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

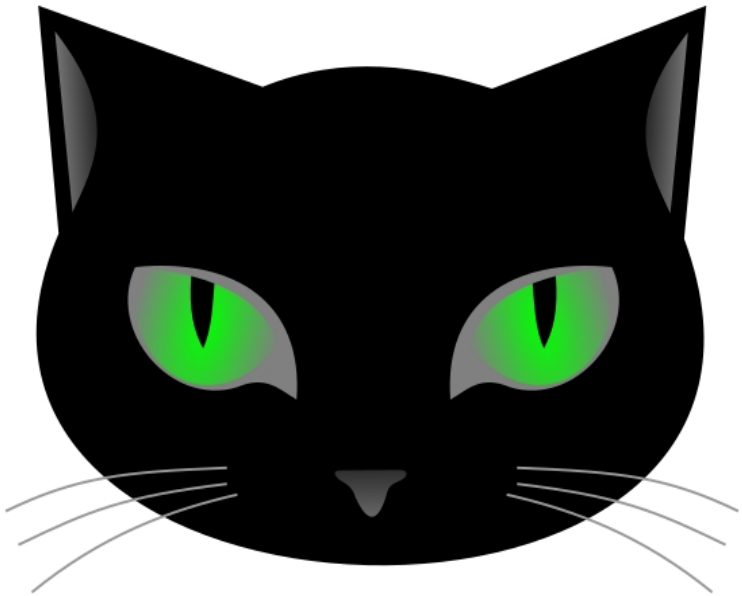
3 de maio de 2019

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

Predicados e Funções

- Definições
- Predicados ou Cláusulas
- Funções
- Contexto de uso
- Exemplos



- Os **predicados** sempre assumem valores valores lógicos `true` (1) ou `false` (0).
- Os **predicados** em seus argumentos, podem passar *n*-termos e receber outros *m*-termos.
- Quanto as **funções**, estas funcionam seguindo as regras de funções matemáticas, sempre retornando um único valor. Logo, um tipo particular de **predicados lógicos** que podem retornar vários termos
- Predicados e funções são definidos com regras de *casamento de padrões*
- Predicados são conhecidos como **regras lógicas**, há dois tipos de regras:

- Regras **sem** *backtracking* (*non-backtrackable*):

```
Cabeça , Condicional  =>  Corpo .
```

- Regras **com** *backtracking*:

```
Cabeça , Condicional  ?=>  Corpo .
```

- A identificação da sintaxe é dada por:
  - *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
- Todas as regras são finalizadas por um ponto final (.), seguido por um espaço em branco ou nova linha.
- Ao longo dos exemplos, detalhes a mais sobre esta construção de **predicados** e **funções**

Regras com e sem Backtracking – Exemplo

```
main => regra_01(7)  ,
        regra_01(-4) ,
        regra_01(44) .

regra_01(0)           ?=> printf("\n  CHEGOU A 0 !!!\n").
regra_01(N) , N < 0   ?=> printf("\n  EH UM NEGATIVO !!!\n").
regra_01(N) , N > 10 ?=> printf("\n  EH MAIOR QUE 10 !!!\n").
regra_01(N) , N <= 10  =>
          printf("\t :%d ", N),
          regra_01(N-1).

%% =<  EH IGUAL a >= :: sobrecarga
%%% $ picat regras_com_sem_backtraking.pi
```

Há uma recursão aqui. Algumas seções a frente.

Casamento de Padrões

- 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.

Casamento de Padrões

- 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.
- Muitos exemplos no curso

Casamento de Padrões – Procedimento

Procedimento de *casamento de padrões*:

- Dado um padrão $p_1(t_1, \ldots, t_m)$, este *casa* 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 tornaram-se equivalentes pela unificação de variáveis em qualquer um dos dois termos;
- Caso essas condições forem satisfeitas, o padrão $p_1(t_1, \ldots, t_m)$ casa com o padrão $p_2(u_1, \ldots, u_n)$.

Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* é conhecido também como *prova do programa*

Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* é conhecido também como *prova do programa*
- Metas ou provas (do inglês: *goal*) são estados que definem o final da execução

Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* é conhecido também como *prova do programa*
- Metas ou provas (do inglês: *goal*) são estados que definem o final da execução
- Uma meta pode ser, um valor lógico, uma chamada de outra regra, uma exceção ou uma operação lógica, um termo ...

Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* é conhecido também como *prova do programa*
- Metas ou provas (do inglês: *goal*) são estados que definem o final da execução
- Uma meta pode ser, um valor lógico, uma chamada de outra regra, uma exceção ou uma operação lógica, um termo ...
- Exemplo: a cláusula `main` é uma meta a ser provada!

Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* é conhecido também como *prova do programa*
- Metas ou provas (do inglês: *goal*) são estados que definem o final da execução
- Uma meta pode ser, um valor lógico, uma chamada de outra regra, uma exceção ou uma operação lógica, um termo ...
- Exemplo: a cláusula `main` é uma meta a ser provada!
- Em resumo, todas cláusulas, de alguma maneira são metas a serem provadas!

- A forma geral de uma função é:

$$Cabeça = X \Rightarrow Corpo.$$

- A forma geral de uma função é:

$$Cabeça = X \Rightarrow Corpo.$$

- Caso tenhamos uma condição *Cond*::

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

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

Funções – II

- Funções são tipos especiais de regras que sempre sucedem com *uma* resposta.

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*).

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

Regras, Metas e Funções – Exemplo

```
main =>
        X = 3,
        Y = 4,
        um_predicado(X,Y,Z),
        R = uma_funcao(X,Y),
        printf("\n Z: %d \t R: %d", Z, R),
        println("\n FIM").

um_predicado(X,Y,Z) => Z = X + Y.

uma_funcao(X,Y) = R => R = X + Y.
```

Regras, Metas e Funções – Exemplo

```
Picat> cl('predicados_funcoes').
Compiling:: predicados_funcoes.pi
predicados_funcoes.pi compiled in 0 milliseconds
loading...

yes

Picat> um_predicado(3,4,Z), write(Z).
7Z = 7
yes

Picat> uma_funcao(3,4) = R, write(R).
7R = 7
yes

Picat>
```

Relembrando as Regras I

- Forma geral de um predicado do tipo Regra:

```
Cabeça , Condicional  =>  Corpo .
```

- Forma geral de um predicado com *backtracking*:

```
Cabeça , Condicional  ?=>  Corpo .
```

Em Prolog, esta condicional (Cond) entra no corpo da regra. Picat é flexível!

Relembrando as Regras II

- Dentro de uma regra, *Cond* só pode ser avaliado uma vez, acessando somente termos dentro do escopo do predicado.
- Sempre estar atento que: regras são **sempre avaliados com valores lógicos** (*true* ou *false*)
- Por outro lado, as variáveis como argumento ou instanciadas dentro dele, podem ser utilizadas dentro do escopo da regra, ou no escopo onde esta regra foi chamada.

Regras do Tipo Fatos I

- As regras que não tem condicionais e nem corpo, estes são conhecidos como: *fatos* ou *regras sem-corpo*
- Estes *fatos* são regras *sempre verdadeiras*
- Formato dos fatos são do tipo:

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

- Os argumentos de um *fato* **não** podem conter **variáveis**.

Regras do Tipo Fatos II

- A declaração de um fato é precedida por uma declaração ***index***, algo como:

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

- Onde um M_{ij} com o simbolo '+', significa que este termo já foi indexado.
- Quanto o '$-$' 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 a ser casada

Regras do Tipo Fatos III

- Quanto ao '_', este é avaliado pelo compilador com uma variável que deverá ser instanciada à um valor. Ou seja, quando se deseja unificar um valor a esta variável
- Dica: o parâmetro '_' no *index* é quase como regra geral
- **Não** pode haver um **predicado** e um **predicado fato** com **mesmo nome**.

Exemplo – Função e Regras

```
index (+,+,+) (+,+, -) (-,+,-) (-,-,+) (-,-, +)
      (+,-,+) (+,+,-)  (-,-,-)
%(-,-,-) %% NENHUM argumento instanciao -- UTIL
%(+,+,+) %% TODOS ARGUMENTOS DEVEM ESTAR INSTANCIADOS
%(+,+, -) (-,+,-) (-,-,+) (-,-, +) (+,-,+) (+,+,-) (-,-,-)

and2(true,true,true).
and2(true,false,false).
and2(false,true,false).
and2(false,false,false).

main ?=>
       and2(X,Y,Z), % and eh reservado
       printf("\n X: %w \t Y: %w \t Z: %w", X, Y, Z),
       fail.
main =>
       println("\n FIM").
```

Este exemplo é muito interessante. Execute ele na console do interpretador excluindo alguns dos parâmetros do *index*

Exemplo de Predicado ou regra

```
1  contas_PO(X1, X2, X3, Z) ?=>
2      number(X1),
3      number(X2),
4      number(X3),
5      X1 < X2,
6      X2 < X3,
7      Z  =  (X2 + X3).
8
9  contas_PO(X1, _, _, Z) =>
10     Z = X1.
```

Exemplo de Funções

```
1 contas_F0(X1, X2, X3) = Z, (number(X1),
2                             number(X2),
3                             number(X3)) =>
4     if (X1 < X2 && X2 < X3) then
5     Z  =  (X2 + X3)
6     else
7     Z  =  X1
8     end.
```

Aperitivo à próxima seção: condicionais e laços!

Mais Exemplos (Fatos e Regras)

```
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).
```

Exemplos de Funções – Equivalentes

```
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.
```

Reflexões

- Esta seção trata dos elementos do Picat: cláusulas

- Esta seção trata dos elementos do Picat: cláusulas
- Predicados ou cláusulas, que são de 2 tipos: **regras predicativas** ou **funções**

Reflexões

- Esta seção trata dos elementos do Picat: cláusulas
- Predicados ou cláusulas, que são de 2 tipos: **regras predicativas** ou **funções**
- *Regras* é um nome genérico a predicados ou cláusulas e funções

Reflexões

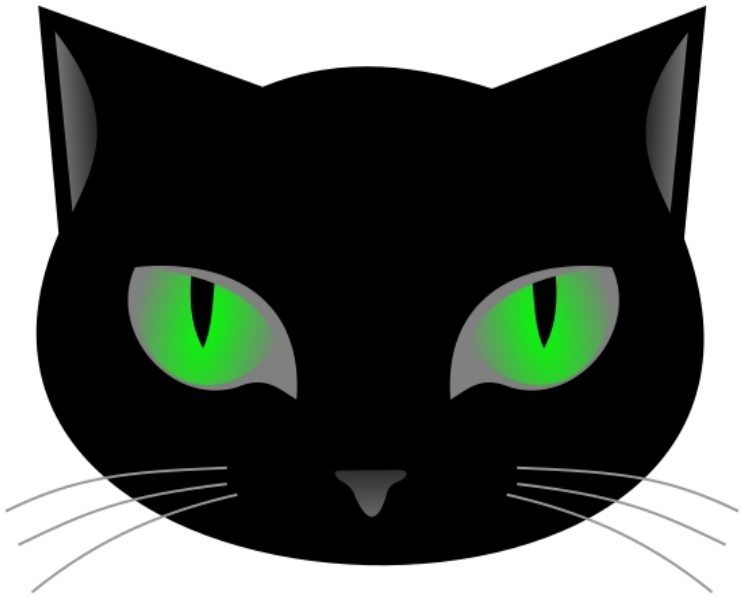
- Esta seção trata dos elementos do Picat: cláusulas
- Predicados ou cláusulas, que são de 2 tipos: **regras predicativas** ou **funções**
- *Regras* é um nome genérico a predicados ou cláusulas e funções
- *Regras* sem corpo são conhecidas como verdades e chamadas de **fatos**
  Uso obrigatório do meta-predicado: *index*

Reflexões

- Esta seção trata dos elementos do Picat: cláusulas
- Predicados ou cláusulas, que são de 2 tipos: **regras predicativas** ou **funções**
- *Regras* é um nome genérico a predicados ou cláusulas e funções
- *Regras* sem corpo são conhecidas como verdades e chamadas de **fatos**
  Uso obrigatório do meta-predicado: *index*
- Lembrando ainda que funções retornam um único valor

Estruturas de Decisão, Laços e Iteradores

- Definições
- Contexto de uso
- Estruturas de decisão
- Estruturas de repetição
- Iteradores
- 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.
```

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 .
9
```

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 de 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 Lista

```
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

Reflexões

- Esta seção avança na sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser praticada

Reflexões

- Esta seção avança na sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela 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
- Embora sua sintaxe não seja muito extensa, ela 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 saída, com Picat, tudo ficou semelhante as demais linguagens imperativas

Reflexões

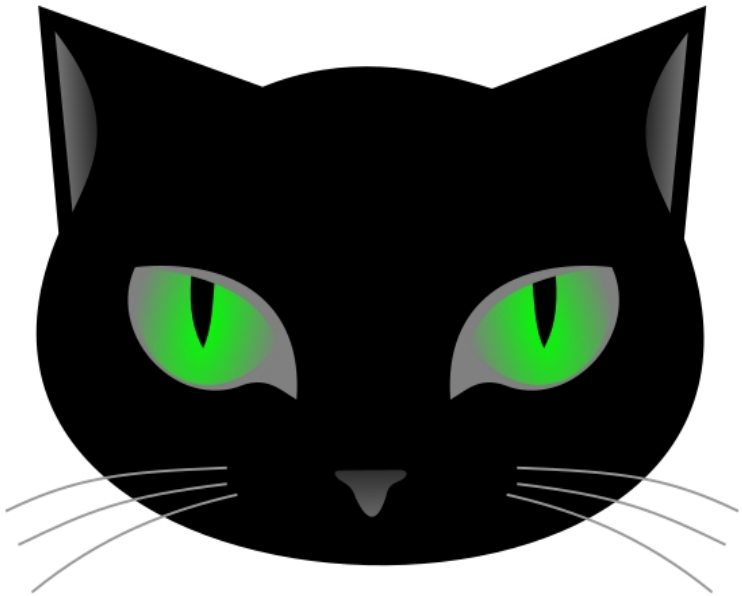
- Esta seção avança na sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para entradas e saída, com Picat, tudo ficou semelhante as demais linguagens imperativas
- 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
- Embora sua sintaxe não seja muito extensa, ela 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 saída, com Picat, tudo ficou semelhante as demais linguagens imperativas
- Em `https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos` tem vários exemplos avançados de entradas e saídas
- Mãos à obra!

Conclusões

- O que foi visto
- O que tem a ser feito
- Oportunidades



Conclusões

- Picat é jovem (nascida em 2013);

Conclusões

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!

Conclusões

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!
- Sua sintaxe é moderna e intuitiva;

Conclusões

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!
- Sua sintaxe é moderna e intuitiva;
- Código aberto, multi-plataforma, e repleta de possibilidades;

Conclusões

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!
- Sua sintaxe é moderna e intuitiva;
- Código aberto, multi-plataforma, e repleta de possibilidades;
- Uso para fins diversos;

Conclusões

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

Conclusões

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

Conclusões

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!
- Sua sintaxe é moderna e intuitiva;
- Código aberto, multi-plataforma, e repleta de possibilidades;
- Uso para fins diversos;
- Muitas bibliotecas específicas prontas: CP, SAT, Planner, etc;
- A sintaxe de PR exige um pouco mais do programador
- Dúvidas: o guia do usuário, livro do Hakan e o Fórum de discussão do Picat

O que ficou faltando:

- Uso do `debug` e `trace` (cansativo – uma oportunidade)

O que ficou faltando:

- Uso do `debug` e `trace` (cansativo – uma oportunidade)
- Explorar uso dos solvers de PO (fácil)

O que ficou faltando:

- Uso do `debug` e `trace` (cansativo – uma oportunidade)
- Explorar uso dos solvers de PO (fácil)
- Explorar a criação e uso de módulos (mais fácil ainda)

O que ficou faltando:

- Uso do `debug` e `trace` (cansativo – uma oportunidade)
- Explorar uso dos solvers de PO (fácil)
- Explorar a criação e uso de módulos (mais fácil ainda)
- Inscreva-se no fórum e consulte o Guia do Usuário (tudo em inglês)

Agradecimentos

- Muito obrigado a voce!

Agradecimentos

- Muito obrigado a voce!
- Algumas pessoas que deram opiniões e me incentivaram a fazer este material

Agradecimentos

- Muito obrigado a voce!
- Algumas pessoas que deram opiniões e me incentivaram a fazer este material
- Claudio Cesar de Sá
- Contacto: `claudio.sa@udesc.br` e `claudio@colmeia.udesc.br`