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

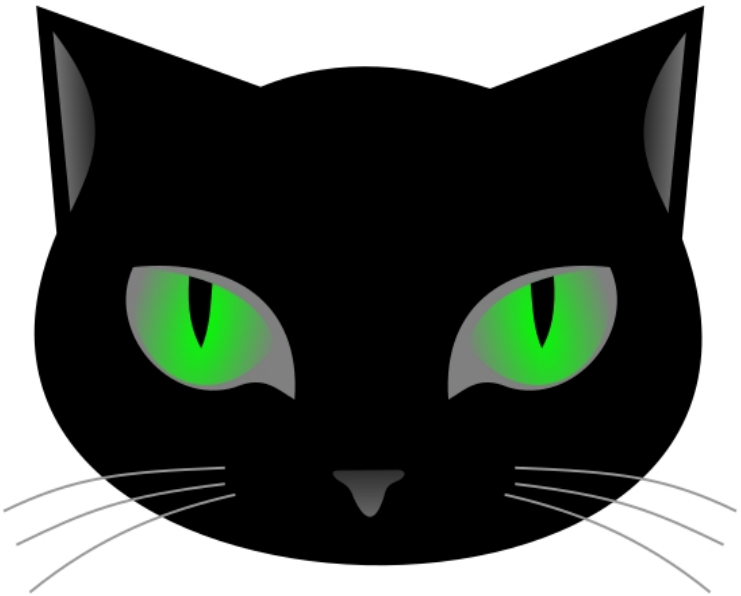
4 de maio de 2019

Contribuições e Agradecimentos

- Miguel Alfredo Nunes
- Jeferson L. R. Souza
- Alexandre Gonçalves
- Hakan Kjellerstrand – (`http://www.hakank.org/picat/`)
- Neng-Fa Zhou – (`http://www.picat-lang.org/`)
- João Herique Faes Battisti
- Paulo Victor de Aguiar
- Rogério Eduardo da Silva
- Outros anônimos que auxiliaram na produção deste documento

Estruturas de Decisão, Laços e Iteradores

- Definições
- Contexto de uso
- Estruturas de decisão
- Estruturas de repetição
- Iteradores
- Entrada e saídas
- Exemplos



- Ao contrário do Prolog, Picat apresenta conceitos e comandos da programação imperativa

- Ao contrário do Prolog, Picat apresenta conceitos e comandos da programação imperativa
- Esta maneira ameniza os obstáculos em se aprender uma linguagem com o paradigma lógico, tendo outros elementos conhecidos

Estruturas de Decisão, Laços e Iteradores

- Ao contrário do Prolog, Picat apresenta conceitos e comandos da programação imperativa
- Esta maneira ameniza os obstáculos em se aprender uma linguagem com o paradigma lógico, tendo outros elementos conhecidos
- Assim, Picat apresenta estruturas clássicas como:
  - `if-then-end`, `if-then-else-end`, `if-then-elseif-then-....end`
  - `foreach`
  - `while`
  - `do-while`
  - Bem como a atribuição, ‘`:=`’, já discutida

- Picat implementa uma estrutura condicional explícita (na programação em lógica, voce faz isto implicitamente)

- Picat implementa uma estrutura condicional explícita (na programação em lógica, voce faz isto implicitamente)
- Sua notação é:

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

Estrutura Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, voce faz isto implicitamente)
- Sua notação é:

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

- Onde *Exp* é uma expressão lógica avaliada como verdadeira ou falsa.

Estrutura Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, voce faz isto implicitamente)
- Sua notação é:

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

- Onde *Exp* é uma expressão lógica avaliada como verdadeira ou falsa.
- A última ação antes de um *else* ou *end* não deve ter vírgula nem ponto e vírgula ao final da linha.

Estrutura Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, voce faz isto implicitamente)
- Sua notação é:

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

- Onde *Exp* é uma expressão lógica avaliada como verdadeira ou falsa.
- A última ação antes de um *else* ou *end* não deve ter vírgula nem ponto e vírgula ao final da linha.

- Tem-se ainda o `elseif` que pode estar embutido no comando `if-then-else-end`

Exemplo: if-then-else-end

```
1       ,
2  if (X <= 100) then
3           println("X e menor que 100")
4       elseif (X <= 1000 && X >= 500) then
5           println("X estah entre 500 e 1000")
6       else
7           println("X estah abaixo de  500")
8       end
9       ,
```

- Picat também implementa 3 estruturas de repetição: `foreach`, `while` e `do-while`

- Picat também implementa 3 estruturas de repetição: `foreach`, `while` e `do-while`
- O laço do `foreach` **itera** sobre termos simples e compostos

- Picat também implementa 3 estruturas de repetição: `foreach`, `while` e `do-while`
- O laço do `foreach` **itera** sobre termos simples e compostos
- O `while` repete um conjunto de ações enquanto uma condição for verdadeira.

- Picat também implementa 3 estruturas de repetição: `foreach`, `while` e `do-while`
- O laço do `foreach` **itera** sobre termos simples e compostos
- O `while` repete um conjunto de ações enquanto uma condição for verdadeira.
- A condição pode ser simples ou combinada: ver exemplos

Estruturas de Repetições

- Picat também implementa 3 estruturas de repetição: `foreach`, `while` e `do-while`
- O laço do `foreach` **itera** sobre termos simples e compostos
- O `while` repete um conjunto de ações enquanto uma condição for verdadeira.
- A condição pode ser simples ou combinada: ver exemplos
- O laço `do-while` é análogo ao `while`, porém ele sempre executa pelo menos uma vez

Estruturas de Repetições: `foreach`

- Um laço `foreach` tem a seguinte forma:

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

Estruturas de Repetições: `foreach`

- Um laço `foreach` tem a seguinte forma:

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

  Esta notação é dada por:

- E_i é um *padrão de iteração* ou *iterador*.
- D_i é uma expressão de *valor composto*. Exemplo: uma lista de valores
- $Cond_i$ é uma condição opcional sobre os **iteradores** E_1 até E_i.
- O `foreach` pode conter múltiplos iteradores usando o "`in`" Caso isso ocorra, o compilador interpreta isso como diversos laços aninhados.

Exemplo: foreach

```
1  imp_tracejados (N) =>
2    nl,
3    foreach(I in 1..N)
4      printf("=")
5    end,
6    nl.
7  ..........................................................................
8  Picat> cl('backtracking_ex_02').
9  Compiling:: backtracking_ex_02.pi
10 ** Warning: singleton variables (backtracking_ex_02.pi, 50-55): I
11 backtracking_ex_02.pi compiled in 3 milliseconds
12 loading...
13
14 yes
15
16 Picat> imp_tracejados(30).
17
18 ==============================
```

O I é iterado com valores de 1 a N

Estruturas de Repetições: `while`

- O laço do `while` tem a seguinte forma:

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

- Enquanto a expressão lógica *Cond* for verdadeira, o conjunto de `Metas` é executado.

Exemplo: `while`

```
1
2 laco_02 =>
3     I = 1,
4     while (I <= 9)
5         println(I),
6         I := I + 2
7     end.
```

Estruturas de Repetições: `do-while`

- O laço `do-while` tem a seguinte forma:

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

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

Exemplo: do-while

```
1
2 laco_03 =>
3       J = 6,
4     do
5         println(J),
6         J := J + 1
7     while (J <= 5).
```

- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado.

Funções e Predicados Especiais

- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado.
- São elas: `compreensão de listas/vetores`, `entrada de dados` e `saída de dados`.
- Na verdade, já fizemos uso delas, porém sem a ênfase de que são funções ora predicados.

Compreensão de Listas e Vetores I

- A função de *compreensão de listas e vetores* é uma função especial que permite a fácil criação de listas ou vetores.
- 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 adicionada a lista, cada E_i é um iterador sobre D_i, o qual é um termo ou expressão, e $Cond_i$ é uma condição sobre cada iterador de E_1 até E_i.
- Há uma seção dedicada a listas. Voltaremos ao assunto.

Compreensão de Listas e Vetores II

- Esta função pode gerar um vetor também, a notação é um pouco diferente:

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

- Neste caso, os delimitadores são { e } de um vetor

Compreensão de Listas e Vetores: Exemplo

```
main =>
 L = [(A, I) : A in [a, b], I in 1 .. 2],
 V = {(I, A) : A in [a, b], I in 1 .. 2},
 printf("\nL: %w  \nV: %w\n", L, V),
 imp_vetor(V).

imp_vetor (M) =>
Tam = M.length,  %% tamanho de M
  nl,
   foreach(I in 1  .. Tam )
     printf("V(%d):%w \t" , I, M[I] )
    end,
  nl.
%%%% $picat vetor_exemplo_01.pi
```

Leitura e Escrita I

- Picat tem diversas funções de leitura de valores, que serve tanto para ler de uma console `stdin`, como de um arquivo qualquer.
- Aos usuários de Prolog, aqui não precisamos do delimitador final de '.' ao final de uma leitura.
- Válido quando editamos no interpretador, o '.' final é opcional

- 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 *Linha* do arquivo *FD*.
- Caso se deseja ler da console, padrão `stdin`, *FD*, o nome do descritor de arquivo, pode ser omitido.

Leitura e Escrita III

- Os dois predicados mais importantes para saída de dados, são `write` e `print`.
- Cada um destes predicados 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.

- 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`, o nome do *FD*, pode ser omitido.

Tabela de Formatação para Escrita

Apenas os mais importantes, há outros como: hexadecimal, notação científica, etc. Ver no apêndice do Guia do Usuário.

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

Comparação entre `write` e `print`

Dados ⇒	"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)

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
```

Exemplos – 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 – Construindo Listas e Vetores

```
import util. % use split
main =>
        le_vetor_01(X1),
        printf("\nVETOR LIDO: %w ", X1),
        le_vetor_02(X2),
        printf("\nVETOR LIDO: %w ", X2),
        le_lista_01(Y),
        printf("\nLISTA LIDA: %w ", Y),
        le_lista_02(W),
        printf("\nLISTA LIDA 2: %w " , W) .
```

- Este exemplo reúne muitos conceitos desta seção.
- https://github.com/claudiosa/CCS/blob/master/picat/input_output_exemplos/leitura_vetores_listas.pi

Exemplo – Leitura de Vetores

```
le_vetor_01 ( V ) =>
  printf("\nDIGITE tamanho da entrada: "),
  Tam = read_int(),
  V = new_array( Tam ), % cria um vetor
  printf("\nDIGITE os %d VALORES do vetor:", Tam),
  foreach (I in 1..Tam)
        V[I] = read_int()
  end,
  printf("\nVETOR: %w ", V).

le_vetor_02 ( V ) =>
  printf("\nLendo um vetor qualquer de inteiros na linha: "),
  V = { to_int(W) : W in read_line().split() }.
  % OU
  %L = [ to_int(W) : W in read_line().split()],
  %V = to_array( L ).
```

Exemplo – Leitura de Listas

```
le_lista_01 ( L ) =>
  printf("\nLendo lista de inteiros na linha: "),
  L = [ to_int(W) : W in read_line().split()].

le_lista_02 (List) =>
    printf("\nLista inteiros e 0 encerra: "),
    L := [] ,
    E := read_int() ,
    while (E != 0)
        L := [E|L],
        E := read_int()
    end,
    List = L .
```

Volte neste exemplo após a seção de Listas.

Saída – via main

```
$ picat leitura_vetores_listas.pi
DIGITE tamanho da entrada: 3
DIGITE os 3 VALORES do vetor: 3 4 5
VETOR LIDO: {3,4,5}
Lendo um vetor qualquer de inteiros na linha: 9 8 7 6
VETOR LIDO: {9,8,7,6}
Lendo lista de inteiros na linha: 1 2 3 4
LISTA LIDA: [1,2,3,4]
Lista inteiros e 0 encerra: 1 2 3 7 0 1 2 3 5
LISTA LIDA 2: [7,3,2,1]
.... removi algumas linhas em branco
```

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para *entradas* e *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para *entradas* e *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado**: as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false**!

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para *entradas* e *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado**: as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false**!
- Felizmente os erros e problemas de sintaxe são prontamente acusados

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para *entradas* e *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado**: as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false**!
- Felizmente os erros e problemas de sintaxe são prontamente acusados
- Em `https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos` tem vários exemplos avançados de entradas e saídas

Reflexões

- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para *entradas* e *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado**: as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false**!
- Felizmente os erros e problemas de sintaxe são prontamente acusados
- Em `https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos` tem vários exemplos avançados de entradas e saídas
- Mãos à obra!