

PICAT: Uma Linguagem de Programação Multiparadigma

Claudio Cesar de Sá

`claudio.sa@udesc.br`

Departamento de Ciência da Computação – DCC
Centro de Ciências e Tecnologias – CCT
Universidade do Estado de Santa Catarina – UDESC

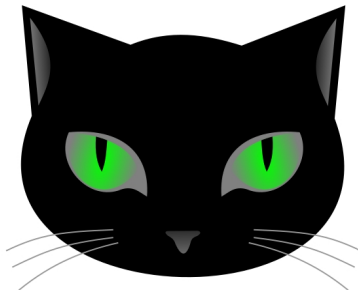
3 de maio de 2019



- 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 Henrique Faes Battisti
- Paulo Victor de Aguiar
- Rogério Eduardo da Silva
- Outros anônimos que auxiliaram na produção deste documento



- Contextualizar a recursão
- Princípios
- 03 exemplos
- *Backtracking*



- A *recursão* é um importante conceito da matemática e presente em muitas linguagens de programação. Exemplo: LISP, Haskell, etc



- A *recursão* é um importante conceito da matemática e presente em muitas linguagens de programação. Exemplo: LISP, Haskell, etc
- Permite expressar conceitos complexos em uma sintaxe abstrata, mas simples de ler.



- A *recursão* é um importante conceito da matemática e presente em muitas linguagens de programação. Exemplo: LISP, Haskell, etc
- Permite expressar conceitos complexos em uma sintaxe abstrata, mas simples de ler.
- Uma regra é dita recursiva quando ela faz auto-referência.



- A *recursão* é um importante conceito da matemática e presente em muitas linguagens de programação. Exemplo: LISP, Haskell, etc
- Permite expressar conceitos complexos em uma sintaxe abstrata, mas simples de ler.
- Uma regra é dita recursiva quando ela faz auto-referência.
- Em Picat, a recursão pode ser usada sob uma notação em *lógica* ou *funcional*



- A *recursão* é um importante conceito da matemática e presente em muitas linguagens de programação. Exemplo: LISP, Haskell, etc
- Permite expressar conceitos complexos em uma sintaxe abstrata, mas simples de ler.
- Uma regra é dita recursiva quando ela faz auto-referência.
- Em Picat, a recursão pode ser usada sob uma notação em *lógica* ou *funcional*
- A funcional apresenta muita clareza ao código!



Somatório dos N naturais

O somatório dos n primeiros números naturais é recursivamente definido como a soma de todos $n-1$ números, mais o termo n . Ou seja:

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

Ou seja:

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



Fatorial

O Fatorial de um número n é definido recursivamente pela multiplicação do fatorial do termo $n - 1$ por n . O fatorial só pode ser calculado para números positivos. Adicionalmente, o fatorial de 0 é igual a 1 por definição.

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

Portanto:

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



Sequência Fibonacci

A sequência Fibonacci é um número calculado a partir da soma dos dois últimos números anteriores a este. Ou seja, o n – *esimo* termo da sequência Fibonacci é definido como a soma dos termos $n - 1$ e $n - 2$. Por definição: os dois primeiros termos, $n = 0$ e $n = 1$ são respectivamente, 0 e 1.

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



- Podemos perceber algo em comum entre estas três definições matemáticas. Todas tem uma ou mais condições que sempre tem o mesmo valor de retorno, ou seja, todas tem uma *regra de aterramento*.



- Podemos perceber algo em comum entre estas três definições matemáticas. 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 (para ou termina).



- Podemos perceber algo em comum entre estas três definições matemáticas. 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 (para ou termina).
- Caso uma regra não tenha uma ou mais *regras de aterramento*, pode ocorrer uma recursão infinita deste regra (infinitas chamadas recursivas sobre a mesma regra provocando um *estouro* da pilha de execução).



Numa visão funcional, estas definições matemáticas podem ser transcritas em Picat como:

```
1 soma_0_N(0) = 0.  
2 soma_0_N(1) = 1.  
3 soma_0_N(N) = N + soma_0_N(N-1).
```

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

```
1 fiboNacci(0) = 0.  
2 fiboNacci(1) = 1.  
3 fiboNacci(N) = fiboNacci(N-1) + fiboNacci(N-2).
```



Numa visão funcional, estas definições matemáticas podem ser transcritas em Picat como:

```
1 main =>
2     writeln( fat = fatorial ( 8 ) ) ,
3     writeln( fib = fiboNacci ( 9 ) ) ,
4     writeln( soma = soma_0_N (10) ) .
5
6 .....
7
8 $ picat recursao_ex_02.pi
9 fat = 40320
10 fib = 34
11 soma = 55
12 $
```



- Caso a definição do fatorial fosse modificada para:

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



- Caso a definição do fatorial fosse modificada para:

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

- Teríamos um caso de *recursão infinita*, pois a regra Fatorial continuaria a ser chamada com $n < 0$
- Nesse caso haveria um erro, pois estaria tentando executar algo indefinido.



Para os exemplos anteriores, reescreva-os as formulações sob uma visão *lógica* e *procedural*.



- O mecanismo de *backtracking* é bem conhecido por algumas linguagens de programação



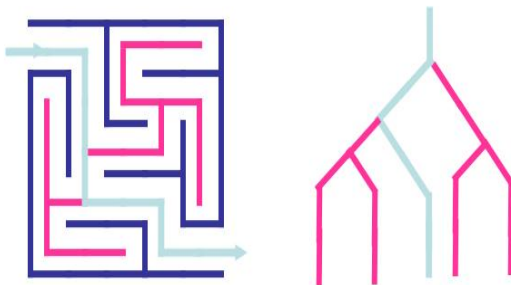
- O mecanismo de *backtracking* é bem conhecido por algumas linguagens de programação
- Em Picat, o *backtracking* é controlável e é habilitado pelo símbolo `?=>` no escopo da regra.



Ilustrando o *Backtracking* no Labirinto

Backtracking Implementation

Backtracking is a modified depth-first search of the solution-space tree. In the case of the maze the start location is the root of a tree, that branches at each point in the maze where there is a choice of direction.



Basicamente o procedimento do *Backtracking* é definido por:

- 1 Inicia-se por um casamento de um predicado *backtrackable* p com um outro predicado p .
- 2 Segue-se a execução da regra p , executando a instância das variáveis da esquerda para direita. **Exemplo** (ilustrativo):

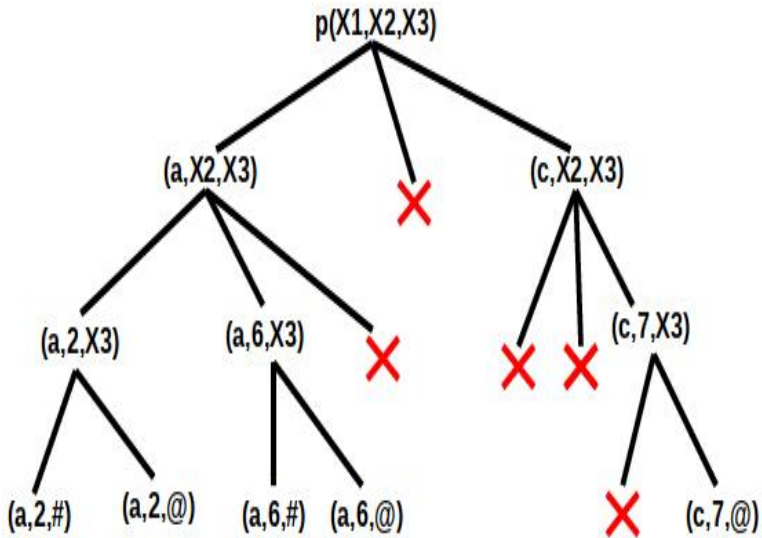
$$p(X_1, X_2, X_3, \dots, X_n) \Rightarrow q_1(X_1), q_2(X_2), \dots, q_n(X_n).$$
- 3 Caso ocorra uma falha durante a execução da regra p , o compilador busca re-instanciar as variáveis do corpo de p que falharem. Esta tentativa segue uma ordem:
 $q_1(X_1) \rightarrow q_2(X_2) \rightarrow \dots \rightarrow q_n(X_n)$, até a variável X_n



- 4 Caso X_n seja instanciada com sucesso, tem-se uma resposta consistente para p
- 5 No caso de uma falha completa na regra corrente p , segue-se para uma próxima regra p ($p \dots ? \Rightarrow \dots$), a qual é avaliada com novas instâncias as suas variáveis.
- 6 Este processo é completo (exaustivo) e se repete até não for mais possível a reinstanciação de variáveis, ou ocorrer uma falha durante a execução.

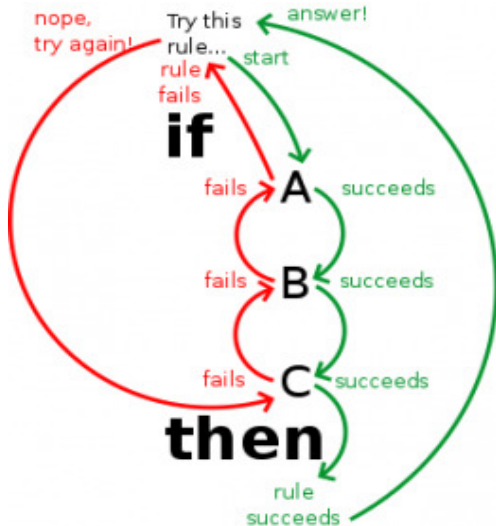


Ilustra o *Backtracking* – Árvore de Busca



Exercício: descubra os domínios possíveis de X_1 , X_2 e X_3 .

Ilustra a Operacionalidade do *Backtracking*



Exemplo 01 – Árvore Geneológica – I

Código: .../picat/uso_ancestral_recurso.pi

```
1 index (-,-) (+,-) (-,+)
2 ancestral(ana,maria).
3 ancestral(pedro,maria).
4 ancestral(maria,paula).
5 ancestral(paula,lucas).
6 ancestral(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 ..... continua
```



Exemplo 01 – Árvore Geneológica – II

```
1 .....
2 mae(X,Y) => ancestral(X,Y), mulher(X).
3 pai(X,Y) => ancestral(X,Y), homem(X).
4 avos(X,Y) => ancestral(X,Z), ancestral(Z,Y).
5
6 descende_de(X,Y) ?=> ancestral(Y,X).
7 descende_de(X,Y) => ancestral(Y,Z), descende_de(X,Z).
8
9 main ?=>
10     descende_de(X,Y),
11     printf("\n => %w descende de %w" , X,Y),
12     false.
13 main => true.
```



- Uma chamada do tipo $mae(maria, X)$, seria como perguntar ao compilador "*Maria é mãe de quem ?*".
- Nesse caso o compilador testa possíveis valores que pudessem ser unificados com X satisfazendo 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 ?".



Código: .../picat/backtracking_ex_02.pi

```
% $ picat backtracking_ex_02.pi
% cl('backtracking_ex_02').
%%% FATOS ...
index(-) % fatos instanciados como retorno
    p(1).  p(3).  p(5).

index(-) % fatos instanciados como retorno
    q(7).  q(4).  q(13).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
algum_num(X, Y, Z) ?=> % ? esta regra tem backtracking
    p(X),
    q(Y),
    Z = 3,
    ((X + Y) mod Z) == 0.

algum_num(X, Y, Z) ?=> % ? esta regra tem backtracking
    p(X),
    q(Y),
    Z = 4,
    ((X + Y) mod Z) == 0.
..... continua
```



```
.....
algum_num(X, Y, Z) => % esta regra NAO tem backtracking
    p(X),
    q(Y),
    Z = 5,
    ((X + Y) mod Z) == 0.

% CUIDAR AQUI
%algum_num(_, _, _) =>
%    printf("\n NAO HA MAIS MULTIPLOS de 3, 4 ou 5").
%%%%%%%%%%%%%%n%%%%%%%%%%%%%% %%%%%%%%%%%%%%%
main ?=> % main com backtracking
    algum_num( X, Y, Z),
    imp(X,Y,Z),
    false.      % força TODAS respostas

main => imp_tracejados(40).
```



```
$ picat backtracking_ex_02.pi
```

```
X:5  Y:7  X+Y:12 é MULTIPL0 de:3  
X:5  Y:4  X+Y:9  é MULTIPL0 de:3  
X:5  Y:13 X+Y:18 é MULTIPL0 de:3  
X:1  Y:7  X+Y:8  é MULTIPL0 de:4  
X:3  Y:13 X+Y:16 é MULTIPL0 de:4  
X:5  Y:7  X+Y:12 é MULTIPL0 de:4  
X:1  Y:4  X+Y:5  é MULTIPL0 de:5  
X:3  Y:7  X+Y:10 é MULTIPL0 de:5
```

```
=====  
$
```

Cuidar em confundir 0 (zero) com O. Perdi horas no código acima.



- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc



- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que **sempre vem antes** das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados (?=>)



- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que **sempre vem antes** das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados (\Rightarrow)
- A avaliação destas regras **são sempre da esquerda para direita**, ocorrendo o *backtracking* em caso de falha ou de uma nova resposta



- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que **sempre vem antes** das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados ($?=>$)
- A avaliação destas regras **são sempre da esquerda para direita**, ocorrendo o *backtracking* em caso de falha ou de uma nova resposta
- As regras recursivas com *backtracking* habilitados ($?=>$), apenas para regras predicativas. As funções não admitem *backtracking*!



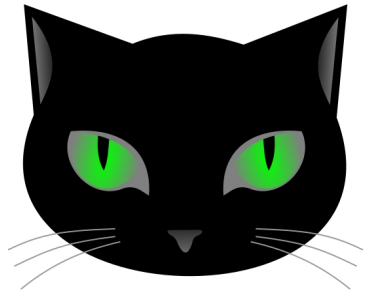
- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que **sempre vem antes** das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados ($?=>$)
- A avaliação destas regras **são sempre da esquerda para direita**, ocorrendo o *backtracking* em caso de falha ou de uma nova resposta
- As regras recursivas com *backtracking* habilitados ($?=>$), apenas para regras predicativas. As funções não admitem *backtracking*!
- A metodologia destas regras e sua construção, seguem esquemas mais avançados da programação declarativa



- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que **sempre vem antes** das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados ($?=>$)
- A avaliação destas regras **são sempre da esquerda para direita**, ocorrendo o *backtracking* em caso de falha ou de uma nova resposta
- As regras recursivas com *backtracking* habilitados ($?=>$), apenas para regras predicativas. As funções não admitem *backtracking*!
- A metodologia destas regras e sua construção, seguem esquemas mais avançados da programação declarativa
- O *main* é uma regra predicativa que pode conter ou não argumentos.



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



- Picat é jovem (nascida em 2013);



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



- 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;



- 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;



- 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;



- 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;



- 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



- 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



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



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



- 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)



- Use o interpretador e o compilador concomitantemente. O interpretador sempre acusa warnings etc. O modo compilado na console, não.



- Use o interpretador e o compilador concomitantemente. O interpretador sempre acusa warnings etc. O modo compilado na console, não.
- No modo interpretado, cada linha de código pode ser testada isoladamente, assim, o efeito global desta é restrita. Qualquer erro ou falha é rapidamente detectada.



- Use o interpretador e o compilador concomitantemente. O interpretador sempre acusa warnings etc. O modo compilado na console, não.
- No modo interpretado, cada linha de código pode ser testada isoladamente, assim, o efeito global desta é restrita. Qualquer erro ou falha é rapidamente detectada.
- Consulte o manual do usuário *on-line* em *html*, mantido pelo Alexandre Gonçalves, UFSC
http://retina.inf.ufsc.br/picat_guide



- Use o interpretador e o compilador concomitantemente. O interpretador sempre acusa warnings etc. O modo compilado na console, não.
- No modo interpretado, cada linha de código pode ser testada isoladamente, assim, o efeito global desta é restrita. Qualquer erro ou falha é rapidamente detectada.
- Consulte o manual do usuário *on-line* em *html*, mantido pelo Alexandre Gonçalves, UFSC
http://retina.inf.ufsc.br/picat_guide
- Consulte o site do Picat e dos grandes mestres Hakan, Neng-Fa, Roman Barták, Sergii Dymchenko, etc



- Use o interpretador e o compilador concomitantemente. O interpretador sempre acusa warnings etc. O modo compilado na console, não.
- No modo interpretado, cada linha de código pode ser testada isoladamente, assim, o efeito global desta é restrita. Qualquer erro ou falha é rapidamente detectada.
- Consulte o manual do usuário *on-line* em *html*, mantido pelo Alexandre Gonçalves, UFSC
http://retina.inf.ufsc.br/picat_guide
- Consulte o site do Picat e dos grandes mestres Hakan, Neng-Fa, Roman Barták, Sergii Dymchenko, etc
- Inscreva-se no fórum e consulte o Guia do Usuário (tudo em inglês)



- Muito obrigado a voce!



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



- 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`

