PICAT: Uma Linguagem Multiparadigma

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

miguel.nunes@edu.udesc.br

jeferson.souza@udesc.br

claudio.sa@udesc.br

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

28 de março de 2019

Contribuições

- Alexandre Gonçalves;
- João Herique Faes Battisti;
- Paulo Victor de Aguiar;
- Rogério Eduardo da Silva;
- Outros anônimos que auxiliaram na produção deste documento;

Apresentação ao Curso de PICAT – I

- O que é o PICAT?

Apresentação ao Curso de PICAT – I

- O que é o PICAT?
  - Uma linguagem de programação de propósitos gerais
  - Uma evolução do PROLOG (consagrada linguagem dos primórdios da IA)
  - Tem elementos de Python, Prolog e Haskell
- Uso e finalidades do PICAT:

Apresentação ao Curso de PICAT – I

- O que é o PICAT?
  - Uma linguagem de programação de propósitos gerais
  - Uma evolução do PROLOG (consagrada linguagem dos primórdios da IA)
  - Tem elementos de Python, Prolog e Haskell
- Uso e finalidades do PICAT:
  - Uso de programas gerais, simples a complexos
  - Provê suporte há vários solvers na área de Pesquisa Operacional
  - Área: IA, programação por restrições, programação inteira, planejamento, combinatória, etc

Apresentação ao Curso de PICAT – II

- Este curso é dirigido a voce?

Apresentação ao Curso de PICAT – II

- Este curso é dirigido a voce?
- Requisitos:

Apresentação ao Curso de PICAT – II

- Este curso é dirigido a voce?
- Requisitos:
  - De conhecimento:
  - Dedicação
  -
- Requisitos computacionais:

Apresentação ao Curso de PICAT – II

- Este curso é dirigido a voce?
- Requisitos:
  - De conhecimento:
  - Dedicação
  -
- Requisitos computacionais: Um computador qualquer (arquitetura 16, 32 ou 64 bits), com Linux ou Windows, que tenha um compilador C instalado completo, preferencialmente.

Apresentação ao Curso de PICAT – III

- Comunidade e ações: `http://picat-lang.org`
- Que tópicos serão cobertos no curso?

Sumário I

Histórico

- Criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman;
- Utiliza o B-Prolog como base de implementação, e ambas utilizam a Lógica de Primeira-Ordem (LPO) como seu fundamento;
- Uma evolução ao Prolog após seus mais de 40 anos de sucesso!
- Sua atual versão é a 2.6 (28 de março de 2019).

Conhecendo PICAT

- Picat é uma linguagem que visa ser simples, mas ainda assim poderosa e multiuso;
- Por isso estão implementadas diversas características normalmente não associadas com linguagens lógicas;
- Isto torna Picat uma linguagem essencialmente multiparadigma, abrangendo partes de ambos os paradigmas declarativo e imperativo;
- Esta combinação de características declarativas e imperativas permite o desenvolvimento de softwares mais produtivos, mas que ainda possam ser altamente otimizados para tarefas específicas, ou softwares mais simples para tarefas mais mundanas;

O que é ser Multiparadigma ?

- Uma linguagem Multiparadigma é uma que contém características de vários paradigmas de programação.
- Picat abrange os seguintes paradigmas:
  - Lógico
  - Funcional
  - Procedural
- Uma boa *mistura* de: Haskell (Funcional) , Prolog (Lógica) e Python (Procedural e Funcional).

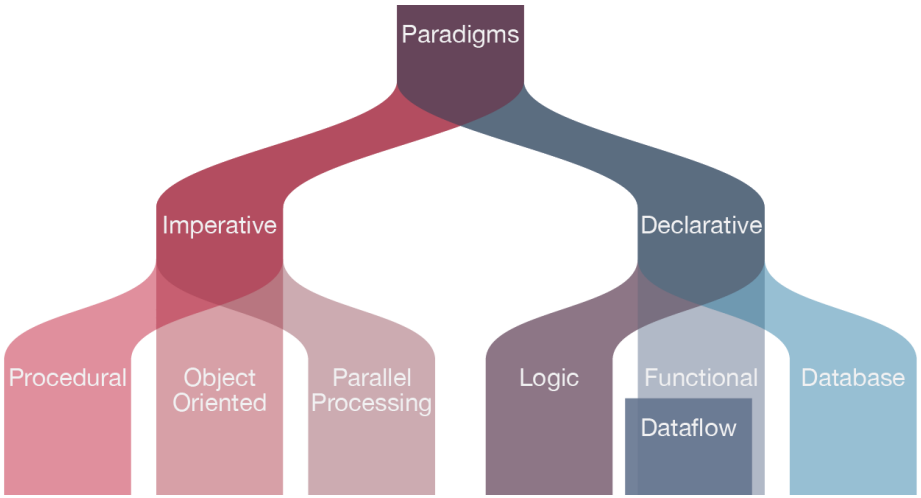


Figura 1: Fluxograma dos paradigmas de programação.

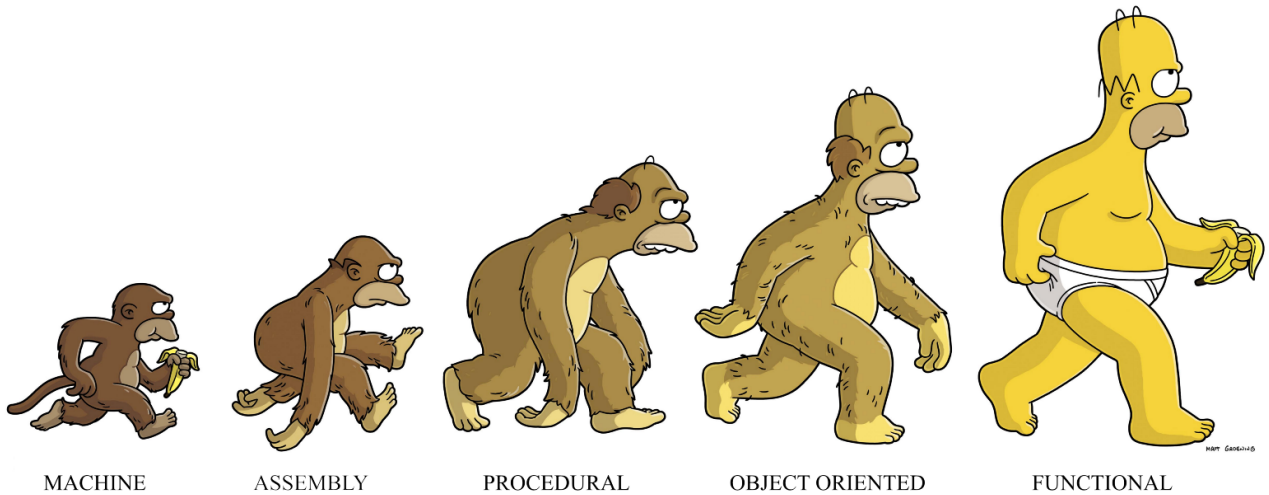
Paradigma Lógico

- Uma linguagem lógica é uma onde o programa é expresso como uma série de predicados lógicos, usadas para expressar fatos e regras sobre um dado domínio;
- Regras são escritas em formas de cláusulas, que são interpretadas como implicações lógicas;
- Este é o principal paradigma de Picat.

Paradigma Funcional

- Uma linguagem funcional é uma onde os elementos do programa podem ser avaliados e tratados como funções matemáticas;
- Um dos principais motivos de se usar linguagens funcionais é a previsibilidade e facilidade de entendimento do estado atual do programa;
- Isso anda lado a lado com a sintaxe simples e intuitiva de Picat, possibilitando que seja possível entender como um programa é estruturado e será executado com muita facilidade.

Figura 2: Comparação do paradigma funcional com outros paradigmas comuns



Paradigma Procedural

- Uma linguagem procedural é uma que pode ser subdividida em *procedimentos*, também chamados de rotinas, subrotinas ou funções;
- Em linguagens procedurais há um procedimento principal (geralmente chamado de *Main*) que controla o uso e a chamada de outros procedimentos, em Picat, o mesmo ocorre;
- Em Picat, cada premissa é tratada como um procedimento, que é resolvido por meio de métodos de inferência lógica;

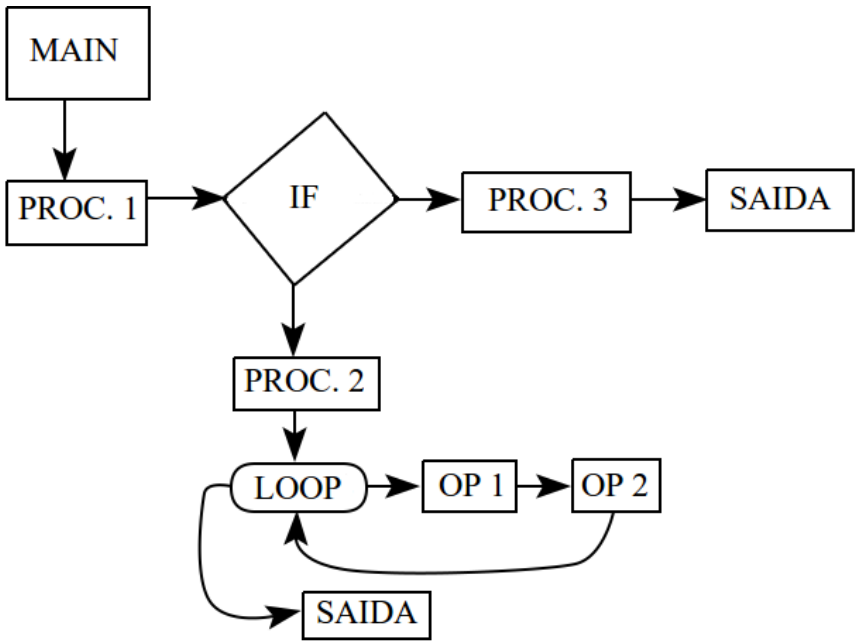


Figura 3: Fluxograma representando a estrutura de um programa Procedural

Algumas Características:

- Sintaxe elegante e simples, facilitando a leitura e entendimento do código;
- Alta velocidade de execução;
- Disponibilidade nos sistemas operacionais e arquiteturas mais importantes;
- *Queries* $\Rightarrow$ Semelhante a Python, podem ser feitas *queries* ou *consultas* ao terminal de Picat, tais consultas podem ser qualquer tipo de programa compilável pela linguagem, por menor que seja;
- Várias bibliotecas da própria linguagem disponíveis, assim como diversas ferramentas externas possibilitam grande extensibilidade à linguagem.

Acrônimo de **P.I.C.A.T.** I

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

I: *Intuitive*: Oferece estruturas de decisão, atribuição e laços de repetição, etc. Análogo a outras linguagens de programação mais populares;

C: *Constraints*: Suporta a programação por restrições (PR) para problemas combinatórios;

A: *Actors*: Suporte as chamadas a eventos, os atores;

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 texto apropriado.
  Sugestão: *Geany*, *Sublime* ou *Atom*.
- Se possível, escolha a sintaxe da linguagem *Erlang*.

Usando Picat

- Picat é uma linguagem de multiplataforma, disponível em qualquer arquitetura de processamento e também de sistema operacional;
- Os seus arquivos fontes utilizam a extensão **.pi**. Exemplo: `programa.pi`
- Há dois modos principais de utilização do Picat:
  - Modo interativo, onde seu código é digitado e compilado diretamente na linha de comando;
  - *Modo console* onde o console só é utilizado para compilar seus programas.
- Códigos executáveis 100% **stand-alone**: ainda não!
- Neste quesito, estamos em igualdade com Java, Prolog e Python

Introdução aos Tipos de Dados

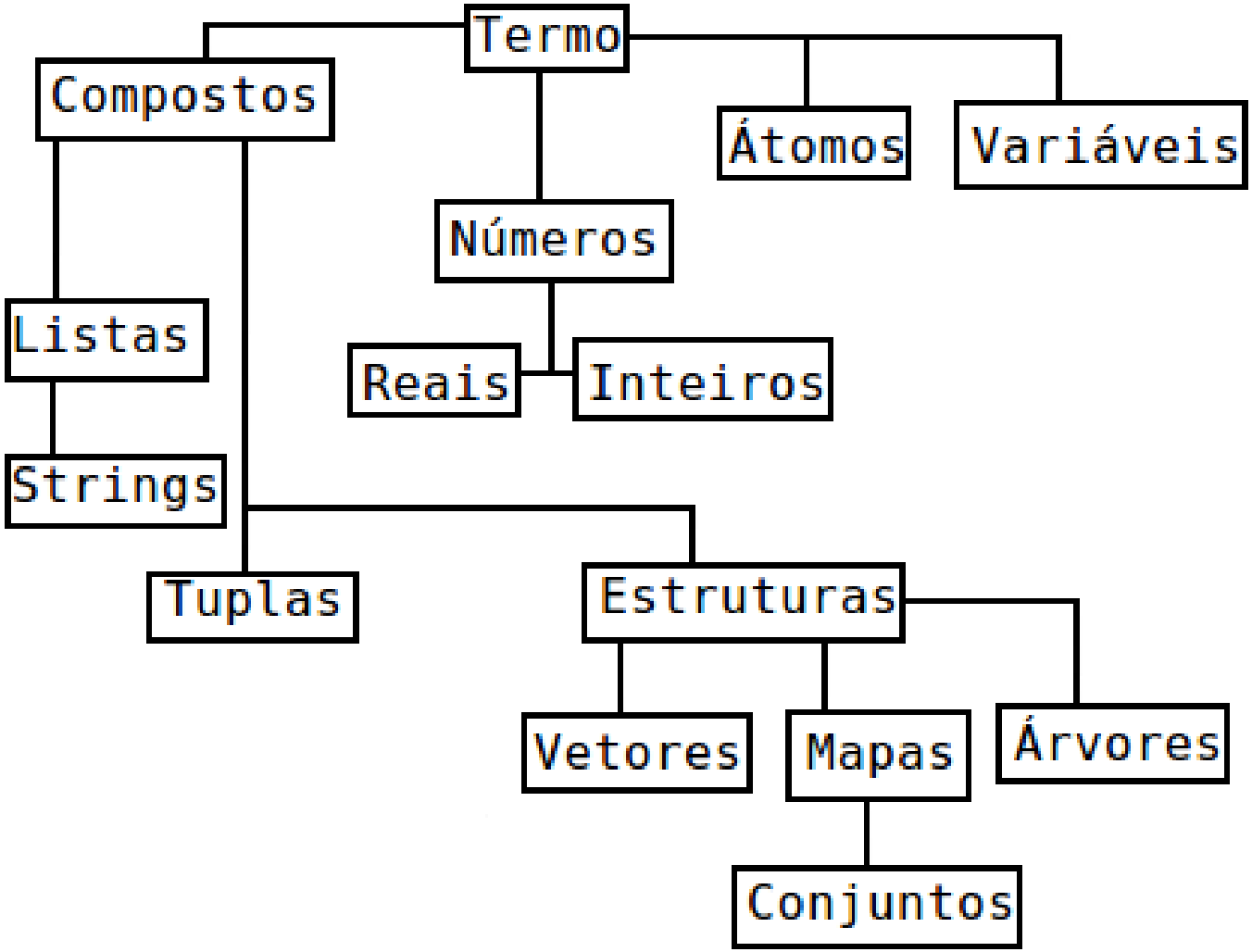


Figura 4: Hierarquia dos Tipos de Dados

Em Picat tanto variáveis quanto valores são considerados *termos*
Mais ainda, valores são subdivididos em duas categorias, números e valores compostos
Números, por suas vez, podem ser inteiros ou reais, e valores compostos podem ser listas ou estruturas

Átomos

Átomos são constantes simbólicas, podendo ser encapsulados, ou não, por aspas simples. Caracteres podem ser representados por átomos de comprimento 1. Átomos não encapsulados por aspas **nunca** começam com uma letra maiúscula, número ou underscore.

Exemplos

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

Números I

Números se dividem em:

- **Inteiro:** Inteiros podem ser representados por números binários, octais, decimais ou hexadecimais. Dígitos em um número podem ser separados por um **underscore**, porém essa separação é ignorada pelo compilador.

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

Números II

- **Real:** Números reais são compostos por um parte inteira opcional, uma fração decimal opcional, um ponto decimal e um expoente opcional.
- 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
```

Compostos

Termos compostos são termos que podem conter mais de um valor ao mesmo tempo. Termos compostos são acessados por 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

Listas são agrupamentos de valores quaisquer sem ordem e sem tamanho pré-definido. Seu tamanho não é armazenado na memória, sendo necessário recalcular 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]

Strings

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

Tuplas são conjuntos de termos não ordenados, podendo ser acessados por notação de índice assim como listas.
Tuplas são imutáveis, 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)

Estruturas (*Functores*)

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

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

Onde *"s"* é um átomo que nomeia 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")
```

Existem 4 estruturas especiais que não necessitam que seja usado o símbolo $, são eles

Vetores

Vetores, ou *arrays*, são estruturas especiais do tipo $\{t_1, \ldots, t_n\}$, cujo nome é simplesmente '{}' e tem aridade n.
Vetores tem comportamento praticamente idêntico à listas, tanto é que quase todas as funções de listas são sobrecarregadas para vetores. Uma importante diferença entre vetores e listas é que vetores tem seu tamanho armazenado na memória, ou seja, o tempo para se calcular o tamanho de um vetor é constante.

Exemplos

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

Mapas, Conjuntos e *Heaps*

- Mapas são estruturas especiais que são conjuntos de relações do tipo `chave-valor`.
- Conjuntos são sub-tipos de mapas onde todos 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 esta na raiz.

Introdução à Variáveis I

- Picat é uma linguagem de Tipagem Dinâmica, ou seja, o tipo de uma variável é checado somente durante a execução de um programa
- Por causa disso, quando uma variável é criada, seu tipo não é instanciado
- Variáveis em Picat, como variáveis na matemática, são símbolos que *seguram* ou representam um valor

Introdução à Variáveis II

- Ao contrário de variáveis em linguagens imperativas, variáveis em Picat não são endereços simbólicos de locais na memória
- Uma variável é dita *livre* se não contém nenhum valor, e dita *instanciada* 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*

Introdução à Variáveis III

- O nome de variáveis devem sempre ser iniciado com letras **maisculas** ou um caractere ***underscore (_)***, porém;
  - Variáveis cujo nome é unicamente um caractere _ são chamadas de *variáveis anônimas*, que são variáveis que podem ser instanciadas valores, mas não os retem durante a execução do programa;
  - Diversas variáveis anônimas podem ser instanciadas durante a execução de um programa, porém todas as ocorrências de variáveis anônimas são tratadas diferentemente.

Unficação e Atribuição

Há dois modos de definir valores a variáveis, a unificação, que usa o operador $=$, e a atribuição, que usa o operador $:=$

Unificação

- A **Unificação** é uma operação que instância uma variável a um termo ou padrão, substituindo toda a ocorrência dessa variável pelo valor a qual ela foi instanciada até que haja uma situação onde esta instanciação falhe, nesse momento a variável será reinstanciada e esse processo se repete.
- 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 instanciado 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)
```

A última consulta demonstra um caso do problema de ocorrência, onde o compilador de Picat não verifica se um termo ocorre dentro de um padrão. Isso cria um termo cíclico que não pode ser acessado.

Atribuição

- A **Atribuição** é uma operação cujo intuito é simular a atribuição em linguagens imperativas, permitindo que variáveis sejam re-atribuídas valores durante a execução do programa
- Para isso, durante a compilação do programa, toda vez que a operação de unificação é encontrada, uma nova variável temporária será criada que irá substituir a variável que seria atribuída.

Exemplo

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

Neste exemplo X é unificado a 0, então, o compilador tenta unificar X a $X + 1$, porém X já foi unificado a um valor, portanto outras operações devem ser feitas para que esta atribuição seja possível.
Nesse caso, o compilador irá criar uma variável temporária, $X1$ por exemplo, e à ela irá unir $X + 1$, depois toda vez que X for encontrado no programa o compilador irá substitui-lo por $X1$.
O mesmo ocorre na atribuição $X1 := X1 + 2$, neste caso uma outra variável temporária será criada, $X2$ por exemplo, e o processo será repetido.
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`	variavel	valida
23	`"correto`	`'termo`
`!numero`	`$valor`	`#comum`

Relembrando, um nome de variável é inválido se começa com números ou símbolos que não sejam _ ou letra minúscula

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 `mod` *Y*	Resto da Divisão
X + *Y*	Adição
X - *Y*	Subtração
Inicio .. *Passo* .. *Fim*	Uma série (lista) de números com um passo
Inicio .. *Fim*	Uma série (lista) de números com passo 1

Tabela 2: Tabela de Operadores Completa em Ordem de Precedência

Operadores Aritméticos	Ver Tabela 1
++	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 Especiais I

Operadores de Termos Não Compostos

1. **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*.
2. **Equivalência Numérica**(=:=): Compara se dois números são o mesmo valor. Não deve ser usada com termos que não são números.
3. **Diferença**(!==): Compara se dois termos são diferentes.
   Mesmo que a negação da equivalência.
4. **Não Unificável**(!=): 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. $a == a, \quad [1,2,3] == [1,2,3], \quad Var1 == Var2$

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. $a == a, \quad [1,2,3] == [1,2,3], \quad Var1 == Var2$
   yes, *yes*, Depende dos Valores (padrão *no*)

2. $1.0 == 1$
   no

3. $1.0 =:= 1, \quad 1.2 =:= 1$
   yes, *no*

Exemplos

1. $a == a, \quad [1,2,3] == [1,2,3], \quad Var1 == Var2$
   yes, *yes*, Depende dos Valores (padrão *no*)

2. $1.0 == 1$
   no

3. $1.0 =:= 1, \quad 1.2 =:= 1$
   yes, *no*

4. $1.0 \; !== 1, \quad Var3 \; !== Var4$

Exemplos

1. $a == a, \quad [1,2,3] == [1,2,3], \quad Var1 == Var2$
   yes, *yes*, Depende dos Valores (padrão *no*)

2. $1.0 == 1$
   no

3. $1.0 =:= 1, \quad 1.2 =:= 1$
   yes, *no*

4. $1.0 \; !== 1, \quad Var3 \; !== Var4$
   yes, Depende dos Valores (padrão *yes*)

Exemplos

1. $a == a, \quad [1,2,3] == [1,2,3], \quad Var1 == Var2$
   yes, *yes*, Depende dos Valores (padrão *no*)
2. $1.0 == 1$
   no
3. $1.0 =:= 1, \quad 1.2 =:= 1$
   yes, *no*
4. $1.0 \; !== 1, \quad Var3 \; !== Var4$
   yes, Depende dos Valores (padrão *yes*)
5. $1.0 \; != 1, \quad \text{aa} \; ! = \text{bb}, \quad Var1 \; != Var5$

Exemplos

1. $a == a, \quad [1,2,3] == [1,2,3], \quad Var1 == Var2$
   yes, *yes*, Depende dos Valores (padrão *no*)

2. $1.0 == 1$
   no

3. $1.0 =:= 1, \quad 1.2 =:= 1$
   yes, *no*

4. $1.0 \; !== 1, \quad Var3 \; !== Var4$
   yes, Depende dos Valores (padrão *yes*)

5. $1.0 \; != 1, \quad \text{aa} \; ! = \text{bb}, \quad Var1 \; != Var5$
   yes, *yes*, *no*

Operadores Especiais II

Operadores de Termos Compostos

1. **Concatenação** (++): concatena duas listas ou vetores, tornando o primeiro termo da segunda lista no termo seguinte ao último termo da primeira lista.
2. **Separador** (`H | 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*).
3. **Iterador** (`X in L`): itera pelo termo composto *L*, instanciando um termo não composto *X* aos termos contidos em *L*. Bastante utilizado para iterar por listas.
4. **Sequência** (`Inicio..Passo..Fim`): Gera uma lista ou vetor, começando (inclusivamente) em *Inicio* incrementando por *Passo* e parando (inclusivamente) em *Fim*. Se *Passo* for omitido, é automaticamente atribuído 1. Se usado dentro do índice de uma lista ou vetor resultará na porção da lista dentro deste intervalo.

Exemplos

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

Exemplos

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

Exemplos

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

Exemplos

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

2. $L = [1,2,3]$, $[H|T] = L$
   $L = [1,2,3]$

Exemplos

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

2. $L = [1,2,3]$, $[H|T] = L$
   $L = [1,2,3]$
   $H = 1$

Exemplos

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

2. $L = [1,2,3]$, $[H|T] = L$
   $L = [1,2,3]$
   $H = 1$
   $T = [2,3]$

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. $foreach(X\ in\ [1,2,3])\ printf("\%w", X)\ end$

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

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. $X = 1..10$, $Y = 0..2..20$, $Z = 10..-1..1$

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. $X = 1..10$, $Y = 0..2..20$, $Z = 10..-1..1$
   $X = [1,2,3,4,5,6,7,8,9,10]$

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. $X = 1..10$, $Y = 0..2..20$, $Z = 10..-1..1$
   $X = [1,2,3,4,5,6,7,8,9,10]$
   $Y = [0,2,4,6,8,10,12,14,16,18,20]$

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. $X = 1..10$, $Y = 0..2..20$, $Z = 10..-1..1$
   $X = [1,2,3,4,5,6,7,8,9,10]$
   $Y = [0,2,4,6,8,10,12,14,16,18,20]$
   $Z = [10,9,8,7,6,5,4,3,2,1]$

Resumindo

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!
- Sua sintaxe é moderna;
- Código aberto, multi-plataforma, e repleta de possibilidades;
- Uso para fins diversos;
- Muitas bibliotecas específicas prontas: CP, SAT, Planner, etc;
-
- .