

PICAT: Uma Linguagem de Programação Multiparadigma

, Claudio Cesar de Sá, Miguel Alfredo Nunes, Jeferson L. R. Souza

`miguel.nunes@edu.udesc.br`

`jeferson.souza@udesc.br`

`claudio.sa@udesc.br`

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

23 de abril de 2019

- Alexandre Gonçalves;
- João Henrique Faes Battisti;
- Paulo Victor de Aguiar;
- Rogério Eduardo da Silva;
- Hakan Kjellerstrand – (<http://www.hakank.org/picat/>)
- Neng-Fa Zhou – (<http://www.picat-lang.org/>)
- Outros anônimos que auxiliaram na produção deste documento;

- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: estática e dinâmica

- Em projetos de linguagens de programação há dois tipos de verificação do tipo de dados: estática e dinâmica
- A verificação de tipos dados estática em *tempo de compilação*.

- Em projetos de linguagens de programação há dois tipos de verificação do tipo de dados: estática e dinâmica
- A verificação de tipos dados estática em *tempo de compilação*.
- Enquanto a dinâmica em *tempo de execução*.

- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: estática e dinâmica
- A verificação de tipos dados estática em *tempo de compilação*.
- Enquanto a dinâmica em *tempo de execução*.
- Linguagens fortemente tipadas, tais como C, Java e Pascal, exigem que o tipo do dado (conteúdo) seja do mesmo tipo da variável ao qual este valor será atribuído. Tudo isto é pré-definido durante a fase da *compilação*.

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa
- Prós e contras sobre a melhor tipagem de dados, esta é a discussão que fica de lado neste momento

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa
- Prós e contras sobre a melhor tipagem de dados, esta é a discussão que fica de lado neste momento
- Em resumo Picat, tem a tipagem **dinâmica**

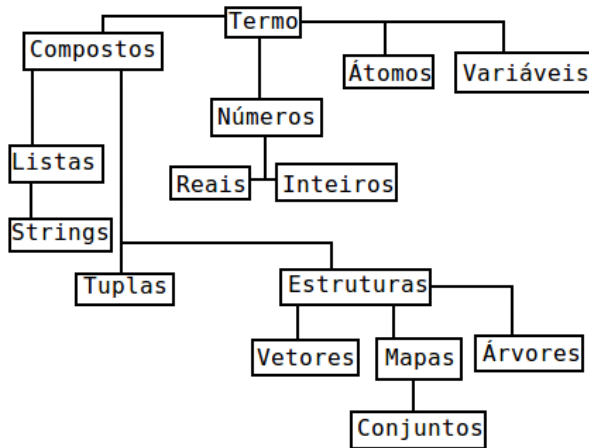


Figura 1: Hierarquia dos Tipos de Dados

- Em Picat, variáveis e valores são *genericamente* chamados de *termos*

- Em Picat, variáveis e valores são *genericamente* chamados de *termos*
- Os valores são subdivididos em duas categorias, números e valores compostos
- Os números, por suas vez, podem ser inteiros ou reais, e valores compostos podem ser listas e estruturas

- Átomos são constantes simbólicas, podendo ser delimitados ou não, por aspas simples.
- Carácteres são representados por átomos de comprimento 1.
- Átomos não delimitados por aspas simples, **nunca** começam com uma letra maiúscula, nem número ou *underscore*.

- Átomos são constantes simbólicas, podendo ser delimitados ou não, por aspas simples.
- Carácteres são representados por átomos de comprimento 1.
- Átomos não delimitados por aspas simples, **nunca** começam com uma letra maiúscula, nem número ou *underscore*.

Exemplos

x x_1 ' _ ' '\\ ' 'a\'b\n' '_ab' '\$%'

Números se dividem em:

- **Inteiro:** Inteiros podem ser representados por números binários, octais, decimais ou hexadecimais.

Exemplos

12_345	12345 em notação decimal, usando _ como separador
0b100	4 em notação binária
0o73	59 em notação octal
0xf7	247 em notação hexadecimal

O **underscore** é ignorado pelo compilador e o interpretador.

- **Real:** Números reais são compostos por um parte inteira, um ponto, seguido por uma fração decimal, ou um expoente.
- Se existe uma parte inteira em um número real então ela deve ser seguida por uma fração ou um expoente. Isso é necessário para distinguir um número real de um número inteiro.

Exemplos

12.345 0.123 12-e10 0.12E10

- Termos compostos podem conter mais de um valor ao mesmo tempo.

- Termos compostos podem conter mais de um valor ao mesmo tempo.
- Termos compostos são acessados pela notação de índice, começando a partir de 1 e indo até N , onde N é o tamanho deste termo.

- Termos compostos podem conter mais de um valor ao mesmo tempo.
- Termos compostos são acessados pela notação de índice, começando a partir de 1 e indo até N , onde N é o tamanho deste termo.
- Se dividem em Listas e Estruturas.

Listas são agrupamentos de valores quaisquer sem ordem e sem tamanho pré-definido. Seu tamanho não é armazenado na memória, sendo necessário recalculá-lo sempre que necessário seu uso. Listas são encapsuladas por colchetes.

Exemplos

[1,2,3,4,5] [a,b,32,1.5,aaac] ["string",14,22]

Há uma seção dedicada a esta poderosa estrutura de dados!

Strings são listas especiais que contém somente caracteres. *Strings* podem ser inicializadas como uma sequência de caracteres encapsulados por aspas duplas, ou como uma sequência de caracteres dentro colchetes separados por vírgulas.

Exemplos

```
"Hello" "World!" "\n" [o,l,a," ",m,u,n,d,o]
```

- Tuplas é um conjunto de termos não-ordenados, podendo ser acessados por notação de índice assim como listas.
- Tuplas são estáticas, ou seja, os termos contidos em uma tupla não podem ser alterados, assim como não podem ser adicionados ou removidos termos de tuplas.
- Tuplas são encapsuladas por parênteses e seus termos são separados por vírgulas.

Exemplos

(1,2,3,4,5) (a,b,32,1.5,aaac) ("string",14,22)

Em geral, usamos as tuplas dentro de listas.

Estruturas são termos especiais que podem ser definidos pelo usuário. Estruturas tomam a seguinte forma:

$$\text{\$}s(t_1, \dots, t_n)$$

Onde '*s*' é um átomo que denomina a estrutura, cada '*t_i*' é um de seus termos, e '*n*' é a aridade ou tamanho da estrutura.

Exemplo

`\$ponto(1,2) \$pessoa(jose, "123.456.789.00", "1.234.567")`

Estruturas são termos especiais que podem ser definidos pelo usuário. Estruturas tomam a seguinte forma:

$$\$s(t_1, \dots, t_n)$$

Onde ' s ' é um átomo que denomina a estrutura, cada ' t_i ' é um de seus termos, e ' n ' é a aridade ou tamanho da estrutura.

Exemplo

`$ponto(1,2)` `$pessoa(jose, "123.456.789.00", "1.234.567")`

Temos 4 outras estruturas que não usam o símbolo \$, são elas:

Vetores ou *arrays* são estruturas especiais do tipo:

$$\{t_1, \dots, t_n\}$$

Vetores ou *arrays* são estruturas especiais do tipo:

$$\{t_1, \dots, t_n\}$$

- Vetor é um conjunto ordenado de tamanho n , delimitado por ' $\{\}$ '.
- Vetores tem comportamentos análogo às listas, tanto é que quase todas as funções de listas são sobrecarregadas para vetores.
- A diferença entre vetores e listas é que vetores tem um tamanho constante.
- Vetores são muito práticos quando se manipula matrizes na entrada

Exemplos

$\{1,2,3,4,5\}$ $\{a,b,32,1.5,aaac\}$ $\{"string",14,22\}$

- **Mapas** são estruturas especiais que são conjuntos de relações do tipo chave-valor.
- **Conjuntos** são sub-tipos de mapas onde todas as chaves estão relacionadas com o átomo `not_a_value`.
- *Heaps* são árvores binárias completas representadas como vetores. Árvores podem ser do tipo *máximo*, onde o maior valor está na raiz, ou *mínimo*, onde o menor valor está na raiz.

- Picat é uma linguagem de Tipagem Dinâmica, ou seja, o tipo de uma variável é validado durante a execução do programa
- Isto é, quando uma variável é criada, seu tipo não é instanciado
- Variáveis são análogas as da matemática, são símbolos que *seguram* ou representam um valor
- Ao contrário de variáveis em linguagens imperativas, variáveis em Picat não são endereços simbólicos de locais na memória
- Uma variável é dita *livre* (*free*) se não contém nenhum valor, e dita *instanciada* (*bound*) se ela contém um valor
- Uma vez que uma variável é instanciada, ela permanece com este valor na execução atual
- Por isso, diz-se que variáveis em Picat são de *atribuição única*

- O nome de variáveis devem sempre ser iniciado com letras **maiúsculas** ou com o carácter ***underscore*** (**_**), porém;
 - Variáveis cujo nome é unicamente um caractere **_** são chamadas de *variáveis anônimas*.
 - As *variáveis anônimas* podem receber qualquer valor não os guardam durante a execução do programa;
 - Num mesmo programa, podem existir diversas variáveis anônimas, instanciadas durante a execução do mesmo

Há dois modos de definir valores às variáveis:

Há dois modos de definir valores às variáveis:

- Unificação usa o operador '='
- Atribuição usa o operador ':='

- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor

- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.

- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.
- Uma instanciação é indefinida até que se encontre um valor que possa ser unificada a uma variável.

- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.
- Uma instanciação é indefinida até que se encontre um valor que possa ser unificada a uma variável.
- Termos são ditos unificáveis se são idênticos ou podem ser tornados idênticos instanciando variáveis nos termos.

Exemplo

```
Picat> X = 1
X = 1
Picat> $f(a,b) = $f(a,b)
yes
Picat> [H|T] = [a,b,c]
H = a
T = [b,c]
Picat> $f(X,b) = $f(a,Y)
X = a
Y = b
Picat> bind_vars({X,Y,Z},a)
Picat> X = $f(X)
```

Cuidar neste último caso, há um laço infinito nesta chamada!

- A atribuição simula a atribuição em linguagens imperativas

- A atribuição simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa

- A atribuição simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa
- O escopo da atribuição da variável é local e volátil

- A atribuição simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa
- O escopo da atribuição da variável é local e volátil
- Na unificação, uma nova variável temporária é criada afim de substituir um valor atribuído ou outra variável.

Exemplo

teste $\Rightarrow X = 0, X := X + 1, X := X + 2, \text{write}(X).$

- Neste exemplo X é unificado a 0.
- Em seguida, há uma atribuição X a $X + 1$, porém X já foi unificado a um termo.
- Então, outras operações devem ser feitas para que esta atribuição seja possível.
- Nesse caso, o compilador cria uma variável temporária, $X1$ por exemplo, e unifica com $X + 1$. Cada vez que X for instanciado, o compilador/programa atualiza em $X1$.
- O mesmo ocorre na atribuição $X1 := X1 + 2$, neste caso uma outra variável temporária é criada, por exemplo $X2$, e o processo se repete.

Exemplo

Portanto, estas atribuições sucessivas são compiladas como:

Exemplo

Portanto, estas atribuições sucessivas são compiladas como:

```
test => X = 0, X1 = X + 1, X2 = X1 + 2, write(X2).
```

Exemplos de Variáveis Válidas

X1	_	_ab
X	A	Variavel
_invalido	_correto	_aa

⇒ Relembrando, um nome de variável é válido se começa com letra **maiúscula** ou **_**

Exemplos de Variáveis Inválidas

1_Var	variável	valida
23	"correto	'termo
!numero	\$valor	#comum

Tabela 1: Operadores Aritméticos em Ordem de Precedência

$X ** Y$	Potenciação
$X * Y$	Multiplicação
X / Y	Divisão, resulta em um real
$X // Y$	Divisão de Inteiros, resulta em um inteiro
$X \text{ mod } Y$	Resto da Divisão
$X + Y$	Adição
$X - Y$	Subtração
<i>Inicio .. Passo .. Fim</i>	Uma série (lista) de números com um passo
<i>Inicio .. Fim</i>	Uma série (lista) de números com passo 1

Tabela 2: Tabela de Operadores Completa em Ordem de Precedência

Ops Aritméticos	Ver Tabela ??
++	Concatenação de Listas/Vetores
= :=	Unificação e Atribuição
== =:=	Equivalência e Equivalência Numérica
!= !===	Não Unificável e Diferença
< =< <=	Menor que
> >=	Maior que
in	Contido em
not	Negação Lógica
, &&	Conjunção Lógica
;	Disjunção Lógica

Operadores de Termos Não-Compostos

- **Equivalência(==)**: compara se dois termos são iguais.
No caso de termos compostos, eles são ditos equivalentes se todos os termos contidos em si são equivalentes. O compilador considera termos de tipos diferentes como totalmente diferentes, portanto a comparação $1.0 == 1$ seria avaliada como falsa, mesmo que os valores sejam iguais. Nesses casos, usa-se a *Equivalência Numérica*.

Operadores de Termos Não-Compostos

- **Equivalência(==)**: compara se dois termos são iguais.
No caso de termos compostos, eles são ditos equivalentes se todos os termos contidos em si são equivalentes. O compilador considera termos de tipos diferentes como totalmente diferentes, portanto a comparação $1.0 == 1$ seria avaliada como falsa, mesmo que os valores sejam iguais. Nesses casos, usa-se a *Equivalência Numérica*.
- **Equivalência Numérica(==)**: Compara se dois números são o mesmo valor. Deve ser usada com termos que sejam números.

Operadores de Termos Não-Compostos

- **Diferença(!=)**: compara se dois termos são diferentes, isto é, a negação da equivalência.

Operadores de Termos Não-Compostos

- **Diferença(\neq):** compara se dois termos são diferentes, isto é, a negação da equivalência.
- **Não-Unificável(\neq):** Verifica se dois termos não são unificáveis. Termos são ditos unificáveis se são idênticos ou podem ser tornados idênticos instanciando variáveis destes termos.

Exemplos

① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- ② $1.0 == 1$
no

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no
- ③ $1.0 ::= 1$, $1.2 ::= 1$

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- ② $1.0 == 1$
no
- ③ $1.0 ::= 1$, $1.2 ::= 1$
yes, no

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- ② $1.0 == 1$
no
- ③ $1.0 == 1$, $1.2 == 1$
yes, no
- ④ $1.0 != 1$, $Var3 != Var4$

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- ② $1.0 == 1$
no
- ③ $1.0 := 1$, $1.2 := 1$
yes, no
- ④ $1.0 != 1$, $Var3 != Var4$
yes, Depende dos valores (padrão yes)

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- ② $1.0 == 1$
no
- ③ $1.0 := 1$, $1.2 := 1$
yes, no
- ④ $1.0 != 1$, $Var3 != Var4$
yes, Depende dos valores (padrão yes)
- ⑤ $1.0 != 1$, $aa != bb$, $Var1 != Var5$

Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- ② $1.0 == 1$
no
- ③ $1.0 == 1$, $1.2 == 1$
yes, no
- ④ $1.0 != 1$, $Var3 != Var4$
yes, Depende dos valores (padrão yes)
- ⑤ $1.0 != 1$, $aa != bb$, $Var1 != Var5$
yes, yes, no

- **Concatenação (++)**: concatena duas listas ou vetores. O termo da esquerda é a primeira parte lista e a segundo a parte final da lista resultante.

- **Concatenação** ($++$): concatena duas listas ou vetores. O termo da esquerda é a primeira parte lista e a segundo a parte final da lista resultante.
- **Separador** ($H \mid T$): separa uma lista L em seu primeiro termo H , chamado de *cabeça* (em inglês *Head*), e o resto da lista T , chamado de *cauda* (em inglês *Tail*).

Na seção de listas este assunto é retomado

- **Iterador** (X in L): itera X no termo composto L , instanciando um termo não-composto X aos termos contidos em L .

- **Iterador** (X in L): itera X no termo composto L , instanciando um termo não-composto X aos termos contidos em L .
- **Sequência** ($\text{Inicio}..\text{Passo}..\text{Fim}$): gera uma lista ou vetor, começando (inclusivamente) em *Inicio* incrementando por *Passo* e parando (inclusivamente) em *Fim*. Se *Passo* for omitido, este é automaticamente atribuído 1.

Na seção de listas este assunto é retomado

Operadores de Termos Compostos – Exemplos

① $[1, 2, 3] ++ [4, 5, 6], \quad [] ++ [1, 2, 3], \quad [] ++ []$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], \quad [] ++ [1, 2, 3], \quad [] ++ []$
 $[1, 2, 3, 4, 5, 6], \quad [1, 2, 3], \quad []$
- ② $L = [1, 2, 3], \quad [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- ④ $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$

- 1 $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- 2 $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- 3 *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- 4 $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- ④ $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
 $Y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- ④ $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
 $Y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$
 $Z = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$

- Picat: tipagem dinâmica. O erro entre variáveis e dados, ocorre no meio da execução do código

- Picat: tipagem dinâmica. O erro entre variáveis e dados, ocorre no meio da execução do código
- As estruturas de dados compostos são as clássicas: listas, vetores, mapas, conjuntos, etc

- Picat: tipagem dinâmica. O erro entre variáveis e dados, ocorre no meio da execução do código
- As estruturas de dados compostos são as clássicas: listas, vetores, mapas, conjuntos, etc
- O conteúdo desta parte do curso pode ser complementado com a **Videoaula 02: Tipos de Dados do PICAT**
<https://www.youtube.com/watch?v=7fPKPd0ZDnc>