Introdução a Prolog

Sobre a linguagem

- Criada em 1972 por Alain Colmerauer e Robert Kowaslki.
- Assim como Haskell, é declarativa.
- Diferente de Haskell, é considerada uma linguagem lógica.
- Linguagem lógica: programas em Prolog especificam relações entre objetos.
- Essas relações ocorrem por meio de axiomas e regras.
- Assim como gcc ou cmake para C, possui diversas implementações, sendo SWI a mais polular.

Instalação (sistema)

- sudo apt install swi-prolog

Introdução

- Um programa Prolog é uma sequência de cláusulas Horn que definem o que é verdadeiro. Há no máximo uma cláusula verdadeira.

Introdução

- Um programa Prolog é uma sequência de cláusulas Horn que definem o que é verdadeiro. Há no máximo uma cláusula verdadeira.
- Cláusulas de Horn têm o seguinte formato:
  - H V ¬B1 V ¬B2 … V ¬Bn

Introdução

- Um programa Prolog é uma sequência de cláusulas Horn que definem o que é verdadeiro. Há no máximo uma cláusula verdadeira.
- Cláusulas de Horn têm o seguinte formato:
  - $H \vee \neg B1 \vee \neg B2 \ldots \vee \neg Bn$
- Que é equivalente a
  - $H \vee \neg(B1 \wedge B2 \ldots \wedge Bn)$

Introdução

- Um programa Prolog é uma sequência de cláusulas Horn que definem o que é verdadeiro. Há no máximo uma cláusula verdadeira.
- Cláusulas de Horn têm o seguinte formato:
  - $H \vee \neg B1 \vee \neg B2 \ldots \vee \neg Bn$
- Que é equivalente a
  - $H \vee \neg(B1 \wedge B2 \ldots \wedge Bn)$
- Que equivale a
  - $(B1 \wedge B2 \ldots \wedge Bn) \rightarrow H$

Introdução

- Em programação lógica, significa que, para provar H, primeiros verifica B1, B2, … Bn.

Estrutura de um programa

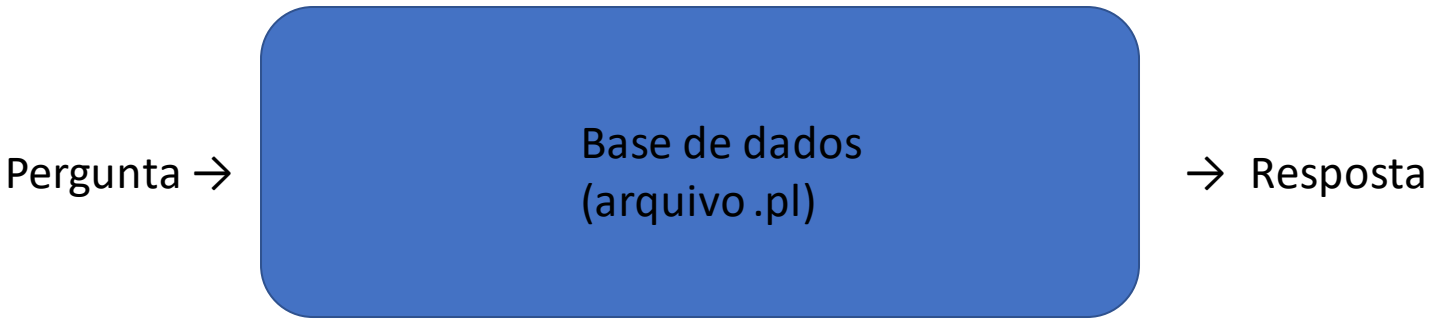
- Um programa Prolog possui uma série de predicados.
- Cada predicado tem um nome, e é seguido de zero ou mais argumentos (o Header da cláusula Horn).
- Cada argumento é um termo.
- Um predicado P com N argumentos é denotado por P/N.
  - Análogo à notação para assinatura de funções.
- Cada predicado é definido por fatos ou regras que formam seu corpo (Body da cláusula Horn).
- Se todas as cláusulas no Body forem verdadeiras, o predicado é verdadeiro.

Tipos

- Dependendo da implementação do Prolog podem variar, mas costumam estar implementados:

Tipo	Valores
boolean	true, fail
integer	Inteiros
real	Números de ponto fluante
variable	Variáveis (começam com *uppercase*)
atom	Sequência de caracteres (aspas ou sem aspas se começarem com *lowercase*)

Fluxo



Estrutura de um programa: fatos

```
%parent(Parent, Child)
parent(albert, jim).
parent(albert, peter).
parent(pat, jim).
parent(amanda, jenny).
parent(jenny, peter).
```

```
male(albert).
male(jim).
male(peter).

female(pat).
female(jenny).
female(amanda).
```

Queries

- Fazemos perguntas e obtemos respostas.

Queries

```
%parent(Parent, Child)
parent(albert, jim).
parent(albert, peter).
parent(pat, jim).
parent(amanda, jenny).
parent(jenny, peter).
```

```
?- [aula].
true.

?- parent(albert, peter).
true.

?- parent(pat, X).
X = jim.

?- parent(albert, X).
X = jim ;
X = peter.
```

Queries

```
%parent(Parent, Child)
parent(albert, jim).
parent(albert, peter).
parent(pat, jim).
parent(amanda, jenny).
parent(jenny, peter).
```

```
?- parent(Parent, Child).
Parent = albert,
Child = jim ;
Parent = albert,
Child = peter ;
Parent = pat,
Child = jim ;
Parent = amanda,
Child = jenny ;
Parent = jenny,
Child = peter.

?-
```

Estrutura de um programa: regras

- Possuem formato Head :- Body (cláusulas Horn)

```
grandparent(Grandparent, Grandchild) :-
    parent(Grandparent, Child),
    parent(Child, Grandchild).
```

Estrutura de um programa: regras

- Não são funções, mas predicados, que apenas avaliam para verdadeiro ou falso.
- Predicados não retornam valores; apenas descrevem relações entre termos.
- Essas relações, que podem ser pensadas como o retorno, são armazenadas nas próprias variáveis passadas como argumento.
- Prolog procura as instâncias na base de dados para as quais o predicado é verdadeiro.

```
grandparent(Grandparent, Grandchild) :-
    parent(Grandparent, Child),
    parent(Child, Grandchild).
```

Aplicando

```
grandparent(Grandparent, Grandchild) :-
    parent(Grandparent, Child),
    parent(Child, Grandchild).
```

```
?- grandparent(Grandparent, Grandchild).
Grandparent = amanda,
Grandchild = peter .

?-
```

Help

- help(algumaCoisa) abre a "man page" do predicado algumaCoisa.

Aritmética

- Em Prolog, tudo é simbólico, o que torna operações matemáticas um pouco diferentes em relação a outras linguagens.
- *Variable* is *<expression>.*
  - X is 11 + 5 * 6 / 3.
    - X = 21
  - X is 10 mod 4.
    - X = 2.
  - 3 + 5 =:= 8.
    - true

Comando	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão (real)
//	Divisão (inteira)
mod	Resto
**	Potência

Comparação	Notação
Menor	X < Y.
Menor ou igual	X =< Y.
Igual	X =:= Y.
Diferente	X =\= Y.
Maior ou igual	X >= Y
Maior	X > Y

Aritmética

```
fatorial(0, 1).
fatorial(N, Valor) :-
    N1 is N - 1, fatorial(N1, Anterior), Valor is Anterior * N.
```

Aritmética

```
fatorial(0, 1).
fatorial(N, Valor) :-
    N1 is N - 1, fatorial(N1, Anterior), Valor is Anterior * N.
```

```
?- fatorial(0,1).
true .

?- fatorial(0,X).
X = 1 .

?- fatorial(5,X).
X = 120 .
```

Aritmética

```
fibonacci(0, 1).
fibonacci(1, 1).
fibonacci(N, V) :- N > 1, N1 is N - 1, N2 is N - 2, fibonacci(N1,V1), fibonacci(N2,V2), V is V1 + V2.
```

Aritmética

```
fibonacci(0, 1).
fibonacci(1, 1).
fibonacci(N, V) :- N > 1, N1 is N - 1, N2 is N - 2, fibonacci(N1,V1), fibonacci(N2,V2), V is V1 + V2.
```

```
?- fibonacci(0, 1).
true .

?- fibonacci(0, X).
X = 1 .

?- fibonacci(5, X).
X = 8 .

?- fibonacci(10, X).
X = 89 .
```

Exercício 1

- Implemente um predicado collatz que recebe dois argumentos, N e R. Se N for par, R recebe a metade de N. Se for ímpar, recebe 3N + 1.

Exercício 1

- Implemente um predicado collatz que recebe dois argumentos, N e R. Se N for par, R recebe a metade de N. Se for ímpar, recebe 3N + 1.

```
collatz(N, R) :-
    Temp is N mod 2,
    Temp =:= 0,
    R is N / 2.
collatz(N, R) :-
    Temp is N mod 2,
    Temp =\= 0,
    R is N*3 + 1.
```

Listas

- Listas são definidas recursivamente, como em Haskell:
  - List --> [ ]
  - List --> [Element|List]
- Sintaxe de uso semelhante a Haskell ou Python:
  - [elemento1, elemento2, elemento3]
  - Não há restrição quanto a o que está na lista.
- Predicados costumam seguir o padrão de conter um caso para a lista vazia e o caso genérico.

Listas: exemplos

- [] é uma lista vazia.
- [a] é uma lista com um átomo a.
- [a, b] é uma lista com dois átomos.
- [a, [], parent(jim, peter)] é uma lista com um átomo, uma lista vazia e um termo composto.

Listas: operações

```
cons(L1, L2, Result) :-
    Result = [L1 | L2].
```

```
?- cons(1, [2,11,10], X).
X = [1, 2, 11, 10].
```

Listas

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

Listas

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

```
?- is_member("Primeiro", ["Primeiro", "segundo", 14]).
true .

?- is_member("14", ["Primeiro", "segundo", 14]).
false.

?- is_member(14, ["Primeiro", "segundo", 14]).
true .
```

Listas

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

```
is_member :: (Eq a) => a -> [a] -> Bool
is_member a [] = False
is_member a (x:xs)
    | a == x = True
    | otherwise = is_member a xs
```

Listas

- Como obter o último elemento da lista?

Listas

- Como obter o último elemento da lista?

```
get_last_element([], Element) :- Element = [].
get_last_element([ _ | Tail], Element) :-
    Tail \== [], get_last_element(Tail,Element).
get_last_element([Head | Tail], Element) :-
    Tail == [], Element = [Head].
```

Listas

- Prolog já oferece diversos predicados que implementam tarefas comuns quando se trabalha com listas.
  - member/2
  - append/3
  - delete/3
  - reverse/2
  - length/2
- E mais.

Exercício 2

- Implemente um predicado comprimento/2. Ele recebe dois argumentos: uma lista e uma variável C. Ele armazena em C o comprimento da lista.

Exercício 2

- Implemente um predicado comprimento/2. Ele recebe dois argumentos: uma lista e uma variável C. Ele armazena em C o comprimento da lista.

```
comprimento :: [a] -> Int
comprimento [] = 0
comprimento (x:xs) = 1 + comprimento xs
```

Exercício 2

- Implemente um predicado comprimento/2. Ele recebe dois argumentos: uma lista e uma variável C. Ele armazena em C o comprimento da lista.

```haskell
comprimento :: [a] -> Int
comprimento [] = 0
comprimento (x:xs) = 1 + comprimento xs
```

- Em Haskell, podemos usar o retorno da função. Em prolog, não temos funções.

Exercício 2

- Implemente um predicado comprimento/2. Ele recebe dois argumentos: uma lista e uma variável C. Ele armazena em C o comprimento da lista.

```
comprimento([], C) :- C is 0.

comprimento([_|T], C) :-
    comprimento(T, Ccauda),
    C is 1 + Ccauda.
```

```
comprimento :: [a] -> Int
comprimento [] = 0
comprimento (x:xs) = 1 + comprimento xs
```

- Em Haskell, podemos usar o retorno da função. Em prolog, não temos funções.

Predicados não são funções

- Você não pode especificar um tipo de retorno para um predicado.
- O "retorno" é armazenado em um dos parâmetros.
- Não pode passar predicados para outros predicados e usá-los diretamente como funções em Haskell.
- Como implementar um map?

Predicados não são funções

- Você não pode especificar um tipo de retorno para um predicado.
- O "retorno" é armazenado em um dos parâmetros.
- Não pode passar predicados para outros predicados e usá-los diretamente como funções em Haskell.
- Como implementar um map?

```
map :: (t -> a) -> [t] -> [a]
map func [] = []
map func (x:xs) = func x : (map func xs)
```

Predicados não são funções

- Você não pode especificar um tipo de retorno para um predicado.
- O "retorno" é armazenado em um dos parâmetros.
- Não pode passar predicados para outros predicados e usá-los diretamente como funções em Haskell.
- Como implementar um map?

```
map(_, [], []).
map(Predicado, [H | T], [N | NS]) :-
    call(Predicado, H, N),
    map(Predicado, T, NS).
```

```
map :: (t -> a) -> [t] -> [a]
map func [] = []
map func (x:xs) = func x : (map func xs)
```

Predicados não são funções

- Você não pode especificar um tipo de retorno para um predicado.
- O "retorno" é armazenado em um dos parâmetros.
- Não pode passar predicados para outros predicados e usá-los diretamente como funções em Haskell.
- Como implementar um map?

```
map(_, [], []).
map(Predicado, [H | T], [N | NS]) :-
    call(Predicado, H, N),
    map(Predicado, T, NS).
```

```
?- map(fatorial, [1,2,3,4,5],L).
L = [1, 2, 6, 24, 120] .
```

Misturando listas e aritmética

- Mesma ideia vale para gerar uma lista de anos bissextos.
- Predicado não retorna uma lista de inteiros.
- Essa lista é armazenada em um parâmetro.

Misturando listas e aritmética

- Mesma ideia vale para gerar uma lista de anos bissextos.
- Predicado não retorna uma lista de inteiros.
- Essa lista é armazenada em um parâmetro.

```
anoBissexto(Y) :-
    Y mod 4 =:= 0,
    Y mod 100 =:= 0,
    Y mod 400 =:= 0.
anoBissexto(Y) :-
    Y mod 4 =:= 0,
    Y mod 100 =\= 0.

bissextos(Ys) :-
    findall(X, (between(1584, 2020, X), anoBissexto(X) ), Ys).
```

Misturando listas e aritmética

- Armazenar retornos em argumentos pode deixar o código mais verboso.
- Por exemplo, na distância de Manhattan:

```
manhattan_distance([], [], D) :- D is 0.
manhattan_distance([H1|T1], [H2|T2], D) :-
    Signed is (H1 - H2),
    abs(Signed, Unsigned),
    manhattan_distance(T1, T2, NovoD),
    D is NovoD + Unsigned.
```

```
manhattan_distance :: [Double] -> [Double] -> Double
manhattan_distance v1 v2 =
    sum $ map (\(a, b) -> abs(a-b) ) $ zip v1 v2
```

Misturando listas e aritmética: exemplo

- Como implementar um predicado log_table que recebe uma lista de inteiros e devolve uma lista de lista de números, em que cada sublista contém, na primeira posição, o elemento original, e na segunda, o seu log?
- O predicado log/2 já existe.
- Exemplo:

```
?- log_table([10,12, 1],R).
R = [[10, 2.302585092994046], [12, 2.484906649788000 4], [1, 0.0]].
```

Misturando listas e aritmética: exemplo

```
% Calcula o log da cabeça e "retorna" uma List [Head, Log]
get_log(Head, List) :-
    Log is log(Head),
    List = [Head,Log].

log_table([], ResultList) :- ResultList = [].
log_table( [ Head | Tail ], ResultList) :-
    get_log(Head,HeadLog),
    log_table(Tail, L),
    ResultList = [ HeadLog | L ].
```

Compilando Prolog

- Até agora escrevemos em um arquivo .pl e carregamos para o interpretador.
- É possível também compilar o código e gerar um executável.
- Defina um predicado main:

```
main :-
    write("Collatz\n"),
    collatz(9, R),
    write(R).
```

- Compile com
  - swipl --goal=main -o nomeExecutavel -c aula.pl

Testes unitários

- Testes são escritos entre as diretivas *begin_tests/1* e *end_tests/1.*
- Arquivo de teste tem extensão .plt.
- Deve ter o mesmo nome do arquivo .pl.

Testes unitários

- Testes são escritos entre as diretivas *begin_tests/1* e *end_tests/1.*
- Arquivo de teste tem extensão .plt.
- Deve ter o mesmo nome do arquivo .pl.

```
:- begin_tests(aula).
:- include(aula).

test(manhattanTest1) :- manhattan_distance([], [], 0).
test(manhattanTest2) :- manhattan_distance([1], [1], 0).
test(manhattanTest3) :- manhattan_distance([1,2], [2,1], 2).

test(consTest1) :- cons(0, [], [0]).
test(consTest2) :- cons(0, [3], [0, 3]).
test(consTest3) :- cons([3,2], [], [[3,2]]).

:- end_tests(aula).
```

Testes unitários

- Testes são escritos entre as diretivas *begin_tests/1* e *end_tests/1.*
- Arquivo de teste tem extensão .plt.
- Deve ter o mesmo nome do arquivo .pl.

```
:- begin_tests(aula).
:- include(aula).

test(manhattanTest1) :- manhattan_distance([], [], 0).
test(manhattanTest2) :- manhattan_distance([1], [1], 0).
test(manhattanTest3) :- manhattan_distance([1,2], [2,1], 2).

test(consTest1) :- cons(0, [], [0]).
test(consTest2) :- cons(0, [3], [0, 3]).
test(consTest3) :- cons([3,2], [], [[3,2]]).

:- end_tests(aula).
```

```
?- [aula].
true.

?- load_test_files([aula]).
true.

?- run_tests.
% PL-Unit: aula ...... done
% All 6 tests passed
true.
```

Functor em Prolog

Functor em Prolog

- **Não** tem relação com functor de Haskell.

Functor em Prolog

- Não tem relação com functor de Haskell.
- Se refere ao átomo de uma estrutura e sua aridade.
- Existe inclusive um predicado *functor* que informa o nome e a aridade de outro predicado.

Functor em Prolog

- Não tem relação com functor de Haskell.
- Se refere ao átomo de uma estrutura e sua aridade.
- Existe inclusive um predicado *functor* que informa o nome e a aridade de outro predicado.

```
?- functor(is_member(3, [2,3,2,1]), Nome, Aridade).
Nome = is_member,
Aridade = 2.
```

Mais

- https://www.sciencedirect.com/science/article/pii/0010448583901288
- https://www.jstor.org/stable/2268661?seq=1#page_scan_tab_contents
- https://www.swi-prolog.org
- https://www.lix.polytechnique.fr/~liberti/public/computing/prog/prolog/prolog-tutorial.html
- https://rextester.com/
- https://www.metalevel.at/prolog/concepts