Y é filho de X.

Cláusulas Prolog desse tipo são denominadas regras. Há uma diferença importante entre regras e fatos. Um fato é sempre verdadeiro, enquanto regras especificam algo que pode ser verdadeiro se algumas condições forem satisfeitas [PAL97]. As regras tem uma parte de conclusão (o lado esquerdo da cláusula), e uma parte de condição (o lado direito da cláusula).

O símbolo "¬" significa ″se″ e separa a cláusula em conclusão, ou cabeça da cláusula, e condição ou corpo da cláusula, como é mostrado no esquema abaixo. Se a condição expressa pelo corpo da cláusula - progenitor (X, Y) - é verdadeira então, segue como conseqüência lógica que a cabeça - filho(Y, X) - também o é. Por outro lado, se não for possível demonstrar que o corpo da cláusula é verdadeiro, o mesmo irá se aplicar à cabeça.

```
filho(Y, X) ¬ progenitor(X, Y)
```

cabeça ----------> se <---------- corpo

(conclusão) (condição)

A utilização das regras pelo sistema Prolog é ilustrada pelo seguinte exemplo: Pergunta-se ao programa se José é filho de Maria:

```
?-filho(josé, maria).
```

Não há nenhum fato a esse respeito no programa, portanto a única forma de considerar esta questão é aplicando a regra correspondente. A regra é genérica, no sentido de ser aplicável a quaisquer objetos X e Y. Logo pode ser aplicada a objetos particulares, como josé e maria. Para aplicar a regra, Y será substituído por josé e X por maria.

A parte de condição se transformou então no objetivo progenitor(maria, josé). Em seguida o sistema passa a tentar verificar se essa condição é verdadeira. Assim o objetivo inicial, filho(josé, maria), foi substituído pelo sub-objetivo progenitor(maria, josé). Esse novo objetivo apresenta-se como trivial, uma vez que há um fato no programa estabelecendo exatamente que Maria é um dos progenitores de José. Isso significa que a parte de condição da regra é verdadeira, portanto a parte de conclusão também é verdadeira e o sistema responde sim.

Para melhor exemplificar, adiciona-se mais algumas relações ao programa. A especificação, por exemplo, da relação mãe entre dois objetos do nosso domínio pode ser escrita baseada na seguinte declaração lógica:

Para todo X e Y

X é mãe de Y se

X é progenitor de Y e

X é feminino.

Que, traduzida para Prolog, conduz à seguinte regra:

```
mãe(X, Y) ¬ progenitor(X, Y), feminino(X).
```

4.2.6 CONSTRUÇÕES RECURSIVAS

Para estudar construções recursivas, adiciona-se ao programa a relação antepassado, definida a partir da relação progenitor. A definição necessita ser expressa por meio de duas regras, a primeira das quais definirá os antepassados diretos (imediatos) e a segunda os antepassados indiretos. Diz-se que um certo X é antepassado indireto de algum Z se há uma cadeia de progenitura entre X e Z.

A primeira regra, que define os antepassados diretos, é bastante simples e pode ser formulada da seguinte maneira:

Para todo X e Z

X é antepassado de Z se

X é progenitor de Z.

ou, traduzindo para Prolog:

```
antepassado(X, Z) ¬
      progenitor(X, Z).
```

Por outro lado, a segunda regra é mais complicada, porque a cadeia de progenitores poderia se estender indefinidamente. Uma primeira tentativa seria escrever uma cláusula para cada posição possível na cadeia. Isso conduziria a um conjunto de cláusulas do tipo:

```
antepassado(X, Z) ¬
      progenitor(X, Y),
      progenitor(Y, Z).
antepassado(X, Z) ¬
      progenitor(X, Y1),
      progenitor(Y1, Y2),
      progenitor(Y2, Z).
antepassado(X, Z) ¬
      progenitor(X, Y1),
      progenitor(Y1, Y2),
      progenitor(Y2, Y3),
      progenitor(Y3, Z).       ... etc.
```

Isso conduziria a um programa muito grande e que, de qualquer modo, somente funcionaria até um determinado limite, isto é, somente forneceria antepassados até uma certa profundidade na árvore genealógica de uma família, porque a cadeia de pessoas entre o antepassado e seu descendente seria limitada pelo tamanho da maior cláusula definindo

essa relação. Há entretanto uma formulação elegante e correta para a relação antepassado que não apresenta qualquer limitação. A idéia básica é definir a relação em termos de si própria, empregando um estilo de programação em lógica denominado recursivo:

Para todo X e Z

X é antepassado de Z se

existe um Y tal que

X é progenitor de Y e

Y é antepassado de Z.

A cláusula Prolog correspondente é:

```
antepassado(X, Z) ¬
      progenitor(X, Y),
      antepassado(Y, Z).
```

Assim é possível construir um programa completo para a relação antepassado composto de duas regras: uma para os antepassados diretos e outra para os indiretos. Reescrevendo as duas juntas tem-se:

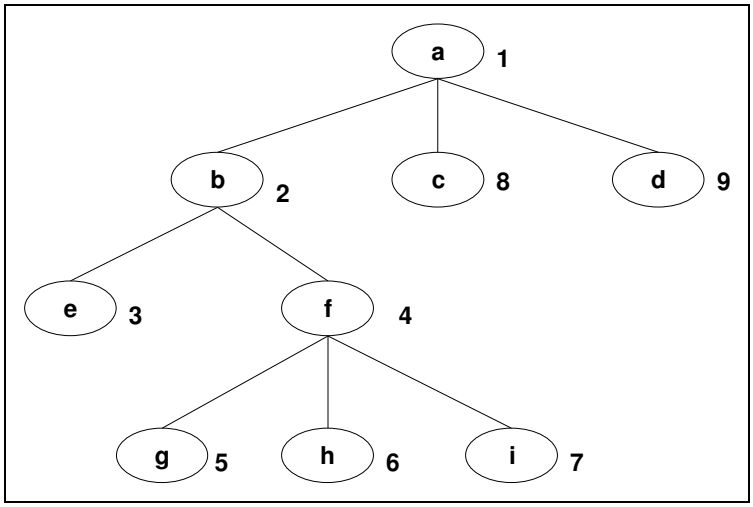
```
antepassado(X, Z) ¬
      progenitor(X, Z).
antepassado(X, Z) ¬
      progenitor(X, Y),
      antepassado(Y, Z).
```

Esta definição pode levar a uma pergunta: Como é possível, ao definir alguma coisa, empregar essa mesma coisa se ela ainda não está completamente definida? Tais definições são denominadas recursivas e do ponto de vista da lógica são perfeitamente corretas e inteligíveis. Por outro lado o sistema Prolog deve muito do seu potencial de expressividade à capacidade intrínseca que possui de utilizar facilmente definições recursivas. O uso de recursão é, em realidade, uma das principais características herdadas da lógica pela linguagem Prolog [CLO94].

4.2.7 O MECANISMO DE BACKTRACKING

De acordo com [PAL97] na execução dos programas Prolog, a evolução da busca por soluções assume a forma de uma árvore - denominada árvore de pesquisa ou *search tree* - que é percorrida sistematicamente de cima para baixo (*top-down*) e da esquerda para direita, segundo o método denominado *depth-first search* ou pesquisa em profundidade. A Figura 2 ilustra esta idéia, ela representa a árvore correspondente à execução do seguinte programa abstrato, onde a, b, c, etc. possuem a sintaxe de termos Prolog:

```
a ¬ b.
a ¬ c.
a ¬ d.
b ¬ e.
b ¬ f.
f  ¬ g.
f  ¬ h.
f  ¬ i.
d.
```



Fonte: [PAL97], p.108

FIGURA 2 - ÁRVORE DE PESQUISA

O programa representado na figura 2 será bem sucedido somente quando o nodo d for atingido, uma vez que este é o único fato declarado no programa. De acordo com a ordenação das cláusulas, d será também o último nodo a ser visitado no processo de execução. O caminho percorrido é dado abaixo

a, b, e, (b), f, g, (f), h, (f), i, (f), (b), (a), c, (a), d

onde o caminho em *backtracking* é representado entre parênteses.

Como visto, os objetivos em um programa Prolog podem ser bem-sucedidos ou falhar. Para um objetivo ser bem-sucedido ele deve ser unificado com a cabeça de uma cláusula do programa e todos os objetivos no corpo desta cláusula devem também ser bem-sucedidos. Se tais condições não ocorrerem, então o objetivo falha [CLO94].

Quando um objetivo falha, em um nodo terminal da árvore de pesquisa, o sistema Prolog aciona o mecanismo de backtracking, retornando pelo mesmo caminho percorrido, na tentativa de encontrar soluções alternativas. Ao voltar pelo caminho já percorrido, todo o trabalho executado é desfeito [CLO94]. O seguinte exemplo, extraído de [PAL97], sobre o predicado gosta/2 (onde 2 indica a aridade do predicado) pode ajudar a esclarecer tais idéias.

```
gosta(joão, jazz).
gosta(joão, renata).
gosta(joão, lasanha).
gosta(renata, joão).
gosta(renata, lasanha).
```

O significado intuitivo do predicado gosta(X, Y) é "X gosta de Y". Supondo o conhecimento acima, quere-se saber do que ambos, joão e renata, gostam. Isto pode ser formulado pelos predicados:

```
gosta(joão, X), gosta(renata, X).
```

O sistema Prolog tenta satisfazer o primeiro objetivo, desencadeando a seguinte execução *top-down*:

a) Encontra que joão gosta de jazz;

b) instancia X com "jazz";

c) tenta satisfazer o segundo objetivo, determinando se "renata gosta de jazz";

d) falha, porque não consegue determinar se renata gosta de jazz;

e) realiza um *backtracking* na repetição da tentativa de satisfazer gosta(joão, X), esquecendo o valor "jazz";

f) encontra que joão gosta de renata;

g) instancia X com "renata";

h) tenta satisfazer o segundo objetivo determinando se "renata gosta de renata";

i) falha porque não consegue demonstrar que renata gosta de renata;

j) realiza um *backtracking*, mais uma vez tentando satisfazer gosta(joão, X), esquecendo o valor "renata";

k) encontra que joão gosta de lasanha;

l) instancia X com "lasanha";

m) encontra que "renata gosta de lasanha";

n) é bem-sucedido, com X instanciado com "lasanha".

O *backtracking* automático é uma ferramenta poderosa e a sua exploração é de utilidade para o programador.

5 GERÊNCIA DE BASES DE DADOS EM PROGRAMAÇÃO LÓGICA

Uma base de dados pode ser entendida como uma coleção de dados interrelacionados, armazenada de modo independente do programa que a utiliza, permitindo a recuperação, inserção, remoção e modificação de forma controlada. Segundo [GRA88] a lógica dos predicados é uma regra importante na teoria das bases de dados relacionais e pode ser usada para expressar formalismos relacionais e as funcionalidades normalmente associadas a estes formalismos.

5.1 ASPECTOS DE GERÊNCIA DE BASES DE DADOS

De acordo com [DAT90] um dos marcos mais importantes no desenvolvimento da pesquisa acerca de bases de dados foi a introdução do modelo relacional por Codd em 1970. Em tal modelo, os dados são definidos por meio de relações sobre domínios e os fatos individuais são representados como tuplas de valores sobre tais domínios. Uma relação com um conjunto de tuplas é também denominada uma "tabela". Segundo [DAT90] o modelo relacional é conceitualmente muito "limpo" e elegante, apoiado por sólida fundamentação matemática.

Diferentemente de outros modelos de bases de dados, o modelo relacional não possui o conceito de "pointer", de modo que a associação entre diferentes tabelas é feita através da identidade explícita de valores de atributos. Este princípio concentra o esforço de implementação em obter maior velocidade de acesso, ao passo que a vantagem natural é a grande flexibilidade e entendimento do processo de modelagem de dados [DAT90].

O modelo relacional puro nem sempre é poderoso o bastante para modelagens avançadas, devido à falta de expressividade semântica . Por exemplo, o modelo relacional não requer que, para cada empregado, o atributo empregador corresponda a uma tupla existente na base de dados. Em modelos reais há dois tipos de regras que relacionam as tabelas uma à outra: as regras genéricas, que definem novas tabelas virtuais que não são explicitamente armazenadas, e as regras restritoras, que estabelecem restrições sobre o que é permitido na base de dados.

Um exemplo de regras restritoras é dada pelas dependências funcionais, que especificam que atributos chave são e devem ser únicos. Um outro exemplo seria uma regra como "Todos os elefantes são cor-de-cinza." Que deduz a cor de um elefante na base de dados, produzindo ainda uma restrição que garante que, nas atualizações subsequentes, nenhum elefante de outra cor será armazenado na base de dados. Tais bases de dados são denominadas "dedutivas".

Questões de semântica são mais importantes para o projeto de uma base de conhecimento do que métodos para a codificação de dados. Segundo [GRA88] quando os projetistas de base de dados adicionam mais informação semântica às bases de dados, os modelos resultantes começam a assemelhar-se aos sistemas de representação de conhecimento desenvolvidos pelos pesquisadores de inteligência artificial. Um desses esquemas de representação de conhecimento é conhecido como "rede semântica". Uma rede semântica é um formalismo para representar fatos e relacionamentos entre fatos por meio de relações binárias.

Os relacionamentos individuais são conectados em uma rede, onde os objetos são representados uma única vez. Para relações binárias, as redes semânticas são um formalismo com uma notação gráfica simples. Quando se tenta, entretanto, representar relações n-árias em redes semânticas é se forçado a empregar construções artificiais, perdendo o formalismo das redes semânticas, grande parte dos seus atrativos.

5.2 BASES DE DADOS EM PROLOG

Qualquer aplicação dispõe de uma base de dados onde são armazenados termos (dados) e cláusulas. Estes dados podem estar em formas de fatos, dentro do próprio programa, ou serem acessados via predicados específicos. A segunda opção, motivo deste estudo, possui largas vantagens sobre a primeira:

- controle da ordem em que os termos aparecem na base de dados. Pode-se incluir termos no início, final ou no meio dos outros termos;
- pode-se navegar livremente pela base de dados para localizar termos. Pode-se navegar para frente, para trás ou para um ponto específico com uma chamada;

- pode-se apagar, substituir um termo por outro, ou simplesmente examinar os termos armazenados na base de dados.
- a base de dados se torna dinâmica, podendo sofrer alterações durante a execução do programa.

Um termo ou um conjunto de termos são armazenados através de uma chave. Esta chave é o nome pelo qual o termo é conhecido, tanto para o usuário, como para o programa.

No caso de cláusulas, a chave é o nome do predicado. O valor dos argumentos das cláusulas não são significantes. Todas as cláusulas de um predicado são recuperadas através da mesma chave. Por exemplo as cláusulas *move(X,Y) :- shift(a,b),* e *move(1,Z)* são todas recuperadas através da chave *move(_,_).*

O gerenciador da base de dados do Arity Prolog atribui um número de referência a cada termo. Diferentemente da chave, o número de referência é único para cada termo armazenado na base de dados. Um número de referência é escrito com um til (~) seguido por um número hexadecimal como:

~0005AD

O Arity Prolog separa cláusulas de dados. Predicados adicionais específicos para gerência da base de dados produzem a habilidade de armazenar informações na base de dados Prolog na forma de termos Prolog. Isto dá ao Prolog a característica de um sistema de gerência de base de dados. Ele dispõe de predicados para armazenar dados em arvores binárias e tabelas *hash*. Estes predicados operam sobre termos e não sobre cláusulas.

Dados e cláusulas devem ser manipulados de forma diferente quando uma base de dados é carregada. Um procedimento especial precisa ser executado para carregar os dados, devido a sua característica de dinamicidade.

A base de dados fica armazenada na memória e todas as manipulações são feitas na memória, isso agrega velocidade aos programas Prolog. O limite de tamanho da base de dados é o limite da memória provido pelo sistema operacional. Esse limite torna o Prolog

apto a ser usado na confecção de bases de dados inteligentes, sistemas de processamento de linguagem natural assim como também sistemas convencionais.

5.2.1 ESTRUTURAS DE DADOS

As estruturas de dados utilizadas no gerenciamento de uma base de dados Prolog são as listas lineares, árvores binárias e pesquisa *hash*. Tem-se em alguns casos a combinação destas duas últimas estruturas.

A lista linear é definida como sendo um conjunto de n >= 0 nós, x_1, x_2, ..., x_n, organizados estruturalmente de forma a refletir as posições relativas dos mesmos: se n > 0, então x_1 é o primeiro nó; para 1 < k < n, o nó x_k é precedido pelo nó x_{k-1} é seguido do x_{k+1}; e x_n é o último nó. Quando n=0, diz-se que a lista é vazia [VEL83].

O método de pesquisa utilizado nas listas lineares consiste na varredura serial da tabela, durante o qual o argumento de pesquisa é comparado com a chave de cada entrada até ser encontrada uma que seja igual, ou ser atingido o final da tabela, caso a chave procurada não esteja presente na tabela [KRU94]. No quadro 1 é mostrado o algoritmo correspondente ao método de pesquisa seqüencial.

```
proc seqüencial (t: tabela; arg: chave; n,i: int)
  { t: tabela onde será feita a pesquisa,
    arg: argumento de pesquisa,
    n: número de entradas da tabela
    i: índice da entrada procurada (i=0: não existe entrada)}

var r: int;

inicio
      I := 0;
      para r de r + 1 até n faça
         se t[r].chave = arg
         então
         início
            i:= r;
            escape
         fim
fim
```

QUADRO 01 : ALGORITMO DE PESQUISA SEQÜENCIAL.

As árvores binárias possuem características de uma relação de hierarquia ou de composição, onde um conjunto de dados é hierarquicamente subordinado a outro. Formalmente uma árvore é um conjunto finito T de um ou mais nós, tais que: existe um nó denominado raiz da árvore; os demais nós formam m >= 0 conjuntos disjuntos S_1, S_2,...,S_m onde cada um desses conjuntos é uma árvore. As árvores S_i (1<=i <=n) recebem a denominação de subárvores [VEL83].

De acordo com [VEL83] existem vários métodos de caminhamento em árvores, que permitem percorrê-la de forma sistemática e de tal modo que cada nó seja visitado apenas uma vez. Um caminhamento completo sobre uma árvore produz um seqüência linear dos nós, de maneira que cada nó da árvore passa a ter um nó seguinte ou um nó anterior, ou ambos, para uma dada forma de caminhamento. Existem determinadas ordens de caminhamento mais freqüentemente utilizadas. As três principais, e seus algoritmos, são relacionados abaixo. O termo visita está sendo utilizado para indicar o acesso a um nó para fins de executar alguma operação sobre ele e será expresso pelo procedimento visita(p) onde p é a referência a um nó da árvore.

Caminhamento pré-fixado: onde faz-se uma visita a raiz, percorre-se a subárvore da esquerda, percorre-se a subárvore da direita.

```
proc pré-fixado (a: árvore)
se a <> nil
então
      início
            execute visita(a);
            execute pré-fixado(esq(a));
            execute pré-fixado(dir(a));
      fim
```

Caminhamento central: onde percorre-se a subárvore da esquerda, visita-se a raiz, percorre-se a subárvore da direita.

```
proc central (a: árvore)
se a <> nil
então
      início
            execute central(esq(a));
            execute visita(a);
            execute central (dir(a));
      fim
```

Caminhamento pós-fixado: onde percorre-se a subárvore da esquerda, percorre-se a subárvore da direita, visita-se a raiz.

```
proc pós-fixado (a: árvore)
se a <> nil
então
      início
            execute pós-fixado(esq(a));
            execute pós-fixado(dir(a));
            execute visita(a);
      fim
```

No entanto, de acordo com [ARI88] o algoritmo utilizado pelo gerenciamento de bases de dados Prolog é a árvore binária de pesquisa. No quadro 02 é apresentado este algoritmo:

```
proc pesquisa (a: árvore, s: simbolo)
se a <> nil
então
      se s < info (a)
      então
        execute pesquisa (esq (a), s)
      senão
        se s > info (a)
        então
            execute pesquisa (dir (a), a)
        senão
        início
            aloque (a);
            execute (poe_info (a, s);
            execute (poe_esq (a, nil);
            execute (poe_dir (a, nil);
        fim
```

QUADRO 02 : ALGORITMO DE PESQUISA EM ÁRVORE.

O procedimento mostrado no quadro 02 recebe como parâmetro o endereço de memória de uma árvore, representado pela letra a e o símbolo a ser incluído na árvore de pesquisa, representado pela letra s. A informação em s é testada com relação a informação do nó apontado por a, caso menor, acontece uma nova pesquisa ao nó da esquerda, chamando-se recursivamente o próprio algoritmo. Caso maior acontece uma nova pesquisa ao nó da direita, chamando-se recursivamente o algoritmo. O procedimento aloque, aloca um novo nó na árvore, poe_info insere a informação no nó alocado, poe_esq e poe_dir coloca informações nos nós da esquerda e direita respectivamente.

O método *hash* ou cálculo de endereço consiste no armazenamento de cada entrada em um endereço calculado pela aplicação de uma função à chave de entrada. O processo de pesquisa sobre uma tabela organizada desta maneira é similar ao processo de inserção de uma entrada e consiste na aplicação da função de cálculo de endereço ao argumento de pesquisa, obtendo como resultado o endereço da entrada procurada [VEL83].

A eficiência da pesquisa neste tipo de organização depende fundamentalmente da função de cálculo de endereço. A função ideal seria aquela que gerasse um endereço diferente para cada um dos n diferentes valores da chave presentes na tabela. No entanto, normalmente não é possível conseguir este tipo de função e usa-se funções que geram colisões, isto é, que atribuem o mesmo endereço a diferentes valores de chave [VEL83].

5.2.2 PREDICADOS DE GERÊNCIA DA BASE DE DADOS

Os predicados de gerência de bases de dados do Prolog são descritos e implementados no Prolog da Universidade de Edimburgo por Davis Warren, Fernando Pereira e Luis Pereira. Esta implementação tornou-se padrão de fato e várias outras implementações tem tomando esta como base [CLO94].

Pode-se dividir os predicados de gerência de base de dados Prolog em quatro grupos: os predicados de manipulação de termos em estruturas simples, os predicados de manipulação de cláusulas, os predicados de manipulação de árvores e os predicados de manipulação de tabelas *hash* [ARI88].

Os predicados de manipulação de termos em estruturas simples são usados quando a base de dados não requer nenhuma ordenação. Um conjunto separado de predicados são disponibilizados para manipular cláusulas. Estes predicados entendem a sintaxe das cláusulas e permitem que um método simples para gravar cláusulas na base de dados seja usado.

Já os predicados de manipulação de arvores são usados em casos mais elaborados quando existe a necessidade de ordenação dos termos e quando a base de dados é de tamanho grande. É uma maneira de classificar uma série de termos conforme determinada categoria. Buscas através destes elementos são mais eficientes porque é especificado a

categoria a qual o termo pertence. Por outro lado, a procura é limitada a termos de uma determinada categoria, os das outras são ignorados.

A tabela *hash* provê um método similar de indexação. Uma tabela *hash* consiste em um número separado de categorias chamadas de *hash buckets*. Um *hash bucket* consiste em uma base de dados de referências apontando para os termos da base de dados. É melhor usar tabela *hash* quando se possui uma base de dados muito extensa e quando esta pode ser separada em categorias. Geralmente o *hash* usa menos espaço que uma árvore. No entanto a árvore retorna os dados de forma ordenada, enquanto o *hash* não.

Pode-se ainda combinar o uso de arvores binárias e da tabela *hash*.

Para melhor visualizar a sintaxe dos predicados de gerência de bases de dados Prolog usa os símbolos de soma (+) para indicar que um argumento é de entrada; subtração (-) para indicar que o argumento é saída após a execução do predicado; e interrogação (?) para indicar que o argumento pode ser um termo ou uma variável que retornará valor. Por exemplo:

```
recorda(+Chave, +Termo, -Ref).
recorded(+Chave, ?Termo, -Ref).
```

No predicado *recorda* entende-se que se precisa fornecer um argumento Chave e um argumento Termo como entrada e o Prolog irá devolver o resultado no argumento de saída Ref. No predicado *recorded* entende-se que é preciso fornecer um argumento Chave, podendo porém fornecer um termo ou uma variável no argumento Termo.

5.2.2.1 SALVANDO TERMOS

O predicado usado para salvar termos na base de dados é o *record*. Este predicado possui variações que o tornam versátil e funcional.

recorda(+Chave,+Termo,-Ref)

Este predicado adiciona um termo no início de um conjunto de termos que tem a mesma chave. Retorna o número de referência atribuído ao termo. O argumento Chave deve conter a chave abaixo da qual se quer inserir o termo. O argumento Termo deve conter

o termo propriamente dito e o argumento Ref pode ser uma variável, onde o Prolog devolverá o número de referência atribuído ao do termo. O argumento Ref pode ainda ser o símbolo *underline* '_' o que indica ao Prolog que a referência não precisará ser retornada.

recordz(+Chave,+Term,-Ref)

Este predicado adiciona um termo no final de um conjunto que tem a mesma chave. Retorna o número de referência atribuído ao termo. O argumento Chave deve conter a chave abaixo da qual se quer inserir o termo. O argumento Termo deve conter o termo propriamente dito e o argumento Ref pode ser uma variável, onde o Prolog devolverá o número de referência atribuído ao do termo. O argumento Ref pode ainda ser o símbolo *underline* '_' o que indica ao Prolog que a referência não precisará ser retornada.

record_after(+Ref,+Termo,-NewRef)

Este predicado adiciona um termo entre dois termos de um conjunto de que tem a mesma chave retornando o número de referência atribuído ao novo termo. O argumento Ref deve conter o número de referência do termo ao qual deseja-se seguir o novo. O argumento Termo deve conter o termo propriamente dito e o argumento NewRef pode ser uma variável, onde o Prolog devolverá o número de referência atribuído ao novo termo. O argumento NewRef pode ainda ser o símbolo *underline* '_' o que indica ao Prolog que a referência não precisará ser retornada.

5.2.2.2 SALVANDO CLÁUSULAS

asserta(+Clause)

Este predicado adiciona cláusulas no início para um mesmo predicado. É como *recorda*, exceto que é usado somente para cláusulas e não para termos ou outros tipos. Se a cláusula especificada não tiver corpo, *asserta* o especifica como verdadeiro. Este predicado ajuda na ordem em que as cláusulas aparecem na base de dados.

assertz(+Clause), assert(+Clause)

Este predicado adiciona cláusulas no final de um mesmo predicado. É como *recordz*, exceto que é usado somente para cláusulas e não para termos ou outros tipos. Se a cláusula especificada não tiver corpo, *asserta* o especifica como verdadeiro.

5.2.2.3 SALVANDO ARVORES

recordb(+Tree_Name, +Sort_Key,+Termo)

Usado para adicionar temos a uma estrutura de árvore. Onde o argumento Tree_Name deve conter o nome da árvore, Sort_Key é a chave pela qual os termos na árvore podem ser acessados. E Termo é o termo propriamente dito que deve ser adicionado á árvore.

5.2.2.4 SALVANDO TABELAS HASH

recordh (+Table_Name,+Sort_Key,+Termo)

É usado para gravar termos em uma estrutura *hash*. O argumento Table_Name indica em qual tabela os dados devem ser incluídos. O argumento Sort_Key indica a categoria onde o termo deve ser adicionado. Em outros termos este argumento especifica o *hash bucket* dentro do qual o termo será armazenado.

5.2.2.5 RESTAURANDO TERMOS

recorded(+Chave,?Termo,-Ref)

Este predicado faz uma busca na base de dados retornando verificando a existência de um determinado termo ou retornando um conjunto de termos. O argumento Chave deve conter a chave dos termos que se deseja consultar. O argumento Termo pode conter o termo que se deseja verificar a existência ou uma variável. No segundo caso toda a lista de termos é mostrada, termo a termo, esperando que o operador tecle ponto e vírgula (;) para continuar a pesquisa ou ponto (.) para encerrar a pesquisa.

5.2.2.6 RESTAURANDO ARVORES

retrieveb(+Tree_Name,?Sort_Key,?Termo)

Usa-se este predicado para retornar uma lista ordenada dos dados de um árvore. Como no predicado *recorded* tem-se a opção de substituir os argumentos Sort_Key e Termo por valores que se deseja consultar ou por variáveis. No segundo caso toda a lista de termos é mostrada, termo a termo, esperando que o operador tecle ponto e vírgula (;) para continuar a pesquisa ou ponto (.) para encerrar a pesquisa. O funcionamento deste termo pode ser de diversas maneiras, dependendo de quais parâmetros são instanciados e os que são usados como variáveis.

betweenb(+Tree_Name,+Key1,+Key2,+Relation1,+Relation2,-Key,-Termo)

Usado para consultar uma parte específica da base de dados armazenada em forma de árvore. O argumento Key1 e Key2 são respectivamente o nó inicial e o nó final. A ordem de pesquisa se dá pela ordem destes argumentos. Por exemplo: se Key1 for 10 e Key2 for 1 a ordem de retorno dos nós é descendente.

Os argumentos Relation1 e Relation2 indicam qual a relação dos termos com Key1 e Key2. Estes argumento podem ser: =, < ou >. Quando se usa > e < o intervalo é não inclusive, e se usar = o Intervalo é inclusive.

Pode-se criar predicados que pesquisem de um certo ponto da árvore para frente. Como por exemplo:

```
forwardb(Arvore,Chave_Inicio,Chave_Retorno,Termo) :-
                betweenb(Arvore,Chave_Inicio,_,_,_,Chave_Retorno,Termo).
```

Ou predicados que permitem caminhar de trás para frente na árvore:

```
backwardb(Arvore,Chave_Inicio,Chave_Retorno,Termo) :-
          betweenb(Arvore,Chave_Inicio,'$begin',_,_,Chave_Retorno,Termo).
```

Portanto para que isto seja possível precisa-se indicar que o limite inferior da procura é a primeira entrada na árvore. A notação especial ‘$begin’ é usada para indicar a primeira entrada em um árvore.

5.2.2.7 RESTAURANDO TABELAS HASH

retrieveh (+Table_Name,?Sort_Key,?Termo)

Este predicado retorna termos de uma tabela *hash.* O argumento Table_Name indica em qual tabela os dados devem ser consultados. O argumento Sort_Key indica a categoria onde o termo deve ser adicionado ou uma variável. O argumento Termo pode conter o termo que se deseja verificar a existência ou uma variável. No segundo caso dos argumentos conterem uma variável toda a lista de termos é mostrada, termo a termo, esperando que o operador tecle ponto e vírgula (;) para continuar a pesquisa ou ponto (.) para encerrar a pesquisa.

5.2.2.8 APAGANDO TERMOS

erase(+Ref)

Remove o termo referenciado. O sistema mantém os termos que foram apagados com o predicado *erase*.

eraseall(+Chave)

Remove todos os termos armazenados com esta chave. Da mesma forma que *erase* os termos são apenas marcados como excluídos, não excluídos fisicamente.

expunge

Este predicado é necessário quando o sistema usa os predicados *erase* ou *retract*. Quando usa-se os predicados *erase* ou *retract* o sistema guarda um registro para cada termo que foi apagado (os ponteiros para os termos excluídos permanecem). Toda vez que os predicados *erase* ou *retract* são usados, o sistema traça um caminho dos termos apagados. O predicado *expunge* elimina as referências aos termos apagados e libera espaço livre para ser usado.

O predicado *expunge* não precisa ser usado toda vez que os predicados *erase* ou *retract* forem usados. Geralmente usa-se expunge nas seguintes circunstâncias:

- Depois do programa usar muitas vezes os predicados *erase* ou *retract*. É muito importante usar *expunge* quando os predicados *erase* ou *retract* estão em loop;
- usa-se *expunge* imediatamente antes de salvar a base de dados usando o predicado *save*, prevenindo que as referências eliminadas não serão usadas após a gravação.

É importante notar que *expunge* deveria somente ser usado quando se tem certeza que a eliminação de termos apagados não vai resultar na eliminação de referências usadas mais tarde no programa. Se o programa usar um procedimento *Backtraking* que usa *recorded* para retornar um termo ou *erase* e *recordz* para eliminar e adicionar termos, não deve usar-se *expunge* até que este procedimento esteja completo. Isso porque *recorded* usa referências do termo retornado para determinar a referência ao próximo termo.

```
do_test(Chave) :-
                  list_and_erase(Chave),
                  expunge.

list_and_erase(Chave) :-
                       recorded(Chave,X,Ref),
                       write(X), nl,

ifthen(X=yes,erase(Ref),recordz(Chave,not_used,_)),

                                  fail.
```

hard_erase(+Ref).

Este predicado apaga o termo com seu número de referência específico. Diferentemente do predicado *erase,* não guarda a referência dos termos que foram eliminados. Assim, o predicado *hard_erase* deveria ser somente usado em programas que não tenha perigo em o sistema se referenciar a elas.

Como o *hard_erase* não guarda dos termos eliminados, não precisa-se usar o predicado *expunge*.

5.2.2.9 APAGANDO CLÁUSULAS

retract(+Clause)

Remove cláusulas da base de dados. O argumento Clause deve conter o nome da cláusula.

abolisch(+Name/Aridade)

Remove todas as cláusulas da base de dados com o nome específico nome e aridade. Aridade é número de argumentos de um predicado.

5.2.2.10 APAGANDO ARVORES

removeb (+Tree_Name,+Sort_Key,+Term)

Remove um termo da árvore. Se algum argumento não for passado, então o primeiro termo da árvore especificada que satisfizer os outros argumentos será deletado.

removeallb (+Tree_Name)

Elimina uma árvore e todos os seus termos.

5.2.2.11 APAGANDO TABELAS HASH

removeh (+Table_Name,+Sort_Key,+Termo)

Usado para apagar um ou um conjunto de termo de uma tabela hash. O argumento Table_Name indica em qual tabela os dados devem ser incluídos. O argumento Sort_Key indica a categoria onde o termo deve ser adicionado ou uma variável. O argumento Termo pode conter o termo que se deseja verificar a existência ou uma variável. No caso dos argumentos conterem uma variável toda a lista de termos é que satisfaça a condição é eliminada, termo a termo, esperando que o operador tecle ponto e vírgula (;) para continuar a eliminação ou ponto (.) para encerrar.

removeallh (+Table_Name)

Remove toda uma tabela hash.

5.2.2.12 OUTROS PREDICADOS DE TERMOS

instance(+Ref,-Termo)

Este predicado retorna no argumento Termo o termo associado a um número de referência específico, passado no argumento Ref.

key(+Chave,-Ref)

Retorna no argumento Ref o número de referência de determinada chave.

Nota: O número de referência retornado por este predicado não deveria ser usado como sendo o argumento Ref para o predicado *instance*.

nref(+Ref,-Proximo)

Este predicado retorna o número de referência do próximo termo. É usado para navegar dentro de uma lista de termos.

Este predicado não retorna o termo e sim o seu número de referência. Assim precisa-se usar *nref* com outros predicados como *instace* para encontrar o termo associado a este número.

Quando Ref for o último termo dentro de uma lista, este predicado falha.

pref(+Ref,-Anterior)

Este predicado retorna o número de referência do termo anterior Ref. É usado para navegar dentro de uma lista de termos.

Este predicado não retorna o termo e sim o seu número de referência. Assim precisa-se usar *pref* com outros predicados como *instace* para encontrar o termo associado a este número.

Quando Ref for o primeiro termo dentro de uma lista, este predicado falha.

replace(+Ref,+Termo).

Procura um termo que tenha a referência passada em Ref e o substitui pelo novo termo que está no argumento Termo.

5.2.2.13 OUTROS PREDICADOS DE CLÁUSULAS

clause(+Head,-Body)

O predicado *clause* unifica Head a cabeça de uma cláusula, e unifica Body ao corpo da cláusula. O argumento Head precisa ser instanciado. Este predicado pode retroceder, retornando todas as cláusulas de um predicado unificado com Head e Body.

5.2.2.14 OUTROS PREDICADOS DE ARVORES

betweekeysb (+Tree_Name,+Key1,+Key2, -Key)

Este predicado é usado para retornar somente a chave dentro de dois determinados pontos da árvore. Se key1 estiver a frente de key2 na ordem padrão, as chaves são retornadas na ordem ascendente, caso contrário em ordem descendente.

defineb (+Tree_Name,+SpliSize,+Uniqueness, +Order)

Usado para definir alguns atributos da árvore. Dependendo da aplicação, a especificação de alguns atributos pode resultar em uma organização mais eficiente ou ajudar a customizar a organização da árvore.

SpliSize Indica quantos ramos um determinado nó pode alcançar antes dos termos serem colocados em um novo nó.

Uniqueness Não deixa com que a chave do termo seja duplicada.

Order Indica a ordem em que termos complexos devem ser armazenados na árvore.

replaceb (+Tree_Name,+Sort_Key,+Old_Term, +NewTerm)

Substitui o termo antigo pelo novo. Onde Tree_Name é o nome da árvore, Sort_Key a chave de acesso, Old_Termo o termo antigo e NewTerm o termo novo.

what_btrees (-BTree)

Retorna, através de backtracking, o nome das árvores existentes.

5.2.2.15 OUTROS PREDICADOS DE TABELAS HASH

defineh (+Table_Name,+HashBuckets)

Usado para especificar quantos *Hash Buckets* devem ser criados para uma tabela. O uso deste predicado é opcional.

5.2.2.16 MANIPULAÇÃO DA BASE DE DADOS EM DISCO

save(+nome_base)

Este predicado salva toda a base de dados em um arquivo em disco. O argumento nome_base deve conter o nome do arquivo sob o qual a base será gravada. Este predicado faz uma cópia dos dados armazenados em memória e os salva em disco.

restore(+nome_base)

Este predicado recupera uma base de dados armazenada em disco pelo predicado *save*, deixando os dados na memória da mesma maneira que estavam na hora em que o comando *save* foi executado. O argumento nome_base deve conter o nome da base a ser recuperada.

6 DESENVOLVIMENTO DO PROTÓTIPO

Uma maneira de consolidar o estudo é implementar uma ferramenta para programação lógica, ou seja, implementar um ambiente Prolog. Um ambiente é composto por várias partes e cada parte é bastante complexa. Desta forma, o trabalho proposto focaliza-se em uma delas: a gerência da base de dados, mais especificamente a implementação dos predicados de manipulação de termos. Estes predicados fornecem ao ambiente Prolog a capacidade de incluir, consultar, alterar e apagar termos de determinada base de dados. A partir do predicado, o protótipo executa o processo necessário para incluir, alterar, apagar ou consultar os termos da base de dados. O protótipo basicamente implementa alguns dos predicados apresentados no capítulo 5 através da manipulação das estruturas de dados de listas apresentadas no mesmo capítulo.

Este capítulo tem por objetivo descrever o protótipo, desde a metodologia empregada para o desenvolvimento, até a sua especificação e utilização, incluindo alguns detalhes da implementação.

6.1 METODOLOGIA UTILIZADA

A metodologia utilizada foi a prototipação. Segundo [MEL90], “a prototipação é um conjunto de técnicas e ferramentas de software para o desenvolvimento de modelos de sistemas”. O principal objetivo da prototipação é antecipar uma versão do sistema para que a sua funcionalidade possa ser avaliada mediante a utilização, percebendo-se os erros e omissões, efetuando de imediato correções ou ajustes com o mínimo de custo operacional. A metodologia de protótipos advoga, basicamente, o retorno ao uso da intuitividade, que é a forma mais natural do homem perceber o “mundo real”. A construção de um sistema é gradual dependendo da aprendizagem sobre a melhor solução [MEL90]. Esta metodologia é descrita pela figura 3.

Na etapa de identificação das necessidades e requisitos do sistema são definidos os objetivos do sistema a ser prototipado e identificados os dados gerados e requisitados para que o objetivo seja alcançado. Para esta etapa são utilizadas as técnicas de análise de dados, de elaboração do modelo lógico dos dados e de análise do funcionamento. Estas técnicas

são utilizadas na especificação do protótipo. Por se tratar de um trabalho acadêmico, a etapa de exame da viabilidade do projeto é a motivação descrita na introdução deste trabalho.

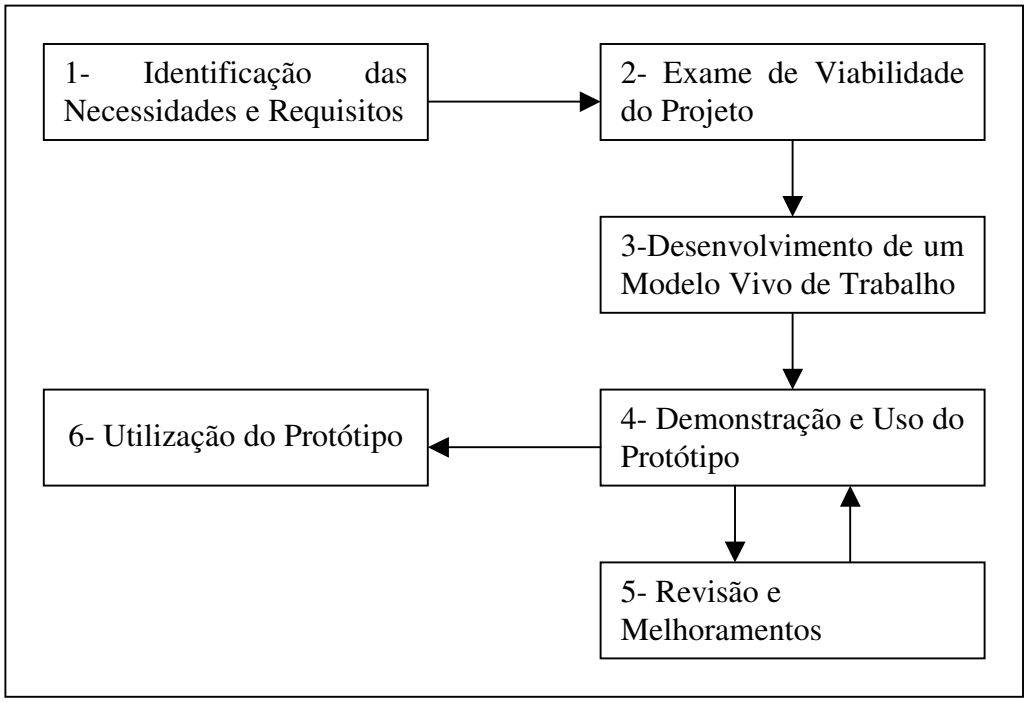


FIGURA 3 - METODOLOGIA DA PROTOTIPAÇÃO

6.2 ESPECIFICAÇÃO DO PROTÓTIPO

Na figura 4 mostra-se o diagrama de fluxo de dados (DFD) em nível de contexto do protótipo.

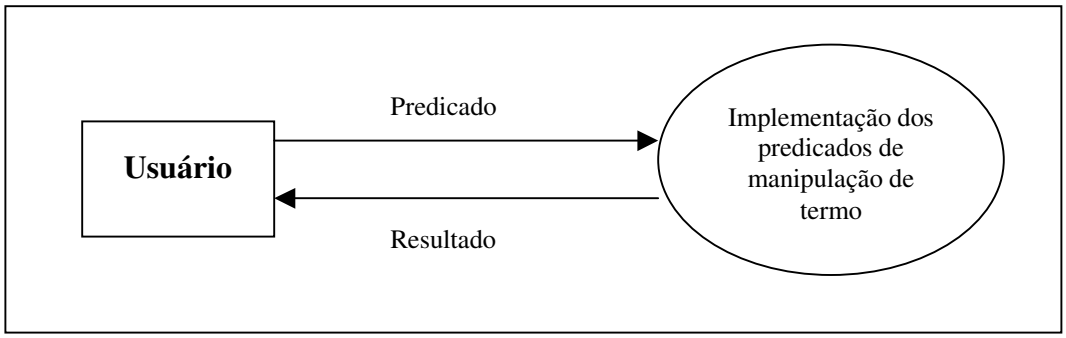


FIGURA 4 – DFD DO PROTÓTIPO

O usuário é a entidade externa. Digitado um predicado ele é desmontado, verifica-se sua sintaxe e chama-se um procedimento específico para sua execução passando os parâmetros necessários. Este é um módulo separado, junto a ele existe toda a gerência da interface com o usuário.

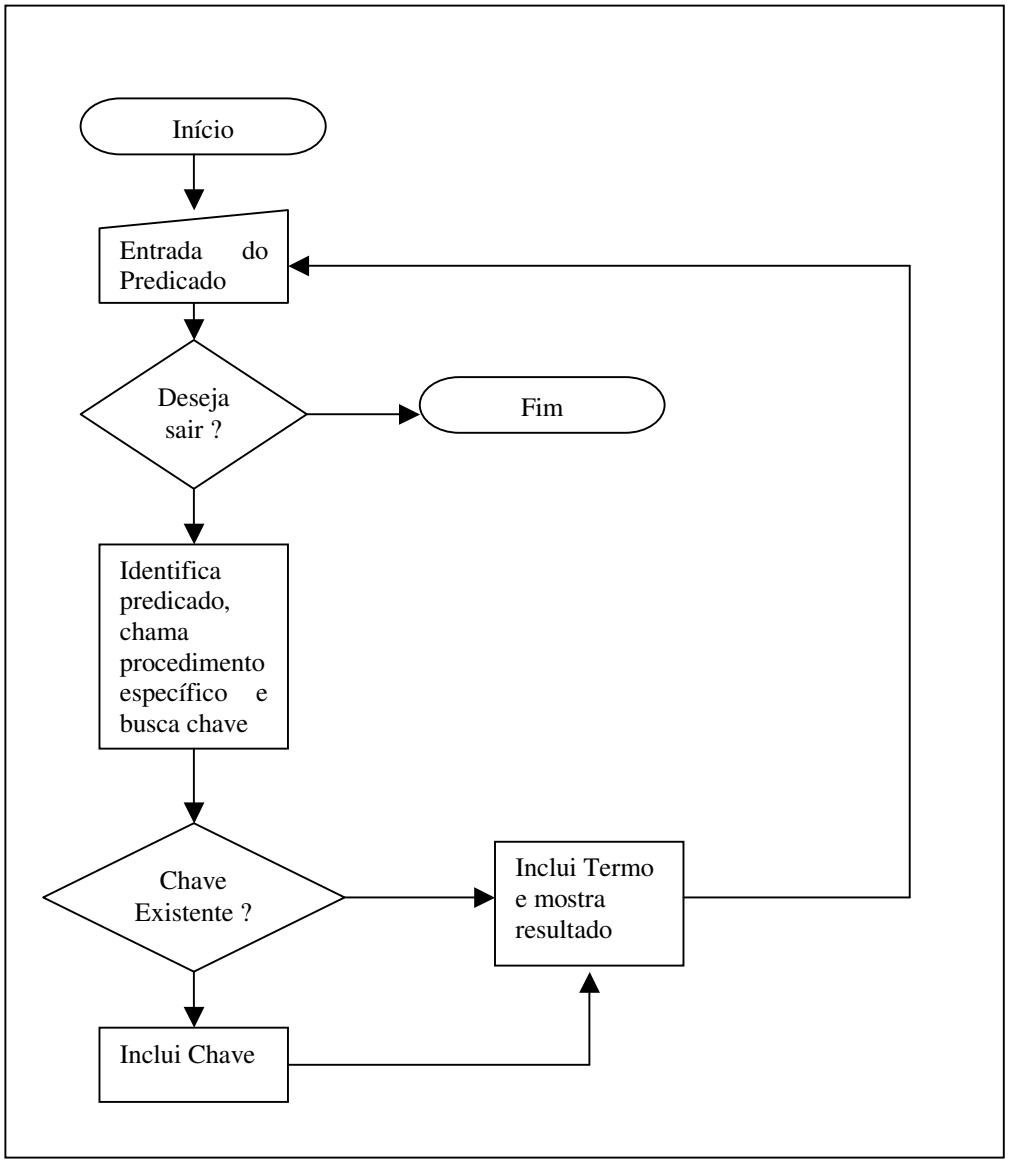


FIGURA 5 - MACRO FLUXO DO PROTÓTIPO

O módulo de execução dos predicados possui um procedimento específico para cada predicado, que pode chamar outros procedimentos. Mostra-se um macro fluxograma do funcionamento do protótipo para inclusão de termos na figura 5. Inicialmente o predicado é

identificado e chamado um procedimento específico que irá fazer a sua execução. No procedimento específico é feita a checagem para ver se a chave digitada já existe, se não existir ela é incluída e o termo é incluído na estrutura de dados de lista.

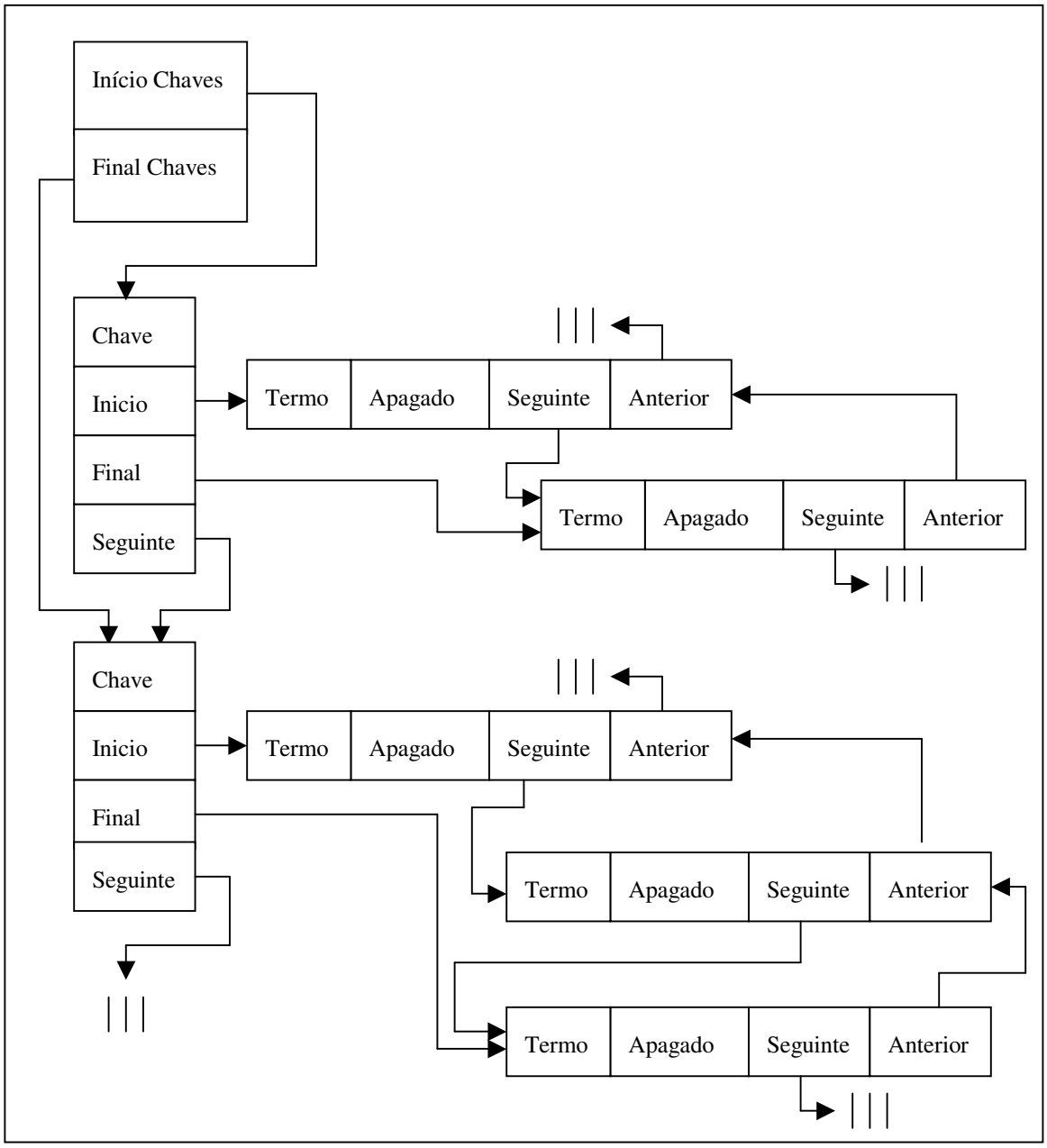


FIGURA 6 - ESTRUTURAS DE DADOS DO PROTÓTIPO

No desenvolvimento do protótipo usou-se duas listas lineares. A primeira que faz o controle das chaves abaixo das quais os termos estão armazenados, e a segunda que contém os termos propriamente ditos.

A lista das chaves é composta pela chave, pelos endereços dos nós inicial e final da lista de termos armazenados sob esta chave e um apontador que aponta para o nó seguinte. Esta lista é acessada toda vez que se faz necessário acessar algum termo, existe um procedimento que é chamado para buscar o endereço inicial e final da lista dos termos da chave.

A lista de termos é composta pelo termo, por um indicador que indica se o termo é apagado logicamente, um apontador que aponta o nó seguinte da lista e um apontador que aponta o nó anterior da lista. Sobre esta lista são executados diversos procedimentos de pesquisa, inclusão, alteração e exclusão. Estas duas listas são mostradas de forma gráfica na figura 6 e em forma de código Deplhi no quadro 03.

```
Aponta   = ^Nodo;

  Nodo  = record
            Termo    : String;
            Apagado  : Boolean;
            Seguinte : Aponta;
            Anterior : Aponta;
           end;

  Descr = record
            Comeco: Aponta;
            Final : Aponta;
          end;

  ApontaChave   = ^NodoChave;

  NodoChave  = record
            Chave             : String;
            DescritorChave    : Descr;
            Seguinte          : ApontaChave;
           end;

  DescrChave = record
            InicioChave: ApontaChave;
            FinalChave : ApontaChave;
          end;
```

QUADRO 03 – ESTRUTURAS DE DADOS DO PROTÓTIPO

6.3 AMBIENTE DE DESENVOLVIMENTO

Para o desenvolvimento do protótipo foi utilizado o ambiente de programação Delphi 4.0 para o sistema operacional Microsoft Windows sobre a plataforma IBM-PC. O Delphi é uma ferramenta de programação visual e a sua linguagem de programação, conhecida como Object Pascal, é baseada na linguagem Pascal. O hardware utilizado foi um Micro-Computador Intel Pentium Celeron de 300 MHz.

O Delphi possui muitas classes implementadas que facilitam a programação, porém elas foram utilizadas apenas na implementação da interface. A implementação dos predicados foi escrita sem o uso de bibliotecas do Delphi para facilitar que este protótipo seja portado para outros ambientes, como por exemplo, para o Free Pascal para Linux. O objetivo não era utilizar todos os recursos do Delphi, mas sim manter o código compatível com a linguagem Pascal padrão, permitindo que o protótipo seja disponibilizado em outros sistemas operacionais.

6.4 PREDICADOS IMPLEMENTADOS

Os predicados implementados são basicamente os de manipulação de termos, mostrados na figura 7.

Inclusão	*Exclusão*	*Alteração*	*Consulta*	*Diversos*	*Base*
recorda	erase	replace	recorded	key	save
recordz	eraseall			instance	restore
record_after	expunge			nref	
	hard_erase			pref	

FIGURA 7 - PREDICADOS IMPLEMENTADOS

6.5 O SISTEMA DESENVOLVIDO

A ferramenta desenvolvida é de uso específico para a experimentação dos predicados de gerência de base de dados para programação lógica desenvolvidos no protótipo. No seu desenvolvimento não houve preocupação com a funcionalidade da digitação dos predicados, nem com a praticidade da forma de mostrar o resultado. Devido a definição do componente do ambiente de programação utilizado para proceder as entradas de dados, não se consegue interagir na própria tela, como acontece no Arity/Prolog. Utilizou-se para isso de caixas de diálogo disponíveis no Delphi para fazer os questionamentos necessários ao usuário do protótipo.

O sistema é composto por um programa TCC.EXE. O sistema pode ser utilizado tanto em disquete como instalado no disco rígido. O único requisito básico é o sistema operacional, que deverá ser Windows 98, tendo em vista o fato do protótipo ter sido desenvolvido neste ambiente.

Para executar o sistema, não importa se em disquete ou disco rígido, basta digitar TCC para que o sistema seja carregado na memória. Em seguida aparecerá a tela principal do sistema como pode ser visto na figura 6.

6.5.1 TELAS

Na tela principal o usuário digita os predicados e os resultados são devolvidos. Reproduz-se esta tela na figura 6.

Todas as opções oferecidas pelo menu, também são oferecidas por botões na barra de ferramentas.

Na figura 7 vê-se o menu de arquivos do sistema, com as opções Abrir, Atualizar, Salvar e Sair.

Ao selecionar a opção Abrir, Atualizar ou Salvar será aberta uma caixa de diálogo para que seja selecionada a unidade e o subdiretório de uma base de dados que será aberta, atualizada ou gravada, respectivamente.

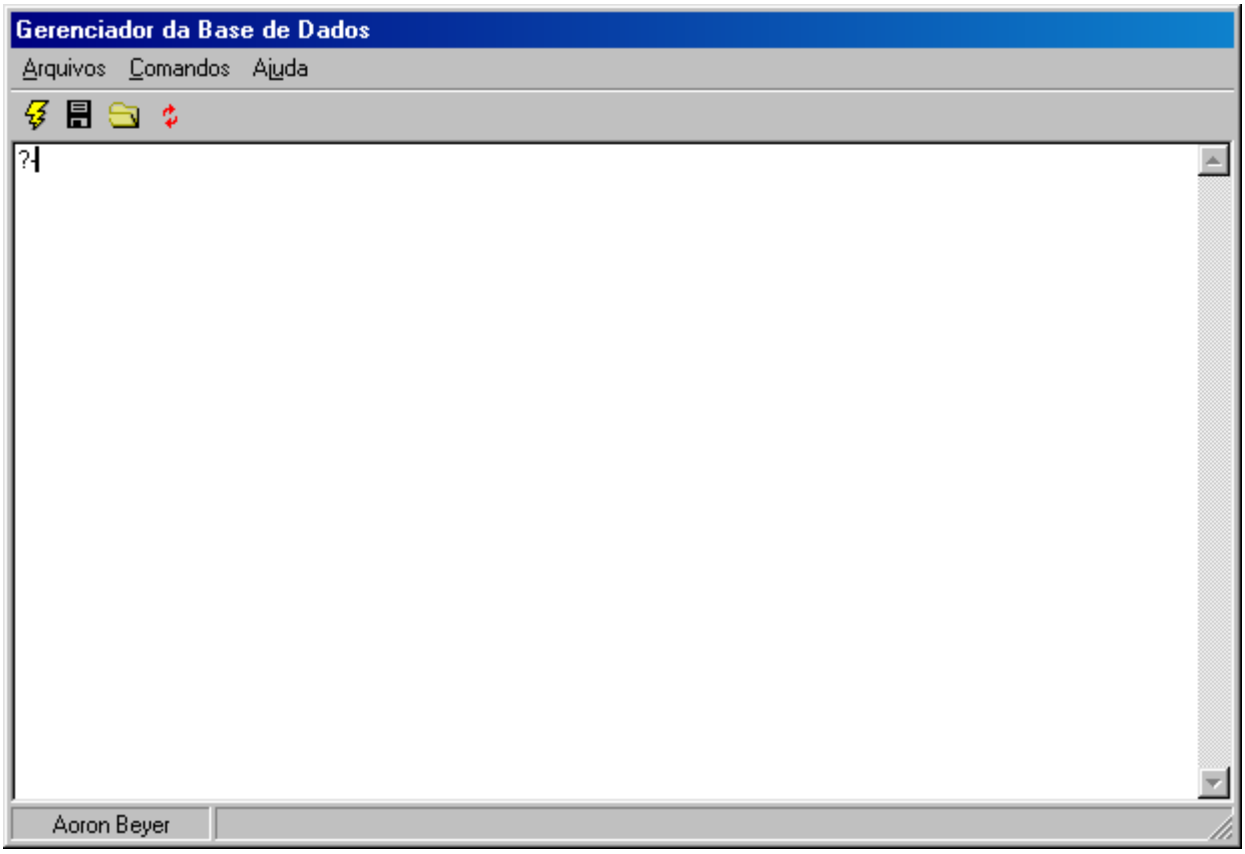


FIGURA 7 - TELA PRINCIPAL DO PROTÖTIPO DESENVOLVIDO

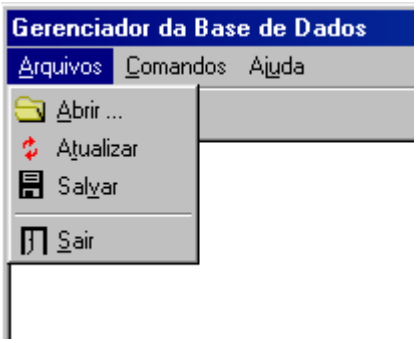


FIGURA 8 - MENU ARQUIVOS

O menu comandos, demonstrado na figura 8, conta com a opção executar. Esta opção também é executada quando se digita Enter no sistema.

O menu ajuda, como mostra a figura 9, contém a opção de sobre, que mostra uma tela com as principais informações sobre o protótipo.

FIGURA 9 - MENU COMANDOS

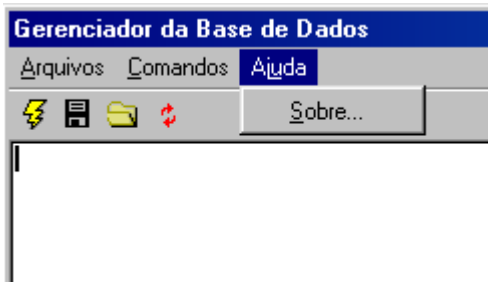


FIGURA 10 - MENU SOBRE

6.5.2 TESTES E RESULTADOS

Inicialmente incluiu-se uma base de dados com o predicados *recorda, recordz, record_after* , que adicionam termos a determinada chave. Na figura 11 mostra-se a entrada dos predicados, sua execução e seus resultados:

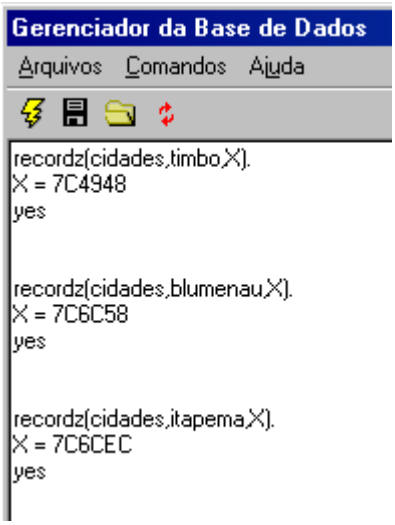


FIGURA 11 - ENTRADA DE DADOS

Após a inclusão destes termos pode-se visualizar os dados armazenados através do predicado recorded, mostrado na figura 12. O estado da estrutura de dados de termos é mostrado no quadro 4. Onde Chave é a chave do Termo, Referência o número de referência do termo e *status* indica se o termo é apagado logicamente.

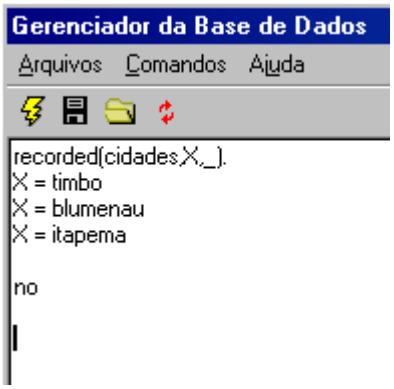


FIGURA 12 - CONSULTA ÀS PRIMEIRAS ENTRADAS

Chave	Termo	Referência	Status
Cidades	Timbo	7D651C	N
	Blumenau	7D6C58	N
	Itapema	7D6CEC	N

QUADRO 04 – MEMÓRIA APÓS A PRIMEIRA ENTRADA

Procede-se com mais algumas inclusões, como mostrado na figura 13.

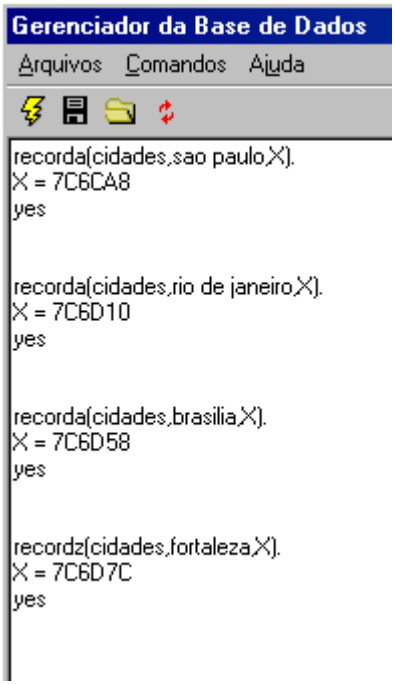


FIGURA 13 - NOVAS ENTRADAS DE DADOS

Após a inclusão destes termos pode-se visualizar os dados armazenados como mostrado na figura 14.

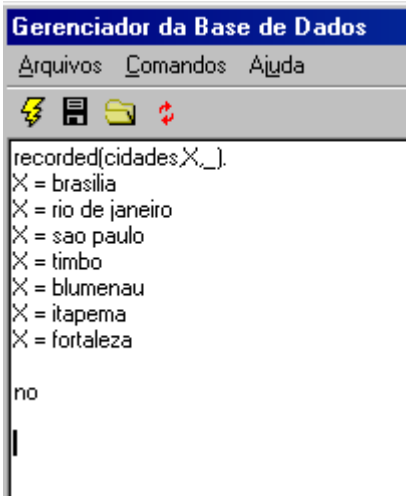


FIGURA 14 - RESULTADOS DA INCLUSÃO

Pode-se observar nos exemplos acima o funcionamento dos predicados *recorda* e *recordz*. Observa-se que o primeiro incluiu os dados ao início da lista e o segundo sempre ao final da lista.

Procede-se com a inclusão de dados, agora usando o predicado *record_after*.

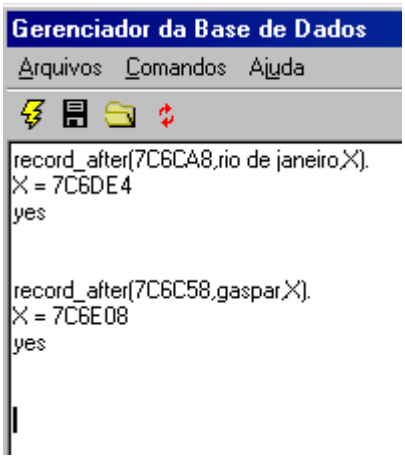


FIGURA 15 - PREDICADO RECORD_AFTER

Após a inclusão destes termos pode-se visualizar os dados armazenados como mostra a figura 16.

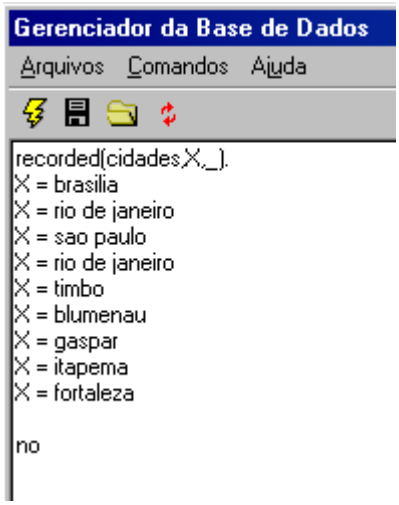


FIGURA 16 - RESULTADOS APÓS INCLUSÕES

O predicado record_after inclui dados no meio de uma lista de termos.

Na figura 16 pode-se observar que a cidade Rio de Janeiro encontra-se duas vezes na tabela, para reverter isso usa-se o predicado *replace* como exemplificado na figura 17.

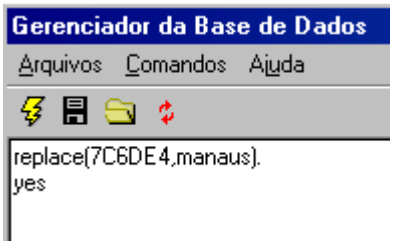


FIGURA 17 - PREDICADO REPLACE

Após a inclusão destes termos pode-se visualizar os dados armazenados como mostrado na figura 18.

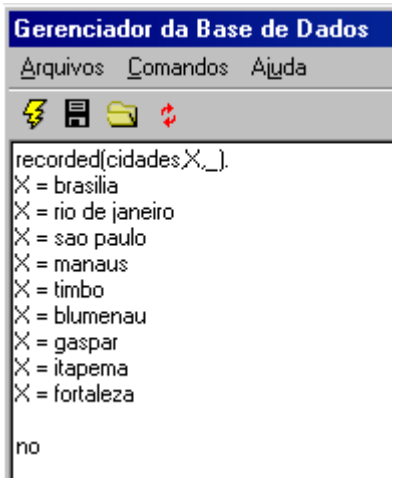


FIGURA 18 - DADOS APÓS USO DO REPLACE

Na figura 19 pode-se observar ainda a inclusão de termos sob outra chave.

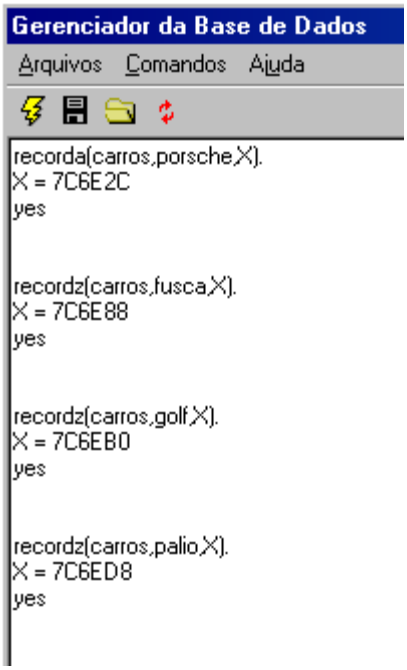


FIGURA 19 - INCLUSÃO DE UMA NOVA CHAVE

Após a inclusão destes termos pode-se visualizar os dados armazenados como mostrado na figura 20.

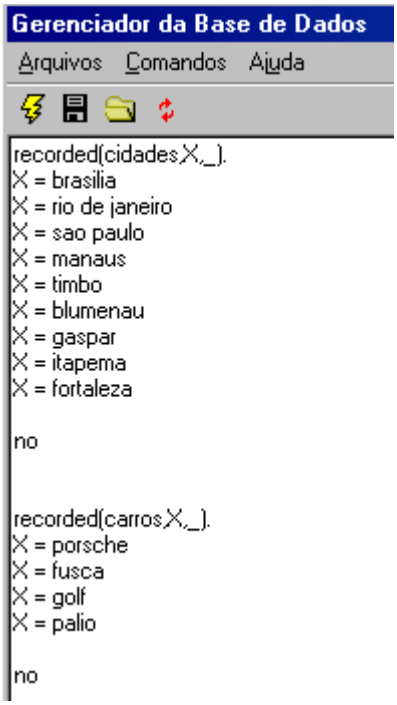


FIGURA 20 - RESULTADOS COM DUAS CHAVES

Outros predicados, como *key, instance, nref e pref* foram testados, pode-se obervar os resultados na figura 21.

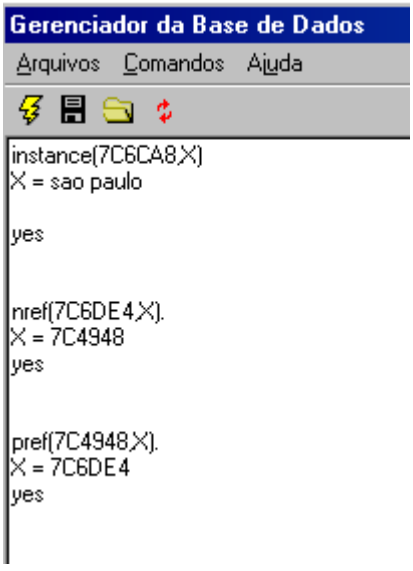


FIGURA 21 - OUTROS PREDICADOS

Testou-se também os predicados *save* e *restore* usados para gravar e restaurar a base de dados, respectivamente. Os resultados são mostrados na figura 22.

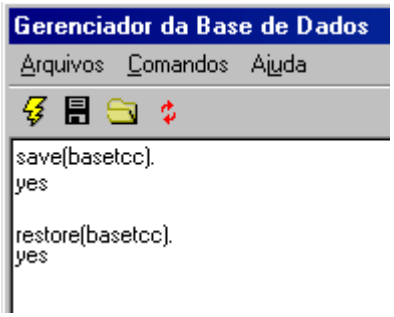


FIGURA 22 - PREDICADOS SAVE E RESTORE

O predicado *save* grava a base de dados em um arquivo texto, cujo formato mostra-se no quadro 05.Onde a informação entre colchetes é a chave, e as informações abaixo os termos e seu *status*.

```
[carros]
porsche,N
fusca,N
golf,N
palio,N
[cidades]
brasilia,N
sao paulo,N
manaus,N
timbo,N
blumenau,N
gaspar,N
itapema,N
fortaleza,N
```

QUADRO 05 – FORMATO DO ARQUIVO GRAVADO EM DISCO

Testou-se ainda os predicados de exclusão. São eles *erase, eraseall, expunge, hard_erase.* Pode observar-se o funcionamento destes predicados nas figuras 23, 25 e 27 e seu resultado nas figuras 24, 26 e 28.

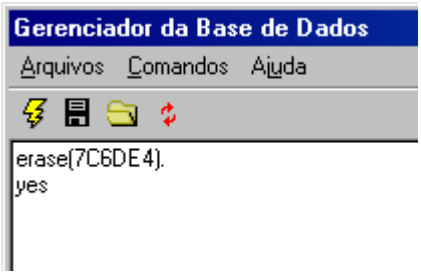


FIGURA 23 - PREDICADO ERASE

Após a exclusão destes termos pode-se visualizar os dados armazenados como mostrado no quadro 06 e figura 24.

Chave	Termo	Referência	Status
Cidades	Brasilia	7D307C	N
	Rio de janeiro	7D3034	N
	São paulo	7D6CA8	N
	Manaus	7D30B8	S
	Timbo	7D651C	N
	Blumenau	7D6C58	N
	Gaspar	7D30A4	N
	Itapema	7D6CEC	N
	Fortaleza	7D307C	N
Carros	Porsche	7D3104	N
	Fusca	7D3160	N
	Golf	7D3188	N
	Palio	7D31B0	N

QUADRO 06 – RESULTADO DA MEMÓRIA APÓS ERASE

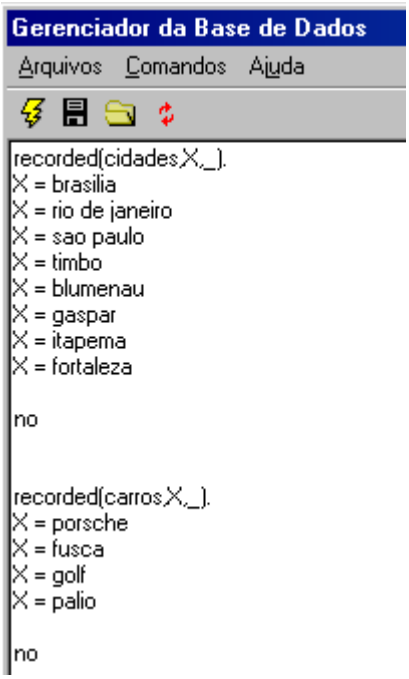


FIGURA 24 - BASE APÓS USO DO ERASE

Na figura 25 pode-se observar o comportamento do predicado *hard_erase* e do predicado *expuge*.

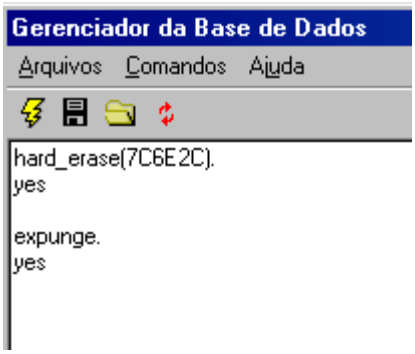


FIGURA 25 - PREDICADOS HARD_ERASE E EXPUNGE

Após a execução destes predicados pode-se visualizar os dados armazenados como mostrado na figura 26.

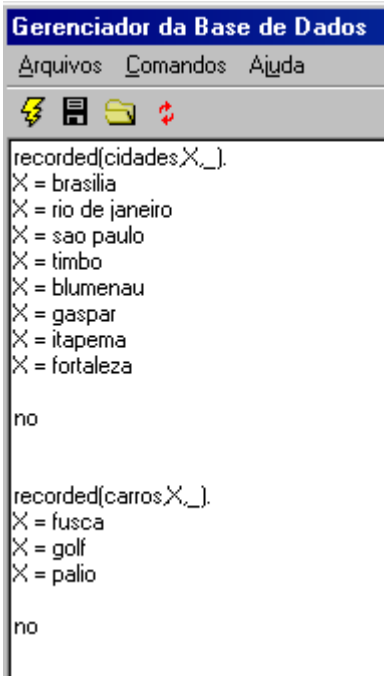


FIGURA 26 - RESULTADO APÓS PREDICADOS DE DELEÇÃO

E finalmente o funcionamento do predicado *eraseall.*

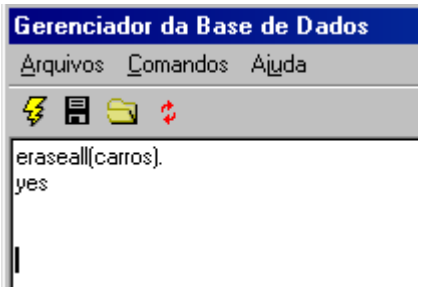


FIGURA 27 - PREDICADO ERASEALL

Após a execução deste predicado pode-se visualizar os dados armazenados como mostrado na figura 28.

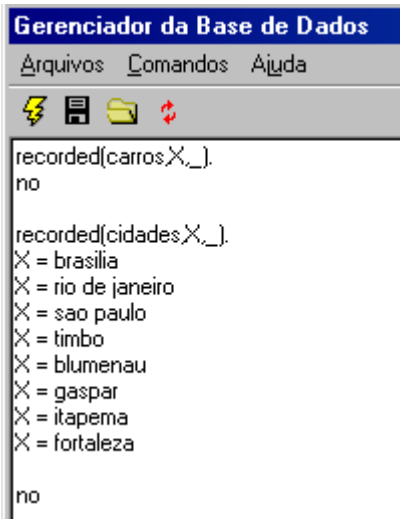


FIGURA 28 - RESULTADO FINAL DA BASE

7 CONCLUSÃO

7.1 CONSIDERAÇÕES FINAIS

O presente trabalho permitiu um estudo dos aspectos relacionados a conceitos de sistemas especialistas. Suas principais características, seus componentes e as ferramentas para construção destes sistemas foram abordados.

Estudou-se programação lógica, mostrando-se suas características, seu histórico e suas principais aplicações. Como seu principal exemplo, a linguagem Prolog foi estudada, passando-se por suas principais características e principais conceitos.

Mostrou-se alguns aspectos da gerência de bases de dados de forma geral e da gerência de base de dados em programação lógica. Abordou-se as estruturas de dados e os principais predicados.

A linguagem Arity/Prolog, usada como base para estudo do Prolog como da gerência de bases de dados forneceu o subsídio necessário para que se pudesse extrair os conhecimentos necessários. E mostrou-se um bom exemplo de gerência de base de dados em programação lógica. Já no ambiente Delphi, utilizada para implementar o protótipo, os algoritmos necessários foram programados sem maiores dificuldades.

A título de limitações do presente trabalho e do protótipo implementado pode-se citar a falta de literatura na área de gerenciamento de dados para programação lógica. O que se observou é que existem poucos estudos nesta área. Os autores que estudam bases de dados e sua interação com a programação lógica, partem geralmente, a utilizar alguma tecnologia de gerência de bases de dados já existente, não criando um novo paradigma para a gerência de dados para este tipo de programação. Outra limitação é o não tratamento sintático e semântico dos predicados no protótipo.

O protótipo desenvolvido tem se mostrado satisfatório, porém, pode ser melhorado com utilização de técnicas de *parsing* no tratamento sintático e semântico. Sua interação com o usuário pode ser melhorada. Devido ao Delphi ser uma linguagem visual, houve algumas dificuldades como a interação com o usuário, que no Arity/Prolog é feita

diretamente na tela. No Delphi houve a necessidade de se usar um componente para obter as respostas do usuário.

7.2 EXTENSÕES

Com a conclusão do presente trabalho, sugere-se:

- Continuar o protótipo implementando os predicados de manipulação de dados em árvores e em tabelas *hash*;
- Implementar os predicados de manipulação de cláusulas, usando lógica dos predicados para resolvê-los;
- Melhorar a implementação do protótipo usando técnicas de *parsing*.

REFERÊNCIAS BIBLIOGRÁFICAS

[ARI88] ARITY Corporation. **The Arity/Prolog Languange Reference Manual.** Massachusetts : Arity Coporation, 1988.

[BEN96] BENDER, Edward A. **Mathematical Methods in Artificial Intelligence.** California : IEEE Computer Society Press, 1996.

[BIR88] BIRD, Richard; WADLER, Philip. **Introduction to Functional Programming.** Great Britain : Prentice Hall, 1988.

[BUD91] BUDDE, Reinhard; KAUTZ, Karlheins; KUHNLENKAMP, Karin; ZÜLLIGHOVEN, Heinz. **Prototyping.** Berlin : Springer-Verlag, 1991.

[BRA90] BRATKO, Ivan. PROLOG – Programming for Artificial Intelligence**.** Englewood Cliffs : Assidson-Wesley Publishers, 1990.

[BRO92] BROUGH, D. R. **Logic Programing - New Frontiers.** Oxford : Kluwer Academic Publishers, 1992.

[CLO94] CLOCKSIN, Wiliam F.; MELLISH, Chistopher S. **Programing in Porlog**. Berlin : Springer-Verlag, 1994. 4 edição

[DAT90] DATE, C. J. **Introdução a Sistemas de Bancos de Dados**. Rio de Janeiro : Editora Campus, 1990.

[GHE91] GHEZZI, Carlo; JAZAYERI, Mehdi. **Conceitos de Linguagens de Programação**. Trad. Paulo A. S. Veloso. Rio de Janeiro : Editora Campus, 1991.

[GRA88] GRAY, Peter M. D.; LUCAS, Robert J. **Prolog and Database Implementation and New Directions**. Chichester : Ellis Horwood, 1988.

[KEL97] KELLER, Robert. **Tecnologia de Sistemas Especialistas: Desenvolvimento e Aplicação.** Trad. Reinaldo Castello. São Paulo : McGraw-Hill, 1991.

[KRU94] KRUSE, Robert Leory. **Data Structures and Program Design.** Halifax : Pretice-Hall International, 1994.

[HAR88] HARMON, Paul; KING, David. **Sistemas Especialistas.** Trad. Antonio Fernandes Carpinteiro. Rio de Janeiro : Campus, 1988.

[HEI95] HEINZLE, Roberto. **Protótipo de uma Ferramenta para Criação de Sistemas Especialistas Baseados em Regras de Produção**. Dissertação de Mestrado. Florianópolis , 1988.

[HOR84] HOROWITZ, Ellis. **Funcamentals od Programming Languages**. California : University of Southern California - Computer Science Press, 1986.

[JAC86] JACKSON, Peter. **Introduction to Expert Systems**. Edinburgh : Addison-Wesley Publishing Company, 1986.

[LUG89] LUGER, George F.; STUBBLEFIELD, William A. **Artificial Inteligence and the Design of Expert Systems**. California : The Benjamin/Cummings Publishing Company, 1989.

[MAI88] MAIER, David; WARREN, David S. **Computing with Logic** – Logic Programming with Prolog. Menlo Park : Benjamin Communigs, 1988.

[MEL90] MELENDEZ, Rubem Filho. **Prototipação de sistemas de informações: fundamentos, técnicas e metodologia**. Rio de Janeiro : Livros Técnicos e Científicos, 1990.

[PRE95] PRESSMAN, Roger S. **Engenharia de Software**. São Paulo : Makron Books, 1995.

[RAB95] RABUSQUE, Renato A. **Inteligência Artificial.** Florianópolis : Editora da UFSC, 1995.

[RIB87] RIBEIRO, Horácio da Cunha e Sousa. **Introdução aos Sistemas Especialistas.** Rio de Janeiro : Livros Técnicos e Científicos Editora, 1987.

[RIC93] RICH, Elaine; KNIGHT, Kevin. **Artificial Intelligence.** São Paulo : Editora McGraw-Hill, 1993.

[PAL97] PALAZZO, Luiz A. M. **Introdução à Programação Prolog.** Pelotas : Editora da Universidade Católica de Pelotas, 1997.

[SET90] SETHI, Ravi. **Programming Languages**. New Jersey : AT&T Bell Laboratories, 1990.

[STE86] STERLING, Leon; SHAPIRO, Ehud. **The Art of Prolog** – Advanced Programming Techniques. Londres : MIT Press, 1986.

[VEL83] VELOSO, Paulo; SANTOS, Clesio dos; AZEREDO, Paulo; FURTADO, Antonio. **Estruturas de Dados**. Rio de Janeiro : Campus, 1983.