Linguagens de programação lógicas

Linguagens de Programação

Marco A L Barbosa

http://github.com/malbarbo/na-lp-copl

Conteúdo

Introdução

Introdução

- Na programação lógica os programas são expressos em termos de lógica simbólica e um processo de inferência lógica é usado para produzir os resultados
- A programação lógica é declarativa ao invés de procedural, apenas a especificação do resultado é dada e não um processo detalhado de como obter o resultado
- Linguagens baseadas em lógica simbólica são chamada de **linguagens de programação lógicas** ou **linguagens declarativas**
- Antes de estudar linguagens de programação lógicas, precisamos estudar o seu fundamento, que é lógica formal

Uma breve introdução ao cálculo de predicados

Uma breve introdução ao cálculo de predicados

- Uma **proposição** pode ser vista como um declaração lógica que pode ou não ser verdadeira
- Uma proposição consiste de objetos e nas relações destes objetos

Uma breve introdução ao cálculo de predicados

- A lógica simbólica pode ser usada para as três necessidades da lógica formal
  - Expressar proposições
  - Expressar relações entre as proposições
  - Descrever como novas proposições podem ser inferidas a partir de outras proposições que são verdadeiras
- A forma particular de lógica simbólica que é usada em programação lógica é chamada de **cálculo de predicados (de primeira ordem)**

Uma breve introdução ao cálculo de predicados

- Objetos em proposições são representados por termos simples: constantes ou variáveis
  - Uma **constante** é um símbolo que representa um objeto
  - Uma **variável** é um símbolo que representa diferentes objetos em diferentes momentos
- A proposição mais simples é chamada de **proposição atômica**, que consiste em um termo composto
- Um **termo composto** é um elemento de uma relação matemática, escrita em uma forma com a aparência de notação de função matemática
- Um termo composto é formado por duas partes
  - Um **functor**, o símbolo de função que nomeia a relação
  - Uma lista ordenada de parâmetros
- Exemplos: man(jake), like(bob, steak)

Uma breve introdução ao cálculo de predicados

- Uma proposição pode ser declarada de dois modos
  - **Fato**: a proposição é definida como verdadeira
  - **Consulta**: a verdade da proposição precisa ser determinada
- As **proposições compostas** tem duas ou mais proposições atômicas ligadas por operadores
- Os operadores lógicos do cálculo de predicados são

Nome	Símbolo	Exemplo	Significado
negação	$\neg$	$\neg a$	não a
conjunção	$\cap$	$a \cap b$	a e b
disjunção	$\cup$	$a \cup b$	a ou b
equivalência	$\equiv$	$a \equiv b$	a é equivalente a b
implicação	$\supset$	$a \supset b$	a implica b
	$\subset$	$a \subset b$	b implica a

Uma breve introdução ao cálculo de predicados

- Exemplos de proposições compostas
  - $a \cap b \supset c$
  - $a \cap \neg b \supset d$
  - O operador $\neg$ tem a maior precedência
  - Os operadores $\cap$, $\cup$ e $\equiv$ tem maior precedência que $\supset$ e $\subset$
  - $(a \cap (\neg b)) \supset d$
- Variáveis podem aparecer nas proposições usando símbolos especiais chamados de quantificadores

Nome	Exemplo	Significado
universal	$\forall X.P$	Para todo X, P é verdadeiro
existencial	$\exists X.P$	Existe um valor X tal que P é verdadeiro

- Exemplos
  - $\forall X$. (woman(X)$\supset$human(X))
  - $\exists X$. (mother(mary,X) $\cap$ male(X))

Uma breve introdução ao cálculo de predicados

- No cálculo de predicados existem muitas formas de expressar a mesma coisa, o que é ruim para implementação

Uma breve introdução ao cálculo de predicados

- No cálculo de predicados existem muitas formas de expressar a mesma coisa, o que é ruim para implementação
- Uma forma padrão simples de proposição e a **forma clausal**
  - $B_1 \cup B_2 \cup \cdots \cup B_n \subset A_1 \cap A_2 \cap \cdots \cap A_m$
  - Onde os A's e B's são termos

Uma breve introdução ao cálculo de predicados

- No cálculo de predicados existem muitas formas de expressar a mesma coisa, o que é ruim para implementação
- Uma forma padrão simples de proposição e a **forma clausal**
  - $B_1 \cup B_2 \cup \cdots \cup B_n \subset A_1 \cap A_2 \cap \cdots \cap A_m$
  - Onde os A's e B's são termos
  - Significa que, se todos os A's forem verdadeiros, pelo menos um B é verdadeiro
  - O lado direito é chamado de **antecedente**
  - O lado esquerdo é chamado de **consequente**
  - O quantificador existencial não é necessário
  - O quantificador universal está implícito no uso de variáveis
  - Apenas os operadores de conjunção e disjunção são requeridos
  - Todas as proposições do cálculo de predicados podem ser convertidas por um algoritmo para a forma clausal

Uma breve introdução ao cálculo de predicados

- Exemplos de proposições na forma clausal
  - likes(bob, trout) $\subset$ likes(bob, fish) $\cap$ fish(trout)
  - father(louis, al) $\cup$ father(louis, violet) $\subset$ father(al, bob) $\cap$ mother(violet, bob) $\cap$ grandfather(louis, bob)
- Como estas proposições podem ser lidas em português?

Cálculo de predicados e demostração de teoremas

Cálculo de predicados e demostração de teoremas

- Um uso para uma coleção de proposições é determinar se algum fato interessante pode ser inferido a partir dela

Cálculo de predicados e demostração de teoremas

- Um uso para uma coleção de proposições é determinar se algum fato interessante pode ser inferido a partir dela
- **Resolução** é uma regra de inferência que permite computar proposições inferidas a partir de proposições dadas
- Ideia geral da resolução
  - Dado duas proposições
    - $P_1 \subset P_2$
    - $Q_1 \subset Q_2$

Cálculo de predicados e demostração de teoremas

- Um uso para uma coleção de proposições é determinar se algum fato interessante pode ser inferido a partir dela
- **Resolução** é uma regra de inferência que permite computar proposições inferidas a partir de proposições dadas
- Ideia geral da resolução
  - Dado duas proposições
    - $P_1 \subset P_2$
    - $Q_1 \subset Q_2$
  - Se P_1 é idêntico a Q_2, então podemos chamá-los de T e reescrever as proposições como
    - $T \subset P_2$
    - $Q_1 \subset T$

Cálculo de predicados e demostração de teoremas

- Um uso para uma coleção de proposições é determinar se algum fato interessante pode ser inferido a partir dela
- **Resolução** é uma regra de inferência que permite computar proposições inferidas a partir de proposições dadas
- Ideia geral da resolução
  - Dado duas proposições
    - $P_1 \subset P_2$
    - $Q_1 \subset Q_2$
  - Se P_1 é idêntico a Q_2, então podemos chamá-los de T e reescrever as proposições como
    - $T \subset P_2$
    - $Q_1 \subset T$
  - Logo $Q_1 \subset P_2$

Cálculo de predicados e demostração de teoremas

- Exemplo
  - older(joanne, jake) $\subset$ mother(joanne, jake)
  - wiser(joanne, jake) $\subset$ older(joanne, jake)
  - A partir destas proposições, a seguinte proposição pode ser construída usando resolução
  - wiser(joanne, jake) $\subset$ mother(joanne, jake)

Cálculo de predicados e demostração de teoremas

- Mecânica da resolução
  - Os termos do lado esquerdo das duas proposições são juntados para formar o lado esquerdo da nova proposição
  - Os termos do lado direito são juntados da mesma forma
  - Qualquer termo que apareça dos dois lados é removido
- Exemplo
  - father(bob, jake) $\cup$ mother(bob, jake) $\subset$ parent(bob, jake)
  - grandfather(bob, fred) $\subset$ father(bob, jake) $\cap$ father(jake, fred)
  - A resolução diz que
  - mother(bob, jake) $\cup$ grandfather(bob, fred) $\subset$ parent(bob, jake) $\cap$ father(jake, fred)

Cálculo de predicados e demostração de teoremas

- O processo de resolução é mais complexo do que os exemplos apresentados
- A presença de variáveis nas proposições implica que a resolução deve encontrar valores para as variáveis que permitam o casamento
- O processo de determinar valores que permitam o casamento é chamado de **unificação**
- A atribuição temporária de valores as variáveis para permitir a unificação é chamada **instanciação**
- Depois da instanciação de uma variável com um valor, se o casamento falhar, pode ser necessário retroceder (backtrack) e fazer a instanciação com outro valor

Cálculo de predicados e demostração de teoremas

- Uma propriedade importante da resolução é a habilidade de detectar inconsistências em um conjunto de proposições
- Esta propriedade pode ser usada para demonstrar teoremas (prova por contradição)
  - As proposições iniciais são chamadas de **hipóteses**
  - A negação do teorema é chamado de **meta**
  - O teorema é demonstrado encontrando-se uma inconsistência
- A demostração de teoremas é a base para a programação lógica

Cálculo de predicados e demostração de teoremas

- Para simplificar a resolução, as proposições devem estar em uma forma clausal restrita, chamada de **cláusulas de Horn**
  - Tem no máximo uma proposição atômica do lado esquerdo
  - Uma cláusula de Horn sem lado esquerdo é chamada de **fato**
  - O lado esquerdo é chamada de **cabeça**
  - O lado direito é chamada de **cauda**
  - A maioria, mas não todas, as proposições podem ser escritas como cláusulas de Horn

Uma visão geral da programação lógica

Uma visão geral da programação lógica

- Semântica declarativa
  - Existe uma maneira simples de determinar o significado de cada declaração
  - Semântica mais simples que linguagens imperativas
- Programação não procedural
  - Os programas não dizem como os resultados são computados, mas a forma dos resultados
  - Para isto, é necessário uma maneira concisa de fornecer as informações relevantes para o computador e um método de inferência para computar os resultados desejados
  - Cálculo de predicados e resolução
- Exemplo: ordenar uma lista com índices $1..n$
  - sort(old_list, new_list) $\subset$ permute(old_list, new_list) $\cap$ sorted (new_list)
  - sorted(list) $\subset \forall j$ tal que $1 \leq j < n$, list(j) $\leq$ list($j+1$)

As origens da linguagem Prolog

As origens da linguagem Prolog

- Universidade de Aix-Marseille
  - Alain Colmerauer e Phillippe Roussel
  - Processamento de linguagem natural
- Universidade de Edinburgh
  - Robert Kowalski
  - Demostração automática de teoremas
- A parceria terminou em meados da década de 70
- 1982-1992 FGCS (Fifth Generation Computins Systems) - projeto Japonês
  - Um dos objetivos era desenvolver máquinas inteligentes (usando Prolog)

Os elementos básicos da linguagem Prolog

Os elementos básicos da linguagem Prolog

- Prolog tem vários dialetos, o livro apresenta a sintaxe de Edinburgh
- Todas as sentenças em Prolog são construídas com termos
- Termos
  - Constante
    - Átomo (símbolo)
    - Inteiro
  - Variável
  - Estrutura

Os elementos básicos da linguagem Prolog

- Um átomo consiste de
  - Uma string de letras, dígitos e underscores, começando com uma letra minúscula
  - Uma string de caracteres ascii imprimíveis delimitada por apóstrofos
- Variável, qualquer string de letras, dígitos e underscores, começando com uma letra maiúscula
- Estrutura, representa uma proposição atômica do cálculo de predicados: functor(lista de parâmetros)

Os elementos básicos da linguagem Prolog

- Declaração de fatos
  - Cláusulas de Horn sem cabeça
  - Usada para definir as hipóteses, ou base de dados de informações
  - As cláusula de Horn sem cabeça em Prolog são chamadas de fatos, proposições que são (assumidas) verdadeiras

Os elementos básicos da linguagem Prolog

- Declaração de fatos
  - Cláusulas de Horn sem cabeça
  - Usada para definir as hipóteses, ou base de dados de informações
  - As cláusula de Horn sem cabeça em Prolog são chamadas de fatos, proposições que são (assumidas) verdadeiras
- Exemplo

```
mulher(maria).
homem(joao).
mulher(paula).
homem(jose).
pai(joao, jose).
pai(joao, maria).
mae(paula, jose).
mae(paula, maria).
```

Os elementos básicos da linguagem Prolog

- Declaração de regras
  - Cláusulas de Horn com cabeça
  - Usada para definir as hipóteses
  - O lado direito é o antecedente ou parte **se**
  - O lado esquerdo é o consequente ou parte **então**
  - Se o antecedente for verdadeiro, então o consequente também precisa ser verdadeiro
  - A forma geral é: `consequente :- antecedentes`
  - As cláusulas de Horn com cabeça em Prolog são chamadas de regras, porque elas definem as regras de implicação entre as proposições

Os elementos básicos da linguagem Prolog

- Declaração de regras
  - Cláusulas de Horn com cabeça
  - Usada para definir as hipóteses
  - O lado direito é o antecedente ou parte **se**
  - O lado esquerdo é o consequente ou parte **então**
  - Se o antecedente for verdadeiro, então o consequente também precisa ser verdadeiro
  - A forma geral é: `consequente :- antecedentes`
  - As cláusulas de Horn com cabeça em Prolog são chamadas de regras, porque elas definem as regras de implicação entre as proposições
- Exemplo

```
ancestral(paula, maria) :- mae(paula, maria).
pais(X, Y) :- mae(X, Y).
pais(X, Y) :- pai(X, Y).
irmaos(X, Y) :- mae(M, X), mae(M, Y), pai(P, X), pai(P, Y).
```

Os elementos básicos da linguagem Prolog

- Declaração de regras
  - Cláusulas de Horn com cabeça
  - Usada para definir as hipóteses
  - O lado direito é o antecedente ou parte **se**
  - O lado esquerdo é o consequente ou parte **então**
  - Se o antecedente for verdadeiro, então o consequente também precisa ser verdadeiro
  - A forma geral é: `consequente :- antecedentes`
  - As cláusulas de Horn com cabeça em Prolog são chamadas de regras, porque elas definem as regras de implicação entre as proposições
- Exemplo

```
ancestral(paula, maria) :- mae(paula, maria).
pais(X, Y) :- mae(X, Y).
pais(X, Y) :- pai(X, Y).
irmaos(X, Y) :- mae(M, X), mae(M, Y), pai(P, X), pai(P, Y).
```

- Para todo X e Y, X é irmão de Y se existem M e P tal que M é mãe de X e Y e P é pai de X e Y.

Os elementos básicos da linguagem Prolog

- Declaração de metas
  - Os fatos e as regras são a base para a demostração de teoremas
  - Um teorema está na forma de uma proposição que o sistema deve provar ser verdadeiro ou falso
  - Estas proposições em Prolog são chamadas de metas ou consultas
  - A forma da declaração de meta é idêntica a da declaração de fatos

    `homem(pedro).`
  - Proposições conjuntivas e proposições com variáveis também são metas válidas

    ```
    pai(X, paulo).
    ```

O processo de inferência do Prolog

- O uso eficiente do Prolog requer que o programador saiba precisamente o que o sistema faz com o seu programa
- Para provar que uma meta é verdadeira, é necessário encontrar uma cadeia de fatos e regras de inferência que conectam a meta com um ou mais fatos na base de dados
- Se a meta é Q, então Q precisa ser encontrado como um fato na base de dados ou o processo de inferência precisa encontrar um fato P_1 e uma sequência de proposições P_2, P_3, ..., P_n tal que
  $P_2 :- P_1$
  $P_3 :- P_2$
  ...
  $Q :- P_n$
- O processo é um pouco mais complicado para regras com o lado direito composto e para regras com variáveis

O processo de inferência do Prolog

- Quando uma meta é uma proposição composta, cada uma das estruturas são chamadas de **submetas**
- O processo de prova de uma submeta é feita através do **casamento** com uma proposição
- Exemplo
  - Meta: `homem(bob).`
  - Se a base de dados contém o fato `homem(bob)` a prova é trivial
  - Se a base de dados contém

```
pai(bob).
homem(X) :- pai(X).
```

  - o Prolog deve usar estas duas declarações para inferir que a meta é verdadeira

O processo de inferência do Prolog

- Existem duas abordagens para tentar casar uma meta com um fato na base de dados

O processo de inferência do Prolog

- Existem duas abordagens para tentar casar uma meta com um fato na base de dados
- Resolução de baixo para cima (bottom-up), encadeamento para frente
  - Inicia com os fatos e regras da base de dados e tenta encontrar a sequência que leva a meta
  - Funciona bem com um conjunto grande de possíveis repostas corretas

O processo de inferência do Prolog

- Existem duas abordagens para tentar casar uma meta com um fato na base de dados
- Resolução de baixo para cima (bottom-up), encadeamento para frente
  - Inicia com os fatos e regras da base de dados e tenta encontrar a sequência que leva a meta
  - Funciona bem com um conjunto grande de possíveis repostas corretas
- Resolução de cima para baixo (top-down), encadeamento para trás
  - Inicia com a meta e tenta encontrar uma sequência que leva a um conjunto de fatos na base de dados
  - Funciona bem para um conjunto pequeno de possíveis repostas corretas

O processo de inferência do Prolog

- Existem duas abordagens para tentar casar uma meta com um fato na base de dados
- Resolução de baixo para cima (bottom-up), encadeamento para frente
  - Inicia com os fatos e regras da base de dados e tenta encontrar a sequência que leva a meta
  - Funciona bem com um conjunto grande de possíveis repostas corretas
- Resolução de cima para baixo (top-down), encadeamento para trás
  - Inicia com a meta e tenta encontrar uma sequência que leva a um conjunto de fatos na base de dados
  - Funciona bem para um conjunto pequeno de possíveis repostas corretas
- As implementações do Prolog usam o encadeamento para trás

O processo de inferência do Prolog

- Quando uma meta tem mais que uma submeta, a busca pode ser
  - Primeiro em profundidade: encontra-se uma prova completa da primeira submeta antes de trabalhar com as outras
  - Primeiro em largura: trabalha-se em todas as submetas em paralelo

O processo de inferência do Prolog

- Quando uma meta tem mais que uma submeta, a busca pode ser
  - Primeiro em profundidade: encontra-se uma prova completa da primeira submeta antes de trabalhar com as outras
  - Primeiro em largura: trabalha-se em todas as submetas em paralelo
- Prolog usa busca primeiro em profundidade, pode ser feito com menos recursos computacionais

O processo de inferência do Prolog

- Com uma meta composta, se a prova de uma submeta falha, é necessário considerar as submetas anteriores para encontrar uma solução alternativa
- Este processo é chamado de **retroceder** (backtrack)
- A busca continua onde a busca anterior parou
- Pode levar muito tempo e ocupar muito espaço pois todas as possível provas para cada submeta podem ter que ser pesquisadas

O processo de inferência do Prolog

- O Prolog tem a estrutura pré-definida `trace` que exibe as instanciações a cada passo
- Existem 4 eventos de rastreamento
  - **call** início da tentativa de satisfazer uma meta
  - **exit** quando uma meta foi satisfeita
  - **redo** quando ocorre retrocesso
  - **fail** quando uma meta falha

O processo de inferência do Prolog

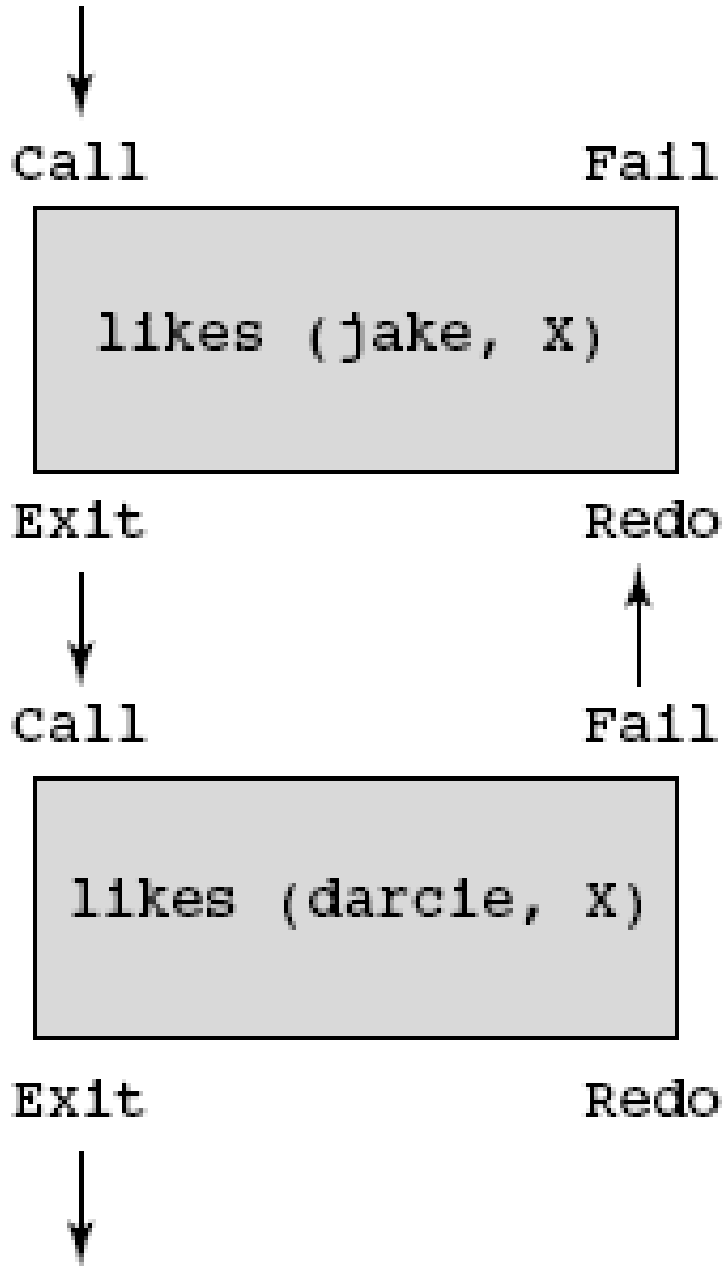
```
likes(jake, chocolate).
likes(jake, apricots).
likes(darcie, licorice).
likes(darcie, apricots).

likes(jake, X), likes(darcie, X).
   Call: (7) likes(jake, _G367) ?
   Exit: (7) likes(jake, chocolate) ?
   Call: (7) likes(darcie, chocolate) ?
   Fail: (7) likes(darcie, chocolate) ?
   Redo: (7) likes(jake, _G367) ?
   Exit: (7) likes(jake, apricots) ?
   Call: (7) likes(darcie, apricots) ?
   Exit: (7) likes(darcie, apricots) ?
X = apricots.
```

O processo de inferência do Prolog

Figure 16.1

Control flow model for the goal `likes (jake, X), likes (darcie, X)`



Aritmética

- Prolog suporta variáveis e aritmética inteira e real
- Operador `is`: uma expressão aritmética como operando do lado direito e uma variável do lado esquerdo
- `A is B / 17 + C`
- Não é a mesma coisa que atribuição!

Aritmética

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed),
                  time(X, Time),
                  Y is Speed * Time.

distance(chevy, D).
   Call: (6) distance(chevy, _G367) ?
   Call: (7) speed(chevy, _G437) ?
   Exit: (7) speed(chevy, 105) ?
   Call: (7) time(chevy, _G437) ?
   Exit: (7) time(chevy, 21) ?
^  Call: (7) _G367 is 105*21 ?
^  Exit: (7) 2205 is 105*21 ?
   Exit: (6) distance(chevy, 2205) ?
D = 2205.
```

Listas

- Além de proposições atômicas, Prolog oferece outra estrutura básica, a lista
- Listas são sequências de elementos
- Os elementos podem ser átomos, proposições atômicas, ou qualquer outro termo, incluindo outras listas

```
[maça, laranja, amora, pera]

[]          % lista vazia

[ X | Y ]   % cabeça X e calda Y
            % usado para construir e desmanchar
            % listas
```

- Segue alguns exemplos de operações com listas

Listas

```
member(Element, [Element | _]).
member(Element, [_ | Tail]) :-
    member(Element, Tail).

member(a, [b, c, d]).
   Call: (6) member(a, [b, c, d]) ?
   Call: (7) member(a, [c, d]) ?
   Call: (8) member(a, [d]) ?
   Call: (9) member(a, []) ?
   Fail: (9) member(a, []) ?
   Fail: (8) member(a, [d]) ?
   Fail: (7) member(a, [c, d]) ?
   Fail: (6) member(a, [b, c, d]) ?
false.

[trace]  ?- member(b, [a, b, c]).
   Call: (6) member(b, [a, b, c]) ?
   Call: (7) member(b, [b, c]) ?
   Exit: (7) member(b, [b, c]) ?
   Exit: (6) member(b, [a, b, c]) ?
true ;
   ...
```

Listas

```
append([], List, List).
append([Head | Tail1], List2, [Head | Tail3]) :-
    append(Tail0, List2, Tail3).

append([bob, jo], [jake, darcie], Family).
   Call: (6) append([bob, jo], [jake, darcie], _G380) ?
   Call: (7) append([jo], [jake, darcie], _G459) ?
   Call: (8) append([], [jake, darcie], _G462) ?
   Exit: (8) append([], [jake, darcie], [jake, darcie]) ?
   Exit: (7) append([jo], [jake, darcie], [jo, jake, darcie]) ?
   Exit: (6) append([bob, jo], [jake, darcie], [bob, jo, jake, darcie])
Family = [bob, jo, jake, darcie].
```

```
reverse([], []).
reverse([Head | Tail], List) :-
    reverse(Tail, Result),
    append(Result, [Head], List).
```

```
reverse([a, b, c], Q).
   Call: (6) reverse([a, b, c], _G376) ?
   Call: (7) reverse([b, c], _G455) ?
   Call: (8) reverse([c], _G455) ?
   Call: (9) reverse([], _G455) ?
   Exit: (9) reverse([], []) ?
   Call: (9) append([], [c], _G459) ?
   Exit: (9) append([], [c], [c]) ?
   Exit: (8) reverse([c], [c]) ?
   Call: (8) append([c], [b], _G462) ?
   Call: (9) append([], [b], _G457) ?
   Exit: (9) append([], [b], [b]) ?
   Exit: (8) append([c], [b], [c, b]) ?
   Exit: (7) reverse([b, c], [c, b]) ?
   Call: (7) append([c, b], [a], _G376) ?
   Call: (8) append([b], [a], _G463) ?
   Call: (9) append([], [a], _G466) ?
   Exit: (9) append([], [a], [a]) ?
   Exit: (8) append([b], [a], [b, a]) ?
   Exit: (7) append([c, b], [a], [c, b, a]) ?
   Exit: (6) reverse([a, b, c], [c, b, a]) ?
Q = [c, b, a].
```

Deficiências do Prolog

Deficiências do Prolog

- Controle da ordem de resolução
  - O Prolog sempre faz o casamento (na resolução) na mesma ordem, o que implica que o programador pode (deve) controlar a ordem da resolução alterando a ordem das proposições e das submetas
  - Exemplo: regra que causa laço infinito na resolução

```
f(X, Y) :- f(Z, Y), g(X, Z).
```

  - O Prolog permite o controle explícito do retrocesso com o operador de corte !, que as vezes é usado para criar controle de fluxo (no estilo imperativo)
  - O Prolog oferece estas possibilidades principalmente para permitir a construção de programas mais eficientes, mas elas contrariam a principal característica das linguagens lógicas, que é a construção de programas de forma declarativa, sem a preocupação com o processo de resolução

Deficiências do Prolog

- Mundo fechado assumido
  - Não tem conhecimento do mundo além daquele que está em sua base de dados
  - O Prolog pode provar que uma determinada meta é verdadeira, mas não pode provar que é falsa
  - O Prolog é um sistema verdadeiro/falha e não um sistema verdeiro/falso

Deficiências do Prolog

- Mundo fechado assumido
  - Não tem conhecimento do mundo além daquele que está em sua base de dados
  - O Prolog pode provar que uma determinada meta é verdadeira, mas não pode provar que é falsa
  - O Prolog é um sistema verdadeiro/falha e não um sistema verdeiro/falso
- O problema da negação
  - O Prolog tem um operador `not`, mas este operador não é equivalente ao not lógico
  - A razão fundamental para isto é que, em uma proposição $A :- B_1 \cap B_2 \cap \cdots \cap B_n$, se algum B é falso, não se pode concluir que A é falso
  - A partir de lógica positiva, só se pode concluir lógica positiva

Deficiências do Prolog

- Limitações intrínsecas
  - Não é possível (ainda) resolver determinados problemas de forma eficiente apenas com a descrição da solução
  - Um exemplo é a ordenação
  - Apenas com a descrição do que é uma lista ordenada não é suficiente para uma solução eficiente

Aplicações da programação lógica

Aplicações da programação lógica

- Sistema de gerenciamento de banco de dados relacional
- Sistemas especialistas
- Processamento de linguagem natural

Referências

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 16.