PICAT e seus Tipos de Dados

Claudio Cesar de Sá
claudio.sa@udesc.br

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

9 de fevereiro de 2017

Objetivos desta Vídeo-Aula – 02

- Contexto dos Tipos de Dados (TD)
- Tipos de Dados
- Usando funções sobres estes TDs
- Referências
- Estes slides e outros:
  `https://github.com/claudiosa/CCS/tree/master/picat/slides_picat`
- **Pré-requisitos: aula 01 de PICAT, noções de lógica, LPs $\Rightarrow$ muitos vídeos e bons!**
- Os exemplos aqui apresentados foram executados diretamente na console interativa do PICAT.

Sumário

Contexto dos Tipos de Dados

- Tipos de dados $\neq$ estruturas de dados
- Lembrar que: predicados apresentam valores V (`yes`) ou F (`no`) e funções retornam valores
- *Funções* em PICAT são análogas as funções das LPs clássicas
- *Predicados* análogo a LPO, a Prolog e seus derivados

Tipos de Dados

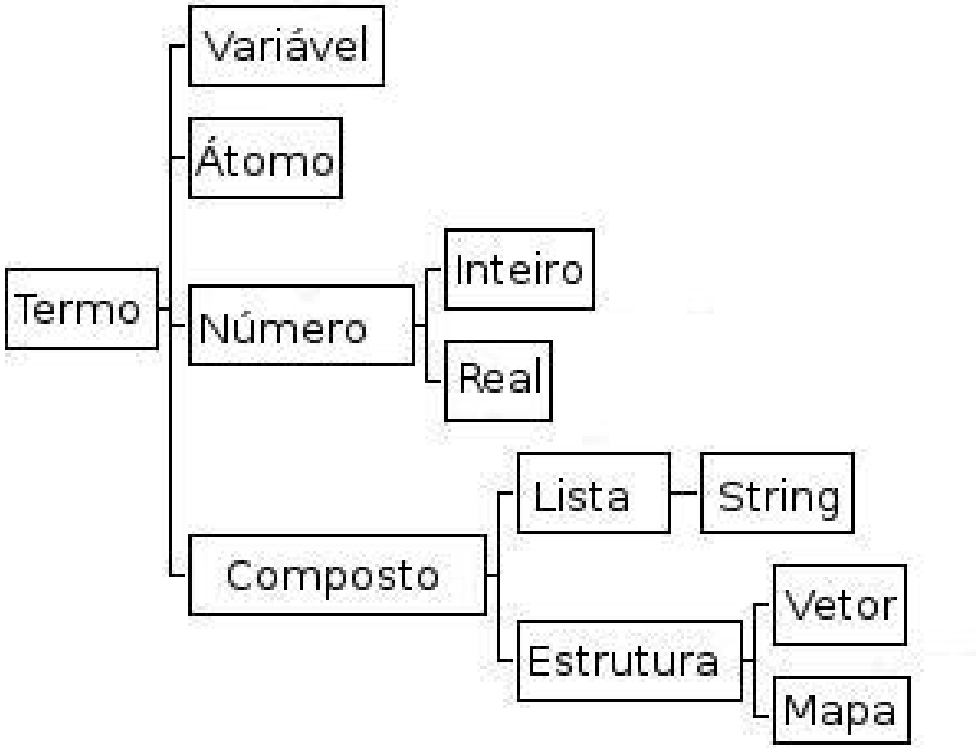

Figura: Hierarquia dos tipos de dados = termos

Variável

- Em PICAT começam por letras MAIÚSCULAS. Ex: Velocidade, TEMPO, etc
- Como na matemática, armazenam valores, outras variáveis, estruturas complexas, etc
- Diferente das outras LPs: não possuem endereço de memória fixo
- A variável está instanciada (*bound*) ou está livre (*free*)
- Uma vez instanciada, permanece com um determinado valor na chamada corrente

Exemplo de Variável (1)

- `X = 34, println(xzinho = X).`
- `X = 34, Y = 34, Z := X + Y.`
- `X = 34, println(xzinho = X), X := 17, println(xzinho = X).`
- Mas `X = 34, X = 17, println(xzinho = X).`
  logo `X = 34` é diferente de `X := 34`
- Assim, cuidar em PICAT no caso de:
  - `=` é o operador de unificação ou casamento de variáveis livres
  - `:=` é a atribuição das LPs clássicas
  - `==` é a comparação entre dois termos
- Predicado útil: `bind_vars({X,Y,Z}, 56.789 )`
  X = 56.789000000000001
  Y = 56.789000000000001
  Z = 56.789000000000001
  yes

Exemplo de Variável (2)

- Igualmente: `X = 234.56, copy_term(X) = Y`
  X = 234.560000000000002
  Y = 234.560000000000002
  yes
  ⇒ Caso X se encontre unificado (venha com um casamento de padrão), e se deseje alguma modificação a partir de X. Então realiza-se uma cópia do mesmo para uma variável temporária Y, e modifica-se Y.
- Pode-se utilizar também o `bind_vars`:
  X = 321.01, bind_vars({Y}, X), Y:= Y + 321.
  X = 321.009999999999991
  Y = 642.009999999999991
  yes

Exemplo de Variável (3)

- Outros predicados úteis: `var`, `nonvar` (retorna *yes* se a variável não estiver livre). Exemplo:
  ```
  ( X = 7, nonvar(X) ) , var(Y)
  X = 7
  ```
  yes – nos dois casos eram *true*
- Uma variável atribuída tem um *mapa* com um par de valores ligados a ela: o seu conteúdo(s) e estado (*true/false*).
- Ver manual alguns predicados específicos para este fim!

Atribuição

- `X := 7, X := X + 7, X := X + 7.`
  `X = 21`
- A atribuição tem um escopo local ao predicado em questão!
- Enfim, cuidar do que se deseja modificar e retornar!

Átomo

- Um átomo é uma constante simbólica
- Seu nome pode ser representado tanto com aspas simples ou sem
- Tamanho de um átomo $\leq$ 1000 caracteres
- Exemplos: `x_20` , `'x_21'` , `'a'` , `a` , `abacate`, etc
- Mas `'ab'== ab` são iguais

Exemplo de Átomos

- `atom('x')` , `atom(x)` cuidar com `atom('x') == atom(x)`
- `atom_chars('x') = X`
- `chr(68) = Valor`
- `ord('D') = Valor` – inverso da anterior
- `digit(1)` $\equiv$ *no* e `digit('1')` $\equiv$ *yes*
- `length(udesc) = X`
- `len(udesc) = X`

Números

- Um número é um átomo inteiro ou real
- Um número inteiro pode ser representado na forma decimal, binária, octal ou hexadecimal
- Um úmero real usa o ponto no lugar da vírgula para separar os valores depois de zero como: `3.1415`

Exemplo de Números Reais e Inteiros

- `X = 3 , number(X).`
- `X = 3 , Y = 4 , X < Y.`
- `number_chars(45) = X`

Exemplo de Números Reais e Inteiros

- `X = 3 , number(X).`
- `X = 3 , Y = 4 , X < Y.`
- `number_chars(45) = X`
  `X = ['4','5']`
- `number_codes(45) = X`

Exemplo de Números Reais e Inteiros

- `X = 3 , number(X).`
- `X = 3 , Y = 4 , X < Y.`
- `number_chars(45) = X`
  `X = ['4','5']`
- `number_codes(45) = X`
  `X = [52,53]`
- `real(5.4321)`
- `int(321)` $\equiv$ `integer(321)` são predicados!

Tipos Compostos (1)

Lista: sequência de termos

- `L = [ a, b, c] , length(L) = X`
- `L = [ a, b, c] , L.length = X`
- `L = [ a, b, c] , get(L,length) = X`
- Em breve uma aula sobre construir funções e predicados sobre listas
- Há uma quantidade de funções e predicados sobre listas embutidos (prontos para uso)

Strings: uma lista de carácteres

```
X = "Oi bom dia!"
X = ['O',i,' ',b,o,m,' ',d,i,a,!]
```

- `X = "Oi bom dia!", to_uppercase(X) = Y`
- Predicado: `string(X)`

Tipos Compostos (2)

Estrutura: um modo de organizar dados heterogêneos em um único termo.

- Uma estrutura tem o formato $\$\texttt{est}(t_1, t_2, ..., t_n)$, onde `est` é um átomo e n é a aridade da estrutura.
- O \$ é usado para diferenciar uma estrutura de uma função em um dos argumentos do predicado (sim, uma função pode ser um argumento de um predicado)
- Cuidados nos casamentos dos termos. Veja o exemplo:

Execute: struct_ex01.pi

Tipos Compostos (3)

```
%=============================================================%
main =>
    X1 = $carro($marca(fiat, palio), $cor(azul), 2007),
    X2 = $carro($marca(toyota, ethios), $cor(prata), 2017),
    X3 = $carro($marca(honda, fit), $cor(branco), 2017),
    L = [X1, X2, X3],
    println(dado_completo = X1),
    println(aridade = arity(X1)),
    println(nome_da_estrutura = name(X1)),
    %% println(todos_os_dados = L),
    %% BUSCA DOS CARROS NOVOS
    foreach (X in L)
            novo_17(X)
    end.


novo_17( carro( marca(X, W), Y, Ano) ) ?=>
     Ano >= 2017,
```

Tipos Compostos (4)

```
19      printf("\n Marca: %w || Modelo: %w || Cor: %w", X, W, Y),
20      printf("\n EH UM CARRO NOVO >= 2017").
21
22  novo_17( carro( marca(X, W), cor(Y), Ano) ) =>
23      Ano < 2017,
24      printf("\n Marca: %w || Modelo: %w || Cor: %w",  X, W, Y),
25      printf("\n NAO EH UM CARRO NOVO, ANO: %w", Ano).
26
27  %=============================================================%
```

Tipos Compostos (5)

Vetores:

- Um vetor ou *array* tem o formato $\{t_1, ..., t_n\}$, o qual é um caso especial de uma estrutura delimitada por '{ }' e aridade n
- Tem seu comprimento delimitado na memória e tempo de acesso constante a seus elementos
- Análogo aos vetores de outras linguagens com uma notação e funções bem fáceis de usar. Exemplo `Vetor[7]` acessa a 7a. posição deste vetor unidimensional
- Para criar um array: `new_array(`$D_1, ..., D_n$`) = Vetor` onde $D_1, \ldots, D_n$ especificam as dimensões do mesmo. Atualmente, $n \leq 10$ (matrizes de 10 dimensões $\Rightarrow$ mais do que suficiente!)
- Como sua implementação tem origem das listas, é de se esperar que: *Listas* $\Leftrightarrow$ *Vetores*

Tipos Compostos (6)

- Logo, há muitas funções e predicados de listas que facilitam o tratamento com vetores
- Mas, algumas estão prontas apenas para vetores unidimensionais. Cuidado aqui. Veja o exemplo para superar estas dificuldades.

Execute: array_example01.pi

```
1  %=============================================================%
2  import os.
3  import util.
4  import math.
5
6  main ?=> Status = command("clear") ,
7   printf("======================================= %d OK", Status),
8   Matriz = f_Array_2D(), % funcao sem argumentos "()" obrigado
9   printf("\n======================================= \n"),
10  printf("\n Soma dos elementos: %d\n", f_soma_2D( Matriz )),
```

Tipos Compostos (7)

```
11  print_matriz(Matriz),
12  printf("\n================================== \n")
13  .
14 main =>  printf("\n Algo errado nas chamadas acima !!!").
15
16 %%-------------------------------------------------------
17 f_Array_2D() = Vetor =>
18          new_array(3,2) = Vetor,
19          Vetor = { {3,4} , {5,6} , {7,8} },
20          printf("\n Primeira linha: %w", first(Vetor) ),
21          printf("\n Ultima linha: %w",  last(Vetor) ),
22          printf("\n Total de linhas: %i",  length(Vetor) ).
23
24 %%-------------------------------------------------------
25 f_soma_2D( M ) = Soma =>
26    L = M.length,  %% Num. de linhas
27    Soma := 0,
28    foreach(I in 1  .. L)
```

Tipos Compostos (8)

```
29      Soma := Soma + sum( M[I] ) %% sum: APENAS PARA VETOR 1D
30    end.
31 %%---------------------------------------------------------
32 %% Imprimindo uma Matriz
33 print_matriz( M ) =>
34  L = M.length,  %% Num. de linhas
35  C = M[1].length,  %% Num. de colunas
36   nl,
37    foreach(I in 1  .. L)
38      foreach(J in 1  ..  C)
39       printf("%w " , M[I,J] )
40      end,
41      nl
42    end.
43 %=========================================================%
```

Mapas: (em breve)

Conjuntos: (em breve)

Conclusão

- Os principais tipos de dados foram apresentados
- Apresentamos o uso de predicados e funções destes TDs, e extensões. Exemplo: uso da função `sum-1D` para `sum-2D`
- Próximo vídeo: laços, predicados e funções

Referências

- O *User Guide* que está no diretório `doc/` da instalação em LaTeX
- Em `http://picat-lang.org/` – *User Guide on-line* está lá também
- Meu GitHub ⇒ `https://github.com/claudiosa/CCS/tree/master/picat`
- Assinem o fórum do PICAT(em inglês: respondo lá também)
- Sítio do Hakan Kjellerstrand ⇒ `http://www.hakank.org/picat/`
- Sítio do Roman Barták ⇒ `http://ktiml.mff.cuni.cz/~bartak/`
- Sítio do Sergii Dimychenko ⇒ `http://sdymchenko.com/blog/2015/01/31/ai-planning-picat/`

Agradecimentos

- Marlon Henry Schweigert
- Paulo Victor Aguiar
- Rogério Eduardo da Silva

Obrigado

π

Retornem os comentários para o próximo vídeo!!!