UNIVERSIDADE REGIONAL DE BLUMENAU
CENTRO DE CIÊNCIAS EXATAS E NATURAIS
CURSO DE CIÊNCIAS DA COMPUTAÇÃO
(Bacharelado)

Protótipo de Software de Apoio ao Aprendizado da Linguagem de Programação Lógica Prolog

RELATÓRIO DO TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO A UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS DE DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO - BACHARELADO

WAGNER MOREIRA STAHNKE

BLUMENAU, DEZEMBRO DE 1999.

1999/2-39

Protótipo de Software de Apoio ao Aprendizado da Linguagem de Programação Lógica Prolog

WAGNER MOREIRA STAHNKE

ESTE RELATÓRIO, DO TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS DA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIO 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

____

Dalton Solano dos Reis

____

Everaldo Artur Grahl

____

Roberto Heinzle

Agradecimentos

Gostaria de agradecer a todos os professores do curso de Bacharelado em Ciências da Computação da Universidade Regional de Blumenau, por todo conhecimento adquirido, e em especial aos Professores **Cláudio Loesch** do departamento de Matemática e **Roberto Heinzle** do departamento de Sistemas e Computação, pelo apoio e orientação na conduta deste trabalho.

Aos meus Pais, Ivo Stahnke e Luzimar Moreira Stahnke, minha irmã Clara Ana Moreira Stahnke, que sem eles, nada disto teria sido possível. Mesmo não tendo conhecimentos da área de informática, foram de vital importância por me apoiarem nas mais diversas situações, pelas palavras de incentivo e dedicação durante todos estes anos.

A minha esposa, **Tania Karger Stahnke** e a minha filha **Evelin Emanueli Karger Stahnke**, pela paciência, carinho e dedicação em todas as horas.

SUMÁRIO

AGRADECIMENTOS iii

SUMÁRIO..iv

LISTA DE FIGURAS 6

RESUMO viii

ABSTRACT...ix

1 INTRODUÇÃO..1

1.1 Motivação..1

1.2 Objetivos..2

1.3 Metodologia.. 3

1.4 Organização 4

2 LINGUAGENS DE PROGRAMAÇÃO..6

2.1 Conceitos...6

2.2 Princípios básicos...6

2.2.1 COMPILADOR...6

2.2.2 INTERPRETADOR..7

2.2.3 MONTADOR..8

2.3 Padrões Principais das Linguagens..9

2.3.1 LINGUAGENS PROCEDURAIS IMPERATIVAS..9

2.3.1.1 PROBLEMAS COM AS LINGUAGENS IMPERATIVAS................................11

2.3.2 LINGUAGENS ORIENTADAS A OBJETOS...11

2.3.3 LINGUAGENS DECLARATIVAS 14

3 PROGRAMAÇÃO LÓGICA 15

3.1 Princípios e Características 16
3.2 Fundamentação Matemática 18
3.2.1 LÓGICA PROPOSICIONAL 20
3.2.1.1 CONCEITOS 20
3.2.1.2 DEDUÇÃO E SOLUÇÃO 21
3.2.2 LÓGICA DOS PREDICADOS 26
3.2.2.1 CONCEITOS 26
3.2.2.2 SINTAXE 28
3.2.2.3 SEMÂNTICA 29
3.2.2.4 DEDUÇÃO E SOLUÇÃO 31
3.3 Áreas Principais de Aplicação 32
3.3.1 MATEMÁTICA 33
3.3.2 SISTEMAS COMPUTACIONAIS 34
3.4 UNIFICAÇÃO 36
3.5 RECURSIVIDADE 39
3.6 BACKTRACKING 41
4 AMBIENTE PROLOG 45
4.1 Fornecedores e Versões 46
4.2 Fatos 46
4.3 Regras 52
4.4 Recursividade 55
4.5 Lista e Árvores 56
4.6 Consultas 59
5 O PROTÓTIPO DESENVOLVIDO 63

5.1 Objetivos....63

5.2 ESPECIFICAÇÃO....63

5.3 componentes e principais programas fontes....65

5.3.1 PREDICADOS INTERNOS....65

5.3.2 PREDICADOS CRIADOS....66

5.4 AMBIENTE DE DESENVOLVIMENTO....68

5.5 TELAS....69

5.5.1 EXEMPLOS E TESTES....71

6 CONCLUSÃO....75

6.1 LIMITAÇÕES....76

6.2 EXTENSÕES....76

REFERÊNCIAS BIBLIOGRÁFICAS....77

LISTA DE FIGURAS

Figura 1 – Escrever os números primos no intervalo 2..n 9

Figura 2 – Movendo-se por um espaço de pesquisa 13

Figura 3 – Programa em Lógica x Convenvional. 17

Figura 4 – Convenções Notacionais.. 22

Figura 5 – Argumento Protágoras x Euathlus 25

Figura 6 – Ordem de visita aos nódos da árvore de pesquisa 42

Figura 7 – Uma árvore genealógica.. 48

Figura 8 – Relação avô em função do progenitor.. 51

Figura 9 – Árvore de Parentes 57

Figura 10 – Árvore do livro / autor 57

Figura 11 – Árvore da Oração. 58

Figura 12 – Árvore da Oração com nomes. 58

Figura 13 - Informação Estruturada sobre uma família..59

Figura 14 – Um programa baseado na relação família 61

Figura 15 - Diagrama de Contexto do Visualizador Prolog...64

Figura 16 - Tela de Carregamento do Protótipo...69

Figura 17 - Tela de Abertura...70

Figura 18 - Tela de Entrada de Arquivo e Predicado...70

Figura 19 - Tela de Visualização de Predicado..71

Figura 20 - Visualização do Avô da família Fontoura..72

Figura 21 - Chamada dos Predicados Envolvidos...73

Figura 22 - Verifica Membro na Lista..73

Figura 23 - Sócio Encontrado...73

RESUMO

Este trabalho visa desenvolver um estudo sobre a programação lógica, mais especificamente a linguagem de programação Prolog. Para tanto, desenvolveu-se um protótipo de software que proporciona uma visualização gráfica dos processos de execução de um programa desenvolvido na linguagem de programação lógica Arity Prolog.

ABSTRACT

This work it seeks to develop a study about the logical programming, more specifically the programming language Prolog, objectifying the development of a software prototype that provides a graphic visualization of the processes of execution of a program developed in the language of logical programming Arity Prolog.

1 INTRODUÇÃO

Prolog, uma abreviação de "PROgramming in LOGic", o que significa programação lógica - uma idéia que emergiu em Edimburgo na Escócia no início da década de 70 para usar a lógica como uma linguagem de programação. Inicialmente sua utilização ficou restrita a algumas universidades e centros de pesquisa europeus tendo alcançado popularidade mundial depois do projeto japonês de computadores de quinta geração ter adotado o PROLOG como sua linguagem básica de desenvolvimento. Hoje é a principal linguagem de programação das que permitem representação lógica. Os fomentadores desta idéia foram Robert Kowalski (no lado teórico), Maarten van Emden (demonstração experimental) e Alain Colmerauer (implementação). A popularidade do Prolog está também presente em grande parte devido a implementação eficiente de David Warren em Edimburgo no meio dos anos setenta [BRA90].

Prolog é uma linguagem de programação centralizada baseada em pequenos conjuntos de mecanismos básicos, estrutura de dados baseados em árvores e regresso automático (*backtracking*). Estes pequenos conjuntos segundo [BRA90], constituem uma estrutura de programação surpreendentemente poderosa e flexível. Prolog é especialmente utilizado para problemas que envolvem objetos, em particular, objetos estruturados e relações entre eles. Por exemplo, é um exercício fácil em Prolog expressar relações de espaço entre objetos. Prolog pode argumentar agora sobre as relações de espaço e a consistência deles a respeito da regra geral. Características como estas fazem do Prolog, na opinião de [BRA90], uma linguagem poderosa para Inteligência Artificial (IA) e programação não numérica em geral.

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. Liberando assim o programador dos problemas associados ao controle de suas rotinas, permitindo-lhe concentrar-se nos aspectos lógicos da situação a representar.

1.2 OBJETIVOS

Os principais objetivos deste trabalho são:

a) realizar um levantamento bibliográfico e estudo teórico sobre programação lógica;
b) realizar um levantamento bibliográfico e estudo teórico sobre as técnicas de programação lógica;
c) desenvolver um protótipo de apoio ao aprendizado da linguagem de programação lógica Prolog. Através de ilustrações gráficas, demonstrando o funcionamento desta linguagem.

O presente trabalho propõe desenvolver um protótipo de software capaz de visualizar o processo de execução de programas escritos na linguagem de programação Prolog. O funcionamento do projeto proposto basear-se-á na interpretação dos comandos Prolog, re-

presentando-os em uma ilustração gráfica, facilitando assim o aprendizado daqueles que queiram aprender sobre esta linguagem.

1.3 METODOLOGIA

Visando atingir os objetivos citados anteriormente, foi realizada uma pesquisa teórica, a especificação e a implementação de um protótipo e finalmente, a elaboração de uma aplicação experimental.

A pesquisa teórica abrangeu uma revisão de literatura especializada, tanto nacional quanto estrangeira. Foram pesquisados livros e dissertações de mestrado.

Na especificação e implementação do protótipo foram observados as recomendações levantadas ao longo de toda a pesquisa teórica.

A técnica empregada para a construção do protótipo será a prototipação de sistemas que segundo [MEL90] é um conjunto de técnicas e ferramentas de software para o desenvolvimento de modelos de sistemas, sendo o principal objetivo da prototipação a antecipação ao usuário final de uma versão (modelo) do sistema para que ele possa avaliar sua funcionalidade, identificar erros e omissões, mediante sua utilização. A vantagem ou benefício econômico desta prática é efetuar as correções e ajustes com um mínimo de custo operacional. O protótipo não é um produto acabado, mas pode ser uma aproximação muito útil deste [KEL97].

A aplicação experimental desenvolveu-se na área da educação. Ela permite facilitar o aprendizado desta linguagem, fazendo com que seu paradigma seja mais facilmente compreendido por aqueles que queiram aprendê-la. O trabalho de elaboração e introdução da construção deste protótipo foi realizado através dos conhecimentos adquiridos nos livros e pelos conselhos dos professores Roberto Heinzle e Cláudio Loesch. Depois de construído o protótipo foram realizados vários testes para comprovar a eficiência do mesmo.

1.4 ORGANIZAÇÃO

O trabalho está organizado da seguinte forma:

O capítulo 1 faz uma introdução, detalhando origem, área, motivação, objetivos, metodologia e a estrutura do trabalho.

O capítulo 2 trata das linguagens de programação. Faz-se um estudo sobre seus conceitos, princípios básico, padrões principais das linguagens e alguns de seus problemas. Este capítulo tem a funções principal de dar ao leitor uma visão dos vários tipos e padrões de linguagens que existem na área da computação, mostrando as particularidades de cada uma.

O capítulo 3 aborda a programação lógica. Mostra-se uma visão geral sobre a programação lógica, seus princípios e características. Faz-se um estudo sobre a fundamentação matemática da lógica, mostrando os conceitos, deduções e soluções da lógica proposicional e a lógica dos predicados. Procura-se mostrar as áreas de aplicações destas lógicas e demonstrar ainda alguns mecanismos básicos da programação lógica, com exemplos e ilustrações sobre a chamada e execução de predicados, a unificação, a recursividade e ainda o mecanismo de backtracking.

O capítulo 4 estuda o Ambiente Prolog, sendo fator estimulante para o desenvolvimento deste protótipo. Procura-se fazer uma análise completa sobre este ambiente, mostrando seus componentes e seu funcionamento. Mostra-se alguns de seus fornecedores e versões deste ambiente, algumas construções básicas desenvolvidas com o ambiente Prolog. Faz-se a distinção do que é um fato de uma regra, como se procede uma pesquisa e o armazenamento de dados em arvores e listas. Como é possível fazer uma consulta através deste ambiente e de que forma ocorre.

O capítulo 5 mostra o protótipo desenvolvido como resultado do estudo e através da especificação, mostram-se seus objetivos, ambiente de desenvolvimento, metodologia utilizada, tipos de telas e menus. E ao final sua utilização e operação através de exemplos e testes.

O capítulo 6 é a conclusão do trabalho, onde é feito uma consideração geral do aprendizado deste protótipo demonstrando as limitações e possíveis extensões para futuros trabalhos.

2 LINGUAGENS E PROGRAMAÇÃO

Uma linguagem de programação é uma notação sistemática através da qual se pode descrever processos computacionais. Entende-se processo computacional como uma série de passos que uma máquina deve percorrer para resolver uma tarefa. Para descrever a solução de um problema a um computador, precisa-se saber um conjunto de comandos que a máquina pode entender e executar [HOR84]. Uma linguagem de programação é uma notação formal para a descrição de algorítmos que serão executados por um computador. Como todas as notações formais, uma linguagem de programação tem dois componentes: Sintaxe e Semântica.

A sintaxe consiste em um conjuntos de regras formais, que especificam a composição de programas a partir de letras, dígitos, e outros símbolos. Por exemplo, regras de sintaxe podem especificar que a cada parênteses aberto em uma expressão aritmética deve corresponder um parênteses fechado, e que dois comandos quaisquer devem ser separados um ponto e vírgula. Já as regras de semântica especificam o significado de qualquer programa, sintaticamente válido, escrito na linguagem.

2.1 CONCEITOS

Neste capítulo serão apresentados alguns dos conceitos abordado na linguagem de programação tais como: compilador, tradutor, montador e os padrões adotados pelas linguagens

2.2 PRINCÍPIOS BÁSICOS

Este módulo irá tratar especificamente dos assuntos mais comuns quanto a linguagem de programação.

2.2.1 COMPILADOR

Nos seus primeiros contatos com o computador, todo programador certamente faz uso dos compiladores, na condição de usuário. Nestas circunstâncias, em geral a máquina e

os programas de sistema, e em particular os compiladores, são vistos de modo extremamente mistificado, devido ao tipo de funções por eles executadas. No entanto, tais módulos de software, apesar de suas dimensões, muitas vezes consideráveis, não passam de programas semelhantes a tantos outros que existem. Apesar de apresentarem funções via de regra não convencionais, os compiladores baseiam-se em princípios relativamente simples, e que se tornam tanto mais compreensíveis quanto maior for o embasamento teórico do interessado. Felizmente, nenhuma espécie de teoria sofisticada é exigida para a compreensão dos seus mecanismos básicos de operação e de projeto.

Compilador é um programa que lê um outro programa escrito em uma determinada linguagem de programação, a chamada linguagem fonte (código de programa) e traduz isto em um programa equivalente em outra linguagem, a linguagem de máquina [PIN95]. Como uma parte importante deste processo de tradução, o compilador informa a seu usuário/programador se seu programa apresenta erros no códigos fonte.

À primeira vista, a variedade de compiladores pode parecer assustadora. Há mil tipos de linguagens de fonte, variando de linguagens tradicionais como Fortran e Pascal para linguagens de mais alto nível como as linguagens virtuais de hoje em dia que tem aplicações em toda a área da computação. Idiomas designados são igualmente como variado; um idioma designado pode ser outro idioma de programação, ou a máquina idioma de qualquer computador entre um microprocessador e um supercomputador.

2.2.2 INTERPRETADOR

O interpretador é um programa que executa repetidamente a seguinte seqüência:

1. Obter o próximo comando do programa.
2. Determinar que ações deve ser executadas.
3. Executar estas ações.

Esta seqüência é bastante semelhante àquela executada por computadores tradicionais, a saber:

1. Obter a próxima instrução (aquela cujo endereço é especificado da próxima instrução a ser executada).

2. Deslocar o indicador de instruções (obtendo o endereço da próxima instrução a ser executada).

3. Decodificar a instrução.

4. Executar a instrução.

Esta semelhança mostra que a interpretação pode ser encarada como a simulação em um computador hospedeiro, de uma máquina especial cuja linguagem de máquina é a linguagem de nível mais alto.

2.2.3 MONTADOR

Os montadores são aqueles tradutores em que a linguagem-fonte é de baixo nível, como é o caso das linguagens de montagem (linguagem assembly). Por tradição, tais tradutores particulares não são chamados de compiladores.

De maneira geral, os tradutores efetuam , portanto, a conversão de textos redigidos em uma linguagem, para formas equivalentes, redigidas em outra linguagem. Se a primeira linguagem for uma linguagem de alto nível, o tradutor receberá o nome de compilador.

O código de montagem é uma versão mnemônica do código de máquina, na qual são usados nomes em lugar do código binário para as operações e fornecidos nomes aos endereços de memória. Uma seqüência típica de instruções de montagem seria (linguagem assembly) [PIN95]:

```
MOV  a, R1
ADD #2, R1
MOV R1, b
```

Este código copia o conteúdo do endereço *a* no *registrador 1*, adiciona a constante *2* ao mesmo, tratando o conteúdo do *registrador 1* como um número em ponto fixo, e, finalmente, armazena o resultado na localização denominada *b*. Computa, então $b := a + 2$.

2.3 PADRÕES PRINCIPAIS DAS LINGUAGENS

Neste capítulo são abordados todas as características de uma linguagem para outra linguagem. Qual sua origem, sua aplicação, vantagens e desvantagens.

2.3.1 LINGUAGENS PROCEDURAIS IMPERATIVAS

As linguagens procedurais imperativas têm este nome devido ao papel dominante desempenhado pelos comandos ou instruções imperativas. 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.

A fim de ilustrar a interação dessas características e seus efeitos, pode-se ver um e-xemplo da figura 1.

Figura 1 - Escrever os números primos no intervalo 2..n

```
program primos;
const n = 50;
var
  i:2..n;
  J:2..25;
  I_é_primo: boolean;

Begin
  For i:=2 to n do
    Begin {i  é primo ?}
      J:=2; i_é_primo:=true;
      While i_é_primo and <= i div 2 do
        If ((i mod j) <> 0) then
          j:= j +1;
        else i_é_primo:=false;
        {se é, escreva seu valor}
          if i_é_primo then write (i)
    end
end.
```

O programa se baseia em duas malhas, uma dentro da outra. A malha mais externa (for i:=2 to n...) percorre os valores no intervalo de interesse (de 2 a n), enquanto que a ma-lha mais interna testa cada um destes números quanto a ser primo ou não. Para entender

cada malha, precisa-se executar mentalmente por algumas iterações pelo menos, examinar suas condições de terminação e as condições sob as quais ela é executada corretamente. Neste caso, a malha mais interna depende de maneira não tão simples assim do índice da malha mais externa (i). A malha mais externa também depende na instrução if da atribuição feita na malha mais interna.

Em outras palavras o programa não é hierárquico no sentido de cada componente ser composto de vários outros componentes (de nível mais baixo). Ao contrário, cada componente usa os feitos dos outros. No exemplo particular, a malha mais interna utiliza as modificações feitas em i na mais externa e esta utiliza as modificações feitas em i_é_primo naquela. Esses componentes estão intimamente relacionados. Um componente é usado não para calcular um valor, mas para produzir um efeito especificamente, o efeito de atribuir valores a variáveis. As estruturas de controle são usadas para ordenar as instruções de maneira que os efeitos combinados atinjam o fim desejado.

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

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

2.3.1.1 PROBLEMAS COM AS LINGUAGENS IMPERATIVAS

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. Sendo este um nível de detalhamento com maior complexidade de entendimento, fazem com que de fato, as linguagens de programação tentem cada vez mais esconder essa natureza de baixo nível da máquina.

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 das células da memória. Para entender as malhas de programa precisamos executá-las mentalmente. Para entender como a computação progride ao longo do tempo precisamos tomar instantâneos da memória em cada passo, após cada instrução. Isto é uma tarefa cansativa quando o programa envolve uma quantidade grande de memória. Vimos que as regras de escopo das linguagens limitam um pouco este problema reduzindo o número de células acessíveis. O uso das variáveis globais pode tornar os programas difíceis de analisar.

Mas deve ser lembrado de que mecanismos como variáveis globais, passagem de parâmetros por referência e (em geral) efeitos colaterais, que parecem ir contra os propósitos de mecanismos de alto nível, são introduzidos nas linguagens com a finalidade de obter eficiência de execução. E esta eficiência está baseada nas características de arquitetura de Von Neumann.

2.3.2 LINGUAGENS ORIENTADAS A OBJETOS

A programação Orientada a Objetos baseia-se na definição de objetos ou classes do mundo real. Um objeto é uma entidade, ou modelo da vida real. Uma classe é um modelo a partir do qual os objetos são criados. Para ser definida como orientada a objetos uma linguagem de programação deve suportar, no mínimo, as seguintes características:

- Herança: uma classe-objeto pode herdar características de outra classe-objeto visando a reutilização de código;

- Encapsulamento: uma classe-objeto não pode ver detalhes de implementação de outra classe-objeto;

- Polimorfismo: uma classe-objeto deve discernir, dentre métodos homônimos, o que deve ser executado.

O Arity Prolog é uma linguagem orientada por objeto, ou direcionada para dados. Isto é, você passa para o programa os dados sobre o problema através de fatos sobre os objetos e as relações entre esses fatos [TOW90]. Por exemplo, poderia-se representar um fato como :

sintoma(paciente,cegueira_noturna).

Que é traduzido para "O paciente possui cegueira noturna". Uma relação entre os fatos é uma regra, e pode ser expressa em inglês/português como:

```
IF    o paciente tem cegueira noturna
AND   a pele do paciente é áspera e seca
THEN  o paciente pode ter uma deficiência de vitamina A
```

Esta regra poderia se expressa em Prolog por:

```
Diagnostico(vitamina_A):-sintoma(cegueira_noturna),
sintoma(pele,aspera_e_seca).
```

Um programa Prolog é um banco de dados constituído de um grupo de fatos e regras. Este tipo de programação é muito diferente dos métodos de programação por procedimento do C e outras linguagens. As linguagens voltadas para o procedimento são quase todas adeptas ao processamento científico e à aplicações comerciais. As linguagens orientadas por objeto são mais adeptas ao raciocínio formal. Um programador com muita experiência no uso de linguagens de procedimento precisa reaprender bastante antes de usar linguagens orientadas por objeto [TOW90].

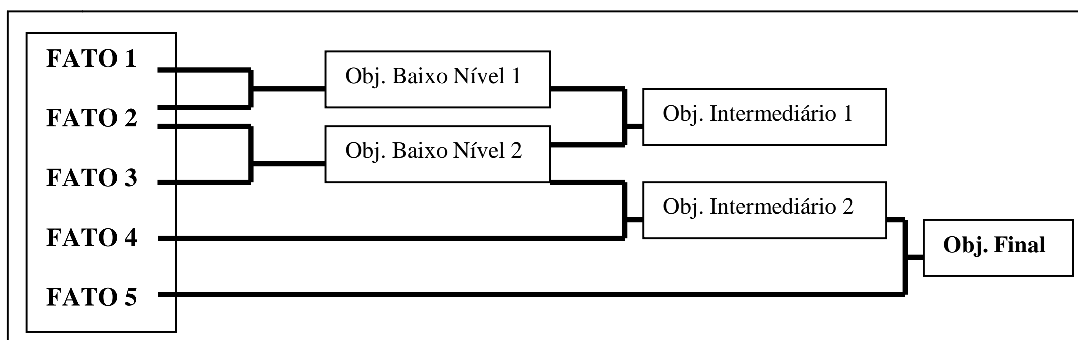
As linguagens de procedimento usam algorítmos bem definidos para resolver um tipo específico de problema. O programador começa tentando definir o algoritmo necessário para resolver um problema, e depois escreve o programa para implementar o algoritmo. Pode-se, então, rodar o programa tantas vezes quantas forem necessárias com diferentes

dados de entrada. Toda vez que é executado, o programa usa o mesmo algoritmo para resolver o problema a partir dos dados de entrada [TOW90].

As linguagens orientadas por objeto, usam procedimentos dinâmicos que não são definidos pelo usuário. O programa é um banco de dados, ou um grupo de fatos e regras. A execução do programa mas se parece com um diálogo com o usuário, com as respostas do usuário servindo para o acréscimo de fatos adicionais, acrescentados ao banco de dados. Os fatos que são conhecidos determinam uma via por meio de um espaço de pesquisa até a conclusão eventual, como mostra figura 2. O programa é escrito por um programador, que tem a responsabilidade de definir os objetos e a estrutura do banco de dados. Normalmente, o programador pode projetar um programa de modo que as heurísticas, reduzam o espaço de pesquisa. Por exemplo, um sistema de diagnóstico médico em Prolog precisaria de um banco de dados extremamente grande para atender a quaisquer objetivos realísticos do projeto [TOW90].

Um sistema poderia, no entanto, ser projetado para que trabalhasse dentro de um espaço de pesquisa limitado.

Figura 2 - Movendo-se por um espaço de pesquisa.



Agora, o que aconteceria se o banco de dados do sistema de diagnósticos fosse expandido para serem incluídos novos sintomas e diagnósticos. Estaria sendo fazendo muitas perguntas que teriam pouca probabilidade de se relacionarem com o diagnóstico final. O programador precisaria projetar o programa de modo que fosse usado a heurística para procurar certos padrões bem no início da consulta, a fim de reduzir o espaço de pesquisa até

um tamanho manuseável. Por exemplo, perguntando primeiro se o paciente teve febre, podemos eliminar metade das conclusões.

As heurísticas podem ser usadas para agilizar a pesquisa, eliminando as vias possíveis que não são eficazes. A heurística não garante a solução mais eficaz, ou até mesmo qualquer solução, mas pode ajudar a reduzir o espaço de pesquisa e aprimorar a eficiência do programa.

O Prolog não é totalmente de fato uma verdadeira linguagem de não-procedimento. Sempre se constatará que algum nível de controle é necessário nos seus sistemas. Na realidade, a beleza do Prolog é que ele é semideclarativo e semiformal, permitindo otimizar a eficiência de programas. Se, no entanto, estiver sendo dispendido muito trabalho de projeto de procedimentos, provavelmente será melhor tentar resolver o problema com uma outra linguagem.

2.3.3 LINGUAGENS DECLARATIVAS

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

3 PROGRAMAÇÃO LÓGICA

Os termos "programação 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, são as seguintes [PAL97]:

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

- **Capacidade Dedutiva:** O conceito de computação confunde-se com o de (passo de) inferência. A execução de um programa é a prova do teorema representado pela consulta formulada, com base nos axiomas representados pelas cláusulas (fatos e regras) do programa.

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

- **Reversibilidade das Relações:** (Ou "computação bidirecional"). Os argumentos de um procedimento podem alternativamente, em diferentes chamadas representar ora parâmetros de entrada, ora de saída. Os procedimentos podem assim ser projetados para atender a múltiplos propósitos. A execução pode ocorrer em qualquer sentido, dependendo

do contexto. Por exemplo, o mesmo procedimento para inserir um elemento no topo de uma pilha qualquer pode ser usado, em sentido contrário, para remover o elemento que se encontrar no topo desta pilha.

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

• **Recursão:** A recursão, 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" (apesar de poderem ser facilmente programados), simplesmente porque eles são absolutamente desnecessários. Também são dispensados comandos de atribuição e, evidentemente, o "goto". Uma estrutura de dados contendo variáveis livres pode ser retornada como a saída de um procedimento. Essas variáveis livres podem ser posteriormente instanciadas por outros procedimentos produzindo o efeito de atribuições implícitas a estruturas de dados. Onde for necessário, variáveis livres são automaticamente agrupadas por meio de referências transparentes ao programador. Assim, as variáveis lógicas um potencial de representação significativamente maior do que oferecido por operações de atribuição e referência nas linguagens convencionais.

3.1 PRINCÍPIOS E CARACTERÍSTICAS

Programação Lógica é um modelo de um determinado problema ou situação expresso através de um conjunto finito de sentenças lógicas. Ao contrário de programas em Fortran ou Pascal, por exemplo, um programa em lógica não é, portanto, a descrição de um procedimento para obter soluções de um problema. De fato, o interpretador ou compilador utilizado para processar os programas em lógica fica inteiramente responsável pelo procedimento adotado para pesquisa de soluções. Um programa em lógica assemelha-se mais a um banco de dados, exceto que as afirmações em um banco de dados descrevem apenas

observações como "João é gerente de Pedro", enquanto que as sentenças de um programa em lógica podem também ter um escopo mais genérico, como "o gerente de um funcionário é superior hierárquico do funcionário " e " o gerente de um superior hierárquico de um funcionário é superior hierárquico do funcionário " [CAS87].

Programação em Lógica exemplifica assim um estilo fundamental, que pode ser chamado de Programação Declarativa (assercional ou não procedimental), em contraste com Programação Procedimental (ou imperativa), típica das linguagens tradicionais. A Programação Declarativa engloba também a Programação Funcional, que tem em LISP o seu exemplo mais conhecido, e quase todas as linguagens mais recentes para consulta a bancos de dados como SQL e QUEL. Lembrando que LISP data de 1960, Programação Funcional é então um estilo conhecido há bastante tempo ao contrário de Programação Lógica, que só ganhou ímpeto depois de 1972 com o advento da linguagem Prolog (PROgramming in LOGic), abordada neste trabalho [CAS87].

Os conceitos de chamada/consulta e de respostas naturalmente também diferem das noções tradicionais. De fato, uma consulta a um programa em lógica é uma afirmação que expressa as condições a serem satisfeitas por um resposta correta em presença da informação descrita pelo programa.

Na figura 3 procura-se explicitar as principais diferenças entre programação em lógica e programação convencional [PAL97].

Figura 3 - Programa em Lógica X Programa Convencional

PROGRAMAS EM LÓGICA	**PROGRAMAS CONVENCIONAIS**
Processamento Simbólico	Processamento Numérico
Soluções Heurísticas	Soluções Algorítmicas
Estruturas de Controle e Conhecimento Separadas	Estruturas de Controle e Conhecimento Integradas
Fácil Modificação	Difícil Modificação
Incluem Respostas Parcialmente Corretas	Somente Respostas Totalmente Corretas
Incluem Todas as Soluções Possíveis	Somente a Melhor Solução Possível

Porém, o ponto fundamental de Programação em Lógica consiste em identificar a noção de computação com a noção de dedução. Mais precisamente, a maioria dos sistemas para Programação em Lógica reduzem a busca de respostas corretas à pesquisa de refuta-

ções a partir das sentenças do programa e da negação da consulta (uma refutação é uma dedução de um contradição). Tais sistemas baseiam-se diretamente em procedimentos para pesquisa de refutações, estudados em Prova Automática de Teoremas, e em resultados de Programação em Lógica mostrando como extrair respostas corretas de refutações [CAS87].

Assim, a resposta de uma consulta a um programa em lógica não se limita apenas a indicar que uma suposição acerca da informação contida no programa é falsa ou verdadeira. A resposta efetivamente exibe informação extraída do programa e pode vir acompanhada de uma explicação sobre como foi obtida, expressa em termos da refutação que a gerou.

Na maioria dos Prologs os nomes dos predicados são identificadores criados com uma seqüência de caracteres alfanuméricos iniciando com minúsculos. As variáveis também são identificadores livremente criados usando-se caracteres alfanuméricos mas devem iniciar com maiúsculo. Assim, poderia-se escrever gosta(mario,maria) para afirmar que Mário gosta de Maria; gosta(X,maria) para afirmar que todos gostam de Maria e gosta(X,maria), gosta(Y,zeca) para representar o conjunto de pessoas que gostam de Maria e de Zeca simultaneamente . Em outra situação poder-se-ia escrever "Ricardo gosta de Vera se Ricardo gostar de Ana" que seria: gosta(ricardo,vera) :-gosta(ricardo,ana) . Já a negação, por exemplo, "Carolina não gosta de Zeca" seria escrito em PROLOG como not(gosta(carolina,zeca)). O predicado gosta, exige dois parâmetros (nomes próprios). O número de parâmetros determina a aridade do predicado, portanto, predicado gosta tem aridade dois.

3.2 FUNDAMENTAÇÃO MATEMÁTICA

A programação lógica fundamenta-se em muito 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.

3.2.1 LÓGICA PROPOSICIONAL

Lógica Proposicional ou Cálculo Proposicional formaliza a estrutura lógica mais elementar do discurso matemático, definindo precisamente o significado dos conectivos lógicos **não , e , ou, se ... então** e outros [CAS87].

3.2.1.1 CONCEITOS

A definição da Lógica Proposicional desdobra-se na especificação do que seja uma linguagem proposicional e na descrição de uma abstração adequada para os princípios lógicos que governam os conectivos [CAS87].

Brevemente, o alfabeto de uma linguagem proposicional consiste dos conectivos lógicos , dos parênteses e de um conjunto de símbolos proposicionais. As regras sintáticas da linguagem definem o conjunto de fórmulas (bem formadas) como sendo ou os próprios símbolos proposicionais ou expressões construídas ligando tais símbolos através dos conectivos lógicos. As regras semânticas da linguagem capturam o significado pretendido dos conectivos e associam a cada fórmula um dos valores-verdade, "falso" ou "verdadeiro".

Há várias formas de abstrair os princípios lógicos que governam os conectivos. A forma mais divulgada, o método da tabela-verdade, permite decidir, entre outras aplicações, se uma fórmula é sempre verdadeira. Uma outra forma consiste em definir um "cálculo" para inferir novas fórmulas a partir de outras. Por fim, existem ainda métodos de refutação para determinar se um conjunto de fórmulas leva a contradições, entre os quais encontramos o método dos tableaux analíticos e o método de resolução para Lógica Sentencial.

A origem da Lógica Sentencial remonta aos trabalhos de Boole (1815-1864) e de De Morgan (1806-1871) sobre o que veio a ser chamado de Álgebra de Boole. Porém, como o próprio nome indica, estes trabalhos estavam mais próximos de outras teorias matemáticas do que de Lógica. Devemos a Frege, no seu "Begriffsschrift", o enfoque de Lógica Senten-

cial como uma ferramenta para formalizar princípios lógicos (Frege chegou a ser criticado por esta mudança de enfoque).

3.2.1.2 DEDUÇÃO E SOLUÇÃO

Em uma linguagem proposicional pode-se capturar parte da estrutura lógica de trechos de discursos. Ou seja, podemos abstrair parágrafos consistindo de sentenças concatenadas por partículas e, ou, se ... então e outras com a mesma função. As sentenças recebem nomes, tirados de um conjunto de símbolos proposicionais. Às partículas correspondem símbolos especiais, chamados de conectivos.

Dado um alfabeto A, uma cadeia sobre A é uma seqüência de símbolos de A . Uma linguagem sobre A é um conjunto de cadeias de A . De posse destas noções, define-se então:

Definição a)

Um alfabeto proposicional A consiste de: símbolos lógicos, pontuação e conectivos:

,
¬ (negação)
^ (conjunção)
∨ (disjunção)
→(implicação)
≡ (bi-implicação ou bi-condicional)

E símbolos não-lógicos: Um conjunto enumerável P de símbolos proposicionais diferentes dos símbolos lógicos.

Definição b)

Os conjuntos das fórmulas proposicionais (ou simplesmente fórmulas) sobre um alfabeto proposicional **A** é o menor conjunto de cadeias de **A** satisfazendo às seguintes condições:

(i) todo símbolo proposicional de **A** é uma fórmula sobre **A**;

(ii) se **P** e **Q** são fórmulas sobre **A**, então (¬ **P**), (**P** ^ **Q**), (**P** ∨ **Q**), (**P** → **Q**) e (**P** ≡ **Q**) também são fórmulas sobre **A** .

Definição c)

Uma fórmula **Q** é uma subfórmula de uma fórmula **P** se e somente se **Q** é uma subcadeia de **P**.

Definição d)

A linguagem proposicional sobre um alfabeto, proposicional **A**, denotada por **L(A)**, é o conjunto das fórmulas proposicionais sobre **A** .

O conjunto das símbolos lógicos e as regras de formação para as fórmulas são fixados para todos os alfabetos proposicionais. Assim, para especificar um alfabeto proposicional **A** e, portanto, a linguagem proposicional **L(A)**, basta fixar o conjunto **P** dos símbolos proposicionais de **A** . Baseando-se nesta observação, freqüentemente não faremos distinção entre **A**, **L(A)** e **P**, como nos textos tradicionais de lógica. Por exemplo, especificaremos **L(A)** listando apenas **P** e diremos que os símbolos proposicionais em **P** são símbolos proposicionais de **L(A)**. Finalmente, se não for necessário identificar o alfabeto **A**, denotaremos a linguagem proposicional **L(A)** sobre **A** simplesmente por **L**.

Figura 4 – Convenções Notacionais

Objeto Sintático	**Convenção Notacional**
símbolos proposicionais	letras maiúsculas do início do alfabeto (**A,B,C,...**)
fórmulas	letras maiúsculas do meio para o fim do alfabeto (**P,Q,R,...**)
conjuntos de fórmulas	letras maiúsculas em negrito do meio para o fim do alfabeto (**P**,**Q**,**R**,...)
linguagens proposicionais	letras maiúsculas em itálico e negrito do meio do alfabeto (***L***,...)

De acordo com a definição do conjunto de fórmulas, todas fórmula deverá estar completamente parentetizada. Porém, adotaremos as seguintes convenções sobre omissão de parênteses:

1. os parênteses mais externos podem ser omitidos;
2. a negação aplica-se à menor fórmula possível

exemplo: **¬ A ^B** abrevia **((¬ A) ^B)**

3. a conjunção e a disjunção aplicam-se à menor fórmula possível

exemplo: **A ^ B → ¬ C ∨ D** abrevia **((A ^ B) → ((¬ C) ∨**

4. quando um conectivo é usado repetidamente, o agrupamento é feito pela direita

exemplo: A → B → abrevia (A → (B → C))

A seguir são apresentados dois exemplos simples do uso de linguagens proposicionais para capturar trechos do discurso. Em ambos os casos, o passo inicial consiste em selecionar um conjunto de símbolos proposicionais adequado, fixando assim a linguagem proposicional a ser usada. Cada símbolo está associado a uma frase, que é o seu significado pretendido. O passo seguinte consiste em traduzir o trecho em questão para uma ou mais fórmulas, respeitando o significado pretendido dos símbolos.

Exemplo a: extraído de [CAS87]

Considere a seguinte afirmação:

"Supõe-se que Sócrates está em tal situação que ele estaria disposto a visitar Platão, só se Platão estivesse disposto a visitá-lo, e que Platão está em tal situação que ele não estaria disposto a visitar Sócrates, se Sócrates estivesse disposto a visitá-lo, mas estaria disposto a visitar Sócrates, se Sócrates não estivesse disposto a visitá-lo".

Pergunta-se então "Sócrates está disposto a visitar Platão ou não?"

Pode-se analisar este problema da seguinte forma. Construa-se um alfabeto proposicional **A** cujos símbolos proposicionais são **A** e **B**, com o seguinte significado pretendido:

A - "Sócrates está disposto a visitar Platão"

B - "Platão está disposto a visitar Sócrates"

O trecho anterior desdobra-se então nos seguintes fatos, já expressos como fórmulas de **L(A)**:

Sócrates: (B → A)

Platão: (**A** → ¬ **B**) ^ (¬ **A** → **B**)

Ou seja, estas duas fórmulas axiomatizam em Lógica Sentencial o nosso conhecimento acerca do estado de espírito de Sócrates e Platão.

A pergunta original resume-se então a saber qual das duas fórmulas **A** ou ¬ **A** segue desta axiomatização, independentemente do verdadeiro significado atribuído aos símbolos **A** e **B**.

Captura-se, após os exemplos, precisamente o que significa "seguir de uma axiomatização" através do conceito de conseqüência lógica. Além disto apresentaremos no final deste capítulo um procedimento mecânico, chamado de método da tabela-verdade, que permitirá resolver o problema apresentado neste exemplo de uma forma simples.

Exemplo b: **extraído de [CAS87]**

Os sofistas, uma espécie de professores viajantes ('sofistas' significava mais ou menos o mesmo que 'professor'), tornaram-se famosos na Grécia clássica por visitarem cidades ensinando, por um soldo, a arte de argumentar. Eles alcançaram grande fama e grande habilidade em argumentar a favor ou contra qualquer afirmação, não importando a sua veracidade. A arte que ensinavam, refletida hoje em dia no termo 'sofismático', que tem um sentido quase pejorativo, naquela época poderia ser vital pois, se um cidadão fosse acusado de um delito e tivesse que se apresentar perante um tribunal, caberia a ele se defender. Outra pessoa poderia naturalmente preparar a sua defesa, mas não o substituir na defesa em si. Além disto, os juízes não eram profissionais treinados, mas simples cidadãos escolhidos aleatoriamente e, portanto, bastante influenciáveis por defesas bem arquitetadas.

O mais famoso dos sofistas, Protágoras, nasceu em Abdera por volta de 480 AC e morreu por volta de 420 AC. Consta que Protágoras teve um discípulo brilhante, chamado Euathlus. Protágoras o ensinou a arte de argumentar por uma certa quantia, metade da qual seria paga imediatamente e metade após Euathlus ganhar o seu primeiro caso. Euathlus, porém, demorou a pagar a Protágoras, que o processou então.

Protágoras argumentou que se Euathlus ganhasse o caso, ganharia então o seu primeiro caso, logo deveria pagá-lo. Mas se Euathlus não ganhasse o caso, deveria pagá-lo também pois era esta a questão em jogo.

Euathlus, que foi um bom aluno, argumentou da seguinte forma. Se ele ganhasse o caso, não deveria pagar pois Protágoras perderia a causa. Mas, se ele não ganhasse o caso, não estaria ganhando o seu primeiro caso e, portanto também não deveria pagar a Protágoras.

Quem está com a razão então?

Os argumentos de Protágoras e Euathlus podem ser expressos em uma linguagem proposicional da seguinte forma. Os símbolos proposicionais do alfabeto são:

A - Euathlus ganha o caso

B - Euathlus ganha o seu primeiro caso

C - Euathlus deve pagar a Protágoras

As fórmulas que capturam os argumentos são demonstradas na figura 5.

Figura 5 – Argumento de Protágoras X Argumento de Euathlus

Argumento de Protágoras	Argumento de Euathlus
$\mathbf{A \rightarrow B}$	$\mathbf{A \rightarrow \neg C}$
$\mathbf{B \rightarrow C}$	$\mathbf{\neg A \rightarrow \neg B}$
$\mathbf{\neg A \rightarrow C}$	$\mathbf{\neg B \rightarrow \neg C}$
Logo **C**	logo $\mathbf{\neg C}$

Ou, sob forma de fórmulas:

Protágoras: $((A \rightarrow B) \wedge (B \rightarrow C) \wedge (\neg A \rightarrow C)) \rightarrow C$

Euathlus : $((A \rightarrow \neg C) \wedge (\neg \mathbf{A} \rightarrow \neg B) \wedge (\neg \mathbf{B} \rightarrow \neg C)) \rightarrow \neg C$

A análise em Lógica Sentencial é inconclusiva, pois nenhum dos dois argumentos está incorreto e a pergunta sobre quem deveria ganhar a causa permanece tão em suspenso quanto antes.

Uma solução para Protágoras receber a quantia devida, sugerida por um advogado, seria proceder da seguinte forma. Protágoras deveria processar Euathlus imediatamente após terminadas as aulas, exatamente como descrito acima, e deixar Euathlus ganhar o caso, que seria o seu primeiro. Em seguida, deveria processar novamente Euathlus para que lhe pagasse a quantia estipulada. Desta vez não haveria dúvida de que Protágoras sairia vencedor, pois Euathlus já havia ganho o seu primeiro caso.

3.2.2 LÓGICA DOS PREDICADOS

Segundo [GRA88] a lógica dos predicados é uma regra importante na teoria das bases de dados relacionais e pode ser usada para expressar formalismos relacionais e as funcionalidades normalmente associadas a estes formalismos.

3.2.2.1 CONCEITOS

Denomina-se ***predicado*** ao conjunto de fatos e regras empregados para descrever uma determinada relação.

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:

$$\forall X \text{ (humano(X)} \Rightarrow \text{tem_mãe_humana(X))}$$

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:

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

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:

$$\forall X \text{ (humano(X)} \Rightarrow \text{ (humano(mãe(X)))}$$

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”.

3.2.2.2 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)).

3.2.2.3 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. Pri-

meiramente, 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.

3.2.2.4 DEDUÇÃO E SOLUÇÃO

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) $((\forall X \alpha) * (\forall X \beta)) \equiv \forall X (\alpha * \beta)$.

3.3 ÁREAS PRINCIPAIS DE APLICAÇÃO

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

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

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

Processamento da Linguagem Natural: A implementação de sistemas de processamento de linguagem natural em computadores requer não somente a formalização sintática, como também a formalização semântica, isto é, o correto significado das palavras, sentenças, frases, expressões, etc. que povoam a comunicação natural humana. Segundo

[BRO92] o uso da lógica das cláusulas de Horn são adequadas à representação de qualquer gramática livre-de-contexto e permitem que questões sobre a estrutura de sentenças em linguagem natural sejam formuladas como objetivos ao sistema, e que diferentes procedimentos de prova aplicados a representações lógicas da linguagem natural correspondam a diferentes estratégias de análise.

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

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

3.3.1 MATEMÁTICA

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

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

Embora a principal força do Prolog seja o raciocínio formal, ele também pode ser usado para o suporte a cálculos aritméticos baseado em um algoritmo especificado [TOW90].

3.3.2 SISTEMAS COMPUTACIONAIS

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

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

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

A expressão "programação em lógica" (*logic programming*, originalmente em inglês) é devido a Robert Kowalski (1974) e designa o uso da lógica como linguagem de programação de computadores. Kowalski identificou, em um particular procedimento de prova de teoremas, um procedimento computacional, permitindo uma interpretação procedimental da lógica e estabelecendo as condições que nos permitem entendê-la como uma linguagem de programação de uso geral. Este foi um avanço essencial, necessário para adaptar os conceitos relacionados com a prova de teoremas às técnicas computacionais já dominadas pelos programadores. Aperfeiçoamentos realizados nas técnicas de implementação também foram de grande importância para o emprego da lógica como linguagem de programação. Segundo [SET90] o primeiro interpretador experimental foi desenvolvido por um grupo de pesquisadores liderados por Alain Colmerauer na Universidade de Aix-Marseille (1972) com o nome de Prolog, um acrônimo para "Programmation en Logique". Seguindo-se a este primeiro passo, implementações mais praticas foram desenvolvidas por Battani e Meloni (1973), Bruynooghe (1976) e, principalmente, David H. D. Warren, Luís Moniz Pereira e outros pesquisadores da Universidade de Edimburgo (U.K.) que, em 1977, formalmente definiram o sistema hoje denominado "Prolog de Edimburgo", usado como referência para a maioria das atuais implementações da linguagem Prolog. Deve-se também a Warren a especificação da WAM (Warren Abstract Machine), um modelo formal empregado até hoje na pesquisa de arquiteturas computacionais orientadas à programação em lógica [CLO94].

3.4 UNIFICAÇÃO

O processo pelo qual o Prolog tenta combinar um termo contra os fatos ou cabeças de outras regras em uma tentativa de provar um objetivo é chamada de ***unificação*** [TOW90] . Um *termo* é a menor parte de uma expressão que pode receber um valor, um objeto simples, variável ou estrutura composta, como uma lista ou objeto composto. A unificação é essencialmente um processo de combinação de formas. Diz-se que um termo está *unificado* com um outro se as seguintes condições forem atendidas:

- Ambos os termos estiverem em predicados com o mesmo número de argumentos (a mesma aridade), e ambos os termos aparecem na mesma posição nos dois predicados.
- Ambos os termos forem argumentos do mesmo tipo. Um tipo de símbolo, por exemplo, só pode ser unificado com um tipo de símbolo.
- Todos os subtermos forem unificados uns com os outros. Um *subtermo* é uma expressão de predicado dentro de um predicado, tal como uma expressão dentro de um objeto composto.

No nosso exemplo a seguir, a premissa *sintoma(cegueira_noturna)* na terceira regra é unificada com a conclusão *sintoma(cegueira_noturna)* na quinta regra.

Programa exemplo:

```
DOMAINS
 Doenca, tipo = symbol
PREDICATES
Diagnostico(doenca)
Sintoma(doenca)
Sintoma!(doenca,tipo)
Run

CLAUSES
run:-  diagnostico(X),
       write("Há indicacao de uma deficiencia de  ",X,"."),nl,
run:-  write("Não posso diagnosticar sua doenca."),nl.
diagnostico(vitamina_A):- sintoma(cegueira_noturna),
                          sintoma(pele,aspera_e_seca ).
diagnostico(vitamina_C):- sintoma(infeccoes),
                          sintoma(cura),
                          sintoma(ites).


sintoma(cegueira_noturna):- write("Há incapacidade na vista em se  "),nl.
                            write("adaptar ao escuro?"),nl.
                            readchar(Resposta),nl,
                            resposta='s'.


sintoma(infeccoes):- write("O paciente possui uma pequena "),nl,
                     write("resistencia a infeccoes?"),nl,
                     readchar(Resposta),nl,
                     resposta='s'.


sintoma(cura):- write("As feridas curam-se lentamente? "),nl,
                readchar(Resposta),nl,
                resposta='s'.


sintoma(ites):-  write("O paciente possui alguma das doenças 'ites' (ar-
                 trite, bursite, etc.), "),nl,
                write("hemorragias ou uma tendencia a contusoes?"),
                readchar(Resposta),nl,
                resposta='s'.

sintoma(pele,aspera_e_seca):-
        write("A pele do paciente parece áspera e seca (velha prematura-
        mente)?"),
        readchar(Resposta),nl,
        resposta='s'.
```

O programa exemplo acima é feito em Turbo Prolog, um outro ambiente de programação lógica.

Cada regra possui duas partes: uma cabeça e um corpo. A cabeça é a conclusão, o restante da regra é o corpo ou antecedente, e consiste de uma ou mais premissas. A conclusão é verdadeira se todas as premissas forem verdadeiras. Se alguma premissa falhar, a regra falha nessa premissa. Por exemplo:

```
diagnostico(vitamina_A):-  sintoma(cegueira_noturna),
                           sintoma(pele,aspera_e_seca ).
```

Em nossa língua portuguesa, isto significa: "Se os sintomas forem cegueira noturna e pele áspera e seca, então é deficiência de vitamina A . Neste caso, a cabeça é: diagnostico(vitamina_A).

Se um termo for uma variável, as seguintes regras se aplicam:

- Uma variável livre será unificada com qualquer termo, tornando-se ligada a esse termo.
- O contrário também é verdadeiro: um termo será unificado com qualquer variável livre, ligando a variável ao termo.
- Uma variável ligada será unificada com qualquer termo do mesmo valor.
- Um termo será unificado com qualquer variável do mesmo valor.

Um outro exemplo seria, *diagnostico(X)* é unificado com *diagnostico(vitamina_A).* ligando a variável X a vitamina_A.

Você pode usar *variáveis anônimas* se quiser forçar a unificação. As variáveis anônimas são representadas por um único caractere de sublinhamento. Qualquer termo será unificado com uma variável anônima, e uma variável anônima será unificada com qualquer termo. Por exemplo reescrever a primeira e terceira cláusulas do exemplo da seguinte forma:

```
Run:- diagnostico( _ ).
diagnostico(vitamina_A):- sintoma(cegueira_noturna),
                          sintoma(pele,aspera_e_seca),
                    write("Há evidência de deficiência"),nl,
                          write("de vitamina ª"),nl.
```

Neste caso, diagnostico(_) será unificado com diagnostico(vitamina_A), e o programa tentará provar a regra como antes. Não há uma variável ligando.

3.5 RECURSIVIDADE

Em Prolog, a recursividade ou recursão refere-se à técnica de se usar uma cláusula para chamar uma cópia de si mesma. A recursividade é útil em Prolog para a criação de estruturas de *loop*.

Por exemplo, um contador simples poderia ser criado em Prolog usando-se o seguinte programa:

```
conta(9).
conta(N):-  write("",N),
            NN= +1,
            conta(NN).
```

Neste caso, o predicado é inicialmente chamado conta(0). Este predicado será unificado com a segunda cláusula, que apresenta o valor, incrementa o contador e depois chama conta(1). Este predicado novamente será unificado com a Segunda cláusula, apresentando o contador, incrementando-o e chamando conta(2). O processo continuará até que o valor do contador atinja 9, quando então a chamada será unificada com a primeira cláusula, e o programa recuará a cada chamada de conta(N).

A chamada de conta(N) dentro da regra é uma chamada recursiva. A regra chama uma cópia, exceto por meio do processo de unificação, como em qualquer outro tipo de unificação. A cláusula é executada nove vezes antes de terminar. Observe que a cláusula deve ser incluída para o término do loop, ou então o loop continuará com a recursividade indefinidamente.

A recursividade envolve duas fases: giro para baixo e giro para cima. As sentenças antes da chamada recursiva são chamadas enquanto ela estiver sendo girada para baixo. Quaisquer sentenças após chamada recursiva são chamadas enquanto ela estiver sendo girada para cima. Para ilustrar o giro para cima, altere o exemplo anterior para:

```
conta(9).
conta(N):-  write("",N),
            NN= +1   ,
            conta(NN),
            write("",NN).
```

Neste caso, os predicados write("",N) para todos os novos loops serão executados durante o giro para baixo antes que o primeiro predicado write("",NN) seja executado durante o giro para cima.

Se a chamada recursiva não for última sentença em uma cláusula, o Prolog deve observar onde ela estava durante a execução, de modo que durante o giro para cima todas as sentenças restantes possam ser executadas. Isto requer espaço de pilha (e memória). Se o número de chamadas recursivas se tornar muito grande, pode haver um estouro de pilha. Recursividade de ponta significa projetar a chamada recursiva de modo que a chamada seja a última sentença da cláusula. Se a chamada recursiva for a última sentença na cláusula, o Prolog sente isto e não cria os ponteiros de pilha necessários para a execução das sentenças adicionais.

Mesmo quando a recursividade de ponta é usada, o Prolog requer um certo espaço de memória para cuidar dos retornos. Um *loop* que continua com a recursividade indefinidamente sempre acabará com uma eventual mensagem de erro.

Há portanto, três regras básicas para o uso da recursividade:

- Um programa deve incluir algum método para determinar o loop recursivo.
- A ligação de variável em um loop recursivo aplica-se apenas ao nível corrente. As variáveis são passadas a outras camadas por meio do processo de unificação, assim como em qualquer outra chamada.
- Na maior parte das aplicações, o procedimento recursivo deve fazer seu trabalho durante o giro para baixo. A chamada recursiva deve ser a última sentença na cláusula.

A recursividade é uma boa prática na programação e gera programas elegantes e simples. No entanto, há uma desvantagem na recursividade. Uma grande recursão dificulta a leitura e o entendimento de um programa. Atualizar ou modificar um programa com muita recursão pode ser difícil. O uso de muitos comentários no decorrer do programa, no entanto, pode ajudar.

3.6 BACKTRACKING

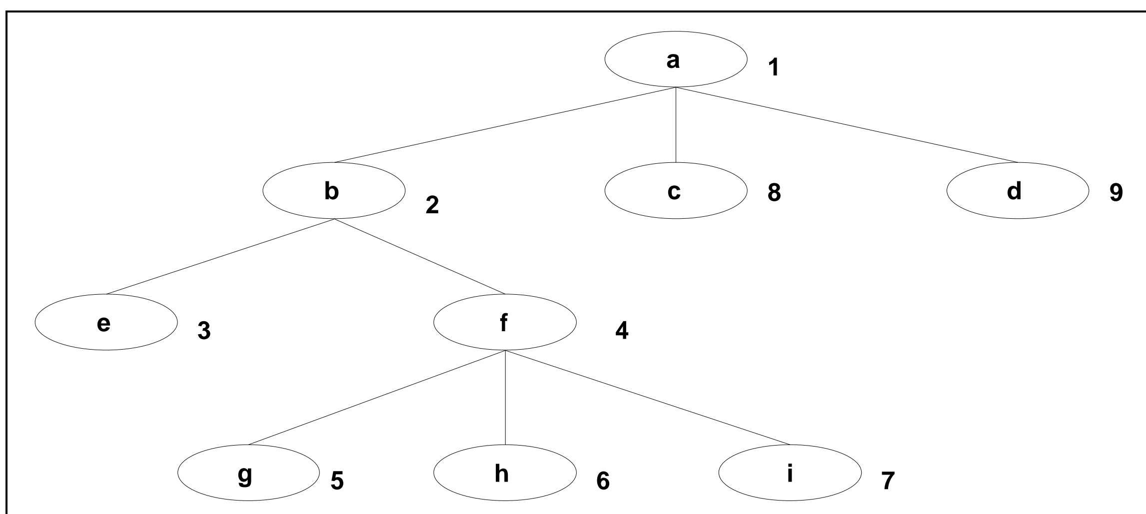
Este é o momento em que o prolog volta a primeira cláusula e começa a pesquisar a lista . A isto denomina-se *retrocesso* e é uma característica importante do Prolog. Sempre que uma sub-meta tem de ser satisfeita, Prolog retrocede pelo banco de dados, sempre procurando de cima para baixo e da esquerda para a direita, para encontrar uma concordância. O retrocesso poderá, facilmente, ficar mais complexo quando a regra e as metas forem mais complexas.

Procurando pelos fatos, Prolog rapidamente encontra o fato procurado, aplica a sub-meta aquele fato e volta à regra originaria, onde sabe que o lado esquerdo da regra poder ser utilizado para concordar com a meta original, porque o lado direito de regra era "True" (verdadeiro).

Na execução dos programas Prolog, a evolução da busca por soluções assume a forma de uma árvore, denominada "árvore de pesquisa" ou "search tree" - que é percorrida sistematicamente de cima para baixo (top-down) e da esquerda para direita, segundo o método denominado "Prof-first search" ou "pesquisa primeiro em profundidade". A figura 6 ilustra esta idéia. Ali é representada a árvore correspondente à execução do seguinte programa abstrato, onde a, b, c, etc. possuem a sintaxe de termos Prolog:

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

Figura 6 - Ordem de visita aos nodos da árvore de pesquisa



O programa representado pela figura 6 será bem sucedido somente quando o nodo **d** for atingido, uma vez que este é o único fato declarado no programa. De acordo com a ordenação das cláusulas, d será também o último nodo a ser visitado no processo de execução. O caminho percorrido é dado abaixo

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

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

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

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

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

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

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

gosta(joão, X), gosta(renata, X).

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

1 Encontra que joão gosta de jazz

2 Instancia X com "jazz"

3 Tenta satisfazer o segundo objetivo, determinando se "renata gosta de jazz"

4 Falha, porque não consegue determinar se renata gosta de jazz

5 Realiza um backtracking na repetição da tentativa de satisfazer gosta(joão, X), esquecendo o valor "jazz"

6 Encontra que joão gosta de renata

7 Instancia X com "renata"

8 Tenta satisfazer o segundo objetivo determinando se "renata gosta de renata"

9 Falha porque não consegue demonstrar que renata gosta de renata

10 Realiza um backtracking, mais uma vez tentando satisfazer gosta(joão, X), esquecendo o valor "renata"

11 Encontra que joão gosta de lasanha

12 Instancia X com "lasanha"

13 Encontra que "renata gosta de lasanha"

14 É bem-sucedido, com X instanciado com "lasanha"

O backtracking automático é uma ferramenta muito poderosa e a sua exploração é de grande utilidade para o programador. Às vezes, entretanto, ele pode se transformar em

fonte de ineficiência e por isto é preciso inserir um mecanismo (CUT) para "podar" a árvore de pesquisa, evitando o backtracking quando este for indesejável.

4 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 em 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.

O Arity Prolog é composto por uma série de menus que são ativados através do mouse ou por uma determinada combinação de teclas. Todas as opções têem suas funções específicas, sendo as principais, as funções para manipulação de arquivos, de edição, de compilação e execução de programas

4.1 FORNECEDORES E VERSÕES

Existem vários fornecedores e ambientes de Prolog, mas dois se destacam com maior freqüência entre os demais pela eficiência e divulgação de seus programas/softwares.

A primeira delas é a Arity , fundada em 1984 por gerentes técnicos sêniores da Lotus Corporação de Desenvolvimento. A visão para a Arity é desenvolver um paradigma de administração de informação novo que permitirá que os negócios adquiram verdadeira compreensão a estimar/utilizar os recursos de dados nele disponível. O primeiro e mais famoso produto é o Arity/Prolog™, cujo qual é o compilador principal de Prolog no mercado com mais de 15.000 licenças em mais de 55 países.

A Arity construiu o seu sucesso com Arity/Prolog e continuaram desenvolvendo aplicações para máquinas de banco de dados sofisticadas. O trabalho culminou na descoberta do Arity Engine™, uma tecnologia sem igual que habilita uma classe nova importante de aplicações baseadas em redes tendo resultado assim no surgimento de outros programas.

O segundo fabricante importante da linguagem Prolog é a LPA Prolog cuja qual é uma software house que provê ferramentas de software inteligentes para soluções de indústrias. Os softwares da LPA tem como objetivos principais trabalhar em duas plataformas específicas, a plataforma PC - Windows e a plataforma da linha Machintoch, como também a internet. O alcance de seus produtos incluem compiladores de Prolog e Servidores ProWeb, dentre outras linhas de programas.

4.2 FATOS

O tipo mais simples de uma declaração é chamado fato. Fatos declaram que existe algum relacionamento entre os objetos. A primeira maneira de combinar um objeto e um

relacionamento é utilizando-os para definir um *fato* [ROB88]. A sintaxe do Arity Prolog é a seguinte: `relação(objeto)`. Um exemplo de fato pode ser: pai(adao,caim).

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

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

Na medida em que vamos utilizando o Prolog, temos que manter sob controle o que os relacionamentos representam. Prolog não pode fazer isto por nós. O programa somente fará sentido se formos consistentes durante um programa inteiro no que diz respeito ao significado de um dado relacionamento. Algumas vezes utilizar de relacionamentos que mais se aproximam daquilo que queremos dizer ajuda. Por exemplo, se quisermos indicar o fato de que charles é um príncipe, poderemos utilizar este relacionamento com ele como objeto:

```
e_um_principe(charles).    Charles é um príncipe.
```

Os travessões indicam ao computador e compilador que isto é uma única palavra longa para um relacionamento

Alguns outros exemplos de fatos:

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

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

dependendo de qual seu objetivo. O número de argumentos de um predicado é chamado de aridade.

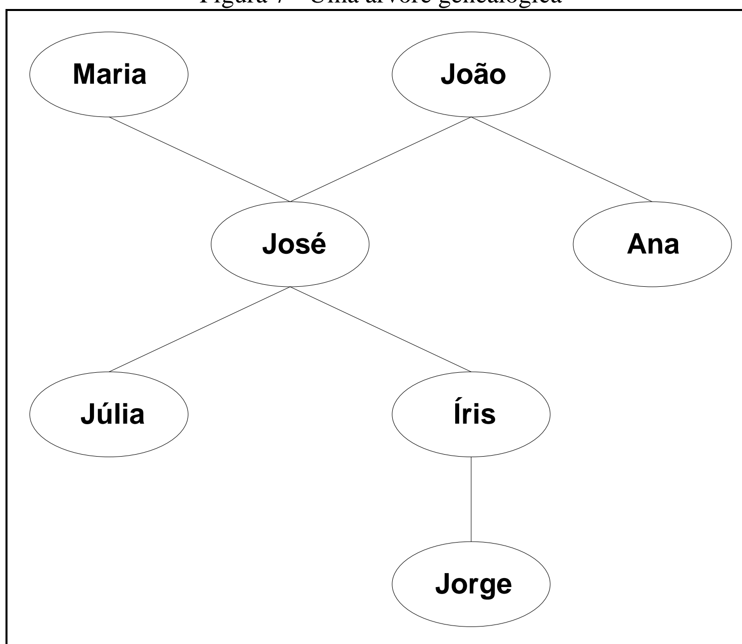
Pode-se declarar fatos que não sejam realidade no mundo real. Pode-se dizer que:

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

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

Uma maneira de compreender melhor o que significa um fato é expresso melhor no exemplo da figura 7.

Figura 7 - Uma árvore genealógica



É possível definir, entre os objetos (indivíduos) mostrados na figura 7 uma relação denominada *progenitor* que associa um indivíduo a um dos seus progenitores. Por exemplo, o fato de que João é 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. Note que os nomes de pessoas (como joão) iniciam-se com letras minúsculas, pois são átomos e não variáveis (como X). A relação progenitor completa, como representada na figura 7 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).
```

O programa acima compõe-se de seis *cláusulas,* cada uma das quais denota um fato acerca da relação *progenitor.* Se o programa for submetido a um sistema Prolog, este será capaz de responder algumas questões sobre a relação ali representada. Por exemplo: *"José é o progenitor de Íris?".* Uma consulta como essa deve ser formulada ao sistema precedida por um "?-". Esta combinação de sinais denota que se está formulando uma pergunta. 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.

```
?-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 feita 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 assertiva *"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 filhos de José. Note que, neste caso, mais de uma resposta verdadeira pode ser encontrada. O sistema 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
```

Aqui a última resposta obtida foi *"não"* significando que todas as soluções válidas 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 (enquanto se desejar, digitando ";") todos os pares progenitor-filho até que estes se esgotem (quando então responde "não") ou até que se resolva encerrar a apresentação de novas soluções (digitando "."). No exemplo a seguir iremos nos satisfazer com 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 complicadas ao programa, como *"Quem são os avós de Jorge?"*. Como nosso programa não possui diretamente a relação *avô*, esta consulta precisa ser dividida em duas etapas, como pode ser visto na figura 8.

A saber:

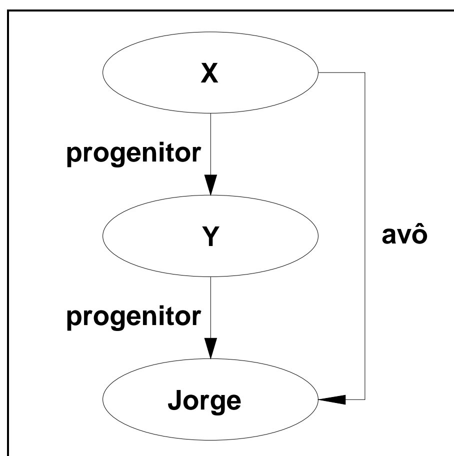
(1) Quem é progenitor de Jorge? (Por exemplo, Y) e

(2) Quem é progenitor de Y? (Por exemplo, X)

Esta 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 8 - A relação avô em função de progenitor



Observe que se é mudada a ordem das consultas na composição, o significado lógico permanece o mesmo, apesar do resultado ser informado na ordem inversa:

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

De modo similar podemos perguntar: *"Quem é neto de João?"*:

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

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
```

Por meio dos exemplos apresentados até aqui acredita-se ter sido possível ilustrar os seguintes pontos:

- Uma relação como *progenitor* pode ser facilmente definida em Prolog estabelecendo-se as *tuplas* de objetos que satisfazem a relação;

- O usuário pode facilmente consultar o sistema Prolog sobre as relações definidas em seu programa;
- Um programa Prolog é constituído de *cláusulas*, cada uma das quais é encerrada por um ponto (.);
- Os argumentos das relações podem ser objetos concretos (como júlia e íris) ou objetos genéricos (como X e Y). Objetos concretos em um programa são denominados *átomos,* enquanto que os objetos genéricos são denominados *variáveis*;
- Consultas ao sistema são constituídas por um ou mais *objetivos,* cuja seqüência denota a sua conjunção;
- Uma resposta a uma consulta pode ser *positiva* ou *negativa*, dependendo se o objetivo correspondente foi alcançado ou não. No primeiro caso dizemos que a consulta foi *bem-sucedida* e, no segundo, que a consulta *falhou;*
- Se várias respostas satisfizerem a uma consulta, então o sistema Prolog irá fornecer tantas quantas forem desejadas pelo usuário.

4.3 REGRAS

Uma regra típica diz que alguma coisa é verdadeira (uma meta será bem sucedida) se algumas outras coisas são verdadeiras. Regras leva Prolog além do estado de um mero dicionário de pesquisa ou banco de dados até chegar a uma máquina lógica pensante [ROB88].

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

```
progenitor(maria, josé).
progenitor(joão, josé).
progenitor(joão, ana).
progenitor(josé, joana).
homem(joão).
homem(josé).
mulher(maria).
mulher(joana).
mulher(ana).
```

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

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

```
filho(josé, joão).
```

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

Para todo X e Y

```
Y é filho de X se
X é progenitor de Y.
```

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

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

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

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

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

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

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

(conclusão)   (condição)

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

```
?-filho(josé, maria).
```

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

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

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

Para todo X e Y

```
X é mãe de Y se
X é progenitor de Y e
X é feminino.
```

Que, traduzida para Prolog, conduz à seguinte regra:

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

Por meio dos exemplos apresentados até aqui acredita-se ter sido possível ilustrar os seguintes pontos:

- As cláusulas Prolog podem ser de três tipos distintos: fatos, regras e consultas;

- Os fatos declaram coisas que são incondicionalmente verdadeiras;
- As regras declaram coisas que podem ser ou não verdadeiras, dependendo da satisfação das condições dadas;
- Por meio de consultas pode-se interrogar o programa acerca de que coisas são verdadeiras;
- As cláusulas Prolog são constituídas por uma cabeça e um corpo. O corpo é uma lista de objetivos separados por vírgulas que devem ser interpretadas como conjunções;
- Fatos são cláusulas que só possuem cabeça, enquanto que as consultas só possuem corpo e as regras possuem cabeça e corpo;
- Ao longo de uma computação, uma variável pode ser substituída por outro objeto. Diz-se então que a variável está instanciada;
- As variáveis são assumidas como universalmente quantificadas nas regras e nos fatos e existencialmente quantificadas nas consultas.

4.4 RECURSIVIDADE

Recursividade ou retrocesso é um elemento essencial do Prolog, quando se pede a um programa para atender uma meta , procura de cima para baixo e da esquerda para a direita pelas cláusulas que combinam com a meta. Caso encontre um beco sem saída, o programa retrocede o suficiente nas cláusulas para encontrar outro ramo que possa ser pesquisado.

Esta não é sempre a melhor forma de um programa trabalhar. Alguns programas consomem muito tempo recuperando informações desnecessárias durante a procura daquilo que importa. Estes programas necessitam da ajuda do programador para diminuir este universo enorme de procura. Os computadores são vulneráveis à "explosão combinatória" que pode acontecer em Prolog com apenas alguns níveis de metas e sub-metas. A física atômica fornece um exemplo ilustrativo do que pode acontecer. Uma bomba atômica é baseada na reação em cadeia de uma desintegração nuclear: um único neutron rompe um átomo, resultando em dois neutrons, que rompem dois átomos, resultando em quatro neutrons, que rompem quatro átomos, resultando em oito neutrons, que rompem oito átomos e assim por diante. Este tipo de cadeia progride geometricamente. Seu programa de computador tam-

bém poderá se tornar uma bomba devido ao aumento geométrico ou logarítmico de usa área de pesquisa. Se cada meta leva as duas sub-metas e cada sub-meta, por sua vez, leva a mais duas sub-metas, e assim em diante, o número de metas a serem satisfeitas pode rapidamente se multiplicar a níveis impraticáveis.

Este tipo de cadeia poderá rapidamente sobrepujar um computador e levar a tempos de pesquisa excessivamente longos. Imagine o número de tentativas de combinações que Prolog terá de fazer se uma meta contiver diversas cláusulas, cada uma das quais tendo de ser testada contra cada linha em um grande banco de dados. Depois, deve-se multiplicar o tempo que isto vai levar, porque a procura vai, possivelmente, encontrar diversas regras e , portanto, sub-metas que , por sua vez, têm de ser comparadas com todo o banco de dados. Brevemente você estará aguardando que Prolog pare de procurar.

4.5 LISTA E ÁRVORES

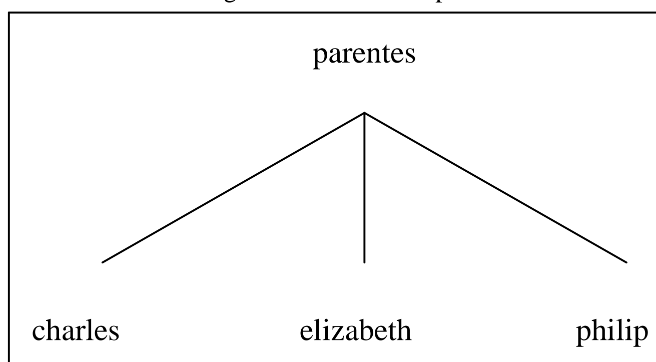
Lista é apenas outra forma de um objeto composto, mas é uma importante estrutura de dados. Parece uma coleção de termos - neste caso, elementos - separados por vírgulas e colocados no interior de colchetes. Aqui está uma lista de inteiros:

[1,2,3,5,8,13]

As listas são estruturas de dados comuns na programação não numérica. A lista é uma sucessão ordenada de elementos que podem ter alguma continuação. Os elementos de uma lista podem ser representadas por - constantes, variáveis ou estruturas. Estas propriedades são úteis quando nós não podemos predizer com antecedência como grande uma lista deveria ser, e que informação que deveria conter. Além disso, listas podem representar praticamente qualquer tipo de estrutura que pode-se ser utilizada em computação simbólica. Listas são amplamente utilizadas para análise de árvores, gramáticas, mapas de cidades, programas de computador e entidades matemáticas como gráficos, fórmulas e funções. Há um idioma de programação chamado LISP no qual uma única estrutura de dados disponível é a constante e a lista. Porém, em Prolog a lista é simplesmente um tipo particular de estrutura. As Listas podem ser representadas por um tipo especial de árvore.

Normalmente é mais fácil de entender a forma de uma estrutura complicada se esta for escrita em forma de uma árvore na qual cada elemento é um nodo, e o componentes são folhas/filhos. Cada folha pode apontar para uma outra estrutura, assim pode-se ter estruturas dentro de estruturas. É habitual escrever um diagrama de árvore com a raiz no topo e as folhas a baixo. Por exemplo, uma estrutura de parentes pode ser escrita assim: parentes(charles, elizabeth ,philip) como demonstra a figura 9.

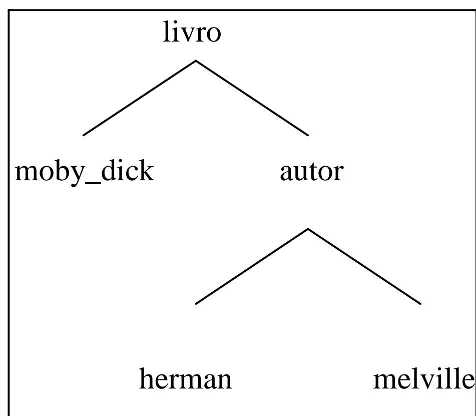
Figura 9 - Árvore de parentes



A estrutura de livros pode ser escrita assim como na figura 10.

livro(moby_dick,autor(herman,melville)):

Figura 10 – Árvore do livro/ autor



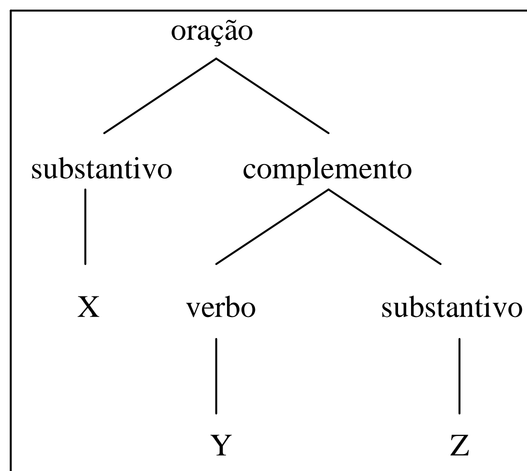
Note que as últimas duas estruturas têm árvores da mesma forma, embora as raízes e folhas são diferentes.

Suponha que precisa-se determinar e representar a oração " João gosta de Maria", uma sintaxe muito simples para o português cuja a oração consiste em um substantivo se-

guido de verbo. Pode-se representar a estrutura de qualquer oração por uma estrutura da forma da figura 11:

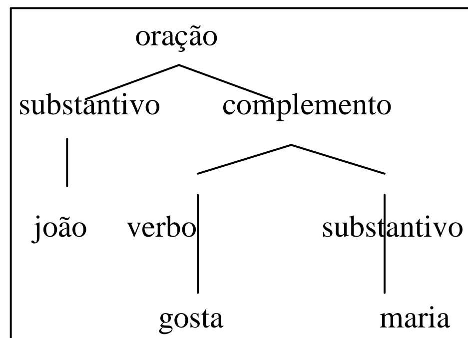
oração (substantivo (X), componentes (verbo (Y), substantivo (Z)));

Figura 11 – Árvore da Oração



Se na oração " João gosta de Maria " fosse instanciada as variáveis da estrutura com as palavras da oração, poderia obter-se o resultado da figura 12:

Figura 12 – Árvore da Oração com nomes

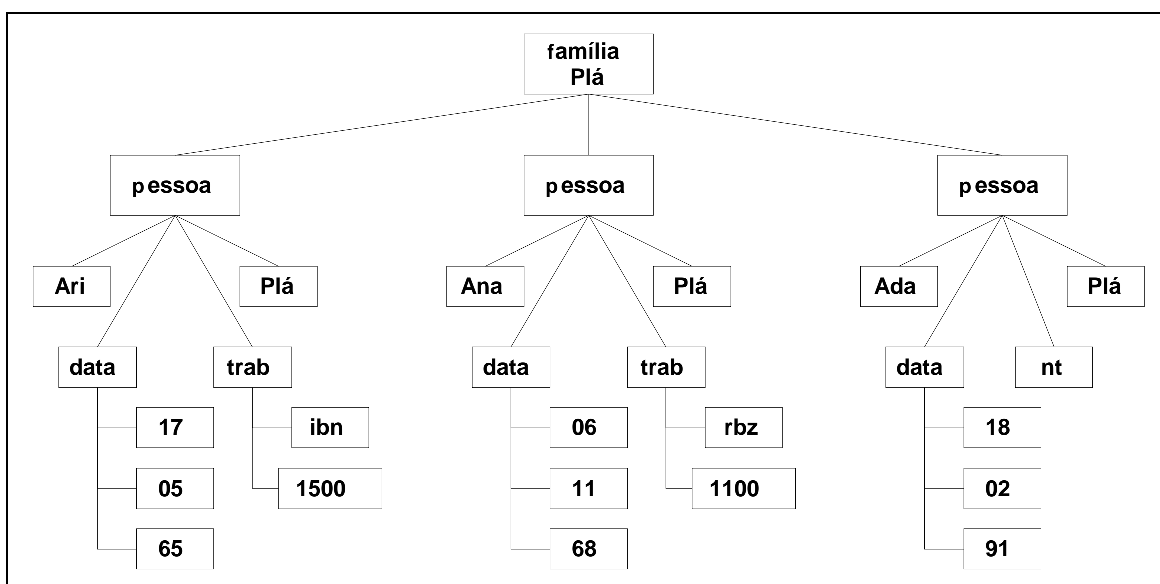


Isto mostra como pode-se usar as estruturas e variáveis do Prolog para representar a sintaxe de uma orações muito simples. Em geral, se nós sabemos as classes de palavras de uma oração é possível escrever uma estrutura de Prolog que faz descreve explicitamente as relações entre palavras diferentes em uma oração. Este é um tópico interessante e com ele podemos usar Prolog para fazer o computador " entender " algumas orações simples.

4.6 CONSULTAS

O exemplo apresentado a seguir desenvolve a habilidade de representar e estruturar objetos de dados e também ilustra a visão do Prolog como uma linguagem natural de consulta a bases de dados, segundo a figura 13:

Figura 13 - Informação estruturada sobre uma família



Uma base de dados pode ser naturalmente representada em Prolog como um conjunto de fatos. Por exemplo, uma base de dados sobre famílias pode ser representada de modo que cada família seja descrita como um termo. A figura 13 mostra como a informação sobre cada família pode ser estruturada em um termo família/3, com a seguinte forma:

```
família(Pai, Mãe, Filhos)
```

Onde Pai e Mãe são pessoas e Filhos é uma lista de pessoas. Cada pessoa é, por sua vez, representada por uma estrutura com quatro componentes: nome, sobrenome, data de nascimento e trabalho. A data de nascimento é fornecida como um termo estruturado data(Dia, Mes, Ano). O trabalho, ou é fornecido por um termo trab(Empresa, Salário), ou pela constante nt, indicando que a pessoa em questão não trabalha. A família exemplificada pode então ser armazenada na base de dados como uma cláusula do tipo:

```
família(
     pessoa(ari, plá, data(17, 05, 65), trab(ibn, 1500)),
```

```
                pessoa(ana, plá, data(06, 11, 68), trab(rbs, 1100)),
                       [pessoa(ada, plá, data(18, 02, 91), nt)]
                   )
```

A base de dados poderia ser vista então como uma seqüência de fatos, descrevendo todas as famílias que interessam ao programa. A linguagem Prolog é, na verdade, muito adequada para a recuperação da informação desejada a partir de uma base de dados. Um detalhe muito interessante é que os objetos desejados não precisam ser completamente especificados. Pode-se simplesmente indicar a estrutura dos objetos que interessam e deixar os componentes particulares apenas indicados. Por exemplo, para recuperar-se todas as famílias "Oliveira", basta especificar:

```
?-família(pessoa(_, oliveira, _, _), _, _).
ou as famílias cujas mães não trabalham:
?-família(_, pessoa(_, _, _, nt), _).
as famílias que não possuem filhos:
?-família(_, _, []).
ou ainda famílias que possuem três ou mais filhos:
?-família(_, _, [_, _, _| _]).
```

As possibilidades de consulta são as mais diversas. Com esses exemplos procura-se demonstrar que é possível especificar os objetos de interesse, não pelo seu conteúdo, mas sim pela sua estrutura, sobre a qual restringimos os componentes conforme nossas necessidades e/ou disponibilidades, deixando os demais indefinidos. Na figura 14 é apresentado um programa demonstrando algumas das relações que podem ser estabelecidas em função de uma base de dados estruturada na forma definida por família.

Figura 14 - Um programa baseado na relação família

```
pai(X) ¬
      família(X, _, _).
mãe(X) ¬
      família(_, X, _).
filho(X) ¬
      família(_, _, Filhos),
      membro(X, Filhos).
membro(X, [X|_]).
membro(X, [_|Y]) ¬
      membro(X, Y).
existe(Pessoa) ¬
      pai(Pessoa);
      mãe(Pessoa);
      filho(Pessoa).
nasceu(pessoa(_, _, Data, _), Data).
salário(pessoa(_, _, _, trab(_, S)), S).
salário(pessoa(_, _, _, nt), 0).
```

Algumas aplicações para os procedimentos mostrados na figura 14 podem ser encontrados nas seguintes consultas à base de dados:

- Achar o nome e sobrenome de todas as pessoas existentes na base de dados:

```
?-existe(pessoa(Nome, Sobrenome, _, _)).
```

- Achar todas as crianças nascidas em 1993:

```
?-filho(X), nasceu(X, data(_, _, 93)).
```

- Achar todas as pessoas desempregadas que nasceram antes de 1976:

```
?-existe(pessoa(_, _, data(_, _, A), nt), A < 76.
```

- Achar as pessoas nascidas após 1965 cujo salário é maior do que 5000:

```
?-existe(Pessoa),
  nasceu(Pessoa, data(_, _, A)),
  A > 65,
  salário(Pessoa, Salário),
  Salário > 5000.
```

Para calcular o total da renda familiar, pode ser útil definir a soma dos salários de uma lista de pessoas como uma relação de dois argumentos:

```
total(L, T)
```

que pode ser declarada em Prolog como mostrado a seguir:

```
total([], 0).
total([Pessoa | Lista], Total) ¬
        salário(Pessoa, Salário)
        total(Lista, Soma),
        Total is Soma + Salário.
```

Esta relação nos permite interrogar a base de dados para saber a renda familiar de cada família:

```
?-família(Pai, Mãe, FIlhos), total([Pai, Mãe | Filhos], RFam).
```

5 O PROTÓTIPO DESENVOLVIDO

Neste capítulo encontra-se o protótipo desenvolvido e procura-se mostrar uma visão geral de como se chegou a realização deste protótipo, bem como os comandos utilizados, rotinas do programa e as telas de saída do mesmo.

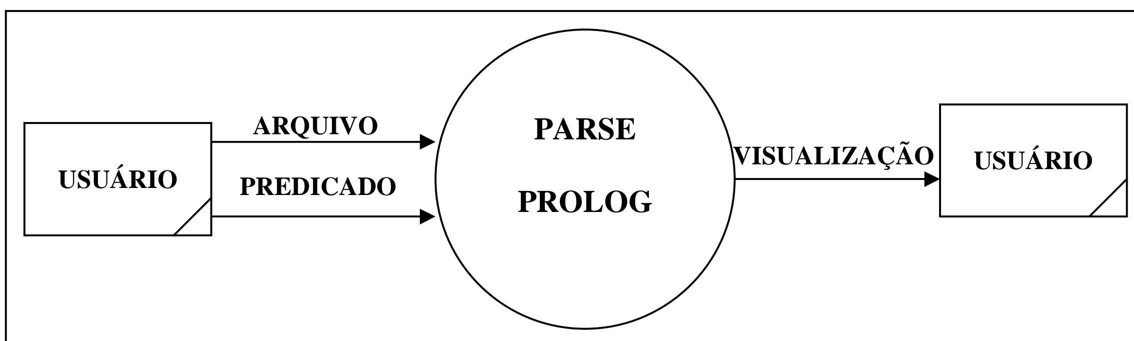
5.1 OBJETIVOS

Os objetivos pretendidos por este trabalho são de auxiliar o aprendizado da linguagem de programação lógica Prolog, mas especificamente o Arity Prolog e fazer com que a compreensão seja a mais clara possível para o usuário. O que será demonstrado através de algumas cláusulas, o que faz e aonde se encontra a execução do programa que pretende-se executar. Pois o mesmo percorre uma série de fatos e regras para validar uma determinada informação, sendo que há 4 estágios/saídas em que o programa pode estar: call, exit, fail, ou sucess. Todos estes estágios são demonstrados pelo protótipo.

5.2 ESPECIFICAÇÃO

O protótipo possui duas entradas, ambas por parte do usuário: o arquivo, que é um programa feito no Arity Prolog e um predicado deste arquivos/programa. Possui também uma saída, que é a visualização do processamento das informações pelo Arity Prolog em vídeo, através de janelas demonstrativas. Isto pode ser observado no diagrama de contexto da figura 15. O usuário fornece, através da interface, um arquivo texto onde está contido um programa no formato Arity Prolog e logo depois é fornecido um predicado válido deste arquivo que deseja-se visualizar. O arquivo é carregado na base de dados e o predicado informado é submetido à regras para validação e visualização dos resultados. O usuário pode então acompanhar o funcionamento do processamento das informações pelo Arity Prolog e verificar se sua regra foi satisfeita ou não. Se ela for bem sucedida retornará uma resposta “yes” de positiva, caso contrário, a resposta será “no” de negativa.

Figura 15 - Diagrama de Contexto do Visualizador Prolog



A especificação do protótipo proposto é apresentada a seguir pela notação BNF (Backus-Naur-Form) uma técnica formal utilizada para especificações:

<regra>::= <condição>

<condição>::= <cláusula1> | <cláusula2> | <cláusula3> | <cláusula4>

<cláusula1>::= <predicado> e <predicado> e <valor>

<cláusula2>::= <sistema> e <valor>

<cláusula3>::= <predicado> e <valor>

<cláusula4>::= <predicado> e <valor>

<predicado>::= = = | !=

<valor>::= 1 | 2 | 3 | 4 | 5 | ....|10

<sistema>::= <predicado_sistema>

<predicado_sistema>::= write | read | keyb | nl | ....| open | ....

Esta notação acima descreve como funciona o protótipo proposto. Como já mencionado anteriormente o usuário entra com um arquivo do Arity Prolog e informa um predicado(regra), este predicado/regra deve satisfazer uma das 4 condições comentadas anteriormente. Se este predicado se enquadrar em uma das 4 situações proposta pelo trabalho, ou seja, ele pode ser um predicado com outros predicados, quer dizer uma conjunção, avo_de(X,Y),sócio(X), pode ser um predicado do sistema, write(X), ou seja, palavras reser-

vadas do sistema, um predicado simples avo_de(X,Y) ou um predicado que não esteja correto ou que não atenda as respostas do usuário, sócio(X), não foi encontrada nenhuma resposta para este predicado, ou predicado não existe.

Dependo de qual for a condição, esta será exibida em vídeo com uma identação proveniente da variável valor.

5.3 COMPONENTES E PRINCIPAIS PROGRAMAS FONTES

Neste capítulo é abordado todos os elementos que fizeram parte da elaboração do protótipo proposto, como comandos, predicados e programas fontes.

5.3.1 PREDICADOS INTERNOS

O Arity Prolog oferece um conjunto de predicados que podem ser utilizados pelo programador. As principais funções destes predicados estão relacionadas as operações de entrada e saída de dados, criação de menus e janelas, manipulação de strings, interface com o ambiente operacional e outros. No desenvolvimento deste protótipo utilizou-se largamente destes predicados. Os predicados mais comuns e suas ações serão amostrados a seguir, agrupados por funções.

a) PREDICADOS RELACIONADOS À ENTRADA E SAÍDA DE DADOS

read(X) – recebe do teclado o termo X, entrada de dados.

write(X) – mostra o termo X no monitor, saída de dados.

nl – provoca o salto para uma nova linha, quebra de linha.

get0_noecho – recebe um caracter sem escrevê-lo na tela, sem ecoá-lo.

open(H,Q,r) – abre o arquivo existente Y para o modo de acesso r associando H como o seu nome lógico.

b) PREDICADOS RELACIONAODS À UTILIZAÇÃO DA TELA

define_windows (N,L,(Lse,Cse),(Lid,Cid),(Wa,Ba)) - define uma janela de nome N com cabeçalho L que inicia na linha número Lse e na coluna número Cse do monitor e Termina na linha Lid e coluna Cid do monitor. Wa define a cor da janela e Ba o tipo e cor das bordas.

current_windows(V,N) – coloca a janela N sobre a janela V na tela. N passa a ser janela corrente.

tmove(L,C) – desloca o cursor para a linha L e coluna C da janela corrente.

cls – limpa a tela.

c) PREDICADOS PARA MANIPULAÇÃO DE STRINGS

concat(S1,S2,Sr) – retorna em Sr a concatenação dos strings S1 e S2.

d) PREDICADO DE INTERFACE COM O AMBIENTE OPERACIONAL

consult(F) – incorpora as cláusulas existentes no arquivo F ao banco de dados.

e) PREDICADOS DIVERSOS

not(X) – falha se X puder ser satisfeito e tem sucesso se falhar.

clause(X,Y) - unifica a cabeça X e o corpo Y com a respectiva cabeça e corpo da cláusula. A cabeça precisa ser instanciada.

nonvar(X) – tem sucesso ser X não for uma variável instanciada.

fail – força o predicado a falhar.

Os predicados internos acima descritos, são os mais utilizados pelo protótipo. Além destes o Arity Prolog oferece dezenas de outros que não foram citados. Para o conhecimento de todos os predicados internos disponibilizados deve ser consultado o manual da linguagem, pois não seria possível relacioná-los neste trabalho.

5.3.2 PREDICADOS CRIADOS

A seguir serão mostrados alguns predicados criados no protótipo. Estes predicados são as principais rotinas implementadas no trabalho. Inicialmente é descrita a finalidade do

predicado e em seguida o seu código e o de outros predicados diretamente relacionado ao mesmo.

mostra – este predicado é responsável por gerar em tela todas as saídas dos predicados, ele gera um tamanho de espaço, depois faz uma tabulação (identamento), escreve o predicado pesquisado, pula uma linha e gera uma pausa até que se pressione uma tecla qualquer.

mostra(A,Prof):- Spacing is 3*Prof, tab (Spacing), write (A), nl, get0_noecho(C).

menu – criado para gerar a tela de abertura, entrada de dados e para dar início ao processo de visualização de execução do Arity Prolog.

menu :- define_window(teste,'Prototipo de Apoio ao Aprendizado do Arity_Prolog',(2,2),(21,79),(7,-47)), current_window(_,teste), tmove(2,2),write('FURB–Fundacao Universidade Regional de Blumenau'), tmove(3,2), write('Centro de Ciencias Exatas e Naturais'), tmove(4,2), write('Depto de Sistemas e Computacao'), tmove(5,2), write('Curso: Bacharelado em Ciencias da Computacao'), tmove(6,2), write('Orientador: Roberto Heinzle'), tmove(8,2), write('Prototipo de Software de Apoio ao Aprendizado'), tmove(9,7), write('de Programacao Logica - PROLOG'),tmove(11,2),write('Academico: Wagner Moreira Stahnke'), tmove(15,2), write('<<pressione qualquer tecla>>'), get0_noecho(W), cls, tmove(2,2), write('Diga o nome do arquivo sem extensao: '), read(T), concat(T,$.ari$,Q),open(H,Q,r), tmove(4,2), write('Diga seu predicado principal: '), read(V), consult(Q), resolve(V).

resolve – este predicado é o predicado responsável em enquadrar em que nível se encaixa o predicado do usuário e também se encarrega de chamar o predicado *mostra* para exibir o resultados na tela e gerar as janelas informativas de estágio através do predicado *define_window*.

```
resolve(Goal) :- resolve(Goal,0).

resolve((A,B),Prof):-
define_window(executa1,'CALL',(3,3),(10,66),(7,-47)),
current_window(_,executa1), cls, tmove(2,2), write($CALL:$), nl,
mostra(A,Prof),resolve(A,Prof),
define_window(executa2,'CALL',(3,3),(10,66),(7,-47)),
current_window(_,executa2), cls, tmove(2,2), write($CALL:$), nl,
mostra(B,Prof), resolve(B,Prof).

resolve(A,Prof) :- system(A), A, mostra(A,Prof), nl.

resolve(A,Prof):- clause(A,B), nonvar(B), B = true,
define_window(executa4,'EXIT',(8,8),(15,72),(7,-47)),
current_window(_,executa4), cls, tmove(2,2), write($EXIT:$), nl,
mostra(A,Prof).

resolve(A,Prof) :- clause(A,B), nonvar(B), B \= true,
Prof1=Prof+1,
define_window(executa5,'CALL',(3,3),(10,66),(7,-47)),
current_window(_,executa5), cls, tmove(2,2), write($CALL:$), nl,
mostra(B,Prof1),resolve(B,Prof1),
define_window(executa6,'EXIT',(8,8),(15,72),(7,-47)),
current_window(_,executa6), cls, tmove(2,2), write($EXIT:$), nl,
mostra(A,Prof).

resolve(A,Prof) :- nonvar(A), not clause(A,B),
define_window(executa7,'FAIL',(14,14),(19,76),(7,-47)),
current_window(_,executa7), cls, tmove(2,2), write($FAIL:$), nl,
mostra(A,Prof), fail.
```

5.4 AMBIENTE DE DESENVOLVIMENTO

Os recursos usados no desenvolvimento do protótipo foram:

a) hardware:
   - Microcomputador Intel MMX 200 MHz;

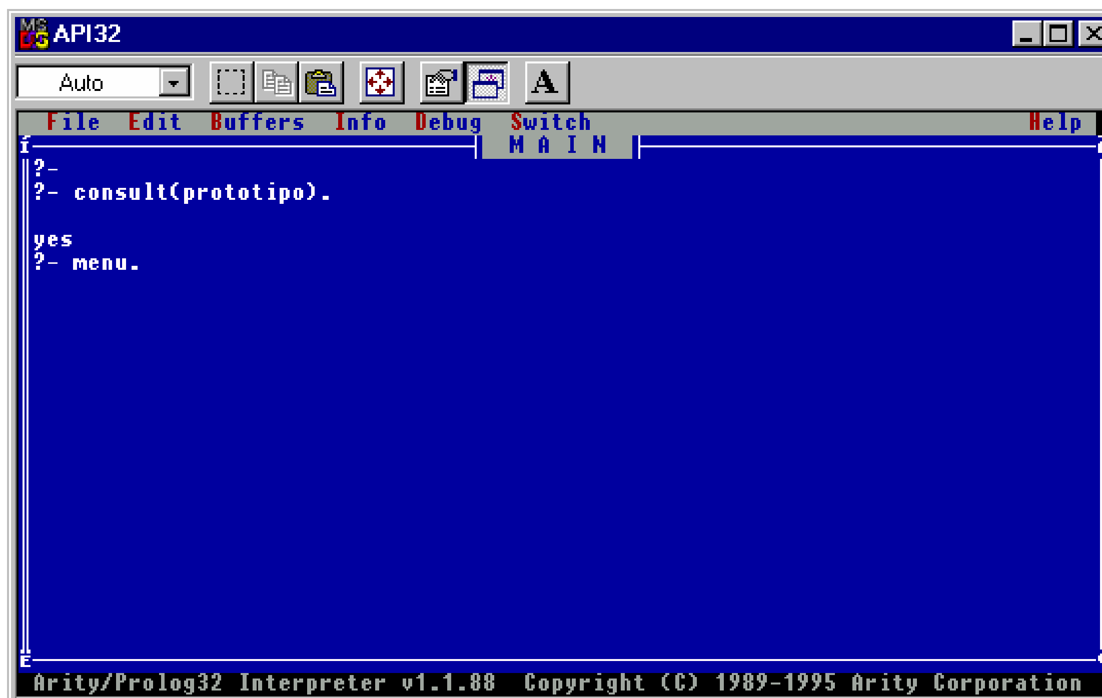
b) software:

- Windows 98;
- um ambiente Prolog completo (Editor, interpretador, SGDB e compilador). Mais especificamente o Arity/Prolog32 V1.1.88.

5.5 TELAS

O protótipo baseia-se em três telas: uma tela de abertura com algumas informações típicas, uma tela de entrada de dados e por último, uma tela de visualização dos predicados. Sendo que é necessário utilizar a janela inicial do Ambiente Arity Prolog, onde o usuário carrega o protótipo através do comando reservado do Arity "consult(nome arquivo).". Depois o usuário deverá informar o nome de um arquivo válido no formato do Arity Prolog e para terminar deve ser digitado o predicado que deseja-se analisar, conforme demonstrado nas figuras 16, 17 e 18.

Figura 16 – Tela de Carregamento do Protótipo



A figura 16 demonstra, como o protótipo deve ser carregado através do comando consult e como deve ser inicializado a execução do mesmo através do predicado principal denominado de menu. Todas as demais opções oferecidas pela barra de menu, como file, edit, etc..., são referentes ao ambiente Arity Prolog.

Figura 17 – Tela de Abertura

```
Prototipo de Apoio ao Aprendizado do Arity Prolog - APLOG

FURB - Fundação Universidade Regional de Blumenau
Centro de Ciências Exatas e Naturais
Depto de Sistemas e Computação
Curso: Bacharelado em Ciências da Computação
Orientador: Roberto Heinzle

Protótipo de Software de Apoio ao Aprendizado da Linguagem
            de Programação Lógica - PROLOG

Acadêmico: Wagner Moreira Stahnke


<<pressione qualquer tecla>>
```

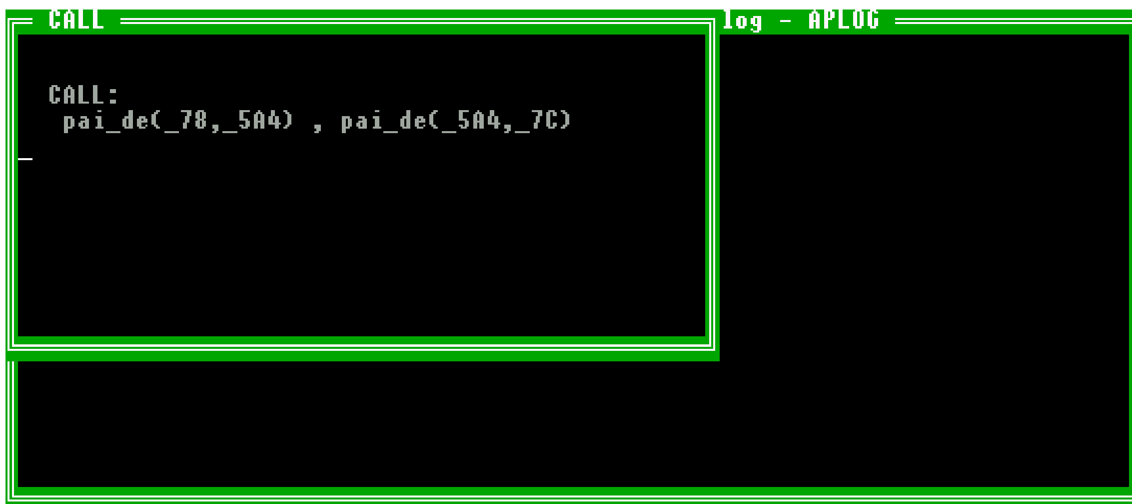
Figura 18 – Tela de Entrada de Arquivo e Predicado

```
Prototipo de Apoio ao Aprendizado do Arity Prolog - APLOG

Diga o nome do arquivo sem extensão: teste.

Diga seu predicado principal: avo_de(X,Y)._
```

A figura 18 demonstra como é feita a entrada dos dados, primeiro deve-se informar o nome de um arquivo feito no Arity Prolog sem extensão, caso o arquivo não for válido o

programa é abortado. Depois deve-se informar o predicado que deseja-se verificar a execução, o mesmo deve se encontrar dentro do arquivo digitado anteriormente, caso também o predicado não seja válido o programa irá detectar e vai gerar uma falha, retornando uma resposta negativa "no" ao usuário.

Figura 19 – Tela de Visualização de Predicados



A figura 19 representa a visualização do predicado digitado anteriormente, nesta situação o predicado está sendo carregado (call) para ser feita a visualização de todos os seus passos. Esta visualização ajuda a compreender aonde se encontra e para onde vai a execução do predicado, ajudando aos principiantes que estejam interessados à entender o funcionamento da programação lógica.

5.5.1 EXEMPLOS E TESTES

Para demonstrar a execução deste protótipo elaborou-se dois predicados para teste, são eles: teste e teste1. Mas, vale lembrar que qualquer outro exemplo funcionará. Pressionando a tecla "enter", a primeira condição que for satisfeita vai ser bem sucedida, caso queira-se outra condição deve ser apertado a tecla ";" (ponto e vírgula) para o Arity buscar outras condições.

Predicado teste

`Árvore genealógica simples`

```
pai_de('Ari Fontoura','Jonas Fontoura').
pai_de('Juliana Fontoura','Jonas Fontoura').
pai_de('Jonas Fontoura','Mario Fontoura').
pai_de('Joao da Silva','Jose da Silva').
pai_de('Jose da Silva','Zeca da Silva').
avo_de(X,Y) :- pai_de(X,Z), pai_de(Z,Y).
```

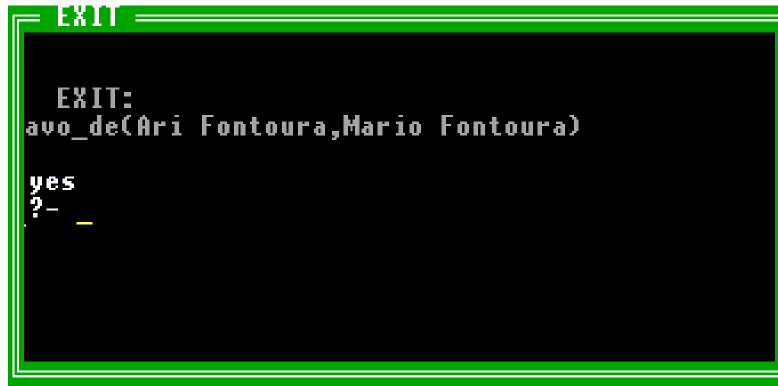
Predicado teste1

Membros de uma lista

```
associados([wagner,tania,evelin,ivo,luzimar,clara,opa]).
socio(X) :- associados(Y),membro(X,Y).
membro(X,[X|_]).
membro(X,[_|Y]) :- membro(X,Y).
```

A seguir será demonstrado uma pesquisa pelo avô da família Fontoura, note que na figura 20 a reposta é positiva, ou seja, a condição encontrada é válida.

Figura 20 – Visualização Avô da Família Fontoura



Na figura 20 a reposta gerada foi positiva “yes”, ou seja, Mário Fontoura é o avô de Arí Fontoura, caso isto não fosse verdadeiro a resposta gerada seria “no” de negativa.

O outro exemplo elaborado para visualizar em qual estágio se encontra o Arity Prolog se baseia em uma determinada lista de sócios, onde tenta-se verificar se uma determinada pessoa pertence a lista. É parecido com uma operação de conjuntos da matemática, conforme demonstrado nas figuras 21 e 22.

Figura 21 – Chamada dos Predicados Envolvidos

```
CALL

CALL:
 associados(_610) , membro(_78,_610)
_
```

Primeiramente o visualizador carrega e mostra todos os predicados envolvidos na operação. No caso acima, figura 21, o exemplo possui dois predicados, associados e membro.

Figura 22 – Verifica membro na lista

```
EXIT

EXIT:
 membro(wagner,[wagner,tania,evelin,ivo,luzimar,clara,opa])
_
```

Já a figura 22 mostra que o visualizador demonstra que o Arity Prolog está tentando verificar um determinado membro com a lista de sócios e caso a resposta seja positiva a palavra “yes” será gerada como demonstra a figura 23.

Figura 23 – Sócio Encontrado

```
EXIT

 EXIT:
socio(wagner)

yes
?-
```

Vale observar também que nos exemplos e telas demonstrados algumas vezes aparecem números e letras, estes nada mais são do que as informações quando são trabalhadas na base de dados do Arity Prolog pelo próprio Arity onde ele tenta satisfazer as regras definidas.

6 CONCLUSÃO

O presente trabalho permitiu um estudo mais aprofundado da linguagem de programação Prolog, bem como alguns de seus componentes e ferramentas para construção deste protótipo aqui demonstrado.

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 demonstra ao usuário em qual estágio e o que está acontecendo com o processamento dos dados pelo Arity Prolog. Mostrando quais os predicados que estão sendo executados e com quais valores está sendo trabalho, gerando assim em vídeo um dos 4 tipos de saídas conforme já comentado.

Utilizando-se de 2 predicados teste onde é concebido uma base de fatos e regras, mais um motor de inferência (*parse*) onde os dados são classificados de acordo com seu estágio e por último um predicado que mostra na tela o estágio propriamente dito.

A ferramenta escolhida para construção do protótipo fez com que o as linhas de código do programa ficassem bem reduzidas pois foi utilizado os recursos do próprio Arity Prolog para validar as informações. Mesmo apesar do protótipo demonstrado ser bem compacto, os objetivos gerais do trabalho foram alcançados.

As principais dificuldades encontradas foram: Paradigma da Linguagem, por se tratar de uma linguagem de programação com uma concepção de construção diferente de programas dos ditos tradicionais com o Pascal e C -- Compreensão do funcionamento de determinados comandos utilizados.

6.1 LIMITAÇÕES

Por se tratar de um protótipo, algumas funções não foram implementadas. Para tornar o programa mais funcional na interpretação de instruções Prolog deverá ser aprimorado as seguintes opções:

a) Visualização Gráfica: Para um maior formalismo e melhor visualização por parte do usuário, deverá ser construído ou gerado algumas telas com melhores recursos gráficos, sendo até mesmo possível que esta parte gráfica seja feito com outro tipo de linguagem pois o Arity Prolog em termos de ambiente gráfico é bem deficiente;

b) Outra Linguagem: Para uma maior compreensão de leigos que queiram seguir as melhorias aqui declaradas para futuras implementações, um outro fator importante será a confecção deste protótipo em outro ambiente/linguagem o que facilita a questão do paradigma da linguagem Prolog.

c) Verificação da Análise Sintática: Poderá ser feito um pequeno analisador sintático para ver se o programa carregado está de acordo com os padrões do Arity Prolog

6.2 EXTENSÕES

Como sugestão para futuros trabalhos propõe-se um estudo mais aprofundado sobre a linguagem Arity Prolog e também um estudo sobre interfaceamento com outras possíveis linguagens de programação, pois a questão de paradigma em relação ao Prolog é muito forte. Seguindo estas dicas, muitas das dificuldades que foram encontrada para a confecção deste protótipo serão amenizadas.

Para o aperfeiçoamento do protótipo, sugere-se implementar as limitações abordadas no item anterior, o que irá gerar um visualizador de processos do Arity Prolog muito mais completo e funcional.

REFERÊNCIAS BIBLIOGRÁFICAS

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

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

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

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

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

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

[CAS87] CASANOVA, Marco Antônio. **Programação em Lógica e a Linguagem Prolog**. Rio de Janeiro : Centro Científico do Rio de Janeiro IBM Brasil, 1987.

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

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

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

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

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

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

[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.

[HOR84] HOROWITZ, Ellis. **Fundamentals Programming Languages**. Califórnia : University of Southern Califórnia - Computer Science Press, 1986.

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

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

[MEL90] MELENDEZ Filho, Rubem. **Prototipação de sistemas de informação.** Rio de Janeiro : LTC 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 : Editora da Universidade Católica de Pelotas, 1997.

[PIN95] PINTO, Daniel de Ariosto. **Compiladores Princípios, Técnicas e Ferramentas.** Rio de Janeiro : LTC Livros Técnicos e Científicos Editora S.A..

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

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

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

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

[ROB88] ROBINSON, Phillip R. **Turbo Prolog: Guia do Usuário**. Trad. Lars Gustav Erik Unonius. São Paulo : McGraw-Hill, 1988.

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

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

[TOW90] TOWNSEND, Carl. **Técnicas Avançadas em Turbo Prolog**. Rio de Janeiro : Editora Campus Ltda, 1990.

[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.