UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

PROTÓTIPO DE UM INTERPRETADOR PARA 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

WENDY KREPSKY

BLUMENAU, JULHO/1999

1999/1-59

PROTÓTIPO DE UM INTERPRETADOR PARA UM AMBIENTE DE PROGRAMAÇÃO LÓGICA

WENDY KREPSKY

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. Marcel Hugo

_____

Prof. Dalton Solano dos Reis

À minha esposa Valéria, com todo o meu amor.

AGRADECIMENTOS

Ao professor Roberto Heinzle por sua dedicação e orientação tornando possível e real o desenvolvimento deste trabalho.

Aos meus pais, que sempre estiveram ao meu lado incentivando, apoiando e dando as condições para que eu pudesse estudar.

À minha esposa Valéria por sua compreensão, cooperação e incentivo neste momento de dedicação aos estudos.

Aos meus amigos que me apoiaram durante este período de ausência, em especial ao Josias e ao Aoron.

Ao meu chefe José Milton da Silva por ter cedido tempo para que eu pudesse me dedicar mais à elaboração deste trabalho.

A Deus, que tem sido o meu mestre, por ter me dado forças para vencer esta etapa tão importante de minha vida.

SUMÁRIO

LISTA DE FIGURAS

RESUMO

Este trabalho apresenta um estudo sobre os conceitos e mecanismos de cálculo da Programação Lógica objetivando a construção de um protótipo. O estudo sobre Programação Lógica inicia-se nas áreas de Programação de Computadores e Inteligência Artificial, que são as áreas nas quais ela está inserida. Estende-se na abordagem da Lógica como linguagem de programação e na sua ligação com a Lógica Matemática, mais especificamente com o Cálculo das Proposições e o Cálculo dos Predicados. Demonstra o funcionamento de um ambiente Prolog detalhando os seus mecanismos. Por fim, especifica e implementa um protótipo de interpretador para um Ambiente de Programação Lógica, que visa experimentar na prática o conhecimento adquirido.

ABSTRACT

This work presents a study on the concepts and mechanisms of calculation of the Logical Programming objectifying the construction of a prototype. The study on Logical Programming starts in the areas of Computer programming and Artificial Intelligence, that is the areas in which it is inserted. It extendeds in the boarding of the Logic as programming language and in its linking with the Mathematical Logic, more specifically with the Propositional Calculus and the Predicate Calculus. It demonstrates the functioning of a Prolog environment detailing its mechanism. Finally, it specifies and implements a prototype of interpreter for a Logical Programming Environment, that aims to try in the practical the acquired knowledge.

1 INTRODUÇÃO

Programação lógica é uma área que tem sido muito discutida e disseminada nos dias atuais, mas ela não é nova. A primeira linguagem de programação lógica foi concebida em 1972 por Alain Colmerauer e aperfeiçoada em 1973 por Gerard Battani e Henri Meloni. Desde então tem sido utilizada para aplicações de computação simbólica, como sistemas de base de dados, compreensão de linguagem natural, automação de projetos, análise de estruturas bioquímicas e sistemas especialistas [STE86].

A programação lógica não é baseada na seqüência de procedimentos, mas na definição de relações. A sua linguagem é declarativa e seus elementos essenciais são os objetos e relacionamentos, sobre os quais pode-se declarar vários fatos, definir regras lógicas e fazer questionamentos, onde as respostas podem ser deduzidas a partir do conjunto de fatos e regras. No mundo nada existe sem relação, portanto esta é uma forma mais natural de representá-lo no computador [BRA90].

A principal linguagem de programação lógica é o Prolog, que ganhou destaque com a proposta dos sistemas de "quinta geração", baseada em técnicas da representação da inteligência artificial e do conhecimento [MAI88]. Também tem se falado muito em processamento da linguagem natural, que é uma das funções do Prolog, como a próxima grande inovação da computação. Isto demonstra que a programação lógica estará presente no futuro da computação [NIL97].

1.1 MOTIVAÇÃO

Existem diversas razões para se fazer um estudo sobre programação lógica. Primeiramente, a programação lógica oferece uma maneira diferente de pensar sobre a resolução de problemas. Tem um significado declarativo e processual, de modo que se possa pensar na exatidão de um programa sem pensar no seu comportamento operacional. Em segundo, as linguagens de programação lógica têm uma semântica formal mais "forte" e mais natural do que a maioria das outras linguagens de programação. Em terceiro, as linguagens de programação lógica são linguagens de muito alto nível. São quase linguagens de especificação, ou seja, linguagens que especificam o problema que deve ser resolvido sem abordar os meios da solução. Em quarto, o conhecimento detalhado de como estas linguagens são executadas permitirá que se programe nelas com mais eficiência e eficácia. Em quinto, as

técnicas da execução e as estratégias podem ser aplicadas no aumento da eficiência de outras linguagens de alto nível [MAI88].

Segundo [PAL97], está se assistindo uma completa transformação do paradigma da quarta geração, ora em fase de esgotamento, para arquiteturas inovadoras, contemplando sistemas de processamento paralelo, sendo a concorrência de processos e *layers* baseados em lógica. Também segundo [PAL97], a programação lógica é uma excelente porta de entrada para a informática do futuro, tendo em vista que é de aprendizado mais fácil e natural do que as linguagens procedimentais convencionais; implementa com precisão todos os novos modelos surgidos nos últimos anos, inclusive redes neurais, algoritmos genéticos, sociedades de agentes inteligentes, sistemas concorrentes e paralelos; e libera o programador dos problemas associados ao controle de suas rotinas, permitindo-lhe concentrar-se nos aspectos lógicos da situação a representar.

1.2 OBJETIVOS

O objetivo deste trabalho é elaborar um estudo sobre programação lógica, concluindo com a especificação e implementação de um protótipo de interpretador para um ambiente de programação lógica, utilizando a técnica de prototipação. O interpretador será composto pelos analisadores léxico, sintático, semântico e o motor de inferências. Será utilizado o padrão Edimburgo Prolog para a definição da linguagem do protótipo, descrito em [STE86].

1.3 ORGANIZAÇÃO DO TEXTO

O capítulo 1 faz uma breve introdução ao assunto, apresentando as origens, motivações e objetivos deste trabalho.

O capítulo 2 descreve os aspectos mais básicos da Programação de Computadores, diferenciando a compilação da interpretação e relatando os principais paradigmas. Aborda as diferenças entre as linguagens procedurais imperativas e as linguagens declarativas, mostrando os conceitos, características, vantagens e desvantagens de cada uma. Dentro do paradigma lógico faz uma breve introdução à programação lógica.

O capítulo 3 aborda a Inteligência Artificial e seus conceitos, descrevendo o seu histórico e citando as principais áreas de aplicação. A programação lógica é uma ferramenta

da Inteligência Artificial, portanto, também são abordadas algumas das suas principais contribuições para a Inteligência Artificial, como o Processamento de Linguagem Natural e Sistemas Especialistas.

O capítulo 4 aborda o assunto central do trabalho, a Programação Lógica. Traz um estudo mais aprofundado dos seus conceitos e características. Mostra as suas origens no passado distante, com a lógica de Aristóteles, e um histórico mais recente, descrevendo as principais áreas de aplicação. Busca também a origem matemática da Programação Lógica, abordando a sintaxe, semântica e dedução da Lógica das Proposições e da Lógica dos Predicados.

O capítulo 5 descreve o funcionamento básico de um ambiente Prolog, ilustrando com exemplos. Mostra como escrever um programa Prolog composto por fatos, regras, construções recursivas e como consultá-lo. Também especifica as regras do padrão Edimburgo.

O capítulo 6 trata do desenvolvimento do protótipo, mostrando detalhes de sua implementação, especificação e uso.

O capítulo 7 apresenta as conclusões e considerações finais do trabalho, assim como suas limitações e sugestões para futuros trabalhos.

2 PROGRAMAÇÃO DE COMPUTADORES

O computador tornou-se um instrumento imprescindível ao comércio, à indústria e à pesquisa científica na execução de tarefas que sem ele não poderiam sequer ser consideradas. Mas o que é um computador? As respostas que surgem com mais freqüência são que um computador é um conjunto de circuitos integrados, uma ferramenta ou uma máquina programável. Todas estas respostas são verdadeiras, no entanto, um rádio também é um conjunto de circuitos integrados, um martelo é uma ferramenta e uma máquina de lavar roupa é uma máquina programável. A principal distinção está em que o computador é programável pelo seu utilizador e não tem nenhuma aplicação específica: é uma máquina genérica. Serve para resolver problemas de muitos tipos diferentes, desde o problema de secretaria mais simples, passando pelo apoio à gestão de empresas, até ao controle de autômatos e à concretização de sistemas inteligentes. O computador é uma máquina que executa operações aritméticas e lógicas segundo regras precisamente estabelecidas [KNU73].

Como se programa um computador? Os computadores entendem uma linguagem própria, usualmente conhecida por “linguagem de máquina”. Esta linguagem caracteriza-se por não ter qualquer tipo de ambigüidades: a interpretação das instruções é única. Por outro lado, é caracterizada também por constituir-se de um conjunto de comandos ou instruções que são executados “cegamente” pelo computador: é uma linguagem imperativa. A resolução de qualquer problema é basicamente a construção de uma seqüência de passos, sendo que estes passos apresentam sempre uma seqüência lógica [KNU73].

A linguagem de máquina caracteriza-se também por ser de difícil leitura para o homem, pois todas as instruções correspondem a códigos numéricos e são muito elementares, aproximando o programador das complexidades da própria máquina. Para que os computadores pudessem ser manejados por pessoas sem conhecimento especializado foram criadas linguagens de programação, que escondem as complexidades da máquina e tornam a sua manipulação mais simples.

É falsa, portanto, a imagem do computador como uma máquina fantástica, manejada por pessoas extraordinárias. O computador não passa de uma máquina que obedece cegamente às instruções dadas pelo homem em uma linguagem semelhante à humana, sendo isto sem dúvida o que ele tem de mais extraordinário [KNU73]. Mas permanece o problema

de como converter um programa escrito numa linguagem de programação em um programa que o computador possa executar.

2.1 COMPILAÇÃO E INTERPRETAÇÃO

Embora seja teoricamente possível a construção de computadores especiais, capazes de executar programas escritos em uma linguagem de programação qualquer, os computadores existentes hoje em dia são capazes de executar somente programas em uma linguagem de baixo nível, a linguagem de máquina. Enquanto as linguagens de máquina são projetadas em função da rapidez de execução de programas, do custo de sua implementação e da flexibilidade com que permitem a construção de programas de nível mais alto, as linguagens de programação são projetadas em função da facilidade na construção e da confiabilidade dos programas [GHE91]. O problema básico, então, é como executar um programa escrito em uma linguagem diferente da linguagem da máquina. Existem basicamente duas alternativas: compilação e interpretação.

O compilador é um programa cuja função é ler um programa escrito em uma linguagem, traduzi-lo para um programa equivalente em outra linguagem e reportar ao seu usuário os erros encontrados no programa origem [AHO87]. Na compilação, programas escritos em linguagem de alto nível são traduzidos para versões equivalentes em linguagem de máquina, antes de serem executados, sendo que esta tradução pode ser feita em várias etapas [GHE91].

Na interpretação, as ações indicadas pelos comandos da linguagem são diretamente executadas. Em geral, existe um subprograma escrito em linguagem de máquina para executar cada ação possível. Assim, a interpretação de um programa é feita pela chamada daqueles subprogramas em uma seqüência apropriada [GHE91]. Ao invés de produzir um programa novo através de uma tradução, um interpretador executa as operações uma a uma. Os interpretadores são usados freqüentemente para executar linhas de comando. Da mesma forma, algumas linguagens de alto nível são interpretadas por possuírem muitos dados que não podem ser deduzidos no momento da compilação [AHO87].

A interpretação pura e a compilação (tradução) pura são dois extremos. Na prática, muitas linguagens são implementadas por uma combinação destas técnicas. Um programa

pode ser traduzido para um código intermediário, que é então interpretado [GHE91]. Por exemplo, o código intermediário poderia ser o código de máquina de uma máquina virtual, que seria depois interpretada por programas.

Em uma solução puramente interpretativa, a execução de um comando pode requerer um processo de decodificação bastante complicado para determinar as operações a serem executadas e seus operandos. Na maioria dos casos, a mesma decodificação é executada cada vez que o comando é encontrado, afetando sensivelmente a rapidez de execução. Por outro lado, na compilação (tradução) pura, o código de máquina é gerado para cada comando de alto nível. Neste caso o compilador decodifica cada comando somente uma vez. Os componentes usados com freqüência são então decodificados, na sua representação em linguagem de máquina, várias vezes. Como isto é feito eficientemente por circuitos internos a compilação pura pode economizar tempo de execução em comparação à interpretação pura. Por outro lado, é possível que a interpretação pura economize memória. Cada uma das soluções possui suas vantagens e desvantagens [GHE91].

2.2 PRINCIPAIS PARADIGMAS

A maioria das linguagens de programação existentes hoje em dia formam uma mesma linha de idéias, cuja característica essencial é a arquitetura da máquina de Von Neumann. Esta arquitetura tem formado a base para o projeto da maioria das linguagens de programação, porém existe uma base alternativa, que dá origem ao grupo das linguagens declarativas. Estas linguagens podem ser subdivididas em funcionais e lógicas.

Existem alguns outros paradigmas, como é o caso da programação concorrente e da programação orientada por objetos, que não serão abordados porque, segundo [WIL93], podem ser consideradas como subparadigmas da programação imperativa.

2.2.1 LINGUAGENS PROCEDURAIS IMPERATIVAS

O que caracteriza esta linha de idéias, ditando suas características essenciais, é a arquitetura de máquina Von Neumann [GHE91]. A maioria das linguagens de programação existentes hoje podem ser consideradas como abstrações construídas em cima desta arquitetura. O paradigma da programação imperativa é o mais antigo, mas é também o dominante [WIL93].

Esta classe de linguagens são assim chamadas por serem baseadas em comandos ou instruções imperativas, que atualizam o conteúdo de variáveis armazenadas na memória. A unidade de trabalho em um programa escrito nessas linguagens é o "comando". Os efeitos de comandos individuais são combinados para a obtenção dos resultados desejados em um programa [GHE91] [WIL93].

A arquitetura dos computadores tem desempenhado um profundo papel em dar forma às linguagens imperativas. Esta influência pode ser vista, conforme [GHE91], em três características bem difundidas das linguagens imperativas:

a) variáveis: um componente fundamental da arquitetura é a memória, que é composta de um grande número de células. É na memória que os dados são armazenados. As células devem ter nomes, de modo que se possa ter sob controle a localização da informação. Uma variável em uma linguagem de programação é essencialmente uma célula de memória, onde se armazenam valores com um nome;
b) operação de atribuição: fortemente ligada à arquitetura da memória existe a noção de que cada valor computado deve ser "armazenado", isto é, atribuído a uma célula. Isto explica a importância da instrução de atribuição nas linguagens de programação. As noções de baixo nível de célula de memória e de atribuição permeiam todas as linguagens de programação, forçando ao programador um estilo de pensamento que é moldado pelos detalhes da arquitetura Von Neumann;
c) 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.

A essência da programação em linguagens imperativas é a computação, passo a passo e repetida, de valores de baixo nível e atribuição destes valores a posição de memória. Isto claramente não é o nível de detalhe com o qual alguém deseja se envolver quando programa uma aplicação complicada. De fato, as linguagens de programação têm tentado cada vez mais esconder essa natureza de baixo nível da máquina [GHE91].

Existe um mecanismo de linguagem que abstrai detalhes do nível de máquina que é o procedimento (*procedure*). O procedimento localiza uma boa parte do raciocínio que precisa

ser feito sobre o comportamento de variáveis, permitindo a construção de componentes independentes. Existem também conceitos que visam obter maior eficiência de execução, como passagem de parâmetro por referência e variáveis globais, que tornam todas as posições de memória endereçáveis e revelam assim a memória baseada em células [GHE91].

Segundo [GHE91], provavelmente o problema mais sério com as linguagens imperativas decorre da dificuldade de se raciocinar sobre a correção de programas. Esta dificuldade é causada pelo fato de a correção de um programa, em geral, depender dos conteúdos de cada célula particular. O estado da computação é determinado pelos conteúdos de cada célula da memória. Para entender uma rotina é preciso executá-la mentalmente. Para entender como a computação progride ao longo do tempo é necessário tomar "estados" da memória em cada passo, após cada instrução. Isto é uma tarefa cansativa quando o programa envolve uma quantidade grande de memória. As regras de escopo das linguagens limitam um pouco este problema reduzindo o número de células acessíveis. O uso de variáveis globais pode tornar os programas difíceis de analisar.

2.2.2 LINGUAGENS DECLARATIVAS

A supremacia das linguagens imperativas está sendo desafiada por linguagens declarativas, embora as linguagens funcionais e lógicas estejam em desvantagem considerável quando são usadas nas atuais arquiteturas de computadores. As principais linguagens declarativas são LISP e PROLOG, do subgrupo das linguagens funcionais e das linguagens lógicas respectivamente.

2.2.2.1 PARADIGMA FUNCIONAL

Enquanto o estilo de programação em uma linguagem orientada para comandos consiste em organizar a execução e repetição de certas seqüências de instruções individuais usando estruturas de controle apropriadas, a essência da programação funcional é combinar funções para produzir outras mais poderosas [GHE91].

Abstrações de dados e funções são idéias diferentes, porém relacionadas. Linguagens funcionais, como o LISP, enfatizam o processo de identificar blocos e partes repetidas de código e constróem funções que encapsulam a funcionalidade dentro de uma simples definição. Isto facilita a manutenção, sendo que a definição de uma rotina é feita uma única

vez no programa, e aumenta a confiabilidade, pois não ocorrem erros de tipagem no caso de outras definições da mesma rotina [HIN99]. As linguagens funcionais levam essa idéia ao extremo, já que sua sintaxe encoraja uma visão totalmente modular do fluxo do programa. Elas reutilizam o código previamente escrito (funções) para construir programas cada vez mais complexos e geralmente não há um identificador para indicar o ponto onde a execução do programa inicia.

O paradigma de programação funcional enxerga todos os subprogramas como funções, eles recebem argumentos e retornam soluções simples. A solução retornada é baseada inteiramente na entrada e o tempo em que uma função é chamada é irrelevante. O programa passa a ser uma lista (no caso do LISP) de funções, sendo o próprio programa principal uma lista. O relacionamento entre as funções é simples: uma função pode chamar outra, ou o resultado de uma função pode ser usado como argumento para outra função. Não há variáveis e comandos, todo o programa é escrito com a linguagem das expressões, funções e declarações [WIL93].

A primeira regra da programação é construir uma função para resolver um problema. Esta função envolve um número de funções subsidiárias que é expressada em uma notação que obedece os princípios normais da matemática. A função do computador é agir como avaliador ou calculador. O resultado é uma estrutura conceitual para programação que é muito simples, muito conciso, muito flexível e muito poderoso [HIN99].

Então, programação em linguagem funcional consiste na construção de definições e o uso do computador para avaliar expressões. Segundo [HIN99], suas principais vantagens são:

a) um alto nível de abstração, especialmente quando as funções são utilizadas, suprimindo muitos detalhes da programação e minimizando a probabilidade da ocorrência de muitas classes de erros;
b) a não dependência das operações de atribuição permite aos programas avaliações nas mais diferentes ordens. Esta característica de avaliação independente da ordem torna as linguagens funcionais as mais indicadas para a programação de computadores maciçamente paralelos;
c) a ausência de operações de atribuição torna os programas funcionais mais simples para provas e análises matemáticas do que os programas procedurais.

A principal desvantagem é a menor eficiência para resolver problemas que envolvam muitas variáveis ou muitas atividades seqüenciais, que são muitas vezes mais fáceis de se trabalhar com programas procedurais ou programas orientados a objeto [HIN99].

2.2.2.2 PARADIGMA LÓGICO

Um programa em lógica é a resolução de um determinado problema através da utilização de sentenças da lógica. É o uso do mecanismo de inferências para obter soluções para problemas declarativos. Em outras palavras, poderia se dizer que é o uso da lógica simbólica como linguagem de programação [BRA90]. Em linguagens imperativas há uma seqüência de instruções executadas uma após a outra, que ao final convergem um resultado. Já um programa em lógica assemelha-se mais a um banco de dados, onde várias informações e relações são registradas, do tipo "o homem é inteligente" ou "X é inteligente se X é homem." A essência da programação lógica são os objetos e seus relacionamentos. Pode-se declarar vários fatos sobre os objetos e relacionamentos, definir regras lógicas e fazer questionamentos, onde as respostas podem ser deduzidas a partir dos fatos e regras.

As linguagens de programação lógica são linguagens de "alto-nível" porque o seu foco está na lógica computacional e não nos seus mecanismos, que estão inclusive inacessíveis ao programador. O resultado é que elas são melhores para expressar idéias complexas porque o trabalho de gerenciamento de memória, pilha de ponteiros e outros, são deixados para o motor computacional. As linguagens de programação lógica são eficientes para rápida prototipação de estruturas de dados e códigos para expressar idéias complexas [BRA90].

A programação em lógica só ganhou ímpeto a partir dos anos 80, quando foi escolhida como a linguagem básica do projeto japonês para o desenvolvimento dos denominados computadores de quinta geração. O seu principal representante é o PROLOG, que é uma abordagem particular mas já é conhecido como sinônimo de programação lógica [WAT90]. A programação lógica, suas características e seus fundamentos são o centro deste trabalho e por isso serão abordados com mais detalhes no capítulo 4.

3 INTELIGÊNCIA ARTIFICIAL

Inteligência Artificial (IA) é um termo que tem sido muito utilizado, porém as vezes de uma forma equivocada devido a distorções da ficção científica, o que tem gerado uma grande expectativa. Falar em IA, entretanto, é sempre polêmico e requer certos cuidados, pois existem muitas controvérsias que envolvem o assunto. A questão "Um máquina pode pensar?" tem interessado filósofos assim como cientistas e engenheiros. Este capítulo procura trazer uma visão do que é a Inteligência Artificial, mostrar de onde ela surgiu e quais são suas principais áreas de aplicação.

3.1 O QUE É

Para entender o que é a IA é necessário definir o que é inteligência, porém esta definição não é simples e geralmente expressa apenas as suas características e sintomas. Segundo Arnold e Bowie *apud* [RAB95]: "Inteligência não é algo que pode ser perfeitamente capturado em uma frase cuidadosamente arrumada. Não é algo que pode ser dissecado em suas partes constituintes, mesmo que desmontemos o cérebro, neurônio por neurônio, e examinemos todas as conexões sinápticas..."

A inteligência abrange o raciocínio, aprendizado, memória, motivação, capacidade de se adaptar e de resolver problemas, sendo que todas estas partes trabalham harmoniosamente para a obtenção de resultados, num processo dinâmico [RAB95]. De forma sintetizada, pode-se dizer que a inteligência é a capacidade de adquirir e manipular o conhecimento para obtenção de resultados.

A inteligência não é totalmente compreendida e entendê-la profundamente seria um grande feito da ciência. James Albus escreveu em resposta a Henry Hexmoor em [ALB95]: "Eu acredito que entender a inteligência envolve entender como o conhecimento é adquirido, representado e armazenado; como o comportamento inteligente é gerado e aprendido; como motivos, emoções e prioridades são desenvolvidos e usados; como sinais sensoriais são transformados em símbolos; como os símbolos são manipulados para realizar lógica, o raciocínio a respeito do passado, e o planejamento para o futuro; e como o mecanismo da inteligência produz o fenômeno da ilusão, crença, esperança, medo, e sonhos – e até mesmo

bondade e amor. Eu acredito que entender estas funções no nível fundamental seria uma proeza científica do nível da física nuclear, da relatividade, e da genética molecular."

O foco da IA é o conhecimento sobre o conhecimento, ou seja, de como ele é adquirido, representado, armazenado e utilizado. Se este conhecimento pode ser formalizado, pode também ser automatizado. Pode-se dizer que a grande atividade da IA é a solução de problemas, usando conhecimento e manipulando-o. Existem inúmeros conceitos, mas nenhum deles é simples e claro. Stuart Russel e Eric Wefald *apud* [BEN96] disseram que cada pessoa que abordar a IA deve decidir o que ela é.

Nils Nilsson conceituou a IA em [NIL97]: "A Inteligência Artificial envolve o comportamento inteligente em artefatos. O comportamento inteligente, por sua vez, envolve percepção, raciocínio, aprendizado, comunicação e ação num ambiente complexo. A IA tem como meta o desenvolvimento de máquinas que podem fazer estas coisas tão bem ou até mesmo melhor que o homem. Outra meta da IA é entender este tipo de comportamento quer ele ocorra em máquinas, em homens ou em outro animal." De acordo com este conceito, a IA é a simulação da inteligência em máquinas.

Segundo [TAF96], que reforça este conceito de forma mais simples: "A inteligência Artificial constitui-se em um conjunto de técnicas de programação para resolver determinados tipos de problemas em informática. Ela procura imitar, através dos programas que comandam estas máquinas, as formas de resolução de problemas do mesmo modo que o homem o faz. É claro que os tais programas são, essencialmente, algoritmos."

Existe uma certa atração dos seres humanos pelo estudo da IA. Quem nunca se encantou com os robôs inteligentes dos filmes de ficção científica? Estudar IA é entender um pouco melhor o próprio ser humano, é estudar as faculdades mentais com o uso de modelos computacionais. As metas da IA são características um tanto gerais na natureza. Seguem algumas metas citadas em [BEN96] que demonstram isto:

a) raciocínio: tendo algum conhecimento geral com alguns fatos específicos, deduzir certas conseqüências. Por exemplo, dado algum conhecimento sobre doenças e sintomas, pode se chegar a um diagnóstico particular com base nos sintomas percebidos. O tipo mais difícil de raciocínio é aquele baseado no que se chama de "senso comum";

b) planejamento: tendo algum conhecimento, a situação atual e um objetivo definido, decidir como alcançá-lo, ou seja, usar o objetivo para direcionar o raciocínio. Enquanto o raciocínio responde “Porque”, o planejamento responde “Como”. Por exemplo, “Que tipo de estudante eu sou?” contra “Como eu posso me tornar um estudante A?”;
c) aprendizagem: adquirir o conhecimento (aprender) é a questão central, porque para poder usá-lo é necessário primeiro adquiri-lo. Em algumas situações, é viável inserir o conhecimento no sistema, em outros, é inviável ou indesejável. Portanto, é necessário que o sistema possa incrementar a sua base de conhecimento de modo coerente, adquirindo fatos novos e integrando-os com o conhecimento já existente, freqüentemente por algum processo de abstração. Por exemplo, se um sistema é exposto a vários exemplos de cadeiras, como ele pode abstrair o conceito “cadeira”?

Existem outras metas, citadas por [BEN96], que podem ser consideradas mais específicas e são tratadas em partes separadas da IA, com suas próprias ferramentas e problemas:

a) compreensão e uso da linguagem: este item tem relação com o raciocínio e aprendizagem, porém merece uma categoria separada. O “senso comum” é muito empregado na linguagem. Infelizmente, o senso comum é um tópico indescritível, que requer uma base de dados considerável. As tentativas de compreender a linguagem falada possuem complicações adicionais;
b) processamento visual: a visão é um dos tipos de entrada sensoriais que precisam ser processados, mas é de longe o mais complexo. Abstrair informações úteis de uma entrada visual é comprovadamente muito difícil;
c) robótica: a robótica precisa casar a IA com a engenharia. A realidade em qualquer cenário industrial é demasiadamente complexo. As técnicas de IA usadas na robótica precisam produzir resultados em tempo real e robôs autônomos não podem requerer um excessivo recursos em computadores.

3.2 HISTÓRICO

Quando os computadores começaram a ser desenvolvidos em 1940 e 1950, muitos pesquisadores escreveram programas que podiam realizar tarefas elementares de raciocínio. Proeminentes entre estes são os estudos que descrevem os primeiros programas que poderiam jogar xadrez (Newell, Shaw e Simon, 1958), jogo de damas (Samuel, 1963 e 1967) e comprovar teoremas no plano da geometria (Gelernter, 1959). Em 1956, John McCarthy e Claude Shannon co-editaram um volume entitulado Estudo de Autômatos (*Automata Studies*). Desapontado por os estudos serem tratados como teorias matemáticas sobre autômatos, John McCarthy decidiu usar a frase Inteligência Artificial como título na conferência de Dartmouth, em 1956, dando origem ao termo [BAR97] [NIL97].

Porém as origens da Inteligência Artificial são muito remotas e o primeiro passo foi dado por Aristóteles (384 a 322 a.C.), quando descreveu o raciocínio dedutivo, que chamou de silogismo, e que tem sua base na linguagem natural. Gottfried Leibniz sonhou com uma álgebra universal, pela qual todo o conhecimento, incluindo as verdades morais e metafísicas, poderia algum dia ser trazido em um único sistema dedutivo. Chamou este seu sistema de cálculo filosófico ou cálculo do raciocínio, dando origem assim a lógica simbólica [NIL97] [RAB95].

Um progresso substancial ocorreu quando George Boole desenvolveu as fundamentações da lógica proposicional. A finalidade de Boole era, entre outras coisas, "coletar... algumas possibilidades prováveis a respeito da natureza e da constituição da mente humana." Para o fim do décimo nono século, Gottlieb Frege propôs um sistema notacional para o raciocínio mecânico e assim inventou muito do que é conhecido hoje como cálculo dos predicados. Chamou sua linguagem "*Begriffsschrift*", que pode ser traduzida como "a escrita de conceito" [NIL97] [BAR97].

Por volta de 1972, em Marseille, Alain Colmerauer implementou o *System Q*, que foi por alguém denominado de PROLOG. Kowalski trabalhou a parte lógica junto com Colmerauer, visando a implementação computacional. A cruzada empreendida por estes dois pesquisadores trouxe muitos adeptos ao uso do PROLOG, e consequentemente, apareceram dezenas de implementações de PROLOG [RAB95].

A entrada dos microcomputadores foi um marco que direcionou de forma irreversível a computação. O crescimento da potência das máquinas está viabilizando a abordagem de problemas "infinitos", assim como a retomada do estudo e simulação de redes neurais. Na década de oitenta, os japoneses lançaram seu projeto de computador de quinta geração, gerando grande excitação na comunidade científica. Se este projeto não produziu os resultados espetaculares que dele se esperava, teve, pelo menos, o mérito de chamar a atenção para o novo rumo tomado pela computação [NIL97] [RAB95].

Atualmente, algumas áreas de aplicação da IA já estão em uso, como é o caso dos sistemas especialista. Outras áreas, onde estão sendo feitos estudos aprofundados, começam a apresentar resultados animadores, como é o caso da robótica e do processamento de linguagem natural. Mas os estudos continuam e devem haver progressos nos próximos anos [NIL97] [RAB95].

3.3 ÁREAS DE APLICAÇÃO

Atualmente, a Inteligência Artificial possui inúmeras áreas de aplicação e tem ocupado espaço da ciência ao lazer. Todas são importantes para a computação como um todo, mas algumas delas se destacam pela quantidade de esforço despendido com o seu desenvolvimento. Assim, destacam-se: Processamento de Linguagem Natural, Reconhecimento de Padrões, Robótica e Sistemas Especialistas.

3.3.1 PROCESSAMENTO DE LINGUAGEM NATURAL (PLN)

Segundo [BAR97], "processamento de linguagem compreende a compreensão de uma seqüência de símbolos e a geração de outra seqüência, não necessariamente no mesmo idioma, ou seja, não necessariamente usando os mesmos símbolos. Assim inclui a criação de uma estrutura mental conseqüência de uma seqüência de palavras, manipulação desta estrutura mental em função do estado emocional e mental, para provocar outra seqüência de palavras."

O PLN não é visto como uma "aplicação". Ele é considerado um ramo de pesquisa dentro da IA, que por sua vez pode ser dividido em duas sub-áreas de trabalho:

a) interpretação de linguagem natural: baseia-se em mecanismos que tentam "compreender" uma sentença em alguma linguagem natural, buscando traduzi-la

para uma representação que possa ser compreendida e utilizada pelo computador [GAL97]. Por exemplo: a Lógica de Predicados. Tal tradução, geralmente se dá em duas etapas:

- processamento sintático (*parsing*), para a obtenção da estrutura sintática da sentença sob análise;
- interpretação semântica, significando a semântica da sentença.

b) na geração de linguagem natural o oposto se verifica: o computador traduz uma representação interna do significado da sentença para a sua realização em alguma língua natural. Esta sub-área busca produzir textos o mais próximo possível de textos que teriam sido produzidos por pessoas [GAL97]. Uma aplicação seria a produção de textos técnicos (relatórios) a partir de dados armazenados em um BD, por exemplo.

Uma aplicação que une técnicas das duas áreas é, por exemplo, a tradução automática de línguas. Neste caso, uma sentença em uma língua natural (e.g., Português) é traduzida para uma representação interna – interpretação, que em seguida é traduzida para uma sentença em outra língua natural (e.g., Inglês) – geração. Acreditava-se que seria suficiente dispor de um potente dicionário e regras gramaticais. Hoje acredita-se ser este um problema muito mais complexo e um dos grandes problemas a serem resolvidos plenamente [BAR97].

A PLN é, sem dúvida, um dos grandes desafios da IA. A linguagem escrita está dominada em certos aspectos, persistindo, contudo, problemas sérios no tocante à linguagem figurada, dupla interpretação e outros. A linguagem falada já está sendo produzida de forma razoável, havendo, no entanto, um problema muito sério quanto à forma humana de falar, emendando as palavras umas às outras [RAB95].

3.3.2 RECONHECIMENTO DE PADRÕES

Reconhecer o dono de uma impressão digital, validar a assinatura num cheque bancário, ler e digitalizar um texto escrito, reconhecer uma palavra falada ou uma fotografia de pessoa são tarefas que envolvem reconhecimento de padrões [BAR97] [RAB95].

Em termos gerais, o reconhecimento de padrões é a ciência que compreende a identificação ou classificação de medidas de informação em categorias. Categorias têm a

característica de representar entidades ou padrões de informação que apresentam similaridades. O reconhecimento de padrões é composto de um conjunto de técnicas e abordagens que são usadas de forma integrada na solução de diversos problemas práticos. Entre as abordagens que podem ser empregadas na solução de problemas pode-se destacar o Reconhecimento de Padrões Estatístico, o Reconhecimento de Padrões Sintático e Redes Neurais Artificiais [VAS99].

O ser humano possui capacidades únicas de reconhecer padrões. Toda a aprendizagem está relacionada com a capacidade do cérebro de identificar, isolar, associar e reconhecer formas, sons ou conceitos. Este processo é de tal forma complexo, que tem apaixonado cientistas e conduziu a tentativas de exploração do seu mecanismo e ao desenvolvimento de metodologias matemáticas como as redes neurais ou a inteligência artificial. De fato a combinação da visão, do processo de memorização e reconhecimento confere aos seres humanos habilidades inultrapassáveis pelo computador mais sofisticado.

3.3.3 ROBÓTICA

Um robô é um dispositivo eletro-mecânico que pode ser programado para realizar tarefas manuais. A diferença entre máquinas automáticas e um robô inteligente é que o robô tem percepção sobre o ambiente e modifica seu comportamento como resultado da informação obtida. Um robô inteligente deve incluir equipamentos sensoriais, tais como uma câmara, que coleta informação sobre as operações do robô e seu ambiente. A parte "inteligente" do robô permite que ele responda e se adapte a mudanças ocorridas no seu ambiente, não se limitando a seguir instruções cegamente [GAL97].

Os robôs "inteligentes" complementam a parte mecânica com dispositivos eletrônicos de suporte, constituindo uma espécie de cérebro, onde são armazenados conhecimentos, os quais podem dar certo grau de autonomia a estes engenhos. Este tipo de robô tem sido usado, em geral, nas tarefas executadas em ambientes hostis ao humano, como é o caso de viagens espaciais, atividades de prospecção de petróleo no fundos dos oceanos, mas também existem robôs que jogam xadrez e que auxiliam no serviço doméstico. A parte de armazenamento de conhecimento e sua execução é semelhante à efetuada em sistemas especialistas. Os robôs são um apelo da mídia para motivar a IA [RAB95].

3.3.4 SISTEMAS ESPECIALISTAS

Sistemas Especialistas (SE) são programas de "aconselhamento" que, com base em conhecimento armazenado e de mecanismos de inferência, busca reproduzir a forma de raciocínio de especialistas humanos na resolução de problemas específicos. Os SEs são desenvolvidos para "domínios" delimitados de acordo com a aplicação [GAL97].

Segundo [HEI95], "os sistemas especialistas são sistemas computacionais projetados e desenvolvidos para solucionarem problemas que normalmente exigem especialistas humanos com conhecimento na área de domínio da aplicação. Tal como um especialista, o sistema deve ser capaz de emitir decisões justificadas acerca de um determinado assunto a partir de uma substancial base de conhecimentos. Para tomar uma decisão o especialista busca em sua memória conhecimentos prévios, formula hipóteses, verifica os fatos que encontra e compara-os com as informações já conhecidas e então emite a decisão. Neste processo o especialista realimenta a sua 'base de conhecimentos' acerca do assunto."

Portanto, os sistemas especialistas são formados por um profundo conhecimento acerca de uma área específica e um mecanismo de raciocínio ou inferência que possibilite responder aos questionamentos, justificar suas conclusões e ainda ter capacidade para adquirir novos conhecimentos [HEI95].

SEs são de grande interesse para empresas, pela capacidade que têm de aumentar a produtividade e ampliar as forças de trabalho em áreas onde especialistas humanos são difíceis de se encontrar. Esta é a aplicação de IA mais difundida comercialmente [GAL97].

Apesar do sucesso dos SEs nas empresas, tais sistemas também são comumente encontrados em outras áreas de atividade, como por exemplo, na medicina. Um dos primeiros SEs desenvolvidos, o sistema MYCIN, foi implementado na Escola de Medicina da Universidade de Stanford em 1970 (Dr. E. H. Shortliffe). Seu objetivo era auxiliar médicos no diagnóstico de doenças infecciosas no sangue e na prescrição de um tratamento adequado a cada paciente específico [BAR97] [GAL97] [RAB95].

Os sistemas especialistas têm, também, aplicações em outras áreas de conhecimento. Há sistemas no campo de prospeção mineral; na matemática, onde se podem provar teoremas ou efetuar derivação e integração formal; na química, onde o estudo de novos compostos

químicos pode ser facilitado; no ensino em geral, podendo o aluno aprofundar seus conhecimentos, abordando sistemas de sua área específica [RAB95].

4 PROGRAMAÇÃO LÓGICA

Para entender a programação lógica é necessário ter uma noção do que há por trás de sua concepção e da sua ligação com a psicologia cognitiva. A primeira concepção pode ser entendida por uma pergunta: o que se relaciona com o quê? Ou, entre que coisas do mundo existe relação? É perceptível que entre todas as coisas existem relações e não se pode conceber a existência de um objeto no mundo que não tenha relação no tempo e no espaço. Praticamente não há limites para uma visão que busca o relacionamento entre os objetos.

Essas relações podem ser de posição física, de posse, de convívio social, de sentimentos, etc. Por exemplo, um vegetal relaciona-se com o seu meio ambiente: com a terra, com o ar e com a luz solar, e sem esta relação o vegetal não existe. Outro exemplo: dois irmãos existem por terem pais em comum, sendo que sem a relação de paternidade a definição de irmãos não existiria. A importância de considerar-se a relação como definidora dos elementos no mundo é que a torna de extrema utilidade na modelagem de representações simbólicas a nível computacional. Isto simplifica a modelagem de elementos que são aparentemente complexos por causa de suas relações, como por exemplo a linguagem natural falada pelo homem [CLO94].

A segunda concepção é a classificação e hierarquização, que são elementos fundamentais para a evolução cognitiva do homem permitindo o agrupamento para discernimento e organização dos elementos no mundo. A identificação de um objeto no mundo depende de sua diferenciação em relação a outros objetos. Essa diferenciação presume classificação, que por sua vez presume hierarquização. O conhecimento depende da distinção entre os atributos dos objetos no mundo. Sem a distinção, sem o relacionamento, não há conhecimento. Relação define, portanto, conhecimento. Relação não é o conhecimento em si, mas um fator fundamental para que o conhecimento exista.

É necessário conhecer estes conceitos porque as relações são a base da Programação Lógica. Ela não é baseada na seqüência de procedimentos, mas na definição de relações, na forma com a qual se representa o mundo que se quer implementar no computador. No presente capítulo procura-se trazer um visão sobre Programação Lógica, mostrando suas origens e sua evolução, até as principais características e aplicações, abordando ainda o seu relacionamento e embasamento matemático.

4.1 O QUE É

Para se compreender como a lógica pode ser empregada na programação de computadores com vantagem sobre as linguagens convencionais é necessário primeiramente definir o que é lógica. Conforme [BAR97], lógica é o estudo dos mecanismos válidos de inferência, ou ainda, segundo [PAL97], a lógica é a ciência do pensamento correto. Isto não implica em afirmar que ela seja a ciência da verdade. Mesmo que tudo o que se permita afirmar dentro da lógica seja supostamente verdadeiro em determinado contexto, as mesmas afirmações podem resultar falsas se aplicadas ao mundo real. Também segundo [PAL97], podemos considerar lógicos os conjuntos de declarações que possuem a propriedade de ser verdadeiros ou falsos independentemente do tempo ou lugar que ocupam no universo considerado.

A lógica é uma linguagem e assim sendo possui sintaxe e semântica, e em adicional possui regras de inferência. O que a torna útil para a computação é que sua notação central é o processo de provar teoremas, que é um conjunto de processos rigorosos, onde uma nova informação pode ser seguramente derivada de uma informação já conhecida como verdadeira. De forma muito simplificada, Ivan Bratko [BRA90] definiu um programa lógico como sendo a especificação executável da solução para um problema.

Segundo [PAL97], um programa em lógica é a representação de determinado problema ou situação expressa através de um conjunto finito de um tipo especial de sentenças lógicas denominadas cláusulas. Ele não é a descrição de um procedimento para se obter a solução de um problema. Na realidade, o sistema utilizado no processamento de programas em lógica é inteiramente responsável pelo procedimento a ser adotado na sua execução. Um programa em lógica pode também ser visto alternativamente como uma base de dados, exceto que as bases de dados convencionais descrevem apenas fatos tais como “Oscar é um avestruz”, enquanto que as sentenças de um programa em lógica possuem um alcance mais genérico, permitindo a representação de regras como “Todo avestruz é um pássaro”, o que não possui correspondência em base de dados convencionais.

Um programa em lógica consiste em conjuntos de regras que descrevem relações entre objetos. Estas relações, chamadas de predicados do programa, são escritas a partir de um subconjunto do cálculo dos predicados. Pode-se então expressar conhecimento por meio de

cláusulas de dois tipos: fatos e regras. Um fato denota uma verdade incondicional, enquanto que as regras definem as condições que devem ser satisfeitas para que uma certa declaração seja considerada verdadeira [PAL97].

Uma das principais idéias da programação em lógica é de que um algoritmo é constituído por dois elementos disjuntos: a lógica e o controle. O componente lógico corresponde à definição do que deve ser solucionado, enquanto que o componente de controle estabelece como a solução pode ser obtida [PAL97]. O programador precisa somente descrever o componente lógico utilizado. Em outras palavras, a tarefa do programador passa a ser simplesmente a especificação do problema que deve ser solucionado, razão pela qual as linguagens lógicas são consideradas linguagens de "alto-nível" e podem ser vistas simultaneamente como linguagens para especificação formal e linguagens para a programação de computadores [PAL97] [MAI88].

O ponto central da programação em lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, os sistemas de programação em lógica reduzem a execução de programas à pesquisa de refutação das sentenças do programa em conjunto com a negação da sentença que expressa a consulta, seguindo a regra: "uma refutação é a dedução de uma contradição." O cálculo proposicional e o cálculo dos predicados, que são os subconjuntos da lógica matemática mais diretamente envolvidos nesse processo, formalizam a estrutura lógica mais elementar do discurso definindo precisamente o significado dos conetivos e, ou, não, se... então e outros.

O estilo declarativo é melhor que o estilo procedural tradicional porque direciona o programador a pensar sobre o objetivo do programa, sobre a descrição estática dos relacionamentos e propriedades contidas no programa, sem se preocupar com o controle. O programa é descrito com um nível de abstração maior, o que diminui o tempo gasto para entender e examinar partes de um programa. Programas em linguagens declarativas são mais fáceis de modificar, podendo-se adicionar novas cláusulas sem afetar as cláusulas já existentes. E ainda, ao contrário das linguagens procedurais, as declaração são independentes do contexto [MAI88].

Separar o significado do programa de um modelo computacional particular é importante. Esta separação resulta em liberdade para alternar entre implementações do

programa. O fato é que quando há implementações alternativas se torna evidente que a linguagem possui um alto nível de abstração.

4.2 HISTÓRICO

O uso da lógica na representação dos processos de raciocínio iniciou-se com os estudos de Boole (1815-1864) e de De Morgan (1806-1871), sobre o que veio a ser mais tarde chamado de "Álgebra de Boole". Esses trabalhos estavam mais próximos de outras teorias matemáticas do que propriamente da lógica [PAL97]. Deve-se ao matemático alemão Gottlieb Frege a primeira versão do cálculo de predicados, proposto por ele como uma ferramenta para formalizar princípios lógicos. Criou uma linguagem chamada "*Begriffsschrift*" (1879), que oferecia uma notação rica e consistente que ele pretendia adequar para a representação de todos os conceitos matemáticos e para a formalização exata do raciocínio dedutivo sobre tais conceitos [NIL97] [PAL97] [RAB95].

No início deste século, 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. Um passo muito importante foi dado em 1930 pelo alemão Kurt Gödel e pelo francês Jacques Herbrand, que demonstraram que o mecanismo de prova do cálculo de predicados poderia oferecer uma prova formal de toda proposição logicamente verdadeira. O resultado de maior impacto foi entretanto produzido por Gödel, em 1931, com a descoberta do "teorema da incompleteza dos sistemas de formalização da aritmética". A prova deste teorema se baseava nos denominados paradoxos de auto-referência (declarações do tipo: "Esta sentença é falsa", que não podem ser provadas nem verdadeiras nem falsas). Em 1934, Alfred Tarski produziu a primeira teoria semântica rigorosamente formal do cálculo de predicados, introduzindo conceitos precisos para "satisfatibilidade", "verdade" (em uma dada interpretação), "conseqüência lógica" e outras noções relacionadas. Ainda na década de 30, diversos outros estudos, entre os quais os de Alan Turing, Alonzo Church e outros, aproximaram muito o cálculo de predicados da forma com que é hoje conhecido e estudado [PAL97].

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. Mas foi somente a partir da

metade dos anos 50 que o desenvolvimento da 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. Tal forma empregava um tipo particular muito simples de sentença lógica denominada "cláusula" [PAL97].

A expressão "programação em lógica" (*logic programming*) 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 condições para entendê-la como uma linguagem de programação de uso geral. 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 práticas 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 definiram formalmente 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 [PAL97].

4.3 CARACTERÍSTICAS

Os termos "programação em lógica" e "programação Prolog" tendem a ser empregados indistintamente. Deve-se, entretanto, destacar que a linguagem Prolog é apenas uma particular abordagem da programação em lógica. As características mais marcantes dos sistemas de programação em lógica em geral – e da linguagem Prolog em particular – abordados em [PAL97] são as seguintes:

a) especificação 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 com qualquer número de argumentos são possíveis. Não há distinção entre o programa e os dados. As cláusulas podem ser usadas com vantagem sobre as construções convencionais para a representação de tipos abstratos de dados;

b) 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;

c) 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;

d) 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;

e) 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;

f) recursão: a recursão, em Prolog, é a forma natural de ver e representar dados e programas. Entretanto, na sintaxe da linguagem não há laços do tipo "*for*" ou "*while*", 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.

A premissa básica da programação em lógica é portanto que "computação é inferência controlada". Tal visão da computação tem se mostrado extremamente produtiva, na medida em que conduz à idéia de que computadores podem ser projetados com a arquitetura de máquinas de inferência. Grande parte da pesquisa sobre computação paralela, conduzida hoje nos EUA, Europa e Japão, emprega a programação em lógica como instrumento básico para a especificação de novas arquiteturas de hardware e o desenvolvimento de máquinas abstratas não-convencionais.

4.4 PRINCIPAIS ÁREAS DE APLICAÇÃO

Um dos primeiros usos da programação em lógica foi a representação e análise de subconjuntos da linguagem natural. Esta foi, inclusive, a aplicação que motivou Alain Colmerauer a desenvolver a primeira implementação da linguagem Prolog. Logo em seguida, outros pesquisadores da área da inteligência artificial propuseram diversas novas aplicações para o novo instrumento. Alguns dos primeiros trabalhos com Prolog envolviam a formulação de planos e a escrita de compiladores, por Pereira e Warren (1977), prova de teoremas em geometria por R. Welhan (1976) e a solução de problemas de mecânica, por Bundy *et al.* (1979). As aplicações relatadas desde então, multiplicaram-se velozmente.

Existe uma área de aplicação da Programação Lógica que originou a maioria das atuais áreas: os sistemas baseados em conhecimento (SBCs), ou *knowledge-based systems*. Os SBCs são sistemas que aplicam mecanismos automatizados de raciocínio para a representação e inferência de conhecimento. Algumas das suas subclasses são: sistemas de base de dados, sistemas especialistas e processamento de linguagem natural. Segue com um conjunto das principais áreas de aplicação da programação em lógica, de acordo com [PAL97]:

a) sistemas de bases de dados (BDs): BDs 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 BD 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 apropriados, se existirem. O potencial da programação em lógica para a representação e consulta à BDs foi simultaneamente investigado, em 1978, por van Emden, Kowalski e Tärnlund. As três pesquisas estabeleceram que a recuperação de dados – um problema básico em BDs convencionais – é intrínseca ao mecanismo de inferência dos interpretadores lógicos. Desde então diversos sistemas tem sido propostos para a representação de BDs por meio de programas em lógica;

b) sistemas especialistas (SEs): um sistema especialista é uma forma de SBC especialmente projetado para emular a especialização humana em algum domínio específico. Tipicamente um SE irá possuir uma base de conhecimento (BC) formada de fatos, regras e heurísticas sobre o domínio, juntamente com a capacidade de entabular comunicação interativa com seus usuários, de modo muito próximo ao que um especialista humano faria. Além disso os SEs devem ser capazes de oferecer sugestões e conselhos aos usuários e, também, melhorar o próprio desempenho a partir da experiência, isto é, adquirir novos conhecimentos e heurísticas com essa interação;

c) processamento da linguagem natural (PLN): o PLN é da maior importância para o desenvolvimento de ferramentas para a comunicação homem-máquina em geral e para a construção de interfaces de SBCs em particular. A implementação de sistemas de PLN 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. O uso da lógica das cláusulas de Horn para este propósito foi inicialmente investigado por Colmerauer, o próprio criador do Prolog (1973), e posteriormente por Kowalski (1974). Ambos mostraram (1) que as cláusulas de Horn eram adequadas à representação de qualquer gramática livre-de-contexto, (2) permitiam que questões sobre a estrutura de sentenças em linguagem natural fossem formuladas como

objetivos ao sistema, e (3) que diferentes procedimentos de prova aplicados a representações lógicas da linguagem natural correspondiam a diferentes estratégias de análise;

d) educação: a programação em lógica poderá vir a oferecer no futuro uma contribuição bastante significativa ao uso educacional de computadores. Esta proposta 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. O sucesso do empreendimento conduziu a um projeto mais abrangente denominado "Lógica como Linguagem de Programação para Crianças", inaugurado em 1980 na Inglaterra com recursos do Conselho de Pesquisa Científica daquele país. 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 linguagens como o Prolog induz;

e) arquiteturas não-convencionais: esta área vem se tornando cada vez mais um campo extremamente fértil para o uso da programação em lógica especialmente 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. Duas implementações iniciais nesse sentido foram o Parlog, desenvolvido em 1984 por Clark e Gregory, e o Concurrent Prolog, por Shapiro em 1983. O projeto da Quinta Geração, introduzido na próxima seção, foi fortemente orientado ao uso da programação em lógica em sistemas de processamento paralelo.

Outras aplicações poderiam ainda ser citadas, principalmente na área da inteligência artificial. Novas tecnologias de hardware e software tais como sistemas massivamente paralelos, redes de computadores, assistentes inteligentes, bases de dados semânticas, etc., tornam o uso do Prolog (e de outras linguagens baseadas em lógica) cada vez mais atraentes.

4.5 ASPECTOS MATEMÁTICOS

A programação lógica fundamenta-se na lógica matemática. Segundo [BEN96], a lógica matemática formaliza a estrutura e os procedimentos usados na manipulação dedutiva da informação, ou segundo [MAI88], a lógica é a formalização de vários aspectos da linguagem. Historicamente, a lógica tem sido motivada por uma tentativa de entender a linguagem natural. Várias lógicas tem sido desenvolvidas para formalizar e capturar os diferentes aspectos desta linguagem. A formalização de uma linguagem consiste de três partes: sintaxe, semântica e dedução.

A sintaxe de uma linguagem é uma especificação precisa de suas expressões válidas, que são geralmente seqüências de símbolos [MAI88]. Por exemplo, a língua portuguesa possui um componente sintático para especificar que uma seqüência de palavras como “O cachorro persegue o gato” é uma sentença válida, enquanto a seqüência “Cachorro gato o o persegue” é uma sentença inválida.

O componente semântico de uma lógica captura o significado de expressões da linguagem [MAI88]. Por exemplo, o componente semântico de um lógica hipotética para a língua portuguesa poderia criar uma função dos estados do mundo, para o conjunto {verdadeiro, falso}, a partir de uma sentença declarativa. Considerando que a função, dado um estado do mundo, retorne o valor “verdadeiro” se nesse estado há um determinado mamífero peludo que corre ao redor de um outro tipo de mamífero peludo menor. A semântica poderia atribuir esta função como significado para a seqüência “O cachorro persegue o gato”.

O componente dedutivo de uma lógica provê regras para manipulação de expressões preservando os aspectos de sua semântica [MAI88]. Por exemplo, a lógica hipotética do Português poderia conter uma regra de “passivização”, que diz precisamente como o sujeito e o objeto de uma sentença podem ser trocados, com as devidas alterações no verbo. Neste caso, a sentença “O cachorro persegue o gato” se tornaria “O gato é perseguido pelo cachorro”. Uma propriedade desta transformação é que o significado da sentença é preservado.

A lógica utilizada na Programação Lógica possui estes três componentes, porém eles são menos ambiciosos que a lógica completa da língua portuguesa, por exemplo. A lógica enfatiza a importância da semântica e da dedução, e sua formalidade é considerada crítica.

O nível mais simples da lógica é chamado de lógica proposicional. Ela formaliza o significado dos conectivos “e”, “ou”, “não”, “se ... então”, e “se e somente se ...” quando aplicados a declarações. A lógica dos predicados adiciona objetos, propriedades e quantificadores aos conectivos já mencionados [BEN96]. Um objeto pode ou não ter uma propriedade particular. Os quantificadores formalizam as notações “para todo” e “existe”, o que permite sentenças do tipo “Todo cachorro persegue gatos”. Estas duas lógicas são também chamadas de cálculos. Um cálculo é simplesmente um método para calcular, ou segundo [BAR97], cálculo designa um sistema simbólico para ajudar o raciocínio.

4.5.1 LÓGICA DAS PROPOSIÇÕES

Algumas perguntas precisam ser respondidas para que se possa entender o que é lógica das proposições. A primeira delas é: o que é uma proposição? Uma proposição é simplesmente uma declaração e pode ser também chamada de cláusula. Por exemplo, “este livro é maçante”, ou “se este livro é maçante, então eu vou adormecer”. As proposições podem ser condicionais ou incondicionais. Uma proposição condicional é uma regra e uma proposição incondicionalmente verdadeira é um fato [BEN96] [MAI88].

A segunda pergunta é: o que é uma regra de inferência? Segundo [MAI88], uma regra de inferência é uma forma de derivar uma declaração implícita por manipulação sintática de um conjunto de declarações explícitas. Tais regras são o primeiro passo para a computação declarativa.

Segundo [BEN96], a lógica das proposições é a lógica mais simples e consiste na formalização e estudo dos conectivos. Ela pode expressar certos fatos e regras simples e pode testar se um determinado fato pode ser inferenciado de um conjunto de fatos e regras. A sua principal finalidade é de, dada uma proposição com sintaxe correta e a semântica dos componentes da proposição, determinar seu valor semântico [BAR97].

4.5.1.1 SINTAXE

A sintaxe do Cálculo das Proposições especifica os símbolos e os modos de combiná-los para formar uma expressão válida da linguagem, as quais costumam ser chamadas "fórmulas bem formadas" (fbf), do inglês "*Well Formed Formulas*" (wff). Os símbolos representam "átomos", "conectivos", "operadores lógicos" e "parênteses" [BAR97]. Por exemplo, a seguinte sentença "Pedro tem um carro" forma um pensamento atômico e por esta razão é chamado fórmula atômica. Uma forma de representá-la poderia ser simplesmente:

PedroTemCarro

Formulas atômicas são o tipo mais simples de fbf. Pode-se formar outras fbfs usando conectivos, dos quais os mais conhecidos são: $\vee$ (ou), $\wedge$ (e), $\Rightarrow$ (implicação – se ... então) e $\equiv$ (se e somente se ...). O nome conectivo vem do fato que eles conectam duas fórmulas atômicas gerando uma nova fórmula. A estes conectivos vem-se juntar a negação, de símbolo $\neg$, que é um operador lógico e não um conectivo. Ele muda o valor verdade da fórmula que precede. E ainda, para resolver ambigüidades, utiliza-se parêntesis. Segundo [BAR97] [BEN96] e [NIL97], a definição de uma fbf é a seguinte:

a) uma fórmula atômica é uma fbf;

b) se α é um fbf, então $(\neg \alpha)$ é uma fbf;

c) se α e β são fbfs, então $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \Rightarrow \beta)$ e $(\alpha \equiv \beta)$ são fbfs.

Pode-se ainda utilizar parêntesis para definir a prioridade dos conectivos. Segue um exemplo de fbf utilizada para representar a sentença "Se Pedro tem um carro e sabe dirigir então ele pode dirigir":

$(\text{PedroTemCarro} \wedge \text{PedroSabeDirigir}) \Rightarrow \text{PedroPodeDirigir}$

E alguns outros exemplos de fbfs:

a) $(A \vee B) \Rightarrow (\neg C)$

b) $A \Rightarrow (\neg B)$

c) $(A \vee B) \Rightarrow C$

d) $(A \Rightarrow B) \Rightarrow (\neg C \Rightarrow \neg D)$

e) $\neg \neg A$

4.5.1.2 SEMÂNTICA

A semântica deve associar elementos de uma linguagem lógica com elementos de um domínio e são estas associações que formam o significado. Na lógica proposicional os átomos são associados com proposições sobre o mundo. Por exemplo, o átomo BAT_OK está associado com a proposição "A bateria está carregada". Uma associação de átomos com proposições é conhecida como interpretação [NIL97].

Logo, um fbf pode ter uma interpretação a qual define a semântica da linguagem. Uma interpretação pode ser considerada como um mapeamento do conjunto das fbfs para um conjunto de valores de verdade que na lógica dicotômica é o conjunto {verdadeiro, falso} ou {V, F} [BAR97] [BEN96]. A semântica destes conectivos mais conhecidos é: $\vee$ (ou), $\wedge$ (e), $\Rightarrow$ (implicação – se ... então) e $\equiv$ (se e somente se ... , que pode também ser representado por $\Leftrightarrow$), ou seja:

a) $(A \wedge B)$ – é verdadeiro se A é verdadeiro e B também é verdadeiro, senão é falso;
b) $(A \vee B)$ – é verdadeiro se A é verdadeiro ou se B é verdadeiro, inclusive se A e B forem verdadeiros, senão é falso;
c) $(A \Rightarrow B)$ – significa que se A é verdadeiro então B também é verdadeiro, entretanto, nada se sabe de B se A for falso. Esta cláusula só será falsa se A for verdadeiro e B for falso;
d) $(A \equiv B)$ – significa que A é verdadeiro se e somente se B também for verdadeiro. Este conectivo representa a igualdade, portanto, esta cláusula será falsa se os valores de A e B forem diferentes;
e) $(\neg A)$ – é verdadeiro se A for falso e é falso se A for verdadeiro. Pode ser considerado um conectivo unário, o qual muda o valor de uma fórmula, lhe dando o valor de interpretação oposto.

É usual definir os conectivos dando sua semântica através de Tabelas Verdades. Por exemplo, para a Tabela Verdade de $\vee$ usa-se uma tabela de dupla entrada, na vertical os valores possíveis da variável A e na horizontal os da variável B. Assim a tabela e definição de $\vee$ é:

$A \vee B$	V	F
V	V	V
F	V	F

Pode-se também usar as tabelas lineares, abaixo representada, para demonstrar os conectivos:

A	B	$A \vee B$	$A \wedge B$	$A \Rightarrow B$	$A \equiv B$	$\neg A$
V	V	V	V	V	V	F
F	V	V	F	V	F	V
V	F	V	F	F	F	F
F	F	F	F	V	V	V

4.5.1.3 DEDUÇÃO

Segundo [MAI88], a dedução é a transformação de fórmulas preservando os aspectos da sua semântica, ou seja, mantendo a fórmula original e a fórmula transformada equivalentes. Duas fórmulas A e B são logicamente equivalentes se possuem o mesmo valor verdade sobre todas as possibilidades. Um exemplo simples é que P and $\neg \neg$ P são logicamente equivalentes, ou seja, qualquer valor que for atribuído a P manterá as duas fórmulas com o mesmo valor. O símbolo usado para denotar equivalência é: ≡.

As regras de inferência são a base do processo de dedução, que é também chamado de prova de teoremas. O processo de dedução inicia com um conjunto de fórmulas e aplica regras de inferência para deduzir novas fórmulas implicadas pelo conjunto inicial [MAI88]. As tabelas verdades permitem provar a equivalência de duas expressões envolvendo conectivos. Para ilustrar, segue a prova de alguns teoremas:

a) teorema da dupla negação: $(\neg (\neg A)) \equiv A$

A	$(\neg A)$	$(\neg (\neg A))$
V	F	V
F	V	F

b) teorema da implicação: $(A \Rightarrow B) \equiv (\neg A) \vee B$

A	B	$(\neg A)$	$((\neg A) \vee B)$	$(A \Rightarrow B)$
V	V	F	V	V
V	F	F	F	F
F	V	V	V	V
F	F	V	V	V

c) teorema de Morgan: $(\neg (A \vee B)) \equiv ((\neg A) \wedge (\neg B))$

A	B	$A \vee B$	$\neg (A \vee B)$	$\neg A$	$\neg B$	$(\neg A) \wedge (\neg B)$
V	V	V	F	F	F	F
V	F	V	F	F	V	F
F	V	V	F	V	F	F
F	F	F	V	V	V	V

Existem outras equivalências lógicas, por exemplo as leis comutativas e associativas para “e” e “ou”:

a) $A \wedge B \equiv B \wedge A$;

b) $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$;

c) $A \vee B \equiv B \vee A$;

d) $(A \vee B) \vee C \equiv A \vee (B \vee C)$.

Segue com mais algumas das principais regras de inferência:

a) modus ponens – se $A \Rightarrow B$ e se A é verdadeiro então B é verdadeiro;

b) modus tollens – se $A \Rightarrow B$ e se B é falso então A é falso;

c) lei do contrapositivo – se $A \Rightarrow B$ é verdadeiro então $\neg B \Rightarrow \neg A$ é verdadeiro;

d) regra de cadeia – se $A \Rightarrow B$ e se $B \Rightarrow C$ são verdadeiros então $A \Rightarrow C$ é verdadeiro.

Mas existe ainda um método especial de dedução chamado resolução. Os processos de dedução que usam as regras de resolução fazem uso do princípio da refutação e consequentemente são chamados de processos de refutação. O princípio da refutação declara que para provar que um conjunto de fórmulas S implica a fórmula f, deve se expressar esta implicação como uma nova fórmula g e deve-se tentar mostrar que a negação de g é insatisfatória [MAI88]. O método de dedução por refutação assim como outros métodos de resolução não são abordados neste trabalho. Uma ampla abordagem pode ser encontrada em [MAI88] e [NIL97].

4.5.2 LÓGICA DOS PREDICADOS

A lógica dos predicados é a extensão da lógica das proposições em que se consideram variáveis e quantificadores sobre as variáveis. Desta forma, a lógica dos predicados permite um detalhamento da estrutura das frases que a lógica proposicional trata como “caixas pretas” denotadas por proposições [BEN96] [MAI88]. Os dois quantificadores mais importantes são o quantificador universal e o existencial, respectivamente representados pelos símbolos: $\forall$ e $\exists$. Na lógica dos predicados, quantificação envolve apenas variáveis [BAR97]. Na lógica proposicional a declaração “Se Pedro é humano, então Pedro possui uma mãe humana”, é representada da seguinte forma:

$(A \Rightarrow B)$, sendo A = “Pedro é humano” e B = “Pedro possui uma mãe humana”

Na lógica dos predicados, “humano” e “tem_mãe_humana” são predicados. Quando os predicados são aplicados aos seus argumentos retornam um resultado verdadeiro ou falsos. Os argumentos dos predicados, que são chamados de termos, podem ser constantes ou variáveis. Desta forma, pode-se criar uma declaração que se aplica a todas as coisas, não apenas a Pedro:

$$\forall X \, (\, humano(X) \Rightarrow tem\_mãe\_humana(X) \,)$$

O quantificador universal $\forall X$ deve ser lido “para todo X”, portanto, esta declaração poderia ser traduzida por: “Para todo X, se X é humano, então X têm mãe humana”. Porém, não é comum pensar em “tem_mãe_humana” como um predicado que tem um único argumento e ao invés de “Pedro tem uma mãe humana” seria melhor “Há alguém que é humano e que é mãe de Pedro”. Isto pode ser capturado por um predicado lógico, que inclusive estende para todos as pessoas, não só a Pedro:

$$\forall X \, (\, humano(X) \Rightarrow (\, \exists Y \, (\, humano(Y) \wedge mãe(Y, X) \,) \,) \,)$$

O quantificador existencial $\exists Y$ deve ser lido “existe Y”, portanto esta declaração poderia ser traduzida por: “Para todo X, se X é humano, então existe Y, tal que Y é humano e Y é mãe de X”. Esta idéia pode ser expressa usando funções como a seguinte: quando o predicado humano(X) é verdadeiro, a função mãe(X) retorna o termo que representa a mãe de X. Usando esta função a declaração poderia ser representada por:

$$\forall X \, (\, humano(X) \Rightarrow (\, humano(\, mãe(X) \,) \,)$$

Onde “humano” é um predicado e “mãe” é uma função. Neste exemplo pode-se notar que uma função produz um valor que é um termo enquanto um predicado produz apenas “verdadeiro” e “falso”.

4.5.2.1 SINTAXE

A sintaxe da lógica dos predicados é similar em sua estrutura com a sintaxe da lógica das proposições e pode também ser vista como uma extensão. Na realidade, a lógica dos predicados usando apenas conectivos, parêntesis e predicados sem argumentos é lógica das proposições. Segundo [MAI88], uma fórmula predicada pode ser uma disjunção de um ou

mais termos, sendo que cada qual é um conjunção de um ou mais átomos. E adicionando-se a isso estão ainda os quantificadores, que possibilitam as generalizações. A linguagem da lógica dos predicados consiste dos seguintes símbolos:

a) um infinito conjunto de variáveis, denotadas por palavras cuja primeira letra deve ser maiúscula. Exemplos: X, A, Nome, TipoSangue, X12;
b) um conjunto de constantes, denotadas por palavras cuja primeira letra deve ser minúscula. Exemplos: pedro, jeep, x, a, jairSantos, x12;
c) um conjunto de predicados, denotados por palavras cuja primeira letra deve ser minúscula. Exemplos: pai, humano, ama, temCarro;
d) um conjunto de funções, denotadas por palavras cuja primeira letra deve ser minúscula. Exemplos: pai, distanciaEntre, tempo;
e) os conectivos $\neg$, $\vee$, $\wedge$, $\Rightarrow$ e $\equiv$;
f) os quantificadores $\forall$ e $\exists$;
g) os parêntesis) e (.

Os termos da lógica dos predicados são definidos da seguinte forma [BEN96]:

a) cada variável e cada constante é um termo;
b) se $t_1, ..., t_n$ são termos e f é uma função que tem n argumentos, então $f(t_1, ..., t_n)$ é um termo. Este tipo de termo é chamado expressão funcional.

As fórmulas bem formadas (fbf) na linguagem da lógica dos predicados são definidas recursivamente pelas seguintes regras [BEN96]:

a) se $t_1, ..., t_n$ são termos e p é um predicado que tem n argumentos, então $p(t_1, ..., t_n)$ é uma fbf, chamada fórmula atômica;
b) se α é uma fbf, então $(\neg \alpha)$ é uma fbf;
c) se α e β são fórmulas, então $(\alpha \vee \beta)$, $(\alpha \wedge \beta)$, $(\alpha \Rightarrow \beta)$ e $(\alpha \equiv \beta)$ são fbfs;
d) se V é uma variável e α é uma fórmula, então $(\forall V\ \alpha)$ e $(\exists V\ \alpha)$ são fbfs.

Seguem alguns exemplos de fórmulas bem formadas:

a) pai(joao, vitor);
b) $\forall X\ \forall Y$ (pai(Z, X) $\wedge$ pai(Z, Y) $\Rightarrow$ irmao(X, Y));
c) $\forall X$ (humano(X) $\Rightarrow$ mortal(X));
d) $\exists X\ \exists Y$ (humano(X) $\wedge$ humano(Y) $\wedge$ ama(X, Y)).

4.5.2.2 SEMÂNTICA

Na lógica das proposições são usados apenas assinalamentos verdadeiros para obter o significado das fórmulas e é necessário apenas abstrair valores verdades de sentenças simples. A lógica dos predicados envolve também os objetos do mundo. A interpretação do mundo provê um conjunto de objetos denominados domínio ou universo. Estes dão significado as constantes e variáveis das fórmulas [MAI88].

A sintaxe define como tudo deve ser construído na linguagem mas, como na lógica proposicional, ela não diz nada sobre o que as fórmulas "significam". Para associar o significado às fórmulas da lógica dos predicados, é necessário saber interpretá-las, o que é mais complicado que o "verdadeiro" e "falso" da lógica proposicional. A extensão de uma função é o conjunto das constantes, o domínio das funções e dos predicados são um subconjunto de constantes, e a definição depende de conhecer as funções e de saber quando os predicados são verdadeiros – uma "interpretação" [BEN96].

A interpretação de uma expressão no cálculo dos predicados mapeia as constantes de objetos em objetos do mundo, as constantes de funções em funções e as constantes de relacionamento em relações. Estas atribuições são chamadas denotações das expressões correspondentes do cálculo dos predicados. O conjunto de objetos onde as atribuições de constantes é feita é chamado de domínio da interpretação. Dada uma interpretação para as partes dos seus componentes, um átomo tem o valor verdadeiro apenas quando a relação denotada prende os indivíduos denotados aos seus termos. Se a relação não prender, o átomos tem valor falso. Os valores verdadeiro e falso de fórmulas não atômicas são determinados pela mesma tabela verdade usada no cálculo das proposições [NIL97].

Para facilitar o entendimento, será usado um exemplo, considerando um mundo onde existem as entidades A, B, C e F, sendo A, B e C blocos e F o chão. Pode-se imaginar alguns relacionamentos entre estas entidades, como: "sobre", que indica que um bloco está sobre outro ou sobre o chão, e "vazio", que indica que um bloco está vazio. Os blocos estão dispostos B sobre A, A sobre C e C sobre o chão e apenas o bloco B está vazio. Os relacionamentos são então mapeados, e o relacionamento "sobre" é composto pelo conjunto dos pares {<B, A>, <A, C>, <C, F>}, e o relacionamento "vazio" por {<B>}. Portanto, as

constantes A, B, C e F, e os relacionamentos “sobre” e “vazio” constituem o domínio deste exemplo. Desta forma é possível determinar o valor de alguns predicados:

a) sobre(A, B) é falso porque <A, B> não está no conjunto do relacionamento “sobre”;
b) vazio(B) é verdadeiro porque <B> está no conjunto do relacionamento “vazio”;
c) sobre(C, F) é verdadeiro porque <C, F> está no conjunto do relacionamento “sobre”;
d) sobre(C, F) $\wedge$ sobre(A, B) é verdadeiro porque tanto sobre(C, F) como sobre(A, B) são verdadeiros.

A lógica dos predicados possui algumas regras semânticas, que são descritas a seguir [BEN96]:

a) se *p* é um predicado e nenhum dos termos $t_1, ..., t_n$ contém variáveis, então $p(t_1, ..., t_n)$ é verdadeiro ou não de acordo com a interpretação;
b) se as verdades de α e β são conhecidas, então a verdade dos conectivos é determinada pelas mesmas regras da lógica proposicional;
c) sendo V uma variável e α uma fórmula, se houver alguma constante *c* tal que substituindo cada ocorrência livre de V em α por *c* resulta em uma fórmula verdadeira, então ($\exists$V α) é verdadeiro;
d) sendo V uma variável e α uma fórmula, se para cada constante *c*, substituindo cada ocorrência livre de V em α por *c* resulta em uma fórmula verdadeira, então ($\forall$V α) é verdadeiro.

Uma fórmula é chamada válida se, e somente se, ela é verdadeira para todas as possíveis interpretações. Deve-se destacar ainda dois fatos importantes sobre a definição. Primeiramente, define-se que uma fórmula é verdadeira ou falsa apenas quando não há nenhuma ocorrência livre das variáveis. Em segundo, como na lógica das proposições, a definição está freqüentemente aplicada no sentido reverso da definição da sintaxe. Por exemplo, considerando:

$$((\forall X\ p(X)) \Rightarrow (\exists X\ p(X)))$$

Para determinar se esta fórmula é verdadeira deve-se primeiro determinar se ($\forall$X p(X)) e ($\exists$X p(X)) são verdadeiros. Há três possibilidades, que são mostradas a seguir com suas conseqüências:

a) p(c) é verdadeiro para todo c. Neste caso, tanto ($\forall$X p(X)) como ($\exists$X p(X)) são verdadeiros, e a fórmula mencionada é também verdadeira;
b) p(c) é falso para todo c. Neste caso, tanto ($\forall$X p(X)) como ($\exists$X p(X)) são falsos, e a fórmula mencionada é verdadeira;
c) p(c) é verdadeira para alguns c e falsa para outros. Neste caso, (($\forall$X p(X)) é falso e ($\exists$X p(X)) é verdadeiro, e a fórmula mencionada é também verdadeira.

4.5.2.3 DEDUÇÃO

As implicações lógicas para a lógica dos predicados se originam das implicações da lógica proposicional. Uma fórmula α implica logicamente a fórmula β se quando α é verdadeiro β é também verdadeiro para qualquer estrutura e para qualquer instanciação. Isto é, o requisito é que tendo qualquer estrutura, se α é verdadeiro para uma instanciação particular *I*, então β é verdadeira para a mesma instanciação *I*. Diferente de dizer que β deve ser verdadeiro sobre qualquer instanciação apenas quando α for verdadeiro sobre qualquer instanciação [MAI88].

Como na lógica das proposições, a dedução é formada por equivalências lógicas e regras de inferência. Segue uma lista das principais equivalências lógicas, onde α é uma fórmula na qual qualquer ocorrência de X e Y são livres, β é uma fórmula sem X livres, e * pode ser o conectivo $\vee$ ou $\wedge$:

a) $\neg (\forall X \alpha) \equiv \exists X (\neg \alpha)$;
b) $\neg (\exists X \alpha) \equiv \forall X (\neg \alpha)$;
c) $\forall X (\forall Y \alpha) \equiv \forall Y (\forall X \alpha)$;
d) $\exists X (\exists Y \alpha) \equiv \exists Y (\exists X \alpha)$;
e) $((\forall X \alpha) * \beta) \equiv ((\forall X (\alpha * \beta))$;
f) $((\exists X \alpha) * \beta) \equiv ((\exists X (\alpha * \beta))$;
g) $((\forall X \alpha) * (\forall X \beta)) \equiv \forall X (\alpha * \beta)$;
h) $((\exists X \alpha) * (\exists X \beta)) \equiv \exists X (\alpha * \beta)$.

5 AMBIENTE PROLOG

O Prolog é a principal implementação de um ambiente para programação lógica e por esta razão sua origem e principais características já foram abordadas nos capítulos anteriores. Este capítulo se resume a mostrar as particularidades de um ambiente Prolog baseado no padrão Edimburgo, descrevendo seus mecanismos básicos e ilustrando-os com exemplos.

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 [PAL97]. A linguagem Prolog reforçou a tese de que a lógica é um formalismo conveniente para representar e processar conhecimento. Seu uso evita a descrição dos procedimentos necessários para a solução de um problema, permitindo que se expresse declarativamente apenas a sua estrutura lógica, através de fatos, regras e consultas [WIL93] [WAT90]. Segundo [PAL97], algumas das principais características da linguagem Prolog são:

a) é uma linguagem orientada ao processamento simbólico;
b) representa uma implementação da lógica como linguagem de programação;
c) apresenta uma semântica declarativa inerente à lógica;
d) permite a definição de programas reversíveis, isto é, programas que não distinguem entre os argumentos de entrada e os de saída;
e) permite a obtenção de respostas alternativas;
f) suporta código recursivo e iterativo para a descrição de processos e problemas, dispensando os mecanismos tradicionais de controle, tais como *while*, *repeat*, etc;
g) permite associar o processo de especificação ao processo de codificação de programas;
h) representa programas e dados através do mesmo formalismo;
i) incorpora facilidades computacionais extralógicas e metalógicas.

5.1 FATOS

Segundo [PAL97], uma relação é definida em Prolog estabelecendo-se as tuplas de objetos que satisfazem a relação. Considerando a árvore genealógica mostrada na Figura 1, é possível definir entre os objetos mostrados uma relação denominada progenitor, que associa um indivíduo a um dos seus progenitores. Por exemplo, o fato de João ser um dos

progenitores de José pode ser denotado por: `progenitor(joão, josé)`, onde "progenitor" é o nome da relação e "joão" e "josé" são os seus argumentos. Os argumentos das relações podem ser objetos concretos, como "joão" e "josé", ou objetos genéricos, como "X" e "Y". Objetos concretos são denominados átomos e os objetos genéricos são denominados variáveis. No padrão Edimburgo, escreve-se os átomos iniciando com letras minúsculas e as variáveis iniciando com letras maiúsculas [BEN96] [PAL97] [WIL93]. A relação progenitor completa, como representada na figura 1 pode ser definida pelo seguinte programa Prolog:

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

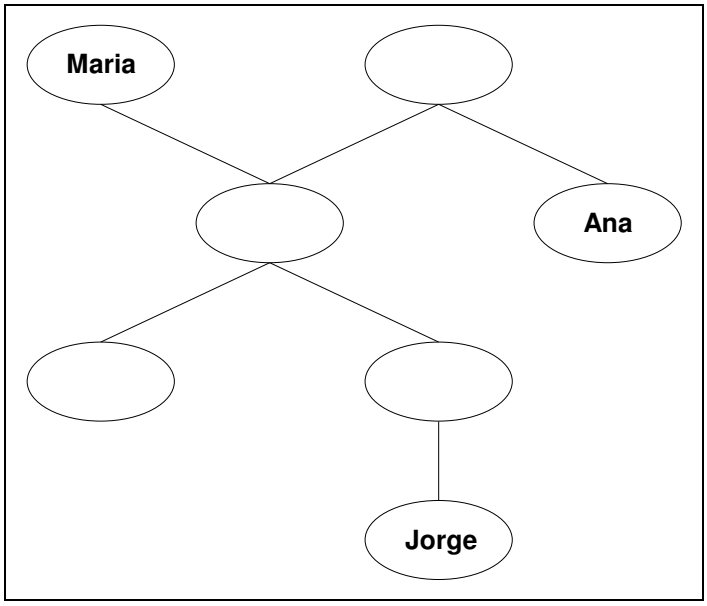

Figura 1 – Uma árvore genealógica

O programa acima é composto de seis cláusulas, cada uma das quais denota um fato acerca da relação progenitor e é encerrada por um ponto (.). Se o programa for submetido a um sistema Prolog, este será capaz de responder algumas questões sobre a relação ali representada [PAL97] [WAT90]. Por exemplo: "José é progenitor de Íris?". Uma consulta como essa deve ser formulada ao sistema precedida por "?-". Esta combinação de sinais denota que se está formulando uma pergunta, permitindo consultas ao sistema Prolog. Como há um fato no programa declarando explicitamente que José é o progenitor de Íris, o sistema responde "sim".

```
?- progenitor(josé, íris).
sim
```

Uma outra questão poderia ser: “Ana é um dos progenitores de Jorge?”. Nesse caso o sistema responde “não”, porque não há nenhuma cláusula no programa que permita deduzir tal fato. Uma resposta a uma consulta pode ser positiva ou negativa, dependendo se o objetivo correspondente foi alcançado ou não. No primeiro caso diz-se que a consulta foi bem sucedida e no segundo diz-se que a consulta falhou [PAL97].

```
?- progenitor(ana, jorge).
não
```

A questão “Luís é progenitor de Maria?” também obteria a resposta “não”, porque o programa nem sequer conhece alguém com o nome Luís.

```
?- progenitor(luís, maria).
não
```

Perguntas mais interessantes podem também ser formuladas, por exemplo: “Quem é progenitor de Íris?”. Para fazer isso introduz-se uma variável, por exemplo “X” na posição do argumento correspondente ao progenitor de Íris. Desta forma o sistema não se limitará a responder “sim” ou “não”, mas irá procurar e informar, caso for encontrado, um valor de X que torne a cláusula “X é progenitor de Íris” verdadeira.

```
?- progenitor(X, íris).
X=josé
```

Da mesma forma a questão “Quem são os filhos de José?” pode ser formulada com a introdução de uma variável na posição do argumento correspondente ao filho de José. Neste caso, mais de uma resposta verdadeira pode ser encontrada. Se várias respostas satisfizerem a uma consulta então o sistema Prolog irá fornecer a primeira que encontrar e aguardar manifestação por parte do usuário. Se este desejar outras soluções deve digitar um ponto-e-vírgula (;), do contrário digita um ponto (.), o que informa ao sistema que a solução fornecida é suficiente.

```
?-progenitor(josé, X).
X=júlia;
X=íris;
não
```

Neste caso, a última resposta obtida foi “não” significando que todas as soluções já foram fornecidas. Uma questão mais geral para o programa seria: “Quem é progenitor de quem?”, ou com outra formulação: “Encontre X e Y tal que X é progenitor de Y”. O sistema,

em resposta, irá fornecer todos os pares progenitor-filho até que estes se esgotem ou até que se resolva encerrar a apresentação de novas soluções, digitando ".". No exemplo a seguir são mostradas apenas as três primeiras soluções encontradas.

```
?- progenitor(X, Y).
X=maria Y=josé;
X=joão  Y=josé;
X=joão  Y=ana.
```

Pode-se formular questões ainda mais complexas ao programa, como "Quem são os avós de Jorge?". Como este programa não possui diretamente a relação avô, a consulta precisa ser dividida em duas etapas: (1) quem é o progenitor (Y) de Jorge e (2) quem é progenitor (X) de Y. Consultas ao sistema são constituídas por um ou mais objetivos, cuja seqüência denota a sua conjunção [PAL97]. Neste caso a consulta em Prolog é escrita como uma seqüência de duas consultas simples, cuja leitura pode ser: "Encontre X e Y tais que X é progenitor de Y e Y é progenitor de Jorge".

```
?- progenitor(X, Y), progenitor(Y, jorge).
X=josé  Y=íris
```

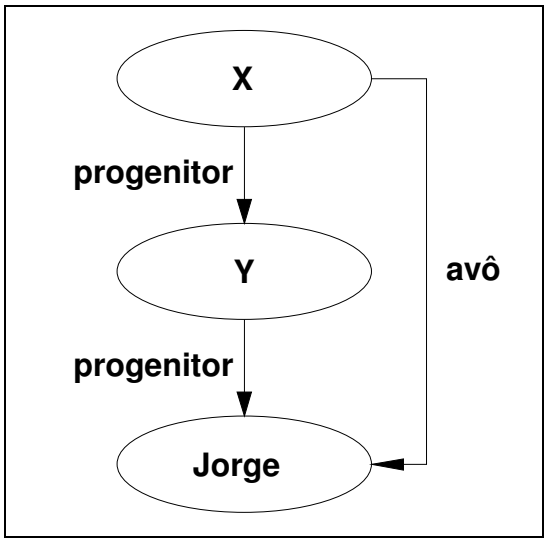

Figura 2 – A relação avó em função de progenitor

Pode-se observar que ao mudar a ordem das consultas na composição, o significado lógico permanece o mesmo e, consequentemente, a resposta também é a mesma, ou seja, foi feita uma alteração sintática que não alterou a semântica da consulta.

```
?- progenitor(Y, jorge), progenitor(X, Y).
X=josé  Y=júlia
```

Ainda uma outra pergunta poderia ser: “José e Ana possuem algum progenitor em comum?”. Novamente é necessário decompor a questão em duas etapas, formulando-a alternativamente como: “Encontre um X tal que X seja simultaneamente progenitor de José e Ana”.

```
?- progenitor(X, josé), progenitor(X, ana).
X=joão
```

5.2 REGRAS

Os programas Prolog podem ser ampliados pela simples adição de novas cláusulas [PAL97]. Desta forma, pode-se ampliar o programa da árvore genealógica, adicionando informações sobre o sexo das pessoas ali representadas, simplesmente acrescentando os seguintes fatos:

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

As relações introduzidas no programa são “masculino” e “feminino”. Tais relações são unárias, isto é, possuem um único argumento. Uma relação binária, como progenitor, é definida entre pares de objetos, enquanto que as relações unárias podem ser usadas para declarar propriedades simples desses objetos [PAL97] [WIL93]. A primeira cláusula unária de relação masculino pode ser lida como: “João é do sexo masculino”. Poderia ser conveniente declarar a mesma informação presente nas relações unárias masculino e feminino em uma única relação binária “sexo”:

```
sexo(joão, masculino).
sexo(maria, feminino).
sexo(josé, masculino).
...
```

A próxima extensão ao programa será a introdução da relação filho como o inverso da relação progenitor. Pode-se definir a relação filho de modo semelhante à utilizada para definir a relação progenitor, isto é, fornecendo uma lista de fatos, cada um dos quais fazendo referência a um par de pessoas tal que uma seja filho da outra. Por exemplo: `filho(josé, joão)`. Entretanto pode-se definir a relação “filho” fazendo uso do fato de que ela é o inverso da relação progenitor e esta já está definida. Tal alternativa pode ser baseada na

declaração lógica “Para todo X e Y, Y é filho de X se X é progenitor de Y”. A cláusula correspondente com a mesma leitura é:

```
filho(Y, X) :- progenitor(X, Y).
```

Segundo [PAL97], as variáveis são assumidas como universalmente quantificadas nas regras e nos fatos, e existencialmente quantificadas nas consultas. Desta forma, a cláusula anterior pode ser lida “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”. As regras tem uma parte de conclusão, que é o lado esquerdo da cláusula, e uma parte de condição, que é o lado direito da cláusula [PAL97] [WIL93] [WAT90].

O símbolo “:-” significa “se” e separa a cláusula conclusão, ou cabeça da cláusula, e condição, ou corpo da cláusula, como é mostrado no exemplo 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 [BEN96] [WIL93]. Segundo [PAL97], fatos são cláusulas que só possuem cabeça, enquanto que as consultas só possuem corpo e as regras possuem cabeça e corpo.

```
filho(Y, X) :- progenitor(X, Y).
```

A utilização das regras pelo sistema Prolog pode ser ilustrada pelo seguinte exemplo: deseja-se saber 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. Diz-se que as variáveis X e Y se tornaram instanciadas: X=maria e Y=josé. 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” [PAL97].

Pode-se adicionar mais algumas relações ao programa como, por exemplo, a relação de mãe, que pode ser escrita baseada na 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).
```

Neste exemplo a vírgula entre as duas condições indica a sua conjunção, significando que para satisfazer o corpo da regra, ambas as condições devem ser verdadeiras [PAL97]. A relação avô, apresentada anteriormente na figura 2, pode agora ser definida em Prolog por:

```
avô(X, Z) :- progenitor(X, Y), progenitor(Y, Z).
```

Neste ponto é interessante comentar alguma coisa sobre o layout dos programas Prolog. Estes podem ser escritos quase que com total liberdade, de modo que pode-se inserir espaços e mudar de linha onde e quando melhor se aprouver. Em geral, porém, deseja-se produzir programas de boa aparência, elegantes e sobretudo fáceis de serem lidos. Com essa finalidade, normalmente se prefere escrever a cabeça da cláusula e os objetivos da condição cada um em uma nova linha [PAL97]. Para destacar a conclusão, indenta-se os objetivos. A cláusula avô, por exemplo, seria escrita:

```
avô(X, Z) :-
        progenitor(X, Y),
        progenitor(Y, Z).
```

Para exemplificar mais uma particularidade da linguagem Prolog será adicionada ainda uma última relação ao programa. Uma cláusula para a relação irmã se embasaria na declaração lógica “Para todo X e Y, X é irmã de Y se X e Y possuem um progenitor comum e X é do sexo feminino”. Ou sob a forma de regra Prolog:

```
Irmã(X, Y) :-
        progenitor(Z, X),
        progenitor(Z, Y),
        feminino(X).
```

Deve-se atentar para a forma sob a qual o requisito “X e Y possuem um progenitor comum” foi expressa. A seguinte formulação lógica foi adotada: “Algum Z deve ser progenitor de X e esse mesmo Z deve também ser progenitor de Y”. Uma forma alternativa, porém menos elegante, de representar a mesma condição seria: “Z1 é progenitor de X e Z2 é progenitor e Y e Z1 é igual a Z2”. Se for feita a consulta “Júlia é irmã de Iris?”, o sistema

retornará um “sim” como resposta. Poderia-se então concluir que a relação irmã, conforme anteriormente definida, funciona corretamente, entretanto há uma falha muito sutil que se revela quando pergunta-se: “Quem é irmã de Íris?”. O sistema irá fornecer duas respostas:

```
?- irmã(X, íris).
X=júlia;
X=íris
```

Íris seria considerada irmã de si própria. Isso não é certamente o que se tinha em mente na definição de irmã, entretanto, de acordo com a regra formulada, a resposta obtida pelo sistema é perfeitamente lógica. A regra sobre irmãs não menciona que X e Y não devem ser os mesmos para que X seja irmã de Y. Como isso não foi requerido, o sistema assume que X e Y podem denotar a mesma pessoa e irá achar que toda pessoa do sexo feminino que possui um progenitor é irmã de si própria. Para corrigir esta distorção é necessário acrescentar a condição de que X e Y devem ser diferentes. Isso pode ser feito de diversas maneiras. Por enquanto será assumido que uma relação diferente(X, Y) seja reconhecida pelo sistema como verdadeira se e somente se X e Y não forem iguais. A regra para a relação irmã fica então definida por:

```
irmã(X, Y) :-
        progenitor(Z, X),
        progenitor(Z, Y),
        feminino(X),
        diferente(X, Y).
```

5.3 CONSTRUÇÕES RECURSIVAS

Será adicionado ao programa a relação “antepassado”, que será definida a partir da relação progenitor. A definição necessita ser expressa por meio de duas regras, a primeira das quais definirá os antepassados diretos 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 como é ilustrado na Figura 3. Na árvore genealógica da Figura 1, João é antepassado direto de Ana e antepassado indireto de Júlia.

A primeira regra que define os antepassados diretos é 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).
```

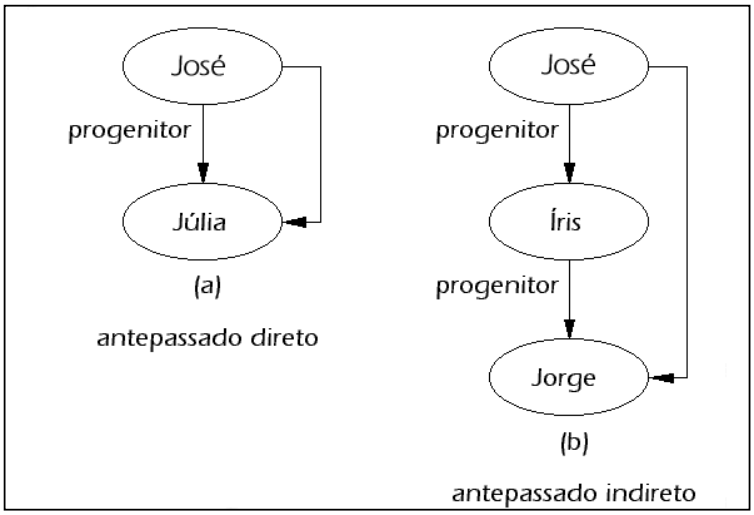

Figura 3 – Exemplos da relação antepassado.

Por outro lado, a segunda regra é mais complexa, 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).
...
```

O resultado seria um programa muito grande e que funcionaria somente 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 denominada recursivo [PAL97]: “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. Rescrevendo as duas tem-se:

```
antepassado(X, Z) :-
        progenitor(X, Z).
antepassado(X, Z) :-
        progenitor(X, Y),
        antepassado(Y, Z).
```

Tal definição pode causar certa surpresa, tendo em vista a seguinte 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, o que fica mais claro pela observação da Figura 4. Por outro lado o sistema Prolog deve muito do seu potencial de expressividade à capacidade intrínseca que possui de utilizar facilmente definição recursivas. O uso de recursão é uma das principais características herdadas da lógica pela linguagem Prolog [PAL97] [WAT90].

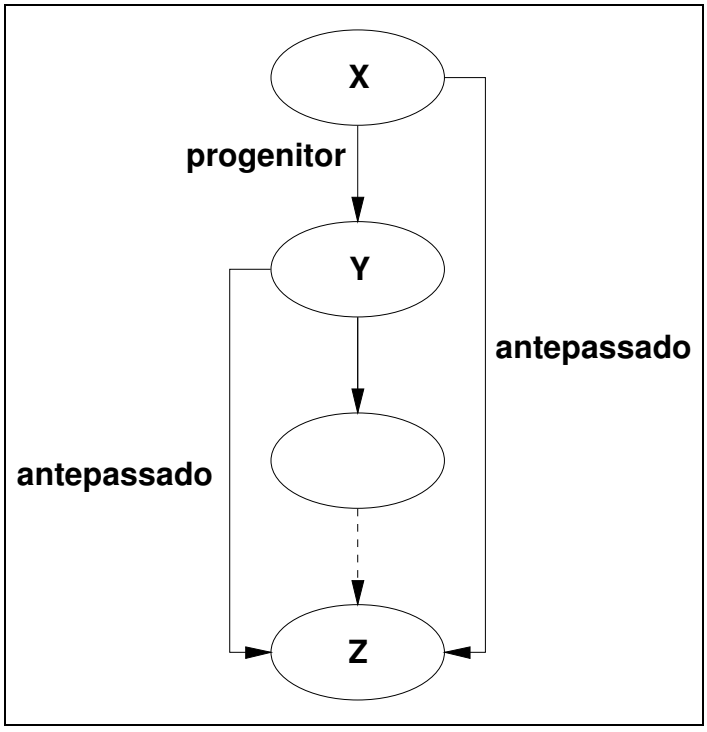

Figura 4 – Formulação recursiva da relação antepassado

Há ainda uma questão importante a ser respondida: Como realmente o sistema Prolog utiliza o programa para encontrar as informações procuradas? Uma explicação informal será fornecida na próxima seção, antes porém, todas as partes do programa que foi sendo gradualmente ampliado pela adição de novos fatos e regras serão reunidos. A forma final do programa é mostrada na Figura 5. O programa ali apresentado define diversas relações: progenitor, masculino, feminino, antepassado, etc. A relação antepassado, por exemplo, é definida por meio de duas cláusulas. Diz-se que cada uma delas é sobre a relação antepassado. Algumas vezes pode ser conveniente considerar o conjunto completo de cláusulas sobre a mesma relação. Tal conjunto de cláusulas é denominado um predicado.

```
progenitor(maria, josé).          % Maria é progenitor de José.
progenitor(joão, josé).
progenitor(joão, ana).
progenitor(josé, júlia).
progenitor(josé, íris).
progenitor(íris, jorge).

masculino(joão).                  % João é do sexo masculino.
masculino(josé).
masculino(jorge).

feminino(maria).                  % Maria é do sexo feminino.
feminino(júlia).
feminino(ana).
feminino(íris).

filho(Y, X) :-                    % Y é filho de X se
       progenitor(X, Y).          % X é progenitor de Y.

mãe(X, Y) :-                      % X é mãe de Y se
       progenitor(X, Y),          % X é progenitor de Y e
       feminino(X).               % X é do sexo feminino.

avô(X, Z) :-                      % X é avô de Z se
       progenitor(X, Y),          % X é progenitor de Y e
       progenitor(Y, Z).          % Y é progenitor de Z.

irmã(X, Y) :-                     % X é irmã de Y se
       progenitor(Z, X),          % X tem um progenitor Z, que
       progenitor(Z, Y),          % é também progenitor de Y e
       feminino(X),               % X é do sexo feminino e
       diferente(X, Y).           % X e Y são diferentes.

antepassado(X, Z) :-              % X é antepassado de Z se
       progenitor(X, Z).          % X é progenitor de Z.     [pr1]
antepassado(X, Z) :-              % X é antepassado de Z se
       progenitor(X, Y),          % X é progenitor de Y e
       antepassado(Y, Z).         % Y é antepassado de Z.    [pr2]
```

Figura 5 – Um programa Prolog

Na Figura 5 as duas regras sobre a relação antepassado foram distinguidas com os nomes [pr1] e [pr2] que foram adicionados como comentário ao programa. Tais nomes serão empregados adiante como referência a essas regras. Os comentários que aparecem em um programa são normalmente ignorados pelo sistema Prolog, servindo apenas para melhorar a legibilidade do programa impresso [PAL97]. Os comentários se distinguem do resto do programa por se encontrarem incluídos entre os delimitadores especiais "/*" e "*/". Um outro método, mas conveniente para comentários curtos, utiliza o caracter de percentual "%": todo o texto informado entre o "%" e o final da linha é interpretado como comentário. Por exemplo:

```
/* Isto é um comentário */
% E isto também.
```

5.4 CONSULTAS

Uma consulta em Prolog é sempre uma seqüência composta por um ou mais objetivos. Para obter a resposta o sistema Prolog tenta satisfazer todos os objetivos que compõem a consulta, interpretando-os como uma conjunção. Satisfazer um objetivo significa demonstrar que esse objetivo é verdadeiro, assumindo que as relações que o implicam são verdadeiras no contexto do programa [WIL93]. Se a questão também contém variáveis, o sistema Prolog deverá encontrar ainda os objetos particulares que, atribuídos às variáveis, satisfazem a todos os sub-objetivos propostos na consulta. A particular instanciação das variáveis com os objetos que tornam o objetivo verdadeiro é então apresentada ao usuário. Se não for possível encontrar, no contexto do programa, nenhuma instanciação comum de suas variáveis que permita derivar algum dos sub-objetivos propostos então a resposta será "não" [PAL97].

Uma visão apropriada da interpretação de um programa Prolog em termos matemáticos é a seguinte: o sistema Prolog aceita os fatos e regras como um conjunto de axiomas e a consulta do usuário como um teorema a ser provado. A tarefa do sistema é demonstrar que o teorema pode ser provado com base nos axiomas representados pelo conjunto das cláusulas que constituem o programa [PAL97] [WAT90]. Essa visão será ilustrada com um exemplo clássico da lógica de Aristóteles. Os axiomas: "todos os homens são falíveis" e "Sócrates é um homem" derivam logicamente o teorema: "Sócrates é falível". O primeiro axioma pode ser rescrito como "Para todo X, se X é um homem então X é falível". Nessa mesma linha o exemplo pode ser escrito em Prolog como se segue:

```
falível(X) :-
  homem(X).
```

```
homem(sócrates).

?- falível(X).
X=sócrates
```

Um exemplo mais complexo extraído do programa apresentado na Figura 5 é:

```
?- antepassado(joão, íris).
```

Sabe-se que o progenitor(josé, íris) é um fato. Usando esse fato e a regra [pr1], pode-se concluir antepassado(josé, íris). Este é um fato derivado, pois não pode ser encontrado explícito no programa, mas pode ser derivado a partir dos fatos e regras ali presentes. Um passo de inferência como esse pode ser escrito em uma forma mais complexa como: progenitor(josé, íris) $\Rightarrow$ antepassado(josé, íris), que pode ser lido assim: "de progenitor(josé, íris) segue, pela regra [pr1] que antepassado(josé, íris)". Além disso sabe-se que progenitor(joão, josé) é um fato. Usando este fato e o fato derivado, antepassado(josé, íris), pode-se concluir, pela regra [pr2], que o objetivo proposto, antepassado(joão, íris) é verdadeiro [PAL97].

Mostrou-se assim o que pode ser uma seqüência de passos de inferência usada para satisfazer um objetivo. Tal seqüência denomina-se seqüência de prova. A extração de uma seqüência de prova do contexto formado por um programa e uma consulta é obtida pelo sistema na ordem inversa da empregada acima. Ao invés de iniciar a inferência a partir dos fatos, o Prolog começa com os objetivos e, usando as regras, substitui os objetivos correntes por novos objetivos até que estes se tornem fatos [PAL97] [WAT90].

Dada por exemplo a questão: "João é antepassado de Íris?", o sistema tenta encontrar uma cláusula no programa a partir da qual o objetivo seja conseqüência imediata. Obviamente, as únicas cláusulas relevantes para essa finalidade são [pr1] e [pr2], que são sobre a relação antepassado, porque são as únicas cujas cabeças podem ser unificadas com o objetivo formulado. Tais cláusulas representam dois caminhos alternativos que o sistema pode seguir. Inicialmente o Prolog irá tentar a que aparece em primeiro lugar no programa:

```
antepassado(X, Z) :- progenitor(X, Z).
```

Uma vez que o objetivo é antepassado(joão, íris), as variáveis na regra devem ser instanciadas por X=joão e Y=íris. O objetivo inicial, antepassado(joão, íris) é então substituído por um novo objetivo:

```
progenitor(joão, íris)
```

Não há, entretanto, nenhuma cláusula no programa cuja cabeça possa ser unificada com progenitor(joão, íris), logo este objetivo falha. Então o Prolog retorna ao objetivo original (*backtracking*) para tentar um caminho alternativo que permita derivar o objetivo antepassado(joão, íris). A regra [pr2] é então tentada:

```
antepassado(X, Z) :-
        progenitor(X, Y),
        antepassado(Y, Z).
```

Como anteriormente, as variáveis X e Z são instanciadas para joão e íris, respectivamente. A variável Y, entretanto não está instanciada ainda. O objetivo original, antepassado(joão, íris) é então substituído por dois novos objetivos derivados por meio da regra [pr2]:

```
progenitor(joão, Y), antepassado(Y, íris).
```

Encontrando-se agora face a dois objetivos, o sistema tenta satisfazê-los na ordem em que estão formulados. O primeiro deles é fácil: progenitor(joão, Y) pode ser unificado com dois fatos do programa: progenitor(joão, josé) e progenitor(joão, ana). Mais uma vez, o caminho a ser tentado deve corresponder à ordem em que os fatos estão escritos no programa. A variável Y é então instanciada com José nos dois objetivos acima, ficando o primeiro deles imediatamente satisfeito. O objetivo remanescente é então antepassado(josé, íris).

Para satisfazer tal objetivo, a regra [pr1] é mais uma vez empregada. Essa Segunda aplicação de [pr1], entretanto, nada tem a ver com a sua utilização anterior, isto é, o sistema Prolog usa um novo conjunto de variáveis na regra cada vez que esta é aplicada. Para indicar isso as variáveis em [pr1] serão renomeadas nesta nova aplicação, da seguinte maneira:

```
antepassado(X1, Z1) :-
        progenitor(X1, Z1).
```

A cabeça da regra deve então ser unificada como o objetivo corrente, que é antepassado(josé, íris). A instanciação de X1 e Y1 fica: X1=josé e Y1=íris e o objetivo corrente é substituído por: progenitor(josé, íris). Esse objetivo é imediatamente satisfeito, porque aparece no programa um fato. O sistema encontrou então um caminho que lhe permite provar, no contexto oferecido pelo programa dado, o objetivo originalmente formulado, e portanto responde “sim”.

5.5 SIGNIFICADO DOS PROGRAMAS PROLOG

Assume-se que um programa Prolog possua três interpretações semânticas básicas. São elas: interpretação declarativa, procedimental e operacional. Na interpretação declarativa as cláusulas que definem o programa descrevem uma teoria de primeira ordem, na interpretação procedimental as cláusulas são entradas para um método de prova e na interpretação operacional as cláusulas são comandos para um procedimento particular de prova por refutação [PAL97] [WAT90].

A interpretação declarativa permite que o programador modele um dado problema através de assertivas acerca dos objetos do universo de discurso, simplificando a tarefa de programação Prolog em relação a outras linguagens tipicamente procedimentais como Pascal ou C. A interpretação procedimental permite que o programador identifique e descreva o problema pela redução do mesmo a subproblemas, através da definição de uma série de chamadas a procedimentos. Por fim, a interpretação operacional reintroduz a idéia de controle da execução, que é irrelevante do ponto de vista da semântica declarativa, através da ordenação das cláusulas e dos objetivos dentro das cláusulas em um programa Prolog. Essa última interpretação é semelhante à semântica operacional de muitas linguagens convencionais de programação, e deve ser considerada, principalmente em grandes programas, por questões de eficiência [PAL97].

Essa habilidade específica do Prolog, de trabalhar em detalhes procedimentais de ação sobre o seu próprio domínio de definição, isto é, a capacidade de ser meta-programado, é uma das principais vantagens da linguagem. Ela encoraja o programador a considerar a semântica declarativa de seus programas de modo independente dos seus significados procedimental e operacional. Uma vez que os resultados do programa são considerados pelo seu significado declarativo, isto deveria ser suficiente para a codificação de programas Prolog. Isso possui grande importância prática, pois os aspectos declarativos do programa são em geral mais fáceis de entender do que os detalhes operacionais. Para tirar vantagem dessa característica o programador deve se concentrar principalmente no significado declarativo e evitar os detalhes de execução. Entretanto, essa interpretação nem sempre é suficiente. Em problemas de maior complexidade os aspectos operacionais não podem ser ignorados. Apesar de tudo, a atribuição de significado declarativo aos programas Prolog deve ser estimulada, na extensão limitada por suas restrições de ordem prática [PAL97].

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: o interpretador, que é o cérebro de um ambiente Prolog. A partir de uma consulta do usuário ele faz as inferências lógicas necessárias e apresenta como resultado uma resposta objetiva. Quando a consulta possui variáveis, ele também retorna o valor das mesmas que tornam a consulta verdadeira. O interpretador é conhecido como motor de inferências por ser justamente ele que faz as inferências lógicas, permitindo-se chegar a conclusões a partir de fatos e regras. O que ele executa é basicamente o cálculo das proposições e o cálculo dos predicados descritos no capítulo 4 e demonstrados no capítulo 5.

Como é necessário alimentar o interpretador com o programa do usuário, foi implementado também o analisador léxico, sintático e semântico para a linguagem Prolog, que lê o programa de um arquivo texto e monta as estruturas em memória necessárias para a execução das consultas. Estes analisadores são também utilizados para interpretar as próprias consultas do usuário. Além destes, foi construída uma interface que permite a interação do usuário, que facilita a entrada do programa e das consultas e mostra os resultados de uma forma visual.

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 6.

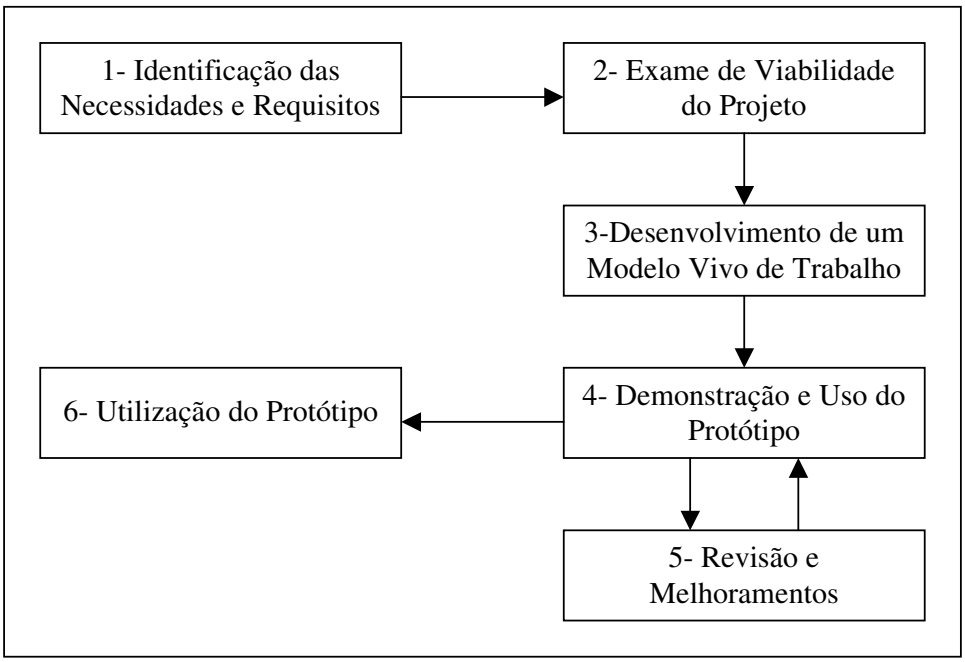

Figura 6 – Metodologia genérica de prototipação

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.

6.2 ESPECIFICAÇÃO

O interpretador possui duas entradas, ambas por parte do usuário: o programa e as consultas. Possui também duas saídas, que são em função das consultas: a resposta objetiva e o conteúdo das variáveis. Isto pode ser observado no diagrama da figura 7. O usuário fornece, através da interface, um programa que está num arquivo texto. Este arquivo é interpretado e gera uma base de dados na memória do interpretador. O usuário então pode entrar com uma cláusula de consulta, que vai ser novamente interpretada e gerará uma meta a ser provada pelo interpretador. O interpretador segue fazendo as inferências necessárias para tentar provar a

cláusula de consulta. Se ele for bem sucedido retornará uma resposta objetiva positiva, caso contrário, a resposta objetiva será negativa. Se a resposta objetiva for positiva e a consulta do usuário possuir variáveis, os valores que tornam a cláusula de consulta verdadeira são também mostradas ao usuário.

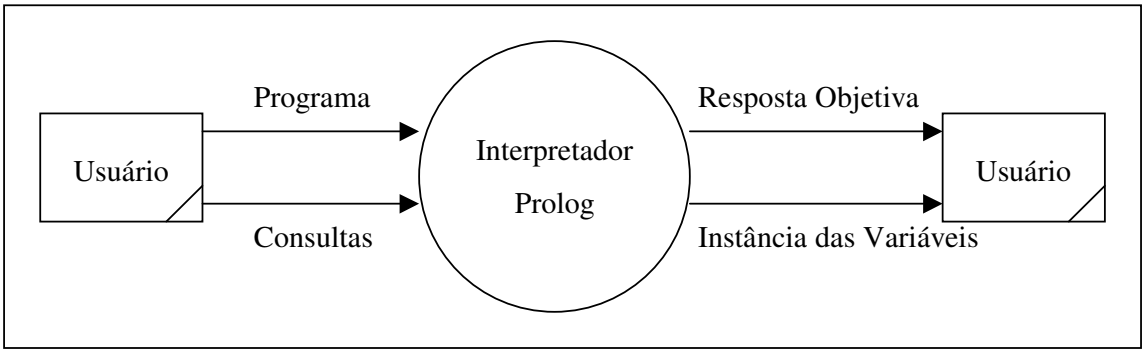

Figura 7 – Diagrama de contexto do interpretador Prolog

Pode-se dividir estas tarefas em três eventos: leitura do programa, pedido de consulta e processamento para obtenção da resposta, sendo que os dois últimos eventos ocorrem juntos mas serão separados para facilitar a representação. A leitura do programa pode ser detalhada pelo diagrama de fluxo de dados representado na figura 8. O usuário fornece ao interpretador um arquivo texto contendo o programa, que é composto por um conjunto de fatos e regras. Este arquivo será interpretado pelos analisadores léxico, sintático e semântico, gerando uma base de dados em memória, em um formato conhecido pelo motor de inferência, de forma que este possa, em seguida, fazer inferências lógicas sobre esta base de dados.

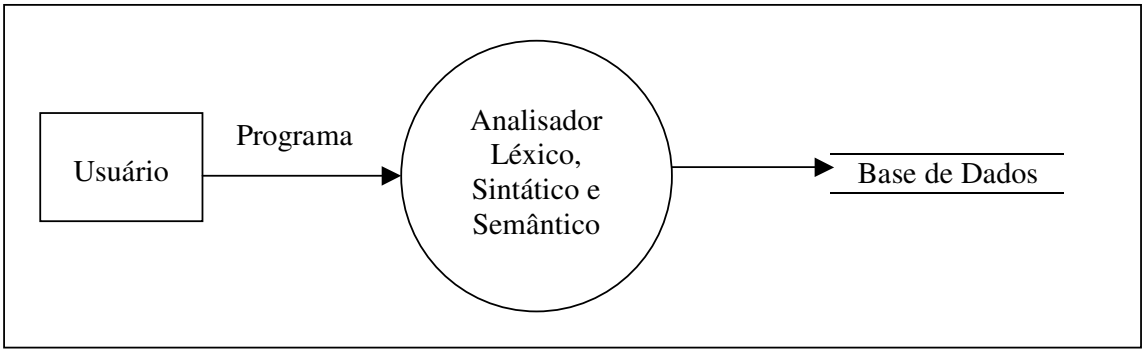

Figura 8 – Diagrama de fluxo de dados da leitura do programa

No pedido de consulta, como pode ser visto na figura 9, o usuário novamente fornece uma cláusula que precisa ser interpretada pelos analisadores léxico, sintático e semântico, gerando uma lista de metas em memória. E em seguida, o motor de inferências utiliza a base

de dados e as metas para fazer as inferências necessárias e apresentar um resultado objetivo, que é simplesmente a resposta “sim” ou “não”. Se a consulta possuir variáveis retorna a lista dos valores ou instância das variáveis, que tornam a cláusula de consulta verdadeira. Isto está representado na figura 10.

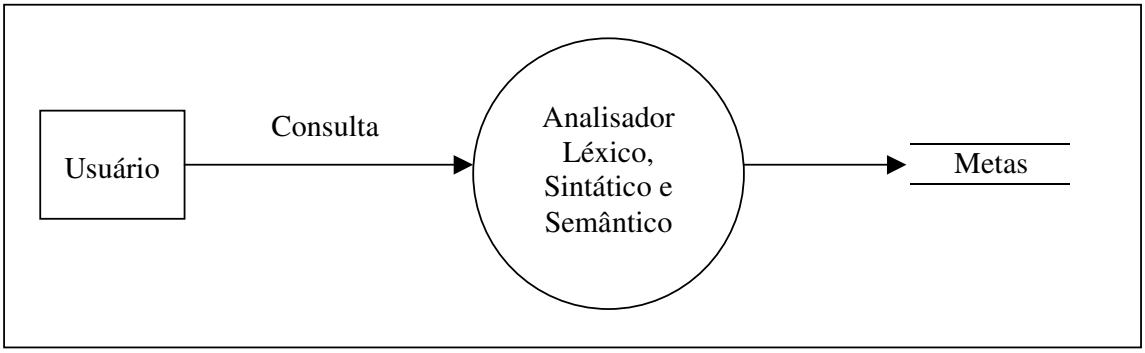

Figura 9 – Diagrama de fluxo de dados do pedido de consulta

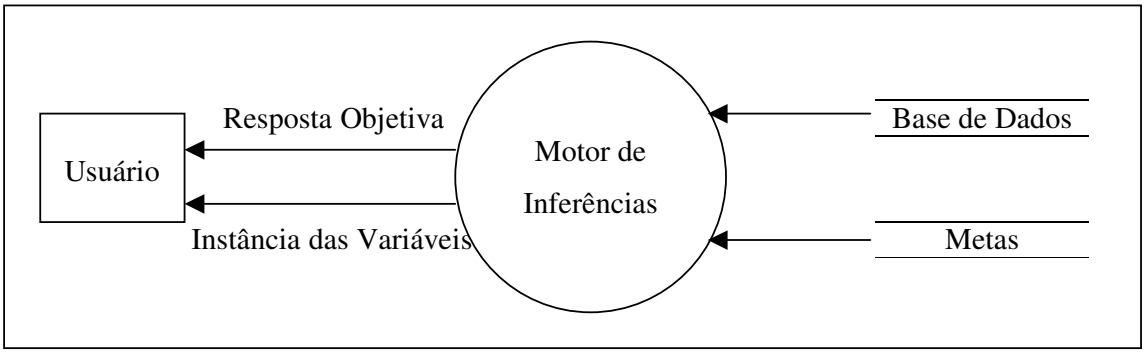

Figura 10 – Diagrama de fluxo de dados da resposta a consulta.

Como mostrado nos diagramas das figuras 7 e 8, as entradas do usuário precisam ser interpretadas. Existe, portanto, uma linguagem na qual o usuário escreve o programa e as cláusulas de consulta, e esta linguagem possui uma sintaxe definida. A sintaxe utilizada no protótipo é a sintaxe do Prolog de Edimburgo, que se tornou um padrão entre vários ambientes Prolog e é descrita em [CLO94] e [MAI88]. Para interpretar esta linguagem é necessário utilizar analisadores léxico, sintático e semântico. Estes recebem as entradas e montam as estruturas dos dados em memória. Em [AHO87] estão descritos todos os passos para se construir os algoritmos para implementar estes analisadores a partir de uma especificação da sintaxe da linguagem. Como não é objetivo deste trabalho abordar a área de compiladores, apenas será registrado aqui a sintaxe da linguagem Prolog do interpretador, utilizando a notação BNF:

```
< tuplas >          →  ∈
< tuplas >          →  < tupla > < tuplas >
< tupla >           →  < predicado > < corpo >.
< predicado >       →  < lit_predicado > < lst_parâmetros >
< lst_parâmetros >  →  ∈
< lst_parâmetros >  →  ( < lst_argumentos > )
< lst_argumentos >  →  < termo > < termos >
< termo >           →  < lit_constante >
< termo >           →  < lit_variável >
< termos >          →  , < lst_argumentos >
< termos >          →  ∈
< corpo >           →  :- < lst_predicados >
< corpo >           →  ∈
< lst_predicados >  →  < predicado > < predicados >
< predicados >      →  , < lst_predicados >
< predicados >      →  ∈
```

Nesta sintaxe, os termos `< lit_predicado >` e `< lit_constante >` são literais que iniciam com uma letra minúscula seguida ou não de outras letras maiúsculas, minúsculas ou dígitos, e `< lit_variável >` é um literal que inicia com uma letra maiúscula seguida ou não de outras letras maiúsculas, minúsculas ou dígitos. Os termos que estão repetidos possuem mais de um valor correto. O símbolo ∈ significa vazio, ou seja, onde ele aparece não precisa haver nenhum texto para que o termo seja reconhecido como válido.

Um programa é representado pelo termo `< tuplas >`, enquanto uma cláusula de consulta é representada pelo termo `< lst_predicados >`. Então pode-se adicionar mais dois termos a sintaxe para facilitar a diferenciação entre os programas e as cláusulas de consulta:

```
< programa >          →  < tuplas >
< clausula_consulta > →  < lst_predicados >
```

Esta sintaxe permite que programas completos sejam escritos, porém ela não é a sintaxe completa do padrão Edimburgo. Apenas um subconjunto foi implementado. Como exemplo será mostrado um programa válido, que é uma parte do programa exemplo do capítulo 5.

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

```
filho(Y, X) :- progenitor(X, Y).
mãe(X, Y) :- progenitor(X, Y), feminino(X).
antepassado(X, Z) :- progenitor(X, Z).
antepassado(X, Z) :- progenitor(X, Y), antepassado(Y, Z).
```

O programa, após passar pelo interpretador, fica armazenado na base de dados. Esta base de dados será mais tarde usada pelo motor de inferências para responder as consultas do usuário. É, portanto, importante conhecer a estrutura desta base de dados para facilitar o entendimento do protótipo. Esta estrutura tem uma ligação direta com a sintaxe da linguagem que foi apresentada e está aqui representadas na linguagem Pascal.

A principal estrutura é uma tabela de todos os símbolos do programa, classificados como predicados, constantes ou variáveis (`pred, cnst, vble`). Todos os literais `< lit_predicado >`, `< lit_constante >` e `< lit_variável >` diferentes encontrados no programa são armazenados nesta tabela em memória. Para os literais que são predicados existe ainda duas informações extras: o número de argumentos e um ponteiro para a primeira cláusula daquele predicado. Esta tabela é armazenada na memória em forma de vetor e o índice deste vetor é representado pelo tipo `TSymTabIndex`, que inicia em 0 e possui um limite fixo, ou seja, o número de símbolos diferentes de um programa é limitado em 512.

```
MaxSyms = 511;

TSymTabIndex = 0..MaxSyms;

TSType = (pred, cnst, vble);
TSTEntry = record
  Name: String;
  case SType: TSType of
    pred: (Arity: Integer;
           Clauses: PClauseList);
    cnst: ();
    vble: ();
end;
```

A segunda estrutura, que já foi utilizada na estrutura anterior, é um registro para armazenar uma cláusula. Esta estrutura separa a cláusula em cabeça e corpo. Se por exemplo, deseja-se armazenar a cláusula `p(X, Y) :- q(Y, Z), r(Z, X)`, a parte `p(X, Y)` é armazenada em *Head* e a parte `q(Y, Z), r(Z, X)` é armazenada em *Body*. Esta estrutura é representada na sintaxe pelo termo `< tupla >`. Possui ainda um ponteiro que aponta para a próxima cláusula, permitindo que se tenha uma lista encadeada de cláusulas.

```
PClauseList = ^TClauseRec;
TClauseRec = record
```

```
  Head: PLit;
  Body: PLitList;
  NxtClause: PClauseList;
end;
```

A estrutura mais simples é a estrutura utilizada para armazenar um literal. Esta estrutura é utilizada para armazenar a cabeça da cláusula na estrutura anterior e é representada na sintaxe pelo termo `< predicado >`. Ela é formada basicamente por um vetor, onde a primeira posição guarda o índice do símbolo do predicado e as demais guardam os índices dos argumentos. Por exemplo, para armazenar o literal `p(X, Y)` nesta estrutura, o índice de p na tabela dos símbolos do programa será armazenado na posição 0 do vetor e o índice de X e Y nas posições 1 e 2 respectivamente. E aqui está a segunda limitação: no protótipo são aceitos apenas literais com até 8 argumentos.

```
MaxItems = 8

TLitRecIndex = 0..MaxItems;

PLit = ^TLitRec;
TLitRec = record
  Items: array [TLitRecIndex] of TSymTabIndex;
end;
```

A última estrutura é a estrutura que armazena uma lista de literais. Ela é utilizada para armazenar o corpo das cláusulas. É composta por um apontador para a estrutura anterior e um apontador para o próximo literal, formando assim uma lista encadeada de literais. É representada na sintaxe pelo termo `< lst_predicados >`.

```
PLitList = ^TLits;
TLits = record
  Rest: PLitList;
  Lit: PLit;
end;
```

Estas são as estruturas utilizadas para armazenar um programa na memória. Retornando à tabela de símbolos pode-se perceber que se houver várias regras para um mesmo predicado elas serão armazenas abaixo de um único item desta tabela. Para que fique mais claro como estas estruturas são utilizadas, será demonstrado o seu conteúdo após a inserção da seguinte cláusula:

```
P(X, Y, Y) :- q(X, a), r(X, Y).
```

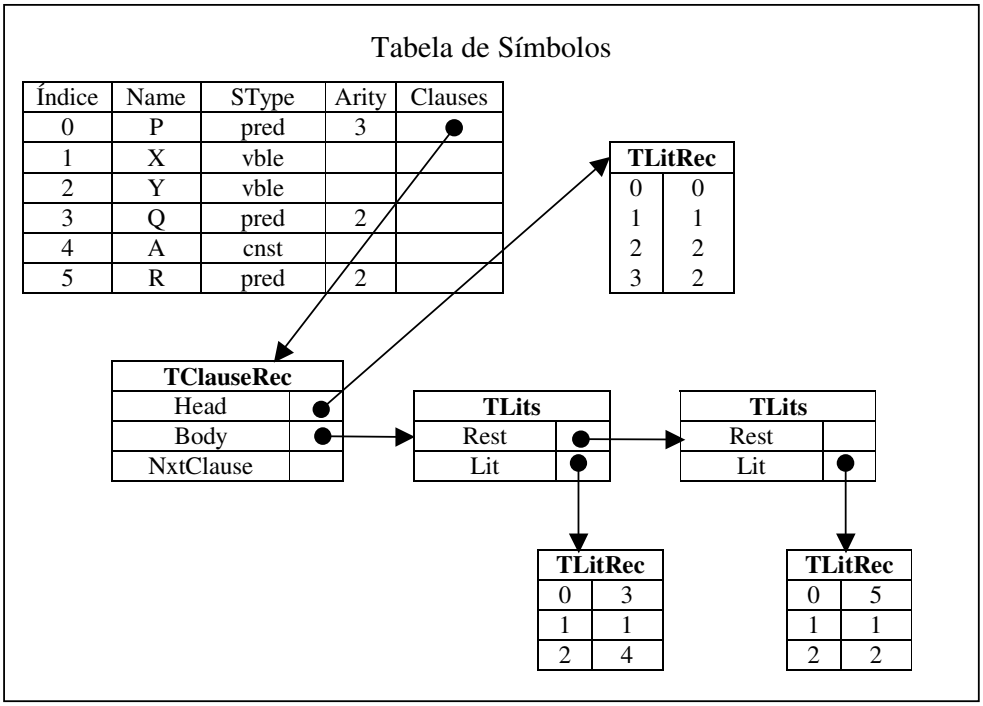

Figura 11 – Estrutura da base de dados após a inserção de uma cláusula.

6.3 IMPLEMENTAÇÃO

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 de 233 MHz.

O Delphi possui muitas classes implementadas que facilitam a programação, porém elas foram utilizadas apenas na implementação da interface. O motor de inferências foi escrito 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.

A principal rotina do motor de inferências é a rotina responsável por demonstrar se a consulta pode ou não ser deduzida do conjunto de fatos e regras contidos no programa Prolog. Esta rotina recebe uma lista de literais, que formam a consulta, e o seu resultado é um valor verdadeiro ou falso. Esta rotina está transcrita na figura 12.

```
function Establish(GoalList: PLList): Boolean;
var
  NextCl: PClauseList;
  LocalSubs: TBAIndex;
  TrailSave: TTrailIndex;
begin
  if IsEmpty(GoalList) then
  begin
    Result := True;
    Exit;
  end;
  GoalMol := First(GoalList);
  NextCl := SymTab[PredSym(GoalMol.Atom)].Clauses;
  LocalSubs := NewVarIndex;
  TrailSave := TrailTop;
  while NextCl <> nil do
  begin
    NewVars(NextCl);
    HeadMol.Atom := NextCl^.Head;
    HeadMol.LSub := LocalSubs;
    if Match(HeadMol, GoalMol) then
    begin
      NewGL := LConcat_Rest(NextCl^.Body, LocalSubs, GoalList);
      if Establish(NewGL) then
      begin
        Result := True;
        Exit;
      end;
    end;
    Restore(TrailSave, LocalSubs);
    NextCl := NextCl^.NxtClause;
  end;
  Result := False;
end;
```

Figura 12 – Principal rotina do motor de inferências

O parâmetro que é recebido pela rotina é a lista de literais. Por exemplo, uma entrada válida seria `p(a, b), p(b, c)`. Se esta lista não contém nenhum literal, o resultado é positivo e a rotina retorna *True*. Tentará se provar cada literal da entrada a partir da lista das cláusulas que está ligada ao símbolo do literal na tabela de símbolos. Qualquer cláusula da lista que possuir uma instância válida no programa e puder ser provada irá tornar o literal verdadeiro, e a partir de então tentará se provar o próximo literal. A rotina *Match* é responsável pela unificação, ou seja, pela prova ou resolução da cláusula. A rotina *Establish* é recursiva. Chamando a si mesma faz com que ela busque a solução em forma de árvore, verificando todas as possibilidades de uma cláusula ser provada.

6.4 UTILIZAÇÃO E OPERAÇÃO

O protótipo possui basicamente uma tela, que é subdividida em duas áreas principais: a área de entrada e a área de saída (*input* e *output*). A área de entrada é o onde o usuário digita as suas cláusulas de consulta e a área de saída mostra as respostas do programa. A área de entrada localiza-se na parte inferior da tela, como pode ser visto na figura 13.

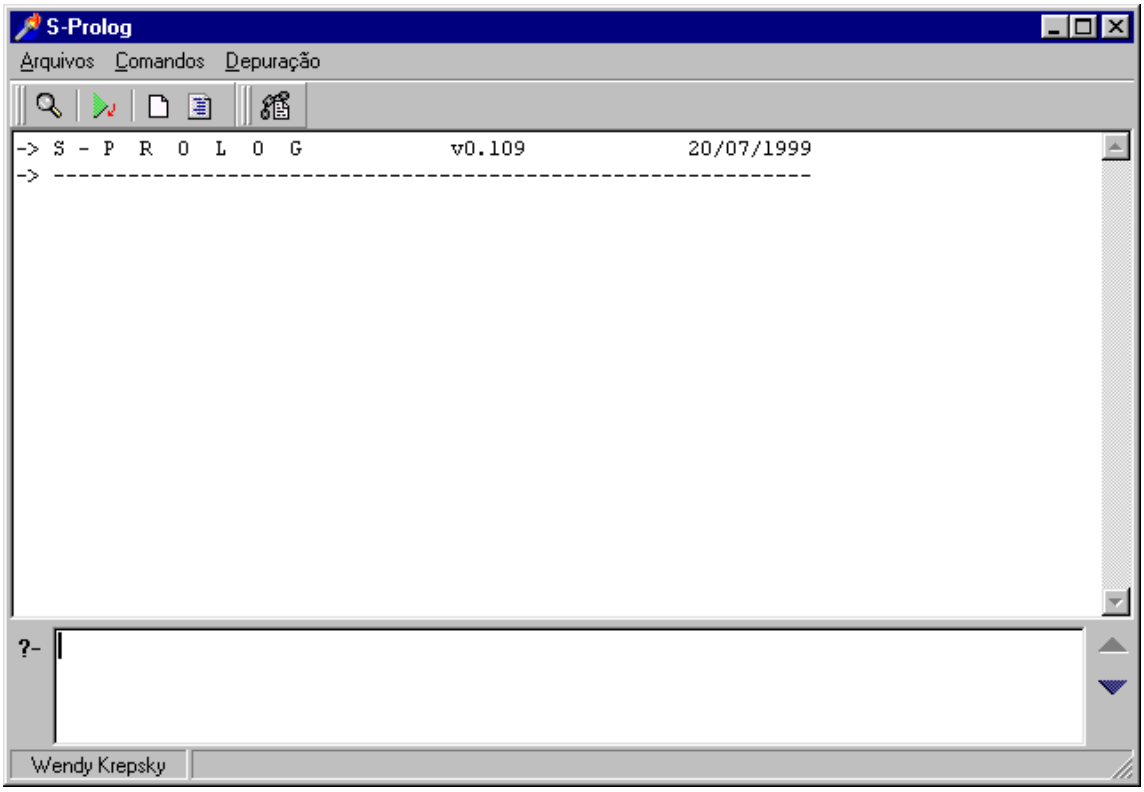

Figura 13 – Tela do protótipo

Os comandos para operação do protótipo são quatro e estão todos na barra de ferramentas. O primeiro botão abre uma caixa de diálogo para que o usuário selecione o arquivo do programa Prolog que será consultado. O segundo executa a consulta que foi digitada pelo usuário. O terceiro limpa a base de dados da memória e o quarto lista o conteúdo desta base na área de saída. Há ainda um quinto botão, que mostra ao usuário a tabela de símbolos do programa que está em memória.

Na figura 14 pode-se observar uma seqüência de passos que foram executados. Um arquivo de programa foi consultado e em seguida listado. As suas cláusulas foram mostradas na área de saída. Em seguida, uma cláusula de consulta foi digitada na área de entrada e a

execução da consulta foi disparada pelo segundo botão. O interpretador retornou uma resposta positiva, que é representada pela palavra “yes”, porque a cláusula de consulta pôde ser provada utilizando-se os fatos e regras que estavam em memória.

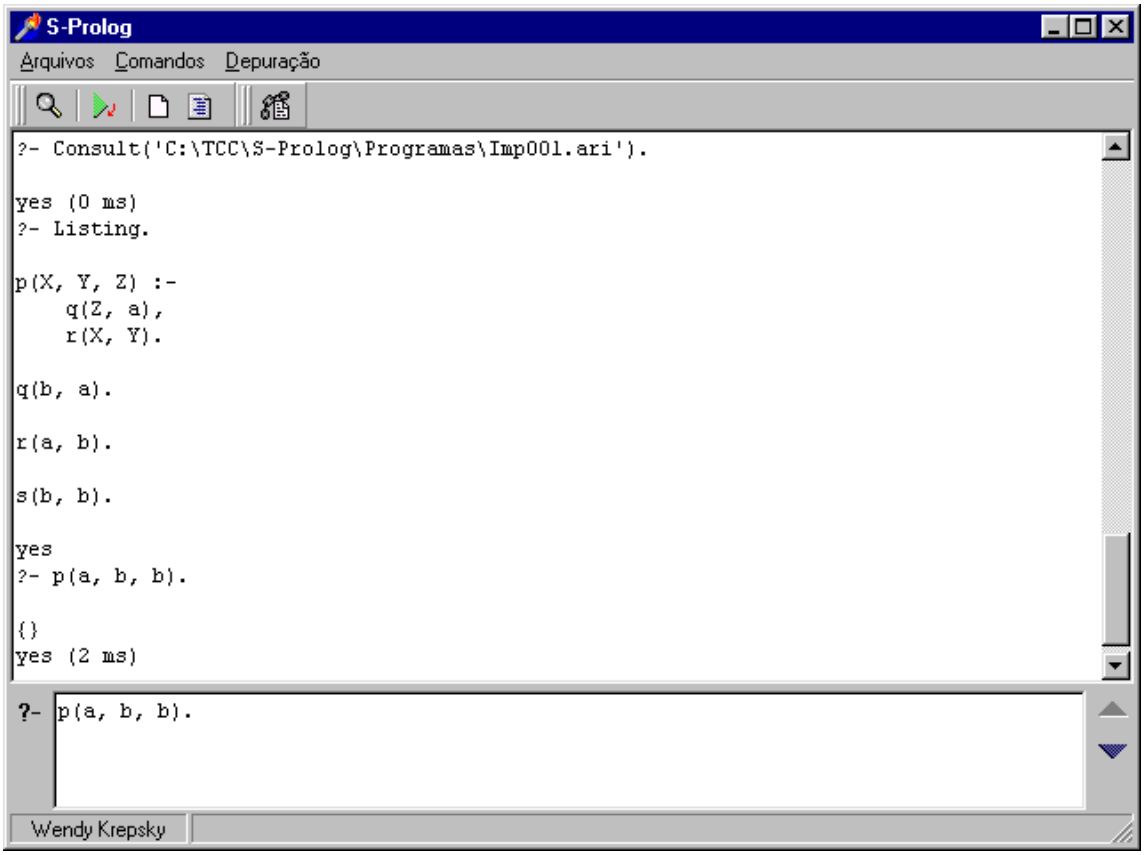

Figura 14 – Exemplo da execução de uma consulta

No menu de depuração existe ainda uma opção que mostra o plano de execução de uma consulta, ou seja, mostra como o interpretador chegou a conclusão. As demais opções do menu são as mesmas que estão disponíveis na barra de ferramentas.

A operação do protótipo é simples pois ele possui poucas funções. Na área de entrada ainda existem duas setas que permitem ao usuário navegar nas consultas já formuladas. Na barra de estado, que se localiza abaixo da área de entrada, são mostradas algumas mensagens de auxílio ao usuário.

7 CONCLUSÃO

O estudo sobre programação lógica iniciou-se pelas áreas de programação de computadores e inteligência artificial, que são as áreas que abrangem a programação lógica. Seguiu com o estudo da própria programação lógica e a sua ligação com a lógica matemática, demonstrando o seu uso através de um ambiente Prolog. Por fim, foi desenvolvido o protótipo do interpretador de um ambiente de programação lógica.

Com o desenvolvimento deste trabalho foi possível compreender melhor o que significa programação lógica e como ela pode ser utilizada como linguagem de programação. O desenvolvimento do protótipo possibilitou o entendimento dos mecanismos utilizados na resolução dos problemas de lógica.

O protótipo atendeu aos requisitos propostos inicialmente. Ele faz inferências lógicas que permitem que conclusões sejam geradas a partir de uma base de fatos e regras, utilizando o cálculo das proposições e o cálculo dos predicados.

As ferramentas escolhidas também trouxeram o resultado esperado. O Delphi se mostrou uma ferramenta eficiente para o tipo de programação exigida na implementação. O estudo do Ambiente Prolog também permitiu que se conhecesse as funções de um interpretador Prolog completo. Os objetivos gerais do trabalho foram alcançados.

7.1 LIMITAÇÕES

Por se tratar de um protótipo, vários itens que fazem parte de um interpretador Prolog não foram implementados:

a) unificação de cláusulas: um interpretador Prolog permite que cláusulas sejam unificadas descobrindo-se o valor das variáveis que as tornam equivalentes;
b) operadores matemáticos: faltam operadores matemáticos como igual, diferente, maior, menor, entre outros;
c) operador *cut*: este operador controla o *backtraking*;
d) todas as respostas possíveis: o interpretador implementado mostra apenas a primeira resposta válida, não havendo como mostrar as respostas alternativas;
e) listas e estruturas: não tem como manipular listas e estruturas.

7.2 EXTENSÕES

Como sugestão para futuros trabalhos propõe-se um estudo mais aprofundado sobre as técnicas de resolução através do cálculo das proposições e cálculo dos predicados. Estas técnicas abrangem o estudo da unificação e principalmente da resolução por métodos de refutação e pelas cláusulas de Horn.

Para o aperfeiçoamento do protótipo, sugere-se implementar as limitações abordadas no item anterior, aproximando-o mais de um interpretador Prolog completo.

REFERÊNCIAS BIBLIOGRÁFICAS

[AHO87] AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compilers: principles, techniques and tools.** Massachusetts : Addison-Wesley, 1987.

[ALB95] ALBUS, James. **AAAI Spring symposium on physical agent architectures** 1995. Endereço eletrônico: http://tommy.jsc.nasa.gov/er/er6/mrl/papers/symposium/albus.txt.

[BAR97] BARRETO, Jorge Muniz. **Inteligência artificial no limiar do século XXI**. Florianópolis : PPP, 1997.

[BEN96] BENDER, Edward A. **Mathematical methods in artificial intelligence.** Washington : IEEE Computer Society, 1996.

[BRA90] BRATKO, Ivan. **Prolog programming for artificial intelligence.** 2 ed. Wokingham : Addison-Wesley, 1990.

[CLO94] CLOCKSIN, W. F.; MELLISH, C. S. **Programming in prolog.** 4 ed. Berlim : Springer-Verlag, 1994.

[GAL97] GALVÃO, Carlos; XAVIER, Mariana; CABRAL, Matheus. **Inteligência artificial: introdução, problemas e paradigmas** 1997. Endereço Eletrônico: http://www.di.ufpe.br/~mcag/icc/index.html.

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

[HEI95] HEINZLE, Roberto. **Protótipo de uma ferramenta para criação de sistemas especialistas baseados em regras de produção**. Florianópolis, 1995. Dissertação de Mestrado. Universidade Federal de Santa Catarina.

[HIN99] HINZ, Marco Antônio; LUCAS, Diogo; GAVIÃO, Wilson. **Paradigma funcional através da linguagem LISP** 1999. Endereço Eletrônico: http://minerva.ufpel.tche.br/~marhinz/index.htm.

[KNU73] KNUTH, Donald E. **Fundamental algorithms.** 2 ed. Massachusetts : Addison-Wesley, 1973. 2ª Edição.

[MAI88] MAIER, David; WARREN, David S. **Computing with logic: logic programming with prolog**. Califórnia : Benjamin Communigs, 1988.

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

[NIL97] NILSSON, Nils J. **Aritificial intelligence: a new synthesis**. San Francisco : Morgan Kaufmann, 1997.

[PAL97] PALAZZO, Luiz A. M. **Introdução à programação PROLOG**. Pelotas : Educat, 1997.

[RAB95] RABUSKE, Renato Antônio. **Inteligência artificial**. Florianópolis : Editora da UFSC, 1995.

[STE86] STERLING, Leon; SHAPIRO, Ehud. **The art of prolog: advanced programming techniques**. 2 ed. Cambridge : MIT, 1986.

[TAF96] TAFNER, Malcon A. XEREZ, Marcos de; RODRIGUES, Ilson W. Filho. **Redes neurais artificiais: introdução e princípios de neurocomputação**. Blumenau : EKO, 1996.

[VAS99] VASCONCELOS, Germano; QUEIROZ, Fausto. **SAPRI – Sistema de aquisição, processamento e reconhecimento de imagens** 1999. Endereço eletrônico: http://www.di.ufpe.br/~sapri.

[WAT90] WATT, David A. **Programming language concepts and paradigms**. New York : Prentice Hall, 1990.

[WIL93] WILSON, Leslie B.; CLARK, Robert G. **Comparative programming languages**. 2 ed. Wokingham : Addison-Wesley, 1993.