PICAT: Uma Linguagem de Programação Multiparadigma

Claudio Cesar de Sá

claudio.sa@udesc.br

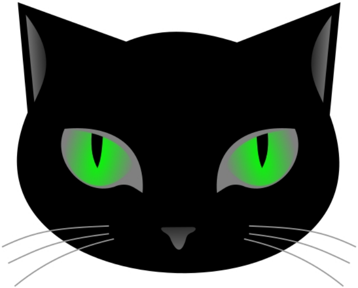
Departamento de Ciência da Computação – DCC
Centro de Ciências e Tecnológias – CCT
Universidade do Estado de Santa Catarina – UDESC

5 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

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



Recursão

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

Recursão

- 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.

Recursão

- 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*

Recursão

- 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!

Conceitos de Recursividade via Exemplos – I

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:

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

Ou seja:

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

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.

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

Portanto:

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

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.

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

Conceitos de Recursividade via Exemplos – IV

- 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*.

Conceitos de Recursividade via Exemplos – IV

- 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 (pára ou termina).

Conceitos de Recursividade via Exemplos – IV

- 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 (pára 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).

Exemplos

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

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

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

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

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

```
main =>
        writeln( fat = fatorial ( 8 ) ) ,
        writeln( fib = fiboNacci ( 9 ) ) ,
        writeln( soma = soma_0_N (10) )  .

...............................................

$ picat recursao_ex_02.pi
fat = 40320
fib = 34
soma = 55
$
```

Recursão Infinita

- Caso a definição do fatorial fosse modificada para:

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

Recursão Infinita

- Caso a definição do fatorial fosse modificada para:

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

- Teríamos um caso de *recursão infinita*, pois a regra Fatorial 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*.

Backtracking

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

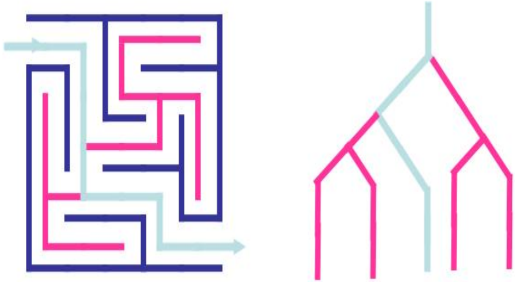
Backtracking

- 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.

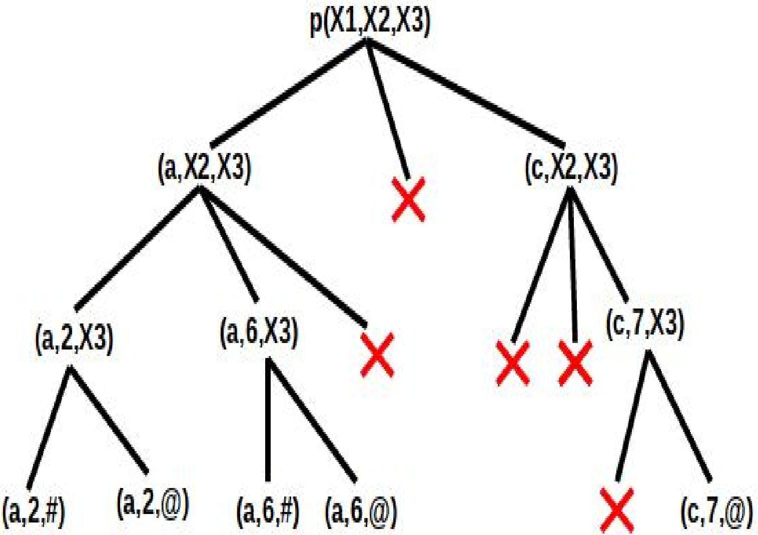


Basicamente o procedimento do *Backtracking* é definido por:

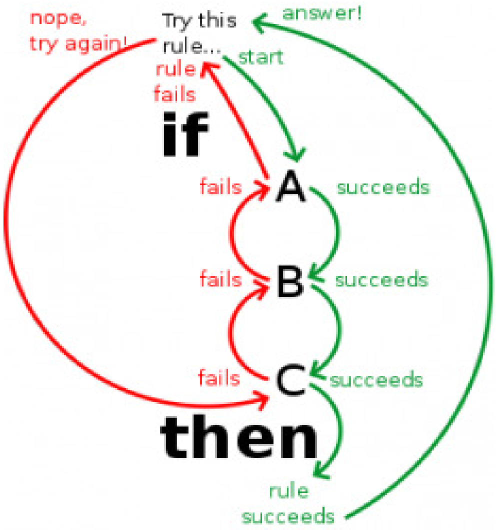
1. Inicia-se por um casamento de um predicado *backtrackable* p com um outro predicado p.
2. Segue-se a execução da regra p, executando a instância das variáveis da esquerda para direita. **Exemplo** (ilustrativo): `p(X1,X2,X3, ....., Xn) ?=> q1(X1), q2(X2), ...., qn(Xn).`
3. Caso ocorra uma falha durante a execução da regra p, o compilador busca re-instanciar as variáveis do corpo de p que falharem. Esta tentativa segue uma ordem: $q1(X1) \rightarrow q2(X2) \rightarrow .... \rightarrow qn(Xn)$, até a variável Xn

4. Caso *Xn* 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 ....?=> ...`), 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 `X1`, `X2` e `X3`


nope,
try again!
Try this
rule...
answer!
start
rule
fails
if
fails
A
succeeds
fails
B
succeeds
fails
C
succeeds
then
rule
succeeds

Exemplo 01 – Árvore Geneológica – I

Código: `.../picat/uso_ancestral_recursao.pi`

```
index (-,-) (+,-) (-,+)
ancestral(ana,maria).
ancestral(pedro,maria).
ancestral(maria,paula).
ancestral(paula,lucas).
ancestral(lucas, eduarda).

index (-)
mulher(ana).
mulher(maria).
mulher(paula).
mulher(eduarda).
homem(pedro).
homem(lucas).

.............................................continua
```

Exemplo 01 – Árvore Geneológica – II

```
....................................................
mae(X,Y) => ancestral(X,Y), mulher(X).
pai(X,Y) => ancestral(X,Y), homem(X).
avos(X,Y) => ancestral(X,Z), ancestral(Z,Y).

descende_de(X,Y) ?=> ancestral(Y,X).
descende_de(X,Y) => ancestral(Y,Z), descende_de(X,Z).

main ?=>
      descende_de(X,Y),
      printf("\n => %w descende de %w" , X,Y),
      false.
main =>  true.
```

Refletindo sobre o Exemplo

- 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).
```

Exemplo 02 – Saída

```
$ picat backtracking_ex_02.pi

 X:5  Y:7  X+Y:12 é MULTIPLO de:3
 X:5  Y:4  X+Y:9 é MULTIPLO de:3
 X:5  Y:13  X+Y:18 é MULTIPLO de:3
 X:1  Y:7  X+Y:8 é MULTIPLO de:4
 X:3  Y:13  X+Y:16 é MULTIPLO de:4
 X:5  Y:7  X+Y:12 é MULTIPLO de:4
 X:1  Y:4  X+Y:5 é MULTIPLO de:5
 X:3  Y:7  X+Y:10 é MULTIPLO 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 (?=>)

Reflexões

- 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

- 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*!

Reflexões

- 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!