PICAT: uma Linguagem Multiparadigma

Claudio Cesar de Sá, Rogério Eduardo da Silva, Alexandre Gonçalves,
João Herique Faes Battisti, Paulo Victor de Aguiar

joaobattisti@gmail.com

pavaguiar@gmail.com

claudio.sa@udesc.br

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

Sumário (1)

Agradecimentos

-
- Ao Google Images ... vários autores

Histórico

- Criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman
- Utiliza o B-Prolog como base de implementação, e ambas utilizam a programação em lógica: Lógica de Primeira-Ordem (LPO)
- Uma evolução ao Prolog após seus mais de 40 anos de sucesso!
- Sua atual versão é a 2.0 (30 de maio de 2017)

O que é multiparadigma?

- Imperativo – Procedural
- Funcional
- Lógico
- Uma boa *mistura* de: Haskell, Prolog e Python

Algumas Características:

- Sintaxe $\Rightarrow$ elegância do código
- Velocidade de execução
- Portabilidade (todas plataformas)
- Extensão há outras ferramentas

Anacrônico de **P.I.C.A.T.**

P: *Pattern-matching*: Utiliza o conceito de *casamento padrão* equivalente aos conceitos de *unificação* da LPO

I: *Intuitive*: oferece estruturas de decisão, atribuição e laços de repetição, etc, análogo as outras linguagens de programação

C: *Constraints*: suporta a Programação por Restrições (PR)

A: *Actors*: suporte as chamadas a eventos, os atores (futuro gráfico)

T: *Tabling*: implementa a técnica de *memoization*, com soluções imediatas para problemas de Programação Dinâmica (PD).

Instalação do PICAT

- Baixar a versão desejada de `http://picat-lang.org/download.html`
- Descompactar. Em geral em **/usr/local/Picat/**
- Criar um link simbólico (linux) ou atalhos (Windows):
  `ln -s /usr/local/Picat/picat /usr/bin/picat`
- Se quiser adicionar (opcional) uma variável de ambiente:
  `PICATPATH=/usr/local/Picat/`
  `export PICATPATH`
- ou ainda adicione o caminho: `PATH=$PATH:/usr/local/Picat`
- Finalmente, tenha um editor de código de programa.
  Sugestão: *geany* ou *sublime*
- Escolha a sintaxe da linguagem *Erlang*

Usando do Picat

- Picat é uma linguagem de multiplataforma, disponível em qualquer arquitetura de processamento e também de sistema operacional. Nesta vídeo-aula: Linux (Manjaro)
- Em seus arquivos fontes utiliza a extensão **.pi**. Exemplo: `programa.pi`
- Existem 2 modos de utilização do Picat: modo linha de comando (ou console) e modo interativo
- Códigos executáveis 100% **stand-alone**: ainda não!
- Neste quesito, estamos em igualdade com Java, Prolog e Python

Fatos e Regras – os pais!

- *pai*(*platao*, *luna*) leia-se: *Platão é o pai de Luna*
- *pai*(*platao*, *pricles*) leia-se: *Platão é o pai de Péricles*
- *pai*(*epimenides*, *platao*) leia-se: *Sócrates é o pai de Platão*
- Codificando tudo isto em Picat

Regras em PICAT (1)

```
1  %%% FATOS ...  desenhe a arvore geneologica
2  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
3  index(-,-)     %% definindo FATOS
4    %%pai(PAI, FILHO)
5          %pai(platao, luna)    .
6          pai(platao, pericles).
7          pai(platao, eratostenes).
8          pai(epimenides, platao).
9          pai(bartolomeu, epimenides).
10 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
11 %% REGRAS: exemplos
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13 %% definindo um avo: pai do pai
14 avo(X,Y) =>  pai(X,Z), pai(Z,Y).
15
16 %% definindo um irmao: alguem que tenha o mesmo pai
17 irmao(X,Y) => pai(Z,X), %%% 1o a ser avaliado
18                pai(Z,Y), %%% 2o a ser avaliado
19                %%X !== Y.
20                !==(X , Y). %%% 3o a ser avaliado
21
```

Regras em PICAT (2)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% MAIS REGRAS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

listar_pais  ?=>    %%% ?=>  regra "backtrackavel"
    pai(X,Y) , %% and
    printf("\n ==> %w  e pai de  %w", X , Y)  ,
    false  .

listar_pais =>
    printf("\n ")  ,
    true. %% the final rule of above

listar_ant  ?=>    %%% ?=>  regra "backtrackavel"
    antepassado(X,Y) ,
    printf("\n ==> %w  e ANTEPASSADO de  %w", X , Y)  ,
    false.

listar_ant =>
    printf("\n ")  ,
    true. %% the final rule of above
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Regras em PICAT (3)

```
44 %% main ... facilidade no uso console
45 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
46 main ?=>    %%% ?=>  regra "backtrackavel"
47  %    listar_pais,
48      listar_ant,
49  %    avo(X,Y), printf("\n ==> %w  eh avo de  %w", X , Y)   ,
50  %    irmao(Z,W), printf(" \n ==> %w  eh irmao de  %w", Z , W),
51      false.
52 main => true.
53 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
54 /*
55 1. Todo x que eh pai de um y implica em x ser um antepassado
56 de y
57 QxQy (pai(x,y) --> antepassado(x,y))
58
59 2. Todo x que eh pai de um z, e z
60 eh um  antepassado de y, entao x eh antepassado de y
61
62 QxQyQz (pai(x,z) and antepassado(z,y) --> antepassado(x,y))
63
64 EM PICAT -- clausulas de HORN (sempre apenas uma conclusao)
65 */
```

Regras em PICAT (4)

```
66 antepassado(X,Y) ?=> pai(X,Y). %%%%%% ?=>  regra "backtrackavel"
67 antepassado(X,Y) => pai(X,Z),
68                     antepassado(Z,Y).
```

Tipos de Dados

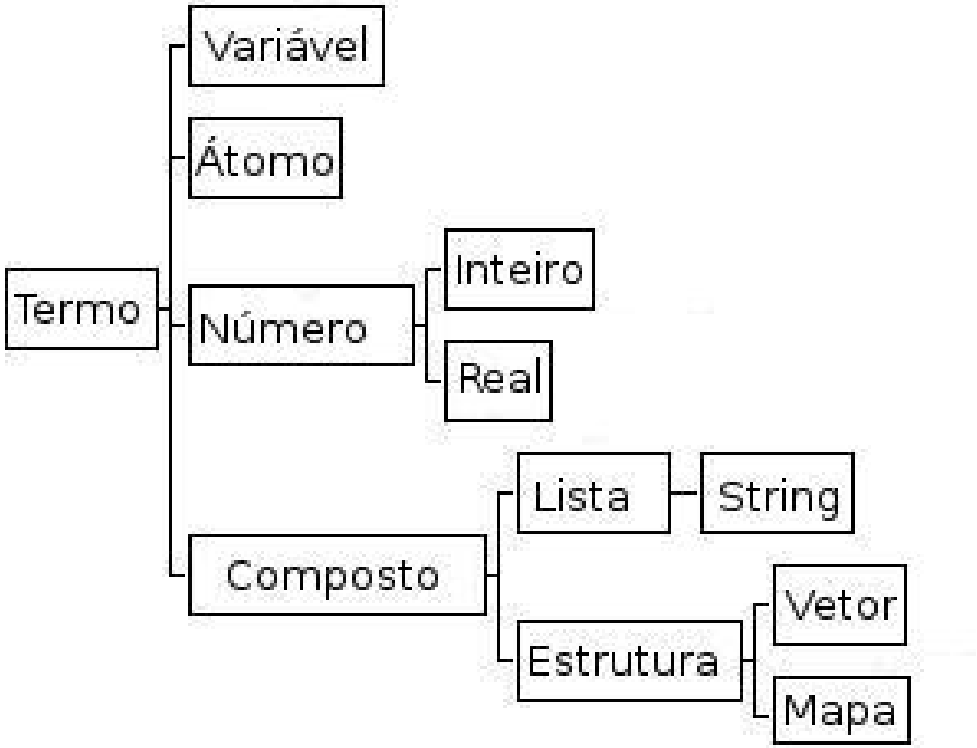

Figura: Hierarquia dos Tipos de dados

Número

```
Picat> A = 5, B = 7, number(A), number(B), max(A, B) =
Maximo, min(A, B) = Minimo.
A = 5
B = 7
Maximo = 7
Minimo = 5
yes.
```

Atribuição

```
Picat> X := 7, X := X + 7, X := X + 7.
X = 21
```

Estruturas de Controle

```
1 teste =>
2       X := 3,
3       Y := 4,
4       if (X >= Y)
5       then
6             printf("\n X eh maior: %d\n" , X)
7       else
8             printf("\n senao Y eh maior: %d\n" , Y)
9       end.
```

Entradas e Saídas

```
1 main =>
2             printf("\n Digite dois numeros:  "),
3             N_real_01 = read_real(),
4             N_real_02 = read_real(),
5             Media = (N_real_01 + N_real_02) / 2,
6             printf(" A media eh: %6.4f ", Media ),
7             printf("\n ..... FIM ..... \n ").
```

Conclusão

- PICAT é uma linguagem nova (2013), desconhecida, revolucionária e com um futuro promissor
- Atualmente há pouco material disponível e uma comunidade pequena de usuários
- Uso muito bom quanto a: Planejamento, Programação por Restrição e PD (diretamente)
- Todos problemas NPs-Completos!

Referências

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

Obrigado

π

Retornem os comentários para o próximo vídeo!!!

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

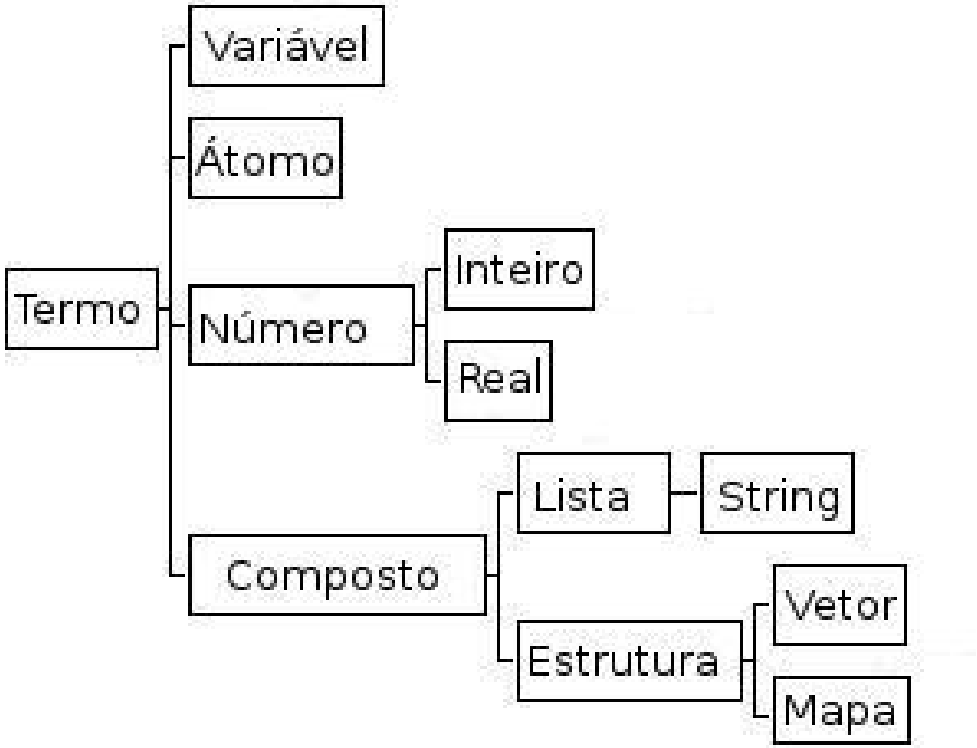

Figura: Hierarquia dos tipos de dados = termos

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

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_example01.pi

Tipos Compostos (3)

```
1  %=========================================================%
2  main =>
3     X1 = $carro($marca(fiat, palio), $cor(azul), 2007),
4     X2 = $carro($marca(toyota, ethios), $cor(prata), 2017),
5     X3 = $carro($marca(honda, fit), $cor(branco), 2017),
6     L = [X1, X2, X3],
7     println(dado_completo = X1),
8     println(aridade = arity(X1)),
9     println(nome_da_estrutura = name(X1)),
10    %% println(todos_os_dados = L),
11    %% BUSCA DOS CARROS NOVOS
12    foreach (X in L)
13            novo_17(X)
14    end.
15
16
17 novo_17( carro( marca(X, W), Y, Ano) ) ?=>
18     Ano >= 2017,
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) ) =>
```

Tipos Compostos (4)

```
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

Tipos Compostos (7)

```
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 )),
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) ).
```

Tipos Compostos (8)

```
23
24 %%-----------------------------------------------------------
25 f_soma_2D( M ) = Soma =>
26    Linhas = M.length,  %% Num. de linhas
27    Soma := 0,
28    foreach(I in 1  .. Linhas)
29      Soma := Soma + sum( M[I] ) %% sum: APENAS PARA VETOR 1D
30    end.
31 %%-----------------------------------------------------------
32 %% Imprimindo uma Matriz
33 print_matriz( M ) =>
34  Linhas = M.length,  %% Num. de linhas
35  Colunas = M[1].length,  %% Num. de colunas
36   nl,
37    foreach(I in 1  .. Linhas)
38      foreach(J in 1  ..  Colunas)
39       printf("%w " , M[I,J] )
40      end,
41      nl
42    end.
43 %============================================================%
```

Tipos Compostos (9)

Mapas: (em breve)

Conjuntos: (em breve)

Conclusões

- Os principais tipos de dados foram apresentados

Conclusões

- 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`

Conclusões

- 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/`