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

5 de novembro de 2018

Contribuições

- Alexandre Gonçalves;
- João Herique Faes Battisti;
  - joaobattisti@gmail.com
- Paulo Victor de Aguiar;
  - pavaguiar@gmail.com
- Rogério Eduardo da Silva;
- Outros autores que auxiliaram na produção deste documento;

Sumário I

Sumário II

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 2.5 (5 de novembro de 2018).

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).

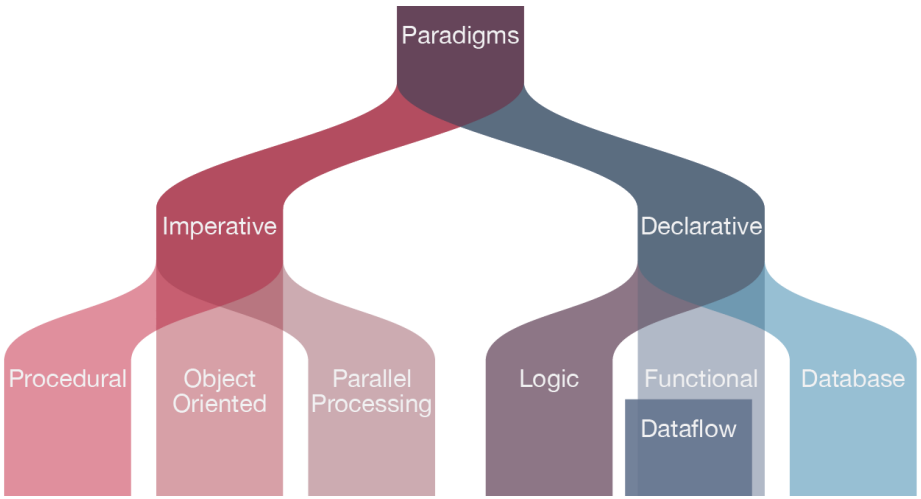


Figura 1: Fluxograma dos paradigmas de programação.

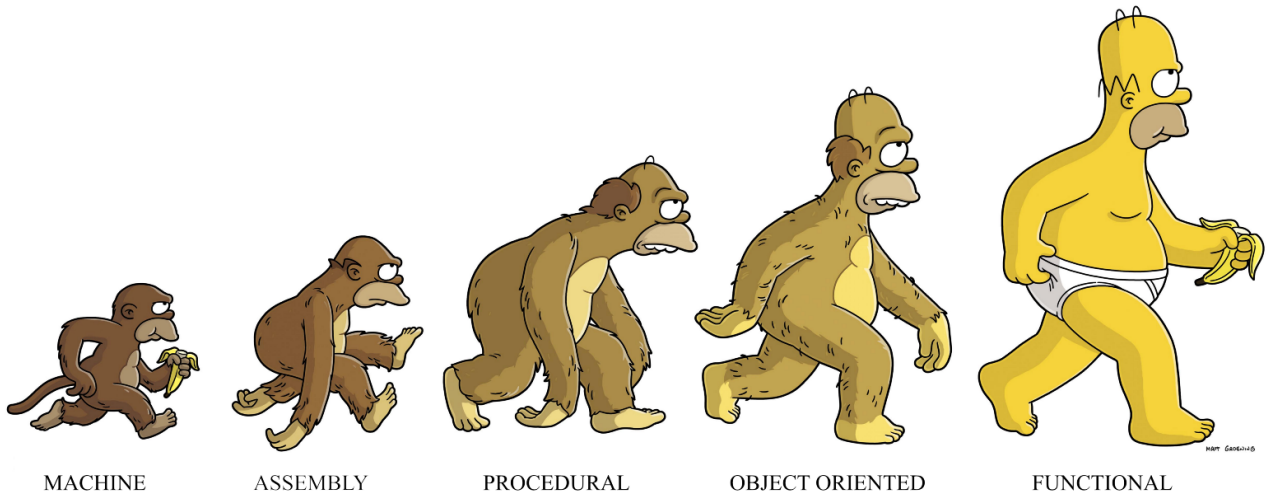
Paradigma Lógico

- 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;

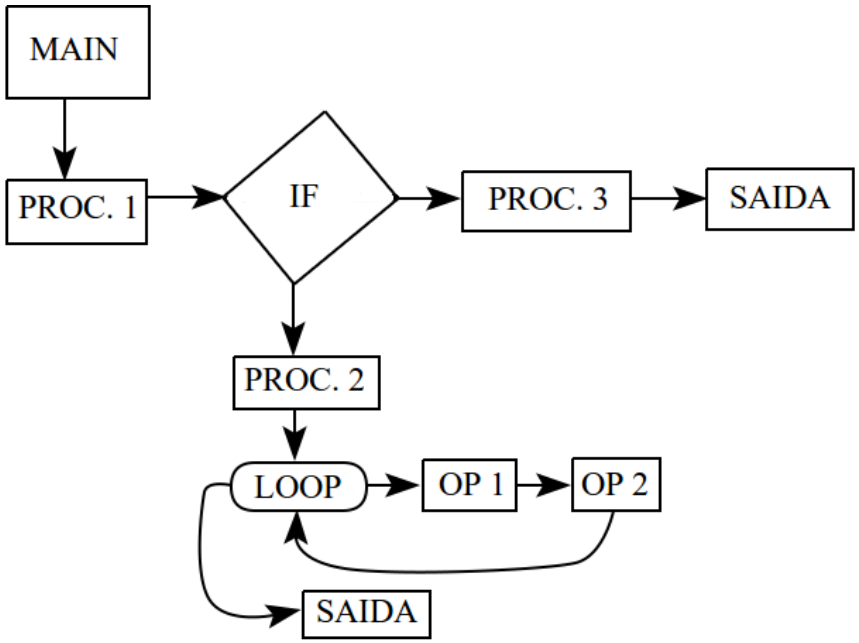


Figura 3: Fluxograma representando a estrutura de um programa Procedural

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.**

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

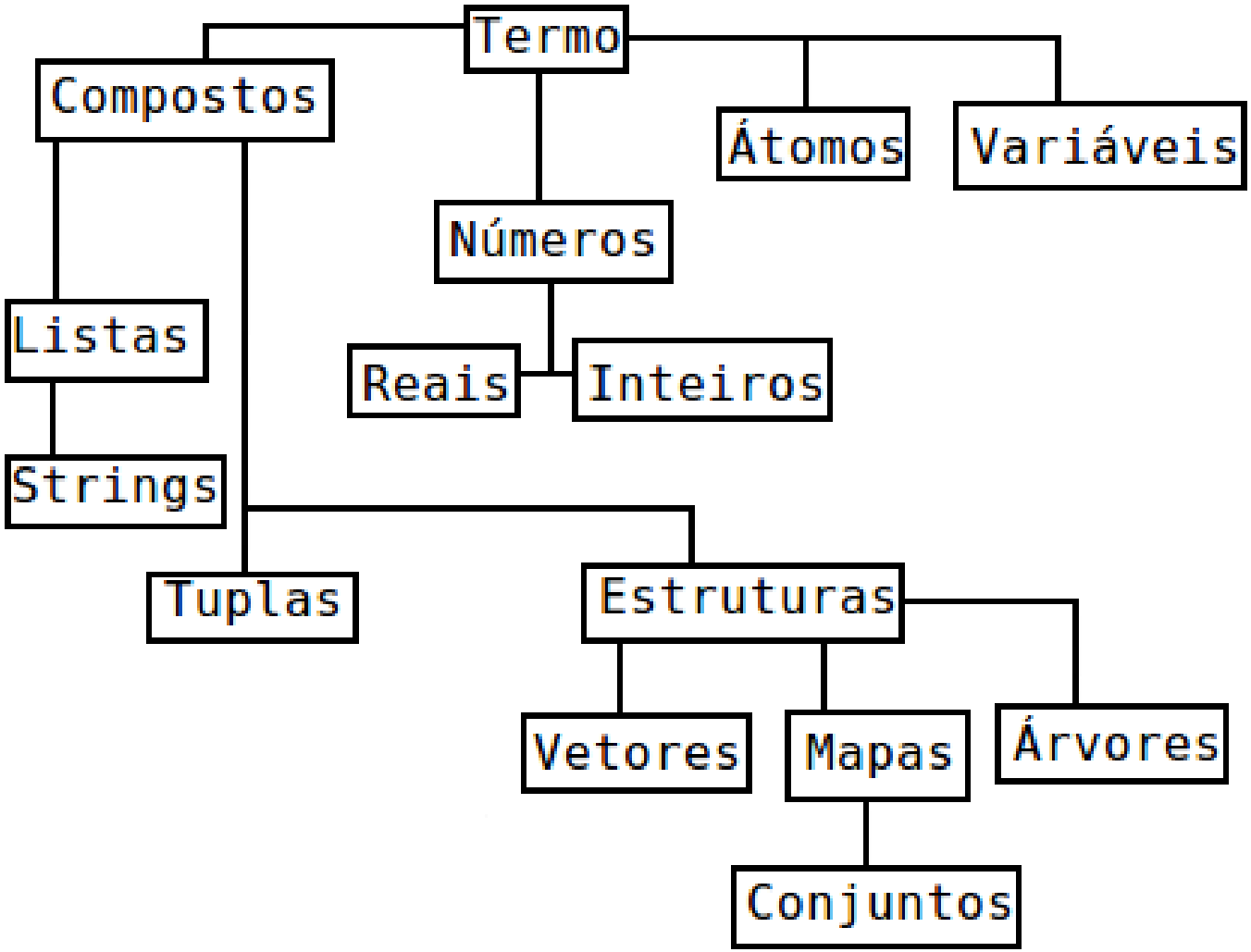


Figura 4: Hierarquia dos Tipos de Dados

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

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

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

Predicados e Funções: Conceitos Iniciais I

- Em Picat, predicados e funções são definidos com regras de casamento de padrões
- Há dois tipos de regras:
  - Regras sem *backtracking* (*non-backtrackable*):

    Cabeça, *Cond* => *Corpo*.

  - Regras com *backtracking*:

    Cabea, *Cond* ?=> *Corpo*
- Seus membros se dividem em:

Predicados e Funções: Conceitos Iniciais II

- *Cabeça*: indica um padrão de regra a ser casada.
  Forma geral:

  $$regra(termo_1, \ldots, termo_n)$$

  Onde:

  - *regra* é um átomo que define o nome da regra.
  - n é a aridade da regra (*i.e.* o total de argumentos)
  - Cada $termo_i$ é um argumento da regra.
- *Cond*: é uma ou várias condições sobre a execução desta regra.
- *Corpo*: define as ações da regra

Predicados e Funções: Conceitos Iniciais III

- Todas as regras são finalizadas por um ponto final (.), seguido por um espaço em branco ou nova linha.
- Ao serem chamadas regras também podem ser denotadas com uma notação semelhante à de métodos em Orientação a Objetos, como tal:

$$termo_1.regra(termo_2, \ldots, termo_n)$$

- Caso, $termo_1$ seja o único termo da regra, denota-se como:

$$termo_1.regra()$$

Casamento de Padrões I

- O algoritmo de *casamento de padrões* para regras é análogo ao algoritmo de unificação para variáveis.
- O objetivo é encontrar dois padrões que possam ser unificados para se inferir alguma ação.
- Quanto ao *casamento de padrões:*

Casamento de Padrões II

- Dado um padrão $p_1(t_1, \ldots, t_m)$, ele será *casado* com um padrão semelhante $p_2(u_1, \ldots, u_n)$ se:
  - p_1 e p_2 forem átomos equivalentes;
  - O número de termos (chamado de aridade) em $(t_1, \ldots, t_m)$ e $(u_1, \ldots, u_n)$ for equivalente.
  - Os termos $(t_1, \ldots, t_m)$ e $(u_1, \ldots, u_n)$ são equivalentes, ou podem ser tornados equivalentes pela unificação de variáveis que possam estar contidas em qualquer um dos dois termos;
- Caso essas condições forem satisfeitas o padrão $p_1(t_1, \ldots, t_m)$ é casado com o padrão $p_2(u_1, \ldots, u_n)$.

Exemplos de regras onde pode ocorrer o casamento

1. A regra $fatorial(Termo, Resultado)$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(abc, 25)$, $fatorial(X, Y)$, *etc.*

Exemplos de regras onde pode ocorrer o casamento

1. A regra $fatorial(Termo, Resultado)$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(abc, 25)$, $fatorial(X, Y)$, *etc.*
2. A regra $fatorial(Termo, Resultado)$, $Termo \geq 0$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(X, Y)$, $fatorial(Z, Z)$, *etc.*

Exemplos de regras onde pode ocorrer o casamento

1. A regra $fatorial(Termo, Resultado)$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(abc, 25)$, $fatorial(X, Y)$, *etc.*
2. A regra $fatorial(Termo, Resultado), Termo \geq 0$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(X, Y)$, $fatorial(Z, Z)$, *etc.*
3. A regra $pai(X, Y)$ pode casar com: $pai(rogerio, miguel)$, $pai(rogerio, henrique)$, $pai(salomao, X)$, $pai(12, 24)$, *etc.*

Exemplos de regras onde pode ocorrer o casamento

1. A regra $fatorial(Termo, Resultado)$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(abc, 25)$, $fatorial(X, Y)$, *etc.*
2. A regra $fatorial(Termo, Resultado), Termo \geq 0$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(X, Y)$, $fatorial(Z, Z)$, *etc.*
3. A regra $pai(X, Y)$ pode casar com: $pai(rogerio, miguel)$, $pai(rogerio, henrique)$, $pai(salomao, X)$, $pai(12, 24)$, *etc.*
4. A regra $pai(salomao, X)$ pode casar com: $pai(salomao, rogerio)$, $pai(salomao, fabio)$.

Exemplos de regras onde pode ocorrer o casamento

1. A regra $fatorial(Termo, Resultado)$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(abc, 25)$, $fatorial(X, Y)$, *etc.*
2. A regra $fatorial(Termo, Resultado)$, $Termo \geq 0$ pode casar com: $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(X, Y)$, $fatorial(Z, Z)$, *etc.*
3. A regra $pai(X, Y)$ pode casar com: $pai(rogerio, miguel)$, $pai(rogerio, henrique)$, $pai(salomao, X)$, $pai(12, 24)$, *etc.*
4. A regra $pai(salomao, X)$ pode casar com: $pai(salomao, rogerio)$, $pai(salomao, fabio)$.
5. A regra $pai(salomao, fabio)$ pode casar com: $pai(X, fabio)$, $pai(salomao, X)$, $pai(X, Y)$

Metas ou Provas

- Metas ou Provas são estados que definem o final da execução de uma regra.
- Uma meta pode ser, entre outros, um valor lógico, uma chamada de outra regra, uma exceção ou uma operação lógica.

1. `true`, `yes` ⇒ Valor lógico para verdade.
2. `false`, `no` ⇒ Valor lógico para falsidade.
3. $p(t_1, \ldots, t_n)$ ⇒ Chamada de uma regra p.
4. `(P,Q)`, `(P;Q)`, `(P&&Q)`, `(P||Q)`, `not P` ⇒ Operação lógica sobre uma ou mais metas P e Q.

Predicados I

- Forma geral de um predicado:

  Cabeça, *Cond* => *Corpo*.

- Forma geral de um predicado com *backtracking*:

  Cabea, *Cond* ?=> *Corpo*

Predicados II

- Predicados são um tipo de regra que definem relações, podendo ter zero, uma ou múltiplas respostas.
- Predicados podem, ou não, ser *backtrable*.
- Caso um predicado tenha $n = 0$, os parenteses que conteriam os argumentos podem ser omitidos.

Predicados III

- Dentro de um predicado, *Cond* só pode ser avaliado uma vez, acessando somente termos dentro do escopo do predicado.
- Predicados são avaliados com valores lógicos (*true* ou *false*), contudo, as variáveis passadas como argumento ou instanciadas dentro dele, podem ser utilizadas dentro do escopo do predicado, ou no escopo onde este predicado foi chamado.

Predicados Fatos I

- Predicados fatos são regras que não tem condições nem corpos.
- Ou seja, são do tipo:

$$p(t_1, \ldots, t_n).$$

- Os argumentos de um *predicado fato* **não** podem ser **variáveis**.

Predicados Fatos II

- A declaração de um predicado fato é precedida por uma declaração *index* do tipo:

$$\texttt{index}\ (M_{11}, M_{12}, \ldots, M_{1n})\ \ldots\ (M_{m1}, M_{m2}, \ldots, M_{mn})$$

- Onde cada M_{ij} é um simbolo $+$, que significa que este termo já foi indexado, o - que significa que este termo deve ser indexado.
- Ou seja, quando ocorre um simbolo $+$ em um grupo do *index*, é avaliado pelo compilador como um valor constante, que não irá gerar uma nova regra durante a execução do programa.

Predicados Fatos III

- Quanto ao -, ele é avaliado pelo compilador como uma variável que deverá ser instanciada à um valor e, para que isso ocorra, será necessária a geração de uma nova regra
- **Não** pode haver um **predicado** e um **predicado fato** com **mesmo nome**.

Funções I

- A forma geral de uma função é:

  $$Cabeça = X \Rightarrow Corpo.$$

- Caso haja alguma condição *Cond*, uma função é denotada de modo:

  $$Cabeça, Cond = X \Rightarrow Corpo.$$

- Funções **não** admitem ***backtracking***.

Funções II

- Funções são tipos especiais de regras que sempre sucedem com *uma* resposta.
- Funções em Picat tem como intuito serem sintaticamente semelhantes a funções matemáticas (vide *Haskell*).
- Em uma função a *Cabeça* é uma equação do tipo $f(t_1, \ldots, t_n) = X$, onde f é um átomo que é o nome da função, n é a aridade da função, e cada termo t_i é um argumento da função.
- X é uma expressão que é o retorno da função.

Funções III

- Funções também podem ser denotadas como fatos, onde podem servir como aterramento para regras recursivas, ou até mesmo como versões simplificadas de uma regra.
- São denotadas como: $f(t_1, \ldots, t_n) = \textit{Expressão}$, onde *Expressão* pode ser um valor ou uma série de ações.

Exemplos I

Exemplos de Predicados

```
1 entre_valores(X1, X2, X3) ?=>
2     number(X1),
3     number(X2),
4     number(X3),
5     X2 < X1,
6     X1 < X3.
7
```

```
1 entre_valores(X1, X2, X3), number(X1), number(X2), number(X3) ?=>
2     X2 < X1,
3     X1 < X3.
4
```

Exemplos II

Exemplos de Predicados Fatos

```
1  index(-,-) (+,-) (-,+)
2  pai(salomao, rogerio).
3  pai(salomao, fabio).
4  pai(rogerio, miguel).
5  pai(rogerio, henrique).
6
7  avo(X,Y) ?=> pai(X,Z), pai(Z,Y).
8  irmao(X,Y) ?=> pai(Z,X), pai(Z,Y).
9  tio(X,Y) ?=> pai(Z,Y), irmao(X,Z).
10
```

Exemplos III

Exemplos de Funções

```
1 eleva_cubo(1) = 1.
2 eleva_cubo(X) = X**3.
3 eleva_cubo(X) = X*X*X.
4 eleva_cubo(X) = X1 => X1 = X**3.
5 eleva_cubo(X) = X1 => X1 = X*X*X.
6
```

```
1 dobra_lista([]) = [].
2 dobra_lista(L), L != [] = L1 =>
3     L1 = L ++ L.
4
```

Condicionais e Repetições I

- Picat, ao contrário de muitas outras linguagens semelhantes, implementa uma estrutura condicional explícita.
- Sua notação é:

```
if (Exp) then
    Ações
else
    Ações
⋮
end
```

- Onde *Exp* é uma expressão lógica que será avaliada como verdadeiro ou falso.
- A última ação antes de um *else* ou *end* não deve ser sucedida por vírgula nem ponto e vírgula.

Condicionais e Repetições II

- Picat também implementa 3 estruturas de repetição, são elas: `foreach`, `while`, e `do-while`.
- O *loop* `foreach` tem como intuito iterar por termos compostos.
- O *loop* `while` irá repetir uma série de ações enquanto uma série de condições forem verdadeiras.
- O *loop* `do-while` é análogo ao loop `while`, porém ele sempre executará pelo menos uma vez.

Condicionais e Repetições III

- Um *loop* `foreach` tem a seguinte forma:

  `foreach (`E_1 `in` D_1`,` $Cond_1$`, ...,` E_n `in` D_n`,` $Cond_n$`)`
  Metas
  `end`

- Onde cada E_i é um *padrão de iteração* ou *iterador*. Cada D_i é uma expressão que gera um *valor composto* ou é um *valor composto*. Cada $Cond_i$ é uma condição opcional sobre os iteradores E_1 até E_i.
- Loops `foreach` podem conter múltiplos iteradores, como apresentado; caso isso ocorra, o compilador irá interpretar isso como diversos loops encapsulados. Maiores detalhes, ver Manual do Usuário.

Condicionais e Repetições IV

- Um *loop* `while` tem a seguinte forma:

```
while (Cond)
    Metas
end
```

- Enquanto a expressão lógica *Cond* for verdadeira, `Metas` será executado.

Condicionais e Repetições V

- Um *loop* `do-while` tem a seguinte forma:

```
do
    Metas
while (Cond)
```

- Ao contrário do *loop* `while` o *loop* `do-while` vai executar `Metas` pelo menos uma vez antes de avaliar `Cond`.

Funções/Predicados Especiais I

- Há algumas funções e predicados especiais em Picat que merecem um pouco mais de atenção.
- São estas as funções/predicados de: `compreensão de listas/vetores`, `entrada de dados` e `saída de dados`.

Funções/Predicados Especiais II

- A função de `compreensão de listas/vetores` é uma função especial que permite a fácil criação de listas ou vetores, opcionalmente seguindo uma regra de criação.
- Sua notação é:

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

- Onde, T é uma expressão que será adicionada a lista, cada E_i é um iterador, cada D_i é um termo composto ou expressão que gera um termo composto, cada $Cond_i$ é uma condição sobre cada iterador de E_1 até E_i.
- A função como dada acima geraria uma lista, para ser gerado um vetor a notação é:

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

Funções/Predicados Especiais III

- Picat tem diversas variações levemente diferentes da mesma função de leitura, que serve tanto para ler de um arquivo quanto de `stdin`.
- As mais importantes são:
  - `read_int(`*FD*`)` = *Int* $\Rightarrow$ Lê um *Int* do arquivo *FD*.
  - `read_real(`*FD*`)` = *Real* $\Rightarrow$ Lê um *Float* do arquivo *FD*.
  - `read_char(`*FD*`)` = *Char* $\Rightarrow$ Lê um *Char* do arquivo *FD*.
  - `read_line(`*FD*`)` = *String* $\Rightarrow$ Lê uma *String* do arquivo *FD*.
- Caso queira ler seu *input* de `stdin`, *FD* pode ser omitido.

Funções/Predicados Especiais IV

- Picat tem dois predicados para saída de dados para um arquivo, são eles `write` e `print`.
- Cada predicado tem três variantes, são eles:
  - `write(`FD`,` T`)` $\Rightarrow$ Escreve um termo T no arquivo FD.
  - `writeln(`FD`,` T`)` $\Rightarrow$ Escreve um termo T no arquivo FD, e pula uma linha ao final do termo.
  - `writef(`FD`,` F`,` $A\ldots$`)` $\Rightarrow$ Este predicado é usado para escrita formatada para um arquivo FD, onde F indica uma série de formatos para cada termo contido no argumento $A\ldots$. O número de argumentos não pode exceder 10.

Funções/Predicados Especiais V

- Analogamente, para o predicado `print`, temos:
  - `print(`*FD*`,` *T*`)` $\Rightarrow$ Escreve um termo *T* no arquivo *FD*.
  - `println(`*FD*`,` *T*`)` $\Rightarrow$ Escreve um termo *T* no arquivo *FD*, e pula uma linha ao final do termo.
  - `printf(`*FD*`,` *F*`,` *A*`...)` $\Rightarrow$ Este predicado é usado para escrita formatada para um arquivo *FD*, onde *F* indica uma série de formatos para cada termo contido no argumento *A*.... O número de argumentos não pode exceder 10.
- Caso queira escrever para `stdout` *FD* pode ser omitido.

Tabela de Formatos

Especificador	Saída
`%%`	Sinal de Porcentagem
`%c`	Caractere
`%d %i`	Número Inteiro Com Sinal
`%f`	Número Real
`%n`	Nova Linha
`%s`	*String*
`%u`	Número Inteiro Sem Sinal
`%w`	Termo

Comparação entre `write` e `print`

	"abc"	[a,b,c]	'a@b'
`write`	[a,b,c]	[a,b,c]	'a@b'
`writef`	[a,b,c] (%s)	abc (%w)	'a@b' (%w)
`print`	abc	abc	a@b
`printf`	abc (%s)	abc (%w)	a@b (%w)

Exemplos I

Condicionais

```
1 main =>
2     X = read_int(),
3     if(X <= 100)then
4         println("X e menor que 100")
5     else
6         println("X nao e menor que 100")
7     end
8 .
9
```

Exemplos II

Repetições

```
1 main =>
2     X = read_int(),
3     println(x=X),
4     while(X != 0)
5         X := X - 1,
6         println(x=X)
7     end
8 .
9
```

```
1 main =>
2     X = read_int(),
3     Y = X..X*3,
4     foreach(A in Y)
5         println(A)
6     end.
7
```

Exemplos III

Compreensão de Listas

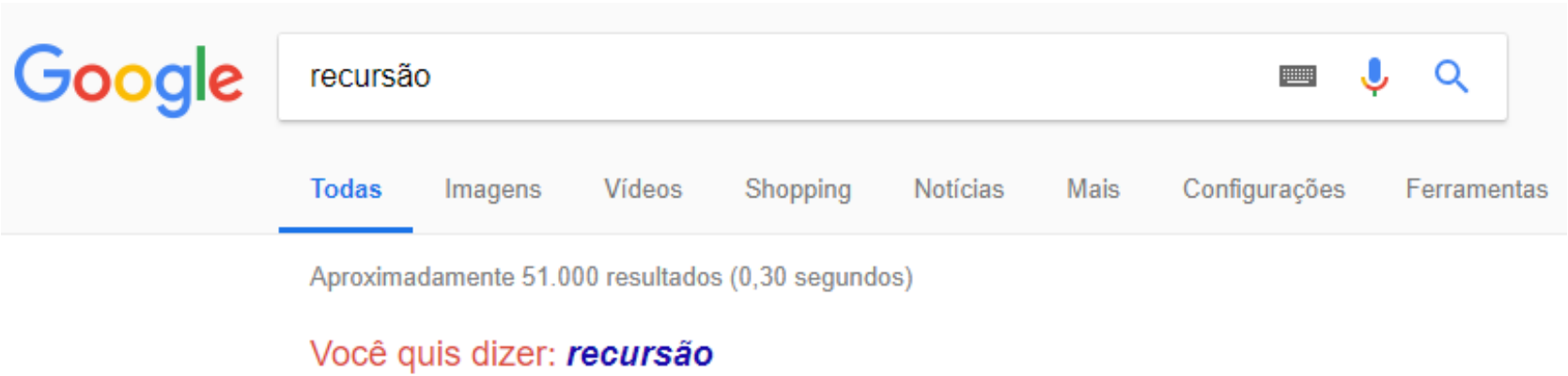
```
1  main =>
2      Tamanho = read_int(),
3      Y = [read_int() : 1..Tamanho],
4      Z = {X : X in Y, X >= 0},
5      foreach(I in 1..Tamanho)
6          if(Y[I] >= 0)then
7              printf("Y[%d] e maior ou igual que 0\n",I)
8          else
9              printf("Y[%d] e menor que 0\n",I)
10         end
11     end
12     printf("Os valores de Y que sao maiores que 0 sao:\n%w",Z)
13 .
14
```

Recursão

- A recursão é um importante conceito não só em Picat, mas também na matemática e em muitas outras linguagens de programação.
- Permite expressar problemas complexos de uma maneira simples.
- Uma regra é dita recursiva quando ela faz auto-referência.
- Exemplo:

Recursão

- A recursão é um importante conceito não só em Picat, mas também na matemática e em muitas outras linguagens de programação.
- Permite expressar problemas complexos de uma maneira simples.
- Uma regra é dita recursiva quando ela faz auto-referência.
- Exemplo:



Regras Matemáticas Definidas Recursivamente I

1. `Somatório`: O somatório é definido como a soma de todos os números em um dado intervalo. O somatório de *N* até *M* pode, então, ser definido como a soma de todos os números de *N* até *M-1* somado com *M*. Ou seja:

$$S(n) = \begin{cases} 1 & \text{para } n = 1 \\ S(n-1) + n & \text{para } n \geqslant 2 \text{ e } n \in \mathbb{N} \end{cases}$$

Ou seja:

$$S(n) = \underbrace{1 + 2 + 3 + ... + (n-1)}_{S(n-1)} \quad +n$$

Regras Matemáticas Definidas Recursivamente II

2. `Fatorial`: O Fatorial de um número é definido como este mesmo número multiplicado pelo fatorial do número anterior. O fatorial só pode ser calculado para números positivos, e é fato que o fatorial de 0 é igual a 1.

$$Fat(n) = \begin{cases} 1 & \text{para } n = 0 \\ Fat(n-1) * n & \text{para } n \geqslant 1 \text{ e } n \in \mathbb{N} \end{cases}$$

Portanto, podemos inferir que:

$$Fat(n) = \underbrace{1 * 2 * 3 * ..... * (n-1)}_{Fat(n-1)} * n$$

Regras Matemáticas Definidas Recursivamente III

3. `Sequência Fibonacci`: A sequência Fibonacci é uma sequência de números calculada a partir da soma dos dois ultimos números anteriores, ou seja o $n-esimo$ termo da Sequência Fibonacci é definido como a soma dos termos $n-1$ e $n-2$. É fato que os dois primeiros termos ($n=0$ e $n=1$) são, respectivamente, 0 e 1.

$$Fib(n) = \begin{cases} 0 & \text{para } n = 0 \\ 1 & \text{para } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{para } n \geqslant 1 \text{ e } n \in \mathbb{N} \end{cases}$$

Regras Matemáticas Definidas Recursivamente IV

- Podemos perceber algo em comum entre estas três regras, todas tem uma ou mais condições que sempre tem o mesmo valor de retorno, ou seja, todas tem uma regra de aterramento.
- Uma condição de aterramento é uma condição onde a chamada recursiva da regra acaba.
- Caso uma regra não tenha uma regra de aterramento, poderá ocorrer uma recursão infinita deste regra, ou seja, são feitas infinitas chamadas recursivas da regra.

Exemplo de Recursão Infinita

- Caso alterássemos a definição da regra fatorial de modo que ela seja:

$$Fat(n) = \quad Fat(n-1) * n, \quad \forall n \in \mathbb{N} \text{ ou } \forall n \geq 0$$

- Teríamos um caso de recursão infinita, pois a regra Fatorial seria continuamente chamada até que $n < 0$, nesse caso haveria um erro, pois estaria tentando executar algo indefinido.

Transcrevendo estas regras para Picat

```
1 fatorial(0) = 1.
2 fatorial(1) = 1.
3 fatorial(n) = n * fatorial(n-1).
4
```

```
1 fibonacci(0) = 0.
2 fibonacci(1) = 1.
3 fibonacci(n) = fibonacci(n-1) + fibonacci(n-2).
4
```

```
1 somatorio(0) = 0.
2 somatorio(1) = 1.
3 somatorio(n) = n + somatorio(n-1).
4
```

Backtracking I

- *Backtracking* é um conceito muito parecido com recursão, porém a principal diferença entre os dois (no escopo da linguagem) é que *backtraking* é exclusivo à predicados, enquanto que ambos funções e predicados podem ser definidos de forma recursiva.
- O que distingue um predicado que pode fazer *backtracking* de um que não pode é o uso do símbolo ?=> no lugar do símbolo =>.
- *Backtring* é uma construção da linguagem, que é definida como segue:

Backtracking II

- O casamento de um predicado *backtrackable* p com outro predicado *backtrackable* $p1$.
- A execução do predicado p.
- Caso ocorra uma falha durante a execução do predicado p o compilador irá reinstanciar todas as variáveis de p, incluindo aquelas que são indexadas a partir de um domínio, com a única exceção sendo variáveis instanciadas a partir de argumentos do predicado.
- O predicado será executado novamente.
- Este processo se repete até não for mais possível a reinstanciação de variáveis, ou ocorrer um erro durante a execução.

Exemplo

- Tomando como exemplo uma relação de parentesco, como a seguinte:

```
1  index(-,-) (+,-) (-,+)
2  antecedente(ana,maria).
3  antecedente(pedro,maria).
4  antecedente(maria,paula).
5  antecedente(paula,lucas).
6  antecedente(lucas, eduarda).
7
8  index(-)
9  mulher(ana).
10 mulher(maria).
11 mulher(paula).
12 mulher(eduarda).
13 homem(pedro).
14 homem(lucas).
15
16 mae(X,Y) ?=> antecedente(X,Y), mulher(X).
17 pai(X,Y) ?=> antecedente(X,Y), homem(X).
18 avos(X,Y) ?=> antecedente(X,Z), antecedente(Z,Y).
19 sucessor(X,Y) ?=> antecedente(Y,X).
20 sucessor(X,Y) ?=> antecedente(Y,Z), sucessor(X,Z).
21
```

- Uma chamada do tipo $mae(maria, X)$, seria como perguntar ao compilador "Maria é mãe de quem ?".
- Nesse caso o compilador iria testar cada possível valor que pudesse ser unificado com X que pudesse satisfazer a regra $mae(maria, X)$.
- Ou seja, seria como se estivéssemos perguntando:
  - "Maria é mãe de Ana ?".
  - "Maria é mãe de Paula ?".
  - "Maria é mãe de Pedro ?".
  - $\vdots$