PICAT: Uma Linguagem de Programação Multiparadigma

Claudio Cesar de Sá

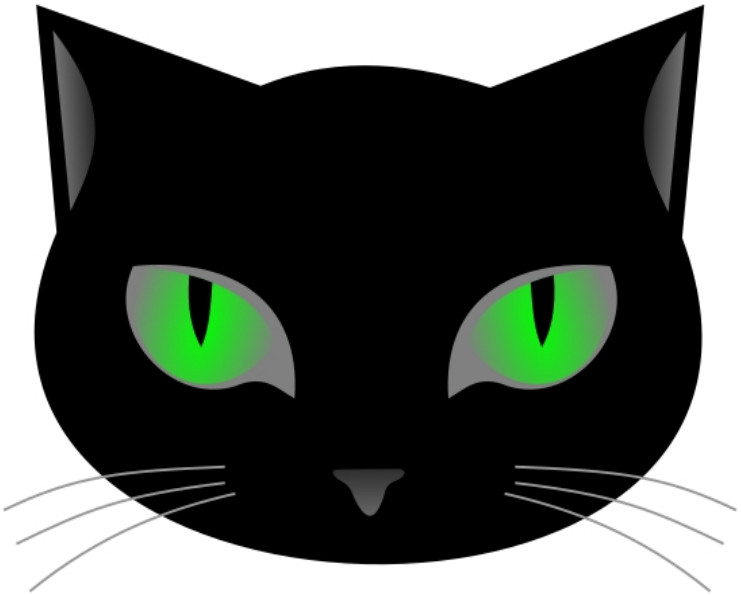
claudio.sa@udesc.br

Departamento de Ciência da Computação – DCC
Centro de Ciências e Tecnológias – CCT
Universidade do Estado de Santa Catarina – UDESC

6 de maio de 2019

Listas

- Definição de listas
- Representação
- Operadores
- Exemplos



- Requisito: conceito de recursividade, *aterramento* etc, dominados!

- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais

- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (cada nó sempre tem duas ramificações)

- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (cada nó sempre tem duas ramificações)
- Lembrando que uma estrutura binária de árvore tem uma equivalência com uma árvore n-ária (ver livro de Estrutura de Dados)

- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (cada nó sempre tem duas ramificações)
- Lembrando que uma estrutura binária de árvore tem uma equivalência com uma árvore n-ária (ver livro de Estrutura de Dados)
- Logo, listas são estruturas flexíveis e poderosas!

Ilustrando uma Lista em Formato Binário

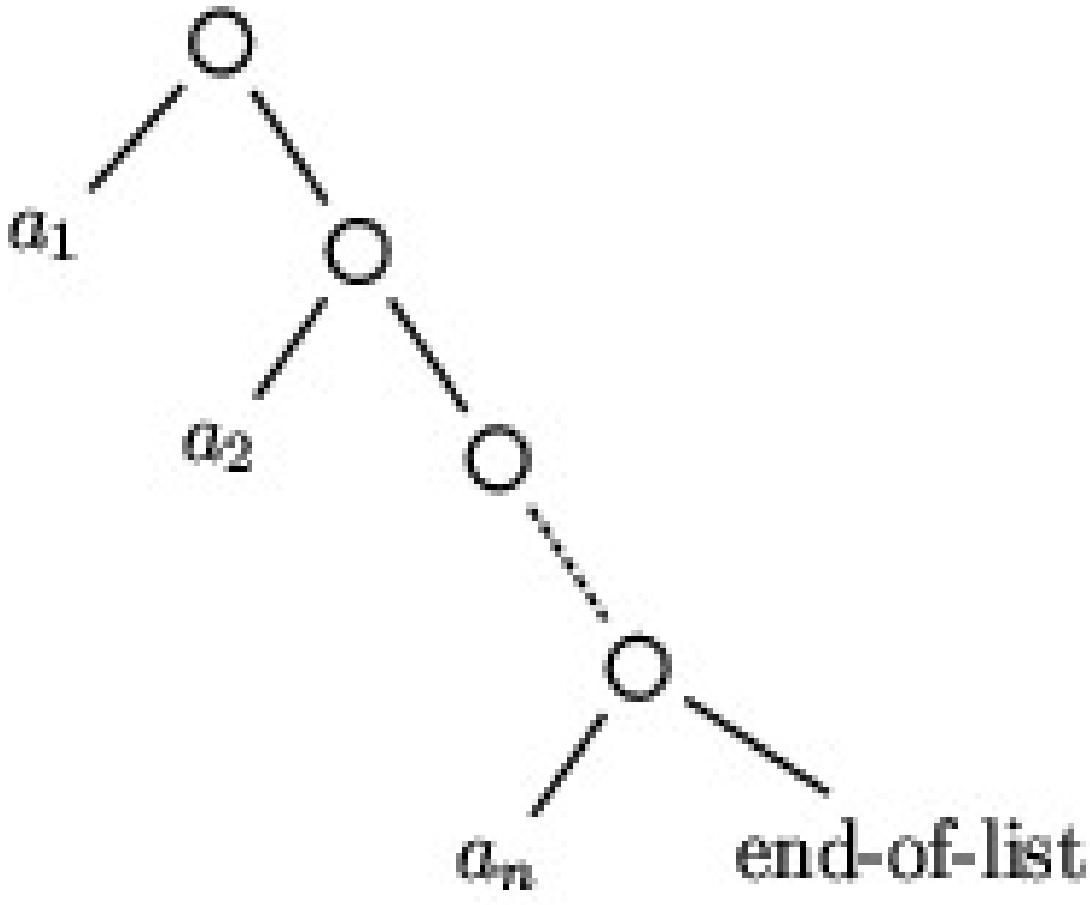


Figura 1: Uma estrutura Lista – Homogênea

Ilustrando Listas e o Operador '|' (ou ':' da figura)

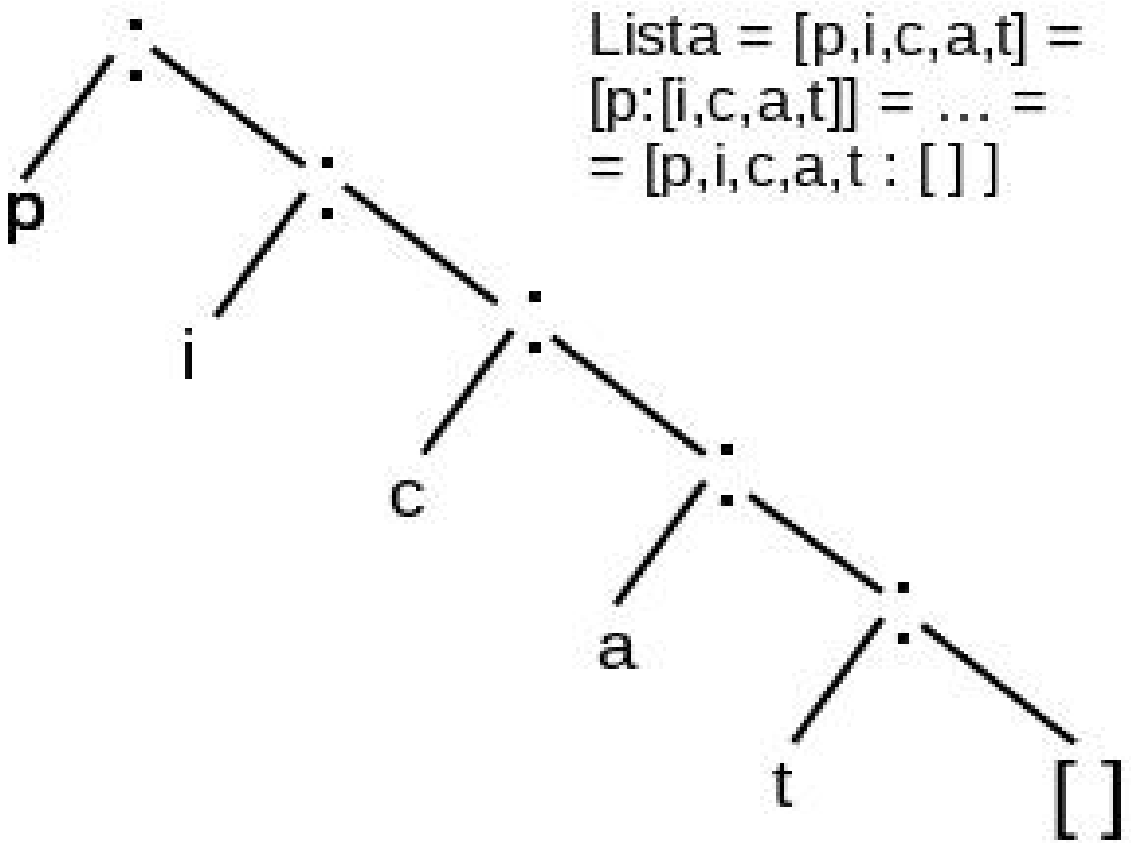


Figura 2: Listas são inerentemente **recursivas**!

Exemplos de Listas

lista:	`[a,b,c,d]`
cabeça:	`a`
cauda:	`[b,c,d]`
lista:	`[[a,b],c,[d,e,f],g]`
cabeça:	`[a,b]`
cauda:	`[c,[d,e,f],g]`
lista:	`[[A11,A12],[A21,A22]]`
cabeça:	`[A11,A12]`
cauda:	`[[A21,A22]]`

Definições iniciais (e recursivas)

- Uma lista é uma sequência de termos (objetos)

Notação:

- O símbolo "[" é usado para descrever o início de uma lista, e "]" para o final da mesma;
- Exemplo: seja a lista [a, b, c, d], logo um predicado cujo argumento seja algumas letras, tem-se uma lista do tipo:
  - letras([a, b, c, d])
  - Onde 'a' é o *cabeça* (primeiro elemento) da lista
  - e [b, c, d]é uma *sub-lista* que é uma lista!
- Os elementos de uma lista são lidos da esquerda para direita;
- A "*sub-lista*" [b, c, d]é conhecida como resto ou "*cauda*" da lista;
- Esta sub-lista é uma lista e toda definição segue-se recursivamente.

Operador "|":

- "*Como vamos distinguir de onde se encontra a cabeça da cauda da lista?*"
- Com as listas novos símbolos foram introduzidos, isto é, além dos delimitadores [...], há um novo operador que **separa** ou **define** quem é a elemento cabeça da lista e cauda.
- Este operador é conhecido como "*pipe*" (ou *barra vertical*), simbolizado por "|", que separa o lado esquerdo da direita da lista.
- Esta separação é necessário para se realizar os *casamentos de padrões* nas linguagens lógicas.

Exemplos de *casamentos*:

```
[ a, b, c, d ] = X
[ X | b, c, d ]  =  [ a, b, c, d ]
[ a | b, c, d ]  =  [ a, b, c, d ]
[ a , b | c, d ]  =  [ a, b, c, d ]
[ a , b , c | d ]  =  [ a, b, c, d ]
[ a , b , c , d | [] ]  =  [ a, b, c, d ]
[] = X
[ [ a | b , c , d] ] = [ [ a , b , c , d ] ]
[  a | b , c , [ d ] ] = [  a , b , c , [ d ] ]
[  _ | b , c , [d ] ] = [  a , b , c , [ d ] ]
[  a | Y ] = [  a , b , c ,  d ]
[  a | _ ] = [  a , b , c ,  d ]
[  a , b | c , d ] = [  X , Y | Z ]
```

Contra-exemplos de ***casamentos***:

```
[ a , b | [c, d] ]  !=  [ a, b, c, d ]
[ [ a , b , c , d] ]  !=  [ a, b, c, d ]
[  a , b , [ c ] , d, e ]  !=  [ a, b, c, d, e ]
[ [ [ a ] | b , c , d] ] != [ [ a , b , c , d] ]
```

Sintaxe das Listas VI

- Estes casamentos de termos de uma lista são também conhecidos por *matching*
- Devido ao fato de listas modelarem qualquer estrutura de dados, invariavelmente, seu uso é extensivo há problemas em geral (dos simples a complexos)
- Porém, alguns cuidados no uso de predicados com *backtracking*. Acompanhe os exemplos.
- Os próximos exemplos encontram-se no arquivo: `../picat/listas.pi`

Exemplo: encontrar o comprimento de uma lista I

- O comprimento de uma lista é o comprimento de sua **sub-lista**, mais **um**
- O comprimento de uma lista vazia ([]) é zero.

Em Picat, sob uma visão funcional, este enunciado é escrito por:

```
comprimento_02( [ ] ) = 0.
comprimento_02([ _ | L ]) = N  =>
                N = 1 + comprimento_02( L ).
```

Exemplo: encontrar o comprimento de uma lista II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
comprimento_01([],N) ?=> N = 0.
%%% em PROLOG, apenas comprimento_01([],0). PORQUÊ?
comprimento_01([_|L],N)  =>
                comprimento_01( L , Parcial ),
                N = 1 + Parcial.
```

Exemplo: encontrar o comprimento de uma lista III

Um *mapa de memória* é dado por:

	Regra	X	T	N	N = N+1
compto([a,b,c,d],N)	#2	a	[b,c,d]	3 →	3+1=4
compto([b,c,d],N)	#2	b	[c,d]	2 →	↖ 2+1
compto([c,d],N)	#2	c	[d]	1 →	↖ 1+1
compto([d],N)	#2	d	[]	0 →	↖ 0+1
compto([],N)	#1	–	–	–	↖ 0

Exemplo: verificar a pertinência de um objeto na lista I

- Verifica se um dado objeto pertence há uma lista
- Um método clássico – muito usado
- Tem embutido no Picat: o *menber*

Em Picat, sob uma visão funcional, esta função é escrita por:

```
pertence_02( _ , [ ]) = false.
pertence_02( A, [A|_]) = true.
% CUIDAR ... em funcoes nao hah ? em ?=> ...
% sem backtracking em funcoes
pertence_02(A, [B|L]) = X =>
                A != B,
                X = pertence_02(A,L).
```

Exemplo: verificar a pertinência de um objeto na lista II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
pertence_01( A, [A|_]) ?=> true.
% Again, backtracking CONTROLADO ... diferente do Prolog
pertence_01(A,[B|L])  =>
                A != B,
                pertence_01(A,L).
```

Exemplo: adicionar um elemento em uma lista I

- Um objeto é adicionado no início da lista (sem repetição) caso este já esteja contido na lista, a lista original é a retornada:

Em Picat, sob uma visão funcional, esta função é escrita por:

```
add_X_lista_02(X, [ ]) = [X].
add_X_lista_02(X, Y) = Z =>
          pertence_02(X, Y) = true,
          Z = Y;
          Z = [ X | Y ].
```

Exemplo: adicionar um elemento em uma lista II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
add_X_lista_01(X, [ ], Z )  ?=> Z = [X].
add_X_lista_01(X, Y, Z) ?=>
          pertence_01(X, Y),
          Z = Y.
add_X_lista_01(X, Y, Z) =>
          Z = [ X | Y ].
```

Exemplo: união de duas listas I

- O método de união ou concatenação entre duas listas, resultando em uma terceira lista
- Este predicado é conhecido como *append* ou *concatena*. O *append* está pronto na biblioteca default do Picat
- Há uma versão simplificada: `L3 = L1 ++ L2`

Em Picat, sob uma visão funcional, esta função é escrita por:

```
uniao_02( [], X ) = X.
uniao_02( [X|L1], L2 ) = L3 =>
                              L3 = [X | uniao_02( L1, L2 )].
```

Exemplo: união de duas listas II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
uniao_01( [] , X, Y ) ?=> Y = X.
uniao_01( A , L2, R  ) =>  A = [X|L1] ,
                           R = [X|L3] ,
                           uniao_01( L1, L2, L3 ).
```

Geração de Listas – *list comprehension*

- O conceito de *list comprehension* veio da programação funcional
- Basicamente serve para criarmos ou gerarmos listas

Geração de Listas – *list comprehension*

- O conceito de *list comprehension* veio da programação funcional
- Basicamente serve para criarmos ou gerarmos listas
- Bastante útil e pode ser usada em qualquer parte de um código

Geração de Listas – *list comprehension*

Um *list comprehension* tem o seguinte formato na criação de listas:

$$[T \ : \ E_1 \ \texttt{in} \ D_1, \ Cond_1, \ \ldots, \ E_n \ \texttt{in} \ D_n, \ Cond_n]$$

- T é uma termo (uma expressão num caso genérico)
- E_i é um padrão de iteração
- D_i é uma expressão de um valor composto, em geral um intervalo de domínio
- Opcionalmente, condições $Cond_1, \ldots, Cond_n$ são chamados de *termos*
- Esta geração de lista tem a seguinte interpretação: *toda tupla de valores* $E_1 \in D_1, \ldots, E_n \in D_n$, *se as condições* $Cond_i$ *forem verdades, então o valor do termo* T *é adicionado na lista em construção*

Um vetor ou matrizes também pode ser construídos com um *array comprehension* e tem o seguinte formato:

$$\{T \ : \quad E_1 \ \texttt{in} \ D_1, \ Cond_1, \ \ldots, \ E_n \ \texttt{in} \ D_n, \ Cond_n\}$$

Geração de Listas – *list comprehension*

Um vetor ou matrizes também pode ser construídos com um *array comprehension* e tem o seguinte formato:

$$\{T \texttt{ : } E_1 \texttt{ in } D_1\texttt{, } Cond_1\texttt{, ..., } E_n \texttt{ in } D_n\texttt{, } Cond_n\}$$

Isto é o mesmo como

$$\texttt{to\_array([}T \texttt{ : } E_1 \texttt{ in } D_1\texttt{, } Cond_1\texttt{, ..., } E_n \texttt{ in } D_n\texttt{, } Cond_n\texttt{])}$$

Exemplos de *list comprehension*

```
main => Status = command("clear") ,
printf("==================================== %d", Status),
    L0 = [I :  I in 10..20],
    L1 = [I :  I in 10..2..20],
    L2 = [I :  I in 1..20, I>10, I<20],
    L3 = [(A,I) : A in [a,b], I in 1..10, I mod 2 == 0],
    L4 = [(I,J,K) :  I in 1..2, J in 3..7, K in 1..10, I+J < K],
    printf("\n L0 : %w " , L0),
    printf("\n L1 : %w " , L1),
    printf("\n L2 : %w " , L2),
    printf("\n L3 : %w " , L3),
    printf("\n L4 : %w " , L4),
    printf("\n FIM\n").

%  $ picat geracao_listas.pi
```

Saída: *list comprehension*

```
======================================== 0
 L0 : [10,11,12,13,14,15,16,17,18,19,20]
 L1 : [10,12,14,16,18,20]
 L2 : [11,12,13,14,15,16,17,18,19]
 L3 : [(a,2),(a,4),(a,6),(a,8),(a,10),(b,2),(b,4),(b,6),(b,8),(b,10)]
 L4 : [(1,3,5),(1,3,6),(1,3,7),(1,3,8),(1,3,9),(1,3,10),(1,4,6),
 (1,4,7),(1,4,8),(1,4,9),(1,4,10),(1,5,7),(1,5,8),(1,5,9),(1,5,10),(1,6
 (1,6,9),(1,6,10),(1,7,9),(1,7,10),(2,3,6),(2,3,7),(2,3,8),(2,3,9),(2,3
 (2,4,7),(2,4,8),(2,4,9),(2,4,10),(2,5,8),(2,5,9),(2,5,10),
 (2,6,9),(2,6,10),(2,7,10)]
 FIM
```

L4 ... *cortada*

Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat

Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos

Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a metodologia de se *programar em lógica*

Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a metodologia de se *programar em lógica*
- Exercitar-se para aprender os detalhes!

Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a metodologia de se *programar em lógica*
- Exercitar-se para aprender os detalhes!
- Usar as listas como estrutura base em problemas complexos

Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a metodologia de se *programar em lógica*
- Exercitar-se para aprender os detalhes!
- Usar as listas como estrutura base em problemas complexos
- Próxima na aula: buscas (**uso extensivo de listas**)