



PICAT: Uma Linguagem de Programação 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 Tecnologias
Universidade do Estado de Santa Catarina

18 de abril de 2019



Contribuições

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



Predicados e Funções em Picat I

- Os **predicados** assumem valores sempre valores lógicos true (1) ou false (0).
- Os **predicados** em seus argumentos, podem passar n -termos e receber outros termos.
- Quanto as **funções**, estas funcionam seguindo as regras de funções matemáticas, sempre retornando um único valor
- Predicados e funções são definidos com regras de *casamento de padrões*
- Há dois tipos de regras:



Predicados e Funções em Picat II

- Regras com *backtracking*:

Cabeça, *Cond* **\Rightarrow** *Corpo*

- A identificação da sintaxe é dada por:

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

Forma geral:

regra(*termo*₁, ..., *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 em Picat III

- 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**



Predicados 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").
```

$\text{um_predicado}(X,Y,Z) \Rightarrow Z = X + Y.$

$\text{uma_funcao}(X,Y) = R \Rightarrow R = X + Y.$



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.
- Quanto ao *casamento de padrões*:
 - Dado um padrão $p_1(t_1, \dots, t_m)$, este *casa* com um padrão semelhante $p_2(u_1, \dots, u_n)$ se:
 - p_1 e p_2 forem átomos equivalentes;
 - O número de termos (chamado de aridade) em (t_1, \dots, t_m) e (u_1, \dots, u_n) for equivalente.
 - Os termos (t_1, \dots, t_m) e (u_1, \dots, 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 $p_1(t_1, \dots, t_m)$ casa com o padrão $p_2(u_1, \dots, u_n)$.



Exemplos de *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 *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*), $\text{Termo} \geq 0$ pode casar com:
fatorial(1, 1), *fatorial*(5, 120), *fatorial*(*X*, *Y*), *fatorial*(*Z*, *Z*),
etc.



Exemplos de *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*), $\text{Termo} \geq 0$ 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 *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 *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* ≥ 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 – (*goals*)

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



Exemplo – Função e Predicado

```
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>
```




Predicados I

- Forma geral de um predicado:
 $Cabea, Cond \Rightarrow Corpo.$
- Forma geral de um predicado com *backtracking*:
 $Cabea, Cond \text{ ?}\Rightarrow Corpo$



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 dos 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 **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 do predicado, ou no escopo onde este predicado foi chamado.



Predicados do Tipo Fatos I

- As regras que não tem condições no corpo, estes predicados são conhecidos como: *fatos*
- Isto é, *fatos* são regras *sempre verdadeiras*
- Os fatos são do tipo:

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

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



Predicados do Tipo Fatos II

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

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

- Onde um M_{ij} com o símbolo $+$, significa que este termo já foi indexado.
- Quanto o $-$ significa que este termo deve ser indexado



Predicados do Tipo Fatos III

- Ou seja, quando ocorre um simbolo $+$ em um grupo do *index*, é avaliado pelo compilador como um valor constante a ser casada
- 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
- O parâmetro $-$ no *index* é quase como regra geral



Predicados do Tipo Fatos IV

- Não pode haver um **predicado** e um **predicado fato** com mesmo nome.



Exemplo – Função e Predicado

```
index (+,+,+) (+,+, -) (-,+, -) (-,-,+) (-,-, +)
      (+,-,+) (+,+,-) (-,-,-)
%(-,-,-) %% NENHUM argumento instanciado -- 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").
```




Funções

- A forma geral de uma função é:

Cabeça = $X \Rightarrow$ *Corpo*.



Funções

- A forma geral de uma função é:

$$\text{Cabeça} = X \Rightarrow \text{Corpo}.$$
- Caso tenhamos uma condição *Cond*::

$$\text{Cabeça}, \text{Cond} = X \Rightarrow \text{Corpo}.$$
- Funções **não** admitem *backtracking*.



Funções

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



Funções

- 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

- 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 $f(t_1, \dots, t_n) = X$, 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

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



Exemplos

Exemplo de Predicado

```

1  contas_P0(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_P0(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.
    
```



Comandos Condicionais, Laços e Repetições

- 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.



Comandos Condicionais, Laços e Repetições

- 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.



Comandos Condicionais, Laços e Repetições

- 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.



Comandos Condicionais, Laços e Repetições

- Um *loop while* tem a seguinte forma:

```
while (Cond)
    Metas
end
```
- Enquanto a expressão lógica *Cond* for verdadeira, *Metas* será executado.
- 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:

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

- 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:

$$\{T : E_1 \text{ in } D_1, Cond_1, \dots, E_n \text{ in } D_n, Cond_n\}$$



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:
 - $\text{read_int}(FD) = \text{Int} \Rightarrow$ Lê um *Int* do arquivo *FD*.
 - $\text{read_real}(FD) = \text{Real} \Rightarrow$ Lê um *Float* do arquivo *FD*.
 - $\text{read_char}(FD) = \text{Char} \Rightarrow$ Lê um *Char* do arquivo *FD*.
 - $\text{read_line}(FD) = \text{String} \Rightarrow$ 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) ⇒` Escreve um termo T no arquivo FD .
 - `writeln(FD, T) ⇒` Escreve um termo T no arquivo FD , e pula uma linha ao final do termo.
 - `writeln(FD, F, A...) ⇒` 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.



Funções/Predicados Especiais V

- Analogamente, para o predicado `print`, temos:
 - `print(FD, T) ⇒` Escreve um termo T no arquivo FD .
 - `println(FD, T) ⇒` Escreve um termo T no arquivo FD , e pula uma linha ao final do termo.
 - `printf(FD, F, A...)` ⇒ 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	Carácter
%d %i	Número Inteiro Com Sinal
%f	Número Real
%n	Nova Linha
%s	<i>String</i>
%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



Exemplos III

```

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

```




Reflexões

- Este conteúdo trata da sintaxe do Picat



Reflexões

- Este conteúdo trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser pratica



Reflexões

- Este conteúdo trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser pratica
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.



Reflexões

- Este conteúdo trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser pratica
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Mãos a obra!