

Paradigma Lógico

Henrique Bergami Orlette
Emanuel Joaquim Ferreira
Enzo Faroni
Felipe Vieira da Silva
Filippo Ferraz
Gabriel Antolini de Vasconcelos
Gabriel da Silva Reboli
Guilherme Bertoli Gasparini
Guilherme Ferreira
Gustavo Briel de Deus
Iago Moschen Resende
João Henrique Alves Silva

June 5, 2024

1 O que é?

1.1 Paradigma

A definição do dicionário Aurélio para “paradigma”:

1. Algo que serve de exemplo geral ou de modelo.
2. Conjunto das formas que servem de modelo de derivação ou de flexão.
3. Conjunto dos termos ou elementos que podem ocorrer na mesma posição ou contexto de uma estrutura.

1.2 Paradigma Computacional

Na computação, um paradigma refere-se a um estilo ou abordagem de programação que oferece uma maneira singular de pensar e estruturar os programas. Paradigmas de programação assim como os paradigmas do mundo real, são molduras ou padrões que orientam como os programas são construídos e como os problemas são resolvidos. Cada paradigma possui princípios e técnicas próprias que definem como os elementos de um programa devem ser organizados e como devem interagir. Existem alguns tipos de paradigmas na computação, sendo

alguns deles o Imperativo; Orientado a Objetos; Funcional; Lógico. Neste material falaremos sobre o Paradigma Lógico e sobre sua utilização em linguagens computacionais

1.3 Paradigma Lógico

O paradigma lógico é uma abordagem de programação baseada em lógica formal. Diferente de paradigmas como o imperativo ou o orientado a objetos, onde os programas são descritos como sequências de comandos ou interações entre objetos, respectivamente, o paradigma lógico utiliza a lógica matemática para expressar a programação. Essa abordagem é fundamental para resolver problemas de maneira declarativa, especificando o que deve ser feito, em vez de como fazer. A base do paradigma lógico é a lógica formal, especificamente a lógica de predicados de primeira ordem. Esta lógica usa predicados para expressar relações entre elementos e regras (ou cláusulas) para definir essas relações. Destaca-se que no paradigma lógico, a programação é declarativa. Isso significa que o programador especifica o que deseja alcançar sem detalhar como alcançar. A execução é gerenciada pelo sistema de inferência da linguagem, que usa as regras e predicados definidos para encontrar soluções.

Exemplo:

```
pai(guilherme, enzo).
```

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

Neste pequeno código em Prolog, estamos estabelecendo um fato onde “guilherme” é pai de “enzo”, e estamos estabelecendo uma regra lógica, onde para X ser filho de Y, Y tem que ser pai de X; Logo enzo é filho de guilherme.

Obs: O comparador “:-” utilizado refere-se ao “se” em Lógica, ou seja, X só será filho de Y se Y for pai de X. Além disso, é importante dizer que as linguagens lógicas possuem um sistema de consultas, ainda utilizando o exemplo acima é possível fazer algumas consultas, sendo uma delas:

```
?- filho(X, guilherme).
```

Aqui a consulta pergunta quem é o filho de guilherme. A resposta desta consulta será:

```
X = enzo.
```

Pois enzo é filho de guilherme.

Em seguimento no Prólogo, vale ressaltar que o coração de uma linguagem de programação lógica é seu sistema de inferência, que deduz novos fatos a partir

de fatos conhecidos e regras. Esse processo é conhecido como resolução. Por exemplo, se temos uma regra que diz que "todos os humanos são mortais" e um fato que diz que "Sócrates é humano", o sistema pode inferir que "Sócrates é mortal".

1.4 Conclusão do prólogo

O paradigma lógico representa uma abordagem poderosa e flexível para resolver problemas complexos de maneira declarativa. Embora possa apresentar desafios em termos de performance e curva de aprendizado, suas aplicações em áreas como inteligência artificial e verificação de programas demonstram sua utilidade e relevância na computação moderna.

2 Como Instalar

2.1 Windows

Para instalar o SWI-Prolog no windows vamos seguir os seguintes passos:

1. **Acesse o site oficial do SWI-Prolog:**
Abra o navegador de sua preferência e vá para SWI-PROLOG.ORG
2. **Navegue até a seção de Downloads:**
No menu do site, clique em "Download".
3. **Baixe o instalador do Windows:**
Na seção "SWI-Prolog for Microsoft Windows", clique no link para baixar o instalador da versão mais recente. Geralmente, o arquivo será algo como `swipl-x.y.z x64.exe` para sistemas de 64 bits, onde `x.y.z` representa a versão específica.
4. **Execute o arquivo para instalar o SWI-Prolog na sua máquina:**
Escolha do diretório de instalação: Escolha o diretório onde deseja instalar o SWI-Prolog, depois clique em "Next".

Componentes adicionais: Escolha os componentes que deseja instalar. Os componentes padrão geralmente são suficientes, então você pode clicar em "Next". Instalar: Clique em "Install" para iniciar a instalação.

5. **Verificar a Instalação**
Abrir o SWI-Prolog: Após a instalação, você pode encontrar o SWI-Prolog no menu Iniciar. Clique em "Iniciar" e digite "SWI-Prolog", depois clique no ícone do SWI-Prolog para abrir.

Verificar a instalação: Quando o SWI-Prolog abrir, você verá o prompt do Prolog. Para verificar se está funcionando corretamente, digite um comando simples como `write('Hello, Prolog!'), nl.` e pressione Enter. Você deve ver a saída `Hello, Prolog!`.

```
1  helloWorld :-  
2      write('Hello World').  
3  
4  :- helloWorld.  
5
```

Figure 1: Exemplo de código

2.2 Linux

Primeiramente podemos verificar se o SWI-Prolog está instalado verificando a versão pelo terminal do linux. Podemos verificar com o seguinte comando no terminal:

swipl --version

Se o SWI-Prolog estiver instalado, este comando retornará a versão instalada, algo como **SWI-Prolog version 8.2.4**. Se não estiver instalado, você verá uma mensagem de erro dizendo que o comando não foi encontrado.

Para instalar o SWI-Prolog no Linux (Ubuntu) siga os seguintes passos:

1. **Adicione ppa:swi-prolog/stable às fontes de software do seu sistema:**

Abra um terminal (Ctrl+Alt+T) e digite:

sudo add-apt-repository ppa:swi-prolog/stable

Depois, atualize as informações do pacote:

sudo apt-get update

2. **Instale o SWI-Prolog através do gerenciador de pacotes:**

Abra um terminal (Ctrl+Alt+T) e digite:

sudo apt-get install swi-prolog

3 Histórico

A programação lógica é uma ideia que começou a ser pensada no contexto da inteligência artificial pelo menos desde o momento em que John McCarthy, em 1958, propôs: "programas para manipular com sentenças instrumentais comuns apropriadas à linguagem formal (muito provavelmente uma parte

do cálculo de predicado)”. Em tese, a ideia seria um modelo que por meio de fatos/informações /conhecimentos (dados) , por meio da programação, pudesse fazer afirmações/declarações com os dados que seriam processados e utilizados em cálculo de predicado.

Nesta seção de histórico citaremos em ordem cronológica desde o surgimento da programação lógica até a sua utilização no cotidiano.

3.1 Fundamentos Teóricos

Lógica Matemática e Teoria da Computação (Década de 1930 a 1960):

A base teórica da programação lógica está na lógica de primeira ordem (ou lógica de predicados), desenvolvida por Gottlob Frege, Kurt Gödel e outros.

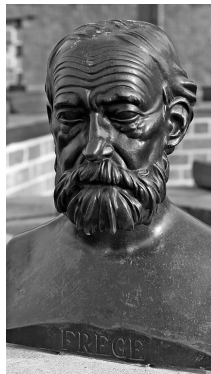


Figure 2: Gottlob Frege



Figure 3: Kurt Gödel

Gödel, Alfred Tarski e Alonzo Church fizeram avanços fundamentais na lógica, mostrando a relação entre a lógica e a computabilidade.

Formalização da Programação Lógica (Anos 1960 e 1970):

Robinson introduziu o método de resolução, um mecanismo fundamental para inferência lógica automática, que se tornaria a base para a programação lógica.

3.2 Origens

Décadas de 1930 a 1950: Durante e após a Segunda Guerra Mundial, houve um aumento significativo no interesse pela automação e pelo processamento de informações. Isso levou a uma busca por métodos eficazes para representar conhecimento e raciocinar sobre ele de forma computacionalmente viável.

1936: Alan Turing propõe a Máquina de Turing, um modelo teórico fundamental para a computação, estabelecendo as bases para a lógica formal na programação.

1956: John McCarthy, introduz o termo "programação lógica" em um workshop na Universidade de Dartmouth, defendendo a utilização da lógica como ferramenta para programar computadores.



Figure 4: John McCarthy

Décadas de 1960 a 1970: O interesse pela inteligência artificial e a necessidade de linguagens de programação mais expressivas impulsionaram o desenvolvimento de linguagens baseadas em lógica, como LISP e Prolog. O trabalho de John McCarthy e outros na inteligência artificial ajudou a popularizar o uso de linguagens de programação lógica.

1969: Criação do Planner, a primeira linguagem de programação lógica, por Carl Hewitt, possibilitando a criação de planos através de asserções e objetivos.

1972: Desenvolvido no início da década de 1970 por Alain Colmerauer e Philippe Roussel na Universidade de Marselha, na França, baseado no trabalho de Robert Kowalski, o Prolog tornou-se uma linguagem de programação declarativa que ganhou grande popularidade, com foco em inteligência artificial e processamento de linguagem natural.



Figure 5: Robert Kowalski



Figure 6: Alain Colmerauer

Década de 1980: Diversas linguagens e ferramentas são criadas, como Datalog e F-logic, impulsionando a pesquisa e o uso prático do paradigma. O paradigma lógico se torna fundamental para o desenvolvimento de sistemas es-

pecialistas, robótica e outras áreas da IA. Durante esta década, o Prolog e suas variantes, como o SWI-Prolog (1987), ganharam destaque em pesquisa acadêmica e aplicações industriais. O Prolog foi amplamente utilizado em áreas como inteligência artificial, processamento de linguagem natural, sistemas especialistas e engenharia de software.

3.3 Atualidade

Atualmente há um crescente interesse na integração de técnicas de programação lógica com aprendizado de máquina, levando ao desenvolvimento de áreas como Inductive Logic Programming (ILP) e Statistical Relational Learning (SRL). Ferramentas de programação lógica, como Prolog, estão sendo usadas para consultas complexas em grandes volumes de dados e na Web Semântica, com o uso de tecnologias como RDF e OWL. Hordienamente O Prolog continua a evoluir, com implementações modernas como SWI-Prolog e GNU Prolog, que oferecem novos recursos e melhor desempenho. A comunidade de programação lógica mantém-se ativa, com conferências e publicações regulares promovendo novos avanços. Embora o paradigma lógico seja distinto, ele frequentemente se integra com outros paradigmas de programação. Por exemplo, é comum ver o Prolog sendo utilizado em conjunto com linguagens funcionais, imperativas ou orientadas a objetos para aproveitar as vantagens de cada paradigma em diferentes partes de um sistema. O paradigma lógico continua a ser objeto de estudo e pesquisa em instituições acadêmicas em todo o mundo. Pesquisadores exploram novas técnicas, algoritmos e aplicações para estender e aprimorar as capacidades do Prolog e do paradigma lógico em geral.

4 Vantagens e Desvantagens

4.1 Vantagens

1. **Expressividade e Clareza:** Descrever problemas em termos de lógica matemática torna os programas mais fáceis de entender e manter, permitindo uma abordagem declarativa focada no que se deseja alcançar, em vez de como alcançá-lo.
2. **Inferência Automática:** Linguagens lógicas oferecem a capacidade de realizar inferências automáticas, permitindo derivar novas informações, solucionar problemas e responder consultas de forma automatizada, o que facilita a busca por soluções coerentes.
3. **Facilidade de Implementação de Algoritmos de Busca e Backtracking:** É especialmente adequado para implementar algoritmos de busca e backtracking, comuns em problemas de inteligência artificial, permitindo a exploração sistemática de todas as possibilidades para encontrar soluções válidas.

4. **Prototipação Rápida:** Devido à natureza declarativa e à capacidade de inferência automática, o desenvolvimento de protótipos para problemas complexos pode ser mais rápido, facilitando a exploração e validação de soluções potenciais. **Raciocínio Simbólico:** Ideal para aplicações que envolvem raciocínio simbólico, como sistemas especialistas e agentes inteligentes, facilitando a manipulação e inferência sobre símbolos e relações lógicas.
5. **Facilidade de Manipulação de Conhecimento:** Eficiente na representação e manipulação de conhecimento, permitindo a adição e atualização de fatos e regras sem a necessidade de reescrever grandes partes do código, o que simplifica a manutenção e melhoria do sistema.
6. **Paralelismo:** A natureza declarativa e independente das regras lógicas permite a execução paralela de muitas operações, aproveitando o poder dos processadores multicore e sistemas distribuídos para aumentar a eficiência e velocidade de execução.
7. **Consistência e Verificação Formal:** Baseado em lógica formal, facilita a aplicação de métodos de verificação formal para garantir a consistência e correção dos programas, sendo importante em sistemas críticos onde a correção é fundamental.

4.2 Desvantagens

1. **Desempenho e Eficiência:** Embora melhore a expressividade do código, o desempenho pode ser prejudicado devido à intensidade computacional de operações como unificação e busca exaustiva.
2. **Escalabilidade:** À medida que a base de conhecimento cresce, a complexidade e o tempo de inferência aumentam, dificultando a escalabilidade para sistemas grandes ou dinâmicos.
3. **Dificuldade de Depuração:** A depuração pode ser complicada devido à complexidade das regras e à natureza implícita do fluxo de execução.
4. **Limitado para Aplicações Numéricas:** Não é naturalmente adequado para cálculos numéricos intensivos ou manipulação de grandes matrizes de dados.
5. **Curva de Aprendizado:** A transição para o paradigma lógico pode ser difícil para programadores acostumados com paradigmas imperativos ou orientados a objetos, exigindo um esforço considerável para dominar a lógica formal e as técnicas de inferência.
6. **Complexidade na Expressão de Algoritmos:** : Expressar algoritmos complexos pode ser menos intuitivo devido à falta de controle explícito sobre o fluxo de execução.

7. **Suporte e Ferramentas Limitadas:** Há menos ferramentas e comunidades de suporte disponíveis para linguagens de programação lógica em comparação com paradigmas mais populares.
8. **Aplicações Restritas:** Mais adequado para problemas específicos como sistemas especialistas e processamento de linguagem natural, limitando sua aplicabilidade em projetos comerciais e industriais comuns.

5 Sintaxe

5.1 Variável

Iniciadas com letra maiúscula ou underscore (`_`), sendo anônimas as variáveis que apresentam apenas o underscore.

Exemplos:

Variável = `Joao` / `_Joao`

Variável anônima = `_` (utilizada para consultas que retornam um valor booleano).

5.2 Átomos

São constantes, começam com letra minúscula ou entre aspas simples.

Exemplo: `joao` / `'João'`

5.3 Inteiros

Sequência numérica sem `(.)` ou caracteres ASCII entre aspas, que serão tratados como lista de inteiros.

Exemplo: `1, 4, "b"`.

5.4 Floats

Números com um ponto `(.)` e pelo menos uma casa decimal.

Exemplo: `6.1`

5.5 Listas

Sequência de elementos entre `[]` separados por vírgula.

Exemplo: `[a, b, c]`

Termo	Classificação
VINCENT	átomo
23	inteiro
variable23	átomo
aulas de lógica	erro
'Joao'	átomo
[1, [2, 3], 4]	lista
Footmassage	variável
65.	erro
23.0	float
_	variável anônima
'aulas de lógica'	átomo
"a"	lista
[]	lista

Figure 7: Tabela para fixação

5.6 Comandos

Write: Utilizado para escrever.

Exemplos:

write('Teste de Impressão'). Correto

Será exibido "Teste de Impressão" no terminal.

write(Teste de impressão). Incorreto

Não possui as aspas simples.

write(X). Correto

Será exibido o valor de X no terminal

Read: utilizado para ler variáveis:

Exemplos:

read(X). Correto

O usuário digitará um valor que será armazenado na variável X.

read(x). Incorreto

Foi feita a tentativa de um “read” em um átomo, porém só é permitido utilizar esse comando em variáveis.

read(Joao). Correto

O usuário digitará um valor que será armazenado na variável Joao.

5.7 Fatos

Fatos são sempre verdadeiros.

Exemplos:

homem(x). = significa que x é um homem

genitor(x, y). = significa que “x é genitor de y” ou “y é gerado por x”.

Outro ponto relevante é que nesses exemplos, “homem” e “genitor” são Functors e “x” e “y” representam a aridade. Por exemplo, a aridade de “homem(x).” é 1, pois possui somente um átomo, já “genitor(x, y)” possui aridade 2, já que possui 2 átomos na sua composição.

No exemplo abaixo temos uma árvore genealógica, sugiro que você tente escrever os fatos presentes nela antes de saber de fato.

Obs.: O círculo vermelho representa o sexo feminino e o círculo azul representa o sexo masculino.

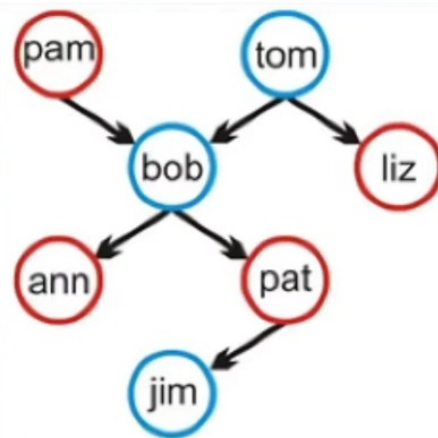


Figure 8: Árvore Genealógica

Fatos:

homem(tom).

homem(bob).

homem(jim).

```
mulher(pam).
mulher(liz).
mulher(ann).
mulher(pat).
genitor(pam, bob).
genitor(tom, bob).
genitor(tom, liz).
genitor(bob, ann).
genitor(bob, pat).
genitor(pat, jim).
```

5.8 Regras

Facilitam as consultas e tornam o programa mais expressivo.

$A(x) \rightarrow B(x) \vee (C(x) \wedge D(x))$, seria escrita em Prolog como: `a(X) :- b(X); (c(x), d(x)).`

Exemplos:

Regra se X é filho de Y: **filho(X, Y) :- genitor(Y, X).**

Em linguagem natural seria: X é filho de Y se Y for genitor de X.

Regra se X é mãe de Y: **mae(X, Y) :- genitor(X, Y), mulher(X).**

Em linguagem natural seria: X é mãe de Y se X for genitor de Y e X for mulher.

Regra se X é avo de Z: **avo(X, Z) :- genitor(X, Y), genitor(Y, Z).**

Em linguagem natural seria: X é avô de Z se X for genitor de Y e Y for genitor de Z.

5.9 Consultas

Não é necessário programar os retornos, apenas são feitas consultas. As consultas são realizadas no terminal.

Exemplos (considerando a árvore genealógica acima):

mulher(pam). Retorno: True.

Em linguagem natural seria: Pam é mulher?

genitor(bob, X). Retorno: X = ann; X = pat; false.

Em linguagem natural seria: Bob é genitor de quem?

Mas como são realizadas essas consultas?

As consultas possuem 3 etapas principais:

- **Matching:** Compara a estrutura das expressões (`pai (joao, maria) = pai(X,Y)`), ou seja, verifica se há um padrão.
- **Unificação:** substitui o valor das variáveis para determinar se a consulta satisfaz os fatos e as regras do programa.
- **Backtracking:** Checagem de todas as possibilidades. Ex.: Se João tem mais de um filho, a consulta retornará todos, caso contrário a consulta retornaria somente o primeiro filho de João encontrado.

Outras etapas de consulta:

- **Resolução:** inferência lógica dos fatos e regras.
- **Recursão:** Regras que chamam a si mesmas. Ex.: Fatorial de algum número

6 Aritmética

Prolog trata as representações de forma equivalente, pois, internamente utiliza árvores para representar expressões. Assim, basta mudar a ordem de caminhamento para obter uma ou outra forma.

É possível utilizar duas notações para representar expressões em Prolog:

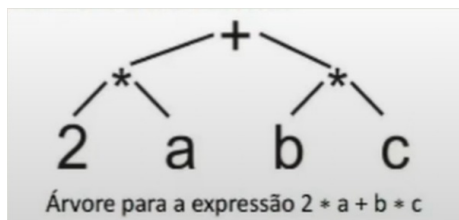


Figure 9: Exemplo de Árvore

Infixa: $(2 + 3)$.

Exemplo:

```
soma_infixa(X, Y, Resultado) :-  
    Resultado is (X + Y).
```

Figure 10: Exemplo de código Infixa

```
?- soma_infixa(2,3, Z).  
Z = 5.
```

Figure 11: Consulta Infixa

Prefixa: $+(2, 3)$.
Exemplo:

```
soma_prefixa(X, Y, Resultado) :-  
    Resultado is +(X,Y) .
```

Figure 12: Exemplo de código Prefixa

```
?- soma_prefixa(2,3, Z).  
Z = 5.
```

Figure 13: Consulta prefixa

6.1 Operadores

São oferecidos diversos operadores para cálculos aritméticos:

Operador	Significado
+, -, *, /	Realizar soma, subtração, multiplicação e divisão, respectivamente
is	Atribui uma expressão numérica à uma variável
mod	Obter o resto da divisão
^	Calcular potenciação
cos, sin, tan	Função cosseno, seno e tangente, respectivamente
exp	Exponenciação
ln, log	Logaritmo natural e logaritmo
sqrt	Raiz

Operador	Significado
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
=,=	igual
\=	diferente
\+	Negação – retorna sucesso se o predicado for falso e vice-versa.

Os operadores = e =,= realizam diferentes tipos de comparação:

“=” atribui valores para as variáveis (unificação de termos)

“=,=” avalia se os valores são iguais

Exemplos:

$$1 + 2 = 2 + 1$$

O retorno dessa expressão seria False, porque o sistema se preocupa com os elementos da adição e não com o resultado

$$1 + 2 =,= 2 + 1$$

Nesse caso o retorno seria True, porque o sistema compara o valor das duas expressões, ou seja, $3 = 3$.

7 Comparativo

Nesta seção iremos comparar o Paradigma Lógico com o Paradigma Funcional, Imperativo e Orientado a Objetos. Vale lembrar que o Paradigma Lógico é um paradigma de programação que faz uso da lógica matemática. O paradigma da programação com apontamento lógico não é composto de instruções e, por isso, se difere bastante dos demais (apesar de derivar do declarativo). É baseado em fatos e usa tudo o que sabe para criar um cenário onde

todos esses fatos e cláusulas são verdadeiros e apontam para algum final.

Cujas características são:

- **Declaração de Conhecimento:** Programas são uma coleção de fatos e regras
- **Resolução Automática:** O sistema deduz automaticamente as soluções através de inferências lógicas.
- **Não Procedural:** Foca no que deve ser resolvido, ao invés da maneira que será resolvido

7.1 Paradigma Lógico x Imperativo

O paradigma imperativo, antigo e usado desde os primórdios da programação, continua predominante hoje. Ele serve como base para outros paradigmas, como o procedimental e o orientado a objetos. Nele, os programas se baseiam na alteração sequencial do estado do programa por meio de comandos como atribuições e estruturas de controle como loops e condicionais. Apesar de sua abordagem intuitiva, pode resultar em efeitos colaterais indesejados e dificuldades de manutenção em programas complexos.

Cujas características são:

- Estados de uma máquina abstrata descritos por valores de um conjunto de variáveis.
- Reconhecedores de estados - expressões compostas por relações entre esses valores ou os resultados de operações utilizando valores.
- Comandos de atribuição e controle que manipulam o estado.

Comparação:

Lógico: A programação lógica se concentra em declarar relações lógicas entre entidades. Utiliza lógica formal para representar problemas e soluções.

Imperativo: Este paradigma se concentra em como as operações são realizadas. Os programas imperativos especificam uma sequência de comandos que modificam o estado do programa.

7.2 Paradigma Lógico x Funcional

O paradigma funcional baseia-se no conceito matemático de função, em que para cada elemento do seu conjunto domínio (entrada) há apenas um elemento no seu conjunto contradomínio (saída). Programas escritos nele são definições de funções e de especificações de aplicação destas, e as execuções consistem em

avaliá-las.

Cujas características são:

- A execução de um programa puramente funcional não tem nenhum estado em termos de semântica operacional e denotacional.
- Imutabilidade: Dados são imutáveis e não mudam após serem criados.
- Retornam resultado de argumento: Funções retornam resultados apenas baseados nos seus argumentos, sem efeitos colaterais.

Comparação:

Lógico: Na programação lógica, os programas são expressos em termos de relações lógicas. As principais construções são cláusulas lógicas e unificação.

Funcional: O paradigma funcional se concentra em funções matemáticas e avaliação de expressões. A imutabilidade e funções de ordem superior são características proeminentes.

7.3 Paradigma Lógico x Orientado a Objetos

O paradigma orientado a objetos organiza o código em torno de objetos, que combinam dados e comportamentos. Exemplos incluem C++, Java e Python. A idéia central deste paradigma é abstrair objetos do mundo real e representá-los com as mesmas características na programação (atributos, ações, modo de fabricação), sendo o programador o responsável por moldar estes objetos, e explicar para estes objetos como eles devem interagir entre si.

Cujas características são:

- Encapsulamento: Dados e métodos são encapsulados dentro de objetos
- Herança: Objetos podem herdar características de outros objetos.
- Objetos podem ser tratados como instancias de suas Superclasses.

Comparação:

Lógico: Programação lógica trata de definir relações e regras que descrevem um problema, sem se preocupar com a sequência de execução. Usa unificação e busca de prova para resolver problemas.

Funcional: Este paradigma se concentra na modelagem de problemas através de objetos que possuem estado e comportamento. A herança, encapsulamento e polimorfismo são conceitos centrais.

8 Exercícios Teóricos

Nesta seção do material, serão apresentados 5 exercícios teóricos sobre paradigmas de programação lógica.

1. Qual das seguintes linguagens de programação é mais associada ao paradigma de programação lógica?

- A) Python
- B) Java
- C) Prolog
- D) C++

Resposta correta: C) Prolog

2. Em programação lógica, o processo de retroceder para um ponto anterior e tentar um caminho alternativo quando uma meta não pode ser provada é chamado de:

- A) Recursão
- B) Iteração
- C) Backtracking
- D) Herança

Resposta correta: C) Backtracking

3. Em Prolog, a operação que combina metas com fatos e regras para provar uma declaração é conhecida como:

- A) Atribuição
- B) Unificação
- C) Compilação
- D) Interpretação

Resposta correta: B) Unificação

4. Ainda em Prolog, qual tipo de dado possui as seguintes características abaixo: “São constantes, devem ser iniciadas com minúsculas seguidas de qualquer caractere alfanumérico ou qualquer sequência entre ‘ ‘ (aspas simples)”

- A) Variáveis
- B) Átomos
- C) Inteiros
- D) Listas

Resposta correta: B) Átomos

5. Em uma consulta, qual é o nome do processo que checa se determinado padrão está presente, para saber quais fatos e regras podem ser utilizados?

- A) Matching

- B) Unificação
- C) Resolução
- D) Recursão

Resposta correta: A) Matching

9 Exercícios Práticos

Nesta seção do material, serão apresentados 5 exercícios práticos sobre paradigmas de programação lógica.

1. Dada a base de fatos abaixo:

```
animal(urso) .  
animal(peixe) .  
animal(raposa) .  
animal(coelho) .  
animal(veado) .  
planta(alga).  
planta(grama) .  
come(urso,peixe).  
come(urso, raposa).  
come(urso, veado) .  
come(peixe,alga).  
come(raposa, coelho) .  
come(coelho,grama).  
come(veado,grama) .
```

Quais são as respostas para as consultas?

- a)?- planta(_).
- b)?- come(raposa,_).
- c)?- come(_,urso).

Resposta correta:

- a>true.
- b>true.
- c>false.

2. Considere a base de fatos e regras em Prolog:

```
gosta(mary,vinho).  
gosta(roberto,cerveja).  
gosta(ana,cereja).  
gosta(augusto, vinho) .
```

`gosta(alberto,X) :- gosta(X,vinho).`

Qual a resposta correta para a consulta:
`?- gosta(alberto,X).`

a) `X = mary ;`
`X = agosto ;`

b) `X = mary ;`
`false.`

c) `X = mary ;`
`X = agosto ;`
`true.`

d) `true.`

e) `X = mary ;`
`X = agosto ;`
`false.`

Resposta correta:

e) `X = mary ;`
`X = agosto ;`
`false.`

3. Considere a seguinte base de fatos em Prolog:

`aluno(joao,calculo).`
`aluno(joel,estrutura).`
`aluno (maria,calculo).`
`aluno (joel, programacao).`
`professor (andre, calculo).`
`Professor (julia, estrutura).`
`Professor(pedro,programacao).`
`frequenta (joao,uvv).`
`frequenta (maria, uvv).`
`frequenta (joel,ufes).`
`funcionario (pedro,ufes).`
`funcionario(julia,uvv).`
`funcionario (andre, uvv).`

Escreva uma regra que mostre quem são os alunos de um professor X.

Resposta correta:

sao_alunos_do_professor(A,X) :- professor (X,Materias), aluno(A,Materias).

4. Crie uma regra em Prolog que peça no console um número inteiro e imprima na tela se o número é maior do que 100 ou se é menor ou igual a 100.

Resposta Correta:

```
maiorQueCem() :- write('Entre como o numero:'),
                 read(X),
                 (
                   (X>100, write('O numero é maior que cem'))
                   ;
                   (X<=100, write('O numero não é maior que cem'))
                 ).
```

5. Suponha os seguintes fatos:

nota(joao,5.0).
nota (joaquim,4.5).
nota(maria,4.0).
nota (mary,10.0).
nota(jose,6.5).
nota(joana,9.0).
nota(cleuza,6.0).

Considerando que:

- Nota de 7.0 a 10.0 = aprovado
- Nota de 5.0 a 6.9 = recuperação
- Nota de 0.0 a 4.9 = reprovado

Identifique o erro da regra abaixo:

```

situacao(A) :- nota(A,Nota), (
    (Nota>=7.0,Nota=<10.0 , write('Aprovado'))
    ,
    (Nota>=5.0,Nota=<6.9 , write('Recuperação'))
    ,
    (Nota>=0.0,Nota=<4.9 , write('Reprovado'))
) .

```

Figure 14: Código para a questão 5


Resposta Correta:

Foi utilizado um conectivo lógico errado, deveria ter sido o conectivo OU (;)

```

situacao(A) :- nota(A,Nota), (
    (Nota>=7.0,Nota=<10.0 , write('Aprovado'))
    ,
    (Nota>=5.0,Nota=<6.9 , write('Recuperação'))
    ,
    (Nota>=0.0,Nota=<4.9 , write('Reprovado'))
) .

```



Após corrigir o erro a regra ficará da seguinte forma:

```

situacao(A) :- nota(A,Nota), (
    (Nota>=7.0,Nota=<10.0 , write('Aprovado'))
    ;
    (Nota>=5.0,Nota=<6.9 , write('Recuperação'))
    ;
    (Nota>=0.0,Nota=<4.9 , write('Reprovado'))
) .

```

Figure 15: Resposta Questão 5

9.1 Explicação das Questões Práticas

1. O símbolo “_” (underline ou underscore) significa uma variável anônima. Esse tipo de variável tem como função retornar um valor booleano referente a existência da relação informada no functor. Por motivos metodológicos, iremos traduzir as 3 consultas em linguagem natural abaixo:

Existe alguma planta?
Existe algo que a raposa coma?
Existe algo que coma o urso?

No caso, o sistema irá verificar a existência da relação utilizando os functors grifados acima, caso ele encontre qualquer um ou vários átomos que satisfaçam a consulta, ele retornará true. Caso nenhum átomo da usa base de conhecimento puder satisfazer as consultas, será retornado false.

2. Para facilitar auxiliar em sua compreensão, iremos traduzir a consulta para linguagem natural abaixo:

Alberto gosta de algo/alguém se esse algo/alguém gosta de vinho.

A lógica dessa consulta é, a primeira parte “gosta(alberto,X)” será verdadeira somente se a segunda consulta for verdadeira, logo iremos focar na parte final. Tal parte, trata-se de uma consulta que, como utiliza uma variável, irá buscar todos os átomos que “gostam de vinho”, logo, o sistema encontrará as respostas Mary e Augusto.

Todavia, devido a utilização do backtracking, após encontrar os resultados Mary e Augusto, o sistema tentará mais uma vez buscar átomos que gostam de vinho na base de conhecimento, porém, não obterá sucesso (pois não tem mais ninguém que gosta de vinho), portanto, trará “False” no final da consulta.

3. A fórmula acima utiliza 2 parâmetros, um para o professor (X) e outro para o aluno (Y). A fórmula `sao_alunos_do_professor` será verdadeira somente se um professor X estiver vinculado a uma matéria E um aluno A TAMBÉM estiver matriculado na mesma matéria.

Logo quando você pesquisar, por exemplo: `sao_alunos_do_professor(A, andre)`.

O sistema retornará:
A = joao;
A = maria;
false.

Pois os átomos joao, maria e andre satisfazem a consulta, visto que o átomo do professor E os átomos dos alunos estão cursando a mesma matéria.

4. A fórmula acima propõe uma função que, conforme indicado no enunciado da questão, determina se um número inteiro inserido pelo usuário é maior que 100 ou se é menor ou igual a 100. Note que não há necessidade de adicionar um parâmetro à fórmula “maiorQueCem”, pois ela pede um valor a ser inserido pelo usuário.

Inicialmente, a regra imprime na tela por meio do comando write, uma solicitação para que o usuário insira um número de sua escolha. Em seguida, o código, por meio do comando read, obtém o valor inserido pelo usuário e o salva na variável X. Após isso, a regra adiciona mais uma condição, que será responsável de realizar duas verificações sobre o dado inserido pelo usuário. Primeiro, verifica se X é maior que 100; se for, imprime na tela que o número é maior que cem. Em seguida, usa o conectivo “ou” para adicionar uma nova verificação: se X for menor ou igual a 100, imprimirá na tela que o número não é maior que cem.

5. Em PROLOG, o conectivo lógico E possui a sintaxe de vírgula (.). Logo, o sistema está tentando achar uma nota que seja:

“Maior ou igual a 7.0” E “Menor ou igual a 10.0”

E

“Maior ou igual a 5.0” E “Menor ou igual a 6.9”

E

“Maior ou igual a 0.0” E “Menor ou igual a 4.9”.

Levando em consideração apenas floats de 1 casa decimal, é impossível encontrar um número que satisfaça tal regra.

Pois, por exemplo, os números “Maior ou igual a 7.0” E “Menor ou igual a 10.0” seriam:

7, 7.1, 7.2, [...], 9.9, 10.0]

E os números “Maior ou igual a 5.0” E “Menor ou igual a 6.9” seriam:

5.0, 5.1, 5.2, [...], 6.8, 6.9

Percebe-se então, que é impossível um número estar em um conjunto E no outro ao MESMO TEMPO.

O correto seria:

“Maior ou igual a 7.0” E “Menor ou igual a 10.0”

OU

“Maior ou igual a 5.0” E “Menor ou igual a 6.9”

OU

“Maior ou igual a 0.0” E “Menor ou igual a 4.9”.

10 Fontes

<https://www.dicio.com.br/paradigma/>
<https://www.treinaweb.com.br>
https://rosettacode.org/wiki/Towers_of_Hanoi
<https://github.com/SWI-Prolog>
<https://pt.wikipedia.org/wiki/Programa>
<https://leandromoh.gitbooks.io/tcc-paradigmas-de-programacao/content/index.html>
https://blog.geekhunter.com.br/quais-sao-os-paradigmas-de-programacao/Paradigma_dedologicaedeprogramacao
<https://www.treinaweb.com.br/blog/linguagens-e-paradigmas-de-programacao>
https://www-pi.github.io/tutorials/lectures/lsp/010_install_wiprolog.html
<https://www.francogarcia.com/pt-br/blog/ambientes-de-desenvolvimento-prolog/>
<https://www.swi-prolog.org/>
<https://www.youtube.com/playlist?list=PLZ-Bk6jzsb-OScKa7vhpcQXoU2uxYGaFx>