Aula 8: Um olhar mais atento aos termos

- Teoria
  - Introduzir o predicado ==
  - Olhar atentamente na estrutura de um termo
  - Introduzir strings em Prolog
  - Introduzir operadores

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true

?- a==b.
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true

?- a==b.
false
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true

?- a==b.
false

?- a=='a'.
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true

?- a==b.
false

?- a=='a'.
true
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true

?- a==b.
false

?- a=='a'.
true

?- a==X.
```

Comparando termos: ==/2

- Prolog contém um importante predicado para comparar termos
- É o predicado identidade ==/2
- O predicado identidade ==/2 não instancia variáveis, isto é, ele comporta-se de forma diferente de =/2

```
?- a==a.
true

?- a==b.
false

?- a=='a'.
true

?- a==X.
false
```

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

```
?- X==X.
```

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

```
?- X==X.
true
```

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

```
?- X==X.
true

?- Y==X.
```

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

```
?- X==X.
true

?- Y==X.
false
```

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

```
?- X==X.
true

?- Y==X.
false

?- a=U, a==U.
```

Comparando variáveis

- Duas variáveis diferentes **não instanciadas** não são termos idênticos
- Variáveis **instanciadas** com um termo *T* são idênticas a *T*

```
?- X==X.
true

?- Y==X.
false

?- a=U, a==U.
U = a
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false

?- a \== b.
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false

?- a \== b.
true
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false

?- a \== b.
true

?- a \== 'a'.
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false

?- a \== b.
true

?- a \== 'a'.
false
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false

?- a \== b.
true

?- a \== 'a'.
false

?- a \== X.
```

Comparando termos: \==/2

- O predicado \==/2 é definido tal que ele tem sucesso precisamente naqueles casos nos quais ==/2 falha
- Em outras palavras, ele tem sucesso sempre que dois termos **não são idênticos**, e falha em todos os outros casos.

```
?- a \== a.
false

?- a \== b.
true

?- a \== 'a'.
false

?- a \== X.
true
```

Termos com uma notação especial

- Algumas vezes os termos parecem diferentes, mas Prolog trata-os como idênticos.
- Por exemplo: **a** e **'a',** mas existem muitos outros casos.
- Porque o Prolog faz isto?
  - Porque isto torna a programação mais agradável
  - É uma forma mais natural de codificar programas Prolog

Termos Aritméticos

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

Arithmetic terms

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

```
?- 2+3 == +(2,3).
```

Arithmetic terms

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

```
?- 2+3 == +(2,3).
true
```

Arithmetic terms

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

```
?- 2+3 == +(2,3).
true

?-  -(2,3) == 2-3.
```

Arithmetic terms

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

```
?- 2+3 == +(2,3).
true

?-  -(2,3) == 2-3.
true
```

Arithmetic terms

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

```
?- 2+3 == +(2,3).
true

?-  -(2,3) == 2-3.
true

?-  (4<2) == <(4,2).
```

Arithmetic terms

- Recorde que na aula 5 foram introduzidos os operadores aritméticos.
- +, -, <, >, etc são funtores e expressões tais como 2+3 são realmente termos complexos comuns.
- O termo 2+3 é idêntico ao termo +(2,3)

```
?- 2+3 == +(2,3).
true

?-  -(2,3) == 2-3.
true

?-  (4<2) == <(4,2).
true
```

Resumo dos predicados de comparação

=	Predicado de unificação
\=	Negação do predicado de unificação
==	Predicado identidade
\==	Negação do predicado identidade
=:=	Predicado de igualdade aritmética
=\=	Negação do predicado de igualdade aritmética

Listas como termos

- Um outro exemplo do Prolog trabalhando com uma representação interna, mas exibindo uma outra para o usuário
- Usando o construtor | , existem muitos modos de se escrever a mesma lista.

```
?- [a,b,c,d] == [a|[b,c,d]].
true
```

Listas como termos

- Um outro exemplo do Prolog trabalhando com uma representação interna, mas exibindo uma outra para o usuário
- Usando o construtor | , existem muitos modos de se escrever a mesma lista.

```
?- [a,b,c,d] == [a|[b,c,d]].
true
?- [a,b,c,d] == [a,b,c|[d]].
true
```

Listas como termos

- Um outro exemplo do Prolog trabalhando com uma representação interna, mas exibindo uma outra para o usuário
- Usando o construtor | , existem muitos modos de se escrever a mesma lista.

```
?- [a,b,c,d] == [a|[b,c,d]].
true
?- [a,b,c,d] == [a,b,c|[d]].
true
?- [a,b,c,d] == [a,b,c,d|[]].
true
```

Listas como termos

- Um outro exemplo do Prolog trabalhando com uma representação interna, mas exibindo uma outra para o usuário
- Usando o construtor | , existem muitos modos de se escrever a mesma lista.

```
?- [a,b,c,d] == [a|[b,c,d]].
true
?- [a,b,c,d] == [a,b,c|[d]].
true
?- [a,b,c,d] == [a,b,c,d|[]].
true
?- [a,b,c,d] == [a,b|[c,d]].
true
```

Listas Prolog internamente

- Internamente, as listas são construídas com dois termos especiais:
  - [] (que representa a lista vazia)
  - '.' (um funtor de aridade 2 usado para construir listas não vazias)
- Estes dois termos são também chamados de *construtores de listas.*
- Uma definição recursiva mostra como eles constroem listas.

Definição de uma lista Prolog

- A lista vazia é o termo []. Ela possui comprimento 0.
- Uma lista não vazia é qualquer termo da forma

  .(*termo*,*lista*)

- No qual *termo* é um termo Prolog, e *lista* é qualquer lista Prolog.
- Se *lista* possui comprimento *n*, então

.(*termo*,*lista*) possui comprimento *n*+1.

Uns poucos exemplos...

```
?- .(a,[]) == [a].
true
```

Uns poucos exemplos...

```
?- .(a,[]) == [a].
true

?- .(f(d,e),[]) == [f(d,e)].
true
```

Uns poucos exemplos...

```
?- .(a,[]) == [a].
true

?- .(f(d,e),[]) == [f(d,e)].
true

?- .(a,.(b,[])) == [a,b].
true
```

Uns poucos exemplos…

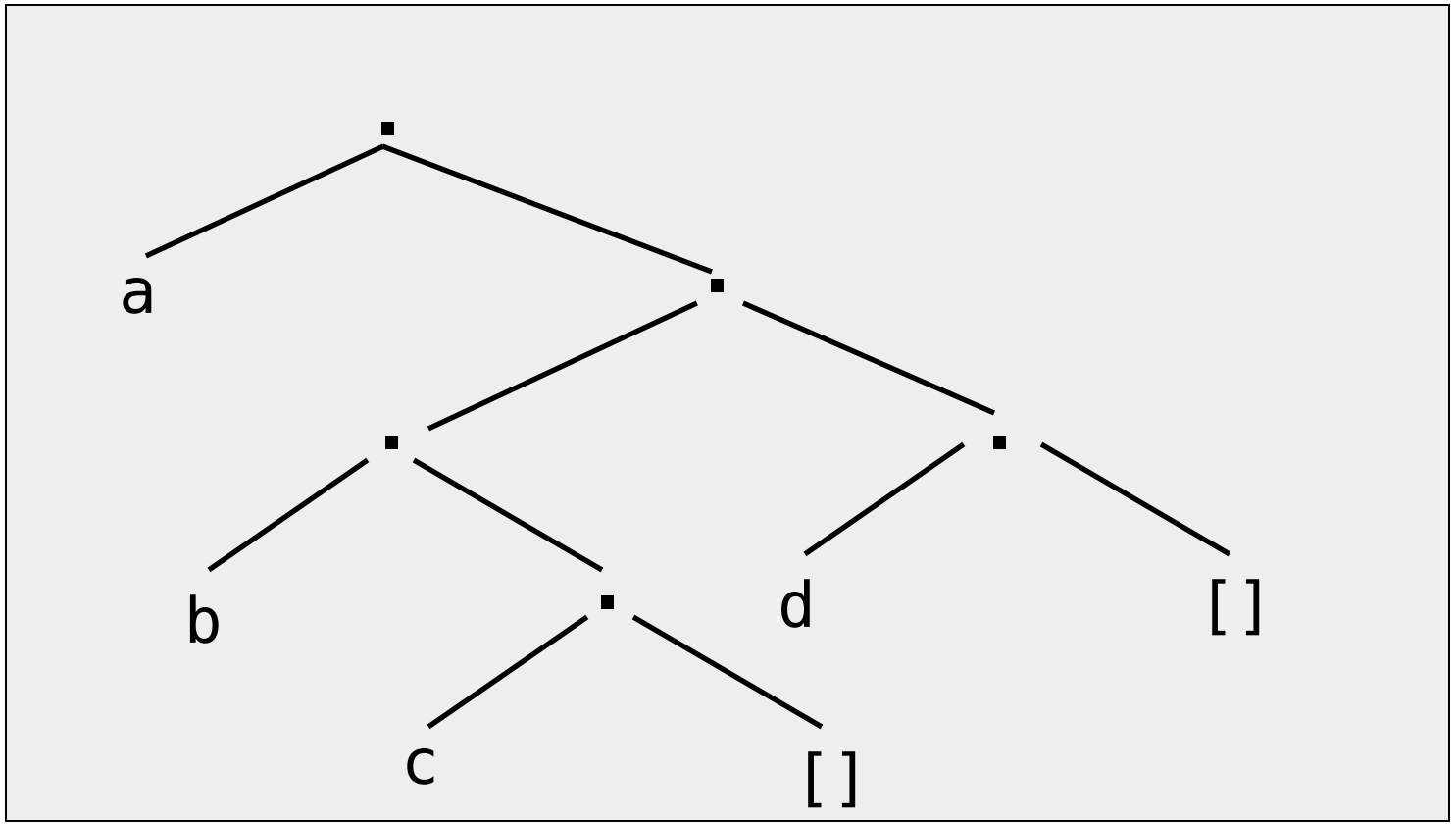
```
?- .(a,[]) == [a].
true

?- .(f(d,e),[]) == [f(d,e)].
true

?- .(a,.(b,[])) == [a,b].
true

?- .(a,.(b,.(f(d,e),[]))) == [a,b,f(d,e)].
true
```

Representação interna da lista

- Funcionamento similar à notação | :
- Representa-se uma lista em duas partes
  - Seu primeiro elemento, a *cabeça*
  - O resto da lista, a *cauda*
- O truque é ler estes termos como árvores
  - Nós internos são rotulados com .
  - Todos os nós possuem dois nós filhos
    - A subárvore sob o filho à esquerda é a cabeça
    - A subárvore sob o filho à direita é a cauda

Exemplo de uma lista como árvore

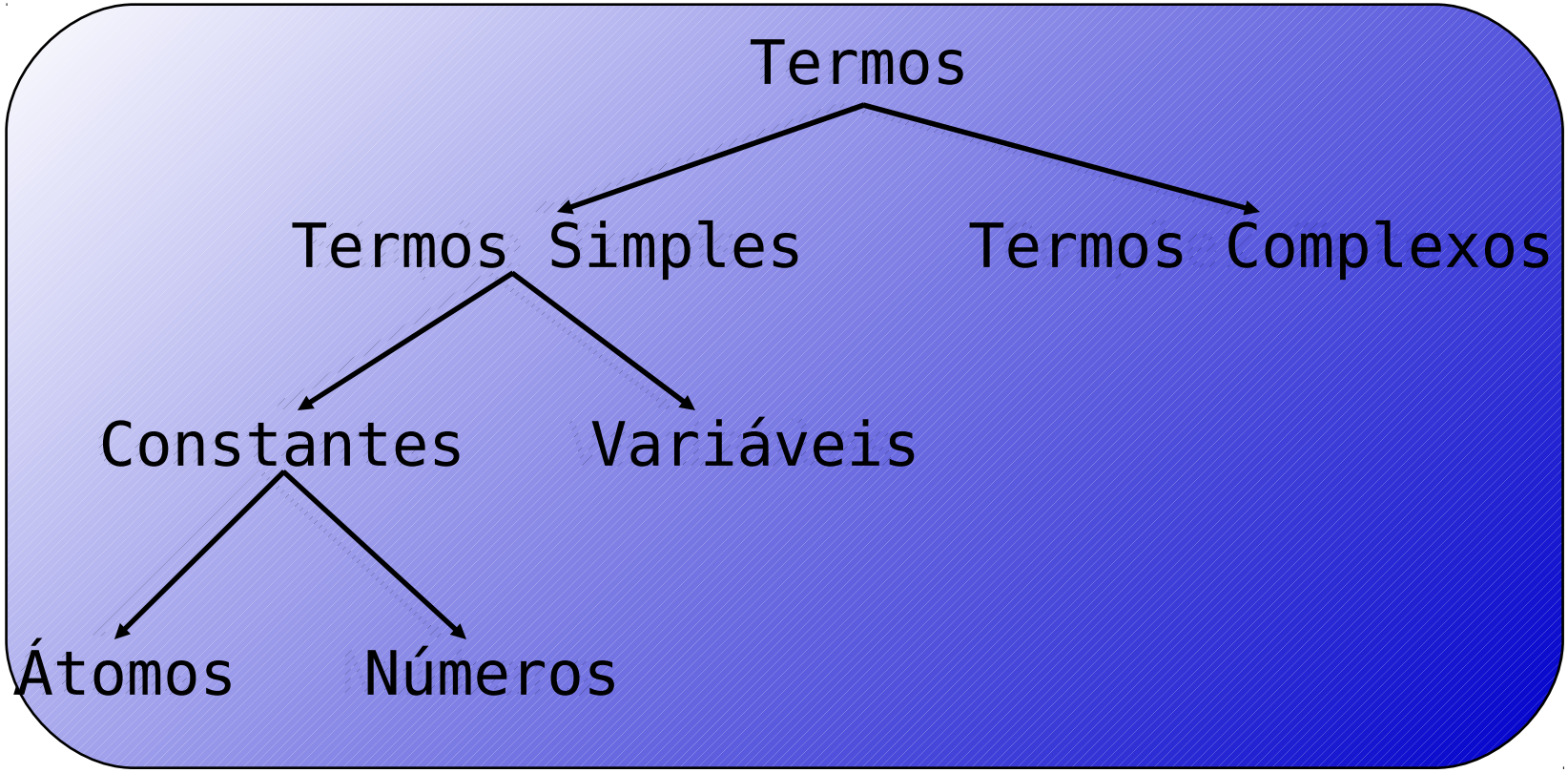
- Exemplo: [a,[b,c],d]



Examinando termos

- Nós nos voltaremos agora para predicados pré-construídos que permitem examinar os termos Prolog mais atentamente
  - Predicados que determinam o tipo dos termos
  - Predicados que nos dizem algo sobre a estrutura interna dos termos

Tipo dos termos



Verificando o tipo de um termo

atom/1	*O argumento é um átomo?*
integer/1	*… um inteiro?*
float/1	*… um número em ponto flutuante?*
number/1	*… um inteiro ou flutuante?*
atomic/1	*… uma constante?*
var/1	*… uma variável não instanciada?*
nonvar/1	*… uma variável instanciada ou um outro termo que não seja uma variável não instanciada*

Verificação de tipo: atom/1

?- atom(a).

Verificação de tipo: atom/1

?- atom(a).

true

Verificação de tipo: atom/1

?- atom(a).

true

?- atom(7).

Verificação de tipo: atom/1

?- atom(a).

true

?- atom(7).

false

Verificação de tipo: atom/1

?- atom(a).

true

?- atom(7).

false

?- atom(X).

Verificação de tipo: atom/1

?- atom(a).

true

?- atom(7).

false

?- atom(X).

false

Verificação de tipo: atom/1

?- X=a, atom(X).

Verificação de tipo: atom/1

?- X=a, atom(X).

X = a

true

Verificação de tipo: atom/1

?- X=a, atom(X).

X = a

true

?- atom(X), X=a.

Verificação de tipo: atom/1

?- X=a, atom(X).

X = a

true

?- atom(X), X=a.

false

Verificação de tipo: atomic/1

```
?- atomic(maria).
```

Verificação de tipo: atomic/1

?- atomic(maria).

true

Verificação de tipo: atomic/1

?- atomic(maria).

true

?- atomic(5).

Verificação de tipo: atomic/1

?- atomic(maria).

true

?- atomic(5).

true

Verificação de tipo: atomic/1

?- atomic(maria).

true

?- atomic(5).

true

?- atomic(ama(vicente,maria)).

Verificação de tipo: atomic/1

```
?- atomic(maria).
true

?- atomic(5).
true

?- atomic(ama(vicente,maria)).
false
```

Verificação de tipo: var/1

?- var(maria).

Verificação de tipo: var/1

?- var(maria).

false

Verificação de tipo: var/1

?- var(maria).

false

?- var(X).

Verificação de tipo: var/1

?- var(maria).

false

?- var(X).

true

Verificação de tipo: var/1

?- var(maria).

false

?- var(X).

true

?- X=5, var(X).

Verificação de tipo: var/1

?- var(maria).

false

?- var(X).

true

?- X=5, var(X).

false

Verificação de tipo: nonvar/1

?- nonvar(X).

Verificação de tipo: nonvar/1

?- nonvar(X).

false

Verificação de tipo: nonvar/1

?- nonvar(X).

false

?- nonvar(maria).

Verificação de tipo: nonvar/1

?- nonvar(X).

false

?- nonvar(maria).

true

Verificação de tipo: nonvar/1

?- nonvar(X).

false

?- nonvar(maria).

true

?- nonvar(23).

Verificação de tipo: nonvar/1

?- nonvar(X).

false

?- nonvar(maria).

true

?- nonvar(23).

true

A estrutura dos termos

- Dado um termo complexo de estrutura desconhecida, qual tipo de informação nós poderíamos extrair dele?
- Obviamente:
  - O funtor
  - A aridade
  - Os argumentos
- Prolog vem com predicados pré-construídos que produzem esta informação.

O predicado functor/3

- O predicado functor/3 informa o funtor e a aridade de um predicado complexo.

O predicado functor/3

- O predicado functor/3 informa o funtor e a aridade de um predicado complexo
  - ?- functor(amigos(luiz,ana),F,A).
    F = amigos,
    A = 2
    true

O predicado functor/3

- O predicado functor/3 informa o funtor e a aridade de um predicado complexo

?- functor(amigos(luiz,ana),F,A).
F = amigos,
A = 2
true

- ?- functor([luiz,ana,vitória],F,A).
F = .
A = 2
true

functor/3 e constantes

- O que acontece quando usamos functor/3 com constantes?

functor/3 e constantes

- O que acontece quando usamos functor/3 com constantes?
  - ?- functor(maria,F,A).
    F = maria,
    A = 0
    true

functor/3 e constantes

- O que acontece quando usamos functor/3 com constantes?
  - ?- functor(maria,F,A).
    F = maria,
    A = 0
    true
  - ?- functor(14,F,A).
    F = 14
    A = 0
    true

functor/3 para a construção de termos

- Pode-se também usar functor/3 para construir termos:
  - ?- functor(Termo,amigos,2).
    Termo = amigos(_G316, _G317)
    true

Verificação para termos complexos

```
termoComplexo(X):-
    nonvar(X),
    functor(X,_,A),
    A > 0.
```

Argumentos: arg/3

- Prolog também fornece o predicado arg/3
- Este predicado nos informa sobre os argumentos de termos complexos.
- Ele possui três argumentos:
  - Um número *N*
  - Um termo complexo *T*
  - O *N-ésimo* argumento de *T*

Argumentos: arg/3

- Prolog também fornece o predicado arg/3
- Este predicado nos informa sobre os argumentos de termos complexos.
- Ele possui três argumentos:
  - Um número *N*
  - Um termo complexo *T*
  - O *N-ésimo* argumento de *T*

```
?- arg(2,gosta(luiz,ana),A).
A = ana
```

Strings

- Strings são representadas em Prolog por uma lista de códigos de caracteres
- Prolog oferece aspas para facilitar a escrita de strings

```
?- S = “Viviane”.
S = [86, 105, 118, 105, 97, 110, 101].
```

Trabalhando com strings

- Existem muitos predicados padrões para trabalhar com strings.
- Um particularmente útil é atom_codes/2

```
?- atom_codes(viviane,S).
S = [118, 105, 118, 105, 97, 110, 101].
```

Operadores

- Como vimos, em certos casos, Prolog nos permite usar notações de operadores que são mais amigáveis.
- Relembre-se, por exemplo, das expressões aritméticas tais como 2+2 que internamente significa +(2,2)
- Prolog também possui um mecanismo para permitir que se adicione operadores definidos pelo programador.

Propriedades dos operadores

- Operadores infixos
  - Funtores escritos entre seus argumentos
  - Exemplos: + - = == , ; . -->
- Operadors prefixos
  - Funtores escritos antes de seus argumentos
  - Exemplo: - (para representar números negativos)
- Operadores pósfixos
  - Funtores escritos após seus argumentos
  - Exemplo: ++ na linguagem de programação C

Precedência

- Todo operador possui uma certa precedência para resolver expressões ambíguas.
- Por exemplo, 2+3*3 significa 2+(3*3), ou (2+3)*3?
- Devido ao fato de que a precedência de + é maior do que a de *, Prolog escolhe + para ser o funtor principal de 2+3*3

Associatividade

- Prolog usa a associatividade para desambiguisar os operadores com o mesmo valor de precedência.
- Exemplo: 2+3+4
  Isto significa (2+3)+4 ou 2+(3+4)?
  - Associativo à esquerda
  - Associativo à direita
- Operadores podem também ser definidos como não-associativos. Neste caso, você é forçado a usar parênteses em casos ambíguos
  - Exemplos em Prolog: :- -->

Definindo operadores

- Prolog permite que você defina seus próprios operadores
- As definições de operadores são da seguinte forma:

```
:- op(Precedência, Tipo, Nome).
```

  - Precedência:
    número entre 0 e 1200
  - Tipo: o tipo do operador

Tipos de operadores em Prolog

- yfx associativo à esquerda, infixo
- xfy associativo à direita, infixo
- xfx não associativo, infixo
- fx não associativo, prefixo
- fy associativo à direit, prefixo
- xf não associativo, pósfixo
- yf associativo à esquerda, pósfixo

Operadores em SWI Prolog

1200	xfx	`-->, :-`
1200	fx	`:-, ?-`
1150	fx	`dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile`
1100	xfy	`;, |`
1050	xfy	`->, op*->`
1000	xfy	`,`
954	xfy	`\`
900	fy	`\+`
900	fx	`~`
700	xfx	`<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, is`
600	xfy	`:`
500	yfx	`+, -, /\, \/, xor`
500	fx	`+, -, ?, \`
400	yfx	`*, /, //, rdiv, <<, >>, mod, rem`
200	xfx	`**`
200	xfy	`^`

Próxima aula

- Cortes e negação
  - Como controlar o comportamento de retrocesso do Prolog com a ajuda do predicado de corte.
  - Explicar como o corte pode ser empacotado em uma forma mais estruturada chamada de *negação como falha*.