

PICAT: Uma Linguagem de Programação Multiparadigma

Claudio Cesar de Sá

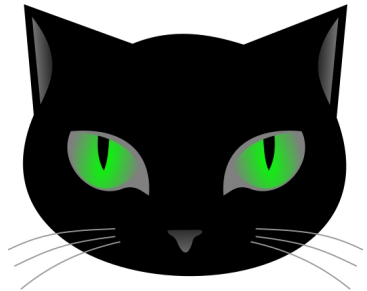
✉ claudio.sa@udesc.br

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

4 de junho de 2019



- O que é depuração?
- Como funciona no Picat?
- Exemplo de uso



- Depuração, *trace* ou *debug* – o que é?



- Depuração, *trace* ou *debug* – o que é? Mostrar estado de variáveis, funções e predicados durante a execução de um código
- Depuração ou *trace* – quando usar?



- Depuração, *trace* ou *debug* – o que é? Mostrar estado de variáveis, funções e predicados durante a execução de um código
- Depuração ou *trace* – quando usar? **Aprender detalhes da linguagem**



- Depuração, *trace* ou *debug* – o que é? Mostrar estado de variáveis, funções e predicados durante a execução de um código
- Depuração ou *trace* – quando usar? **Aprender detalhes da linguagem**
- Depuração ou *trace* \Rightarrow **para descobrir o erro!**



- Depuração, *trace* ou *debug* – o que é? Mostrar estado de variáveis, funções e predicados durante a execução de um código
- Depuração ou *trace* – quando usar? **Aprender detalhes da linguagem**
- Depuração ou *trace* \Rightarrow **para descobrir o erro!**
- Em Picat não há um ambiente gráfico de depuração, ao contrário do SWI-Prolog.



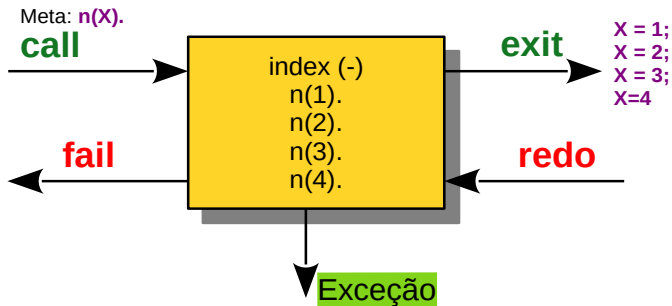
- Depuração, *trace* ou *debug* – o que é? Mostrar estado de variáveis, funções e predicados durante a execução de um código
- Depuração ou *trace* – quando usar? **Aprender detalhes da linguagem**
- Depuração ou *trace* \Rightarrow **para descobrir o erro!**
- Em Picat não há um ambiente gráfico de depuração, ao contrário do SWI-Prolog. Eis uma oportunidade de um projeto aqui....
- Contudo, o *trace* do Picat faz tudo ...



- Depuração, *trace* ou *debug* – o que é? Mostrar estado de variáveis, funções e predicados durante a execução de um código
- Depuração ou *trace* – quando usar? **Aprender detalhes da linguagem**
- Depuração ou *trace* \Rightarrow **para descobrir o erro!**
- Em Picat não há um ambiente gráfico de depuração, ao contrário do SWI-Prolog. Eis uma oportunidade de um projeto aqui....
- Contudo, o *trace* do Picat faz tudo ...



Depuração – Como funciona um predicado?



Depuração – Internamente tem o *backtracking*!

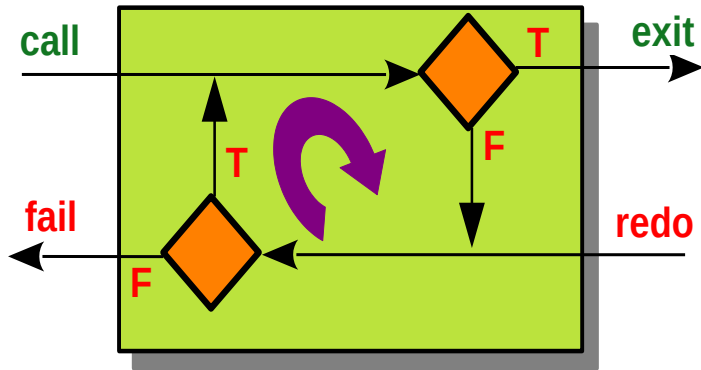
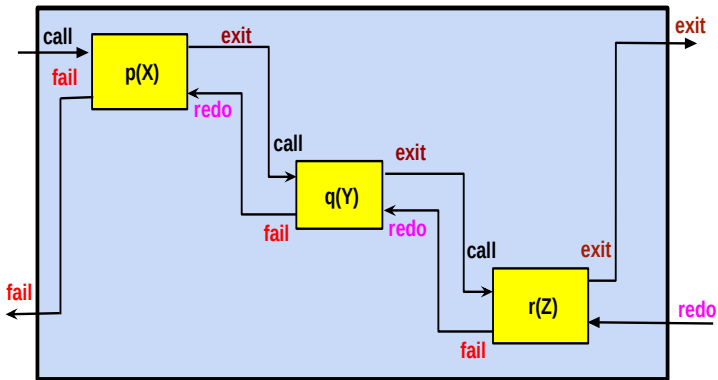


Figura 1: Levemente diferente do que é encontrado na bibliografia!



Depuração – uma regra composta

$\text{regra}(X,Y,Z) \Rightarrow p(X), q(Y), r(Z).$



- Picat tem **3 modos de execução**: *trace*, *non-trace* e *spy*



- Picat tem **3 modos de execução**: *trace*, *non-trace* e *spy*
- *non-trace*: é a execução normal ou *default* – vista até o momento



- Picat tem **3 modos de execução**: *trace*, *non-trace* e *spy*
- *non-trace*: é a execução normal ou *default* – vista até o momento
- *trace* ou *debug*: vê passo-a-passo de cada variável, predicado ou função, do código inteiro



- Picat tem **3 modos de execução**: *trace*, *non-trace* e *spy*
- *non-trace*: é a execução normal ou *default* – vista até o momento
- *trace* ou *debug*: vê passo-a-passo de cada variável, predicado ou função, do código inteiro
- *spy*: *salta* para um predicado ou função específica, se iniciando a depuração a partir deste ponto



- Picat tem **3 modos de execução**: *trace*, *non-trace* e *spy*
- *non-trace*: é a execução normal ou *default* – vista até o momento
- *trace* ou *debug*: vê passo-a-passo de cada variável, predicado ou função, do código inteiro
- *spy*: *salta* para um predicado ou função específica, se iniciando a depuração a partir deste ponto
- Devem ser habilitados e desabilitados no modo interpretado
- **Há funções e predicados que não tem *backtracking*.**



- Picat tem **3 modos de execução**: *trace*, *non-trace* e *spy*
- *non-trace*: é a execução normal ou *default* – vista até o momento
- *trace* ou *debug*: vê passo-a-passo de cada variável, predicado ou função, do código inteiro
- *spy*: *salta* para um predicado ou função específica, se iniciando a depuração a partir deste ponto
- Devem ser habilitados e desabilitados no modo interpretado
- **Há funções e predicados que não tem *backtracking*.**
Exemplos: `nl`, `printf`, `read_...`, etc



- 1 Entre no interpretador do Picat
- 2 Ative o modo *trace* (ou *debug*), com o comando: *trace*
- 3 Compile e carregue o programa desejado a depurar, com o comando: `cl('....')`.
- 4 Execute a parte desejada do código
- 5 Vá dando Enter no modo *passo-a-passo*
- 6 Digite 'h' para o help e ver opções disponíveis
- 7 Digite 'a' para o abortar o modo trace

Em qualquer modificação do código fonte no editor, repita os passos acima, desde o início!



```

main ?=>
    %%%% ACHE X e Y tal que  X + Y = 22
    regra(X,Y, 22) ,
    printf("SAIDAS ==> X: %w Y: %w \n", X, Y) ,
    false.

main => printf("\n Não há mais soluções! \n").
%%%%%%%%%%%%%%
regra(X,Y, R) =>
    p(X),
    q(Y),
    R = X + Y.
%%%%%%%%%%%%%%
index(-)
    p(2).      p(1).      p(0).
index(-)
    q(20).     q(22).
%%%%%%%%%%%%%%

```



```

Picat> trace
Note: you need to recompile programs in debug mode for tracing
yes
{Trace mode}
Picat> cl('trace_exemplo_01')
Compiling:: trace_exemplo_01.pi
trace_exemplo_01.pi compiled in 3 milliseconds
loading...
yes
{Trace mode}
Picat> q(X), writeln(x = X), false.
    Call: (1) q(_10b38) ?
? Exit: (1) q(20) ?
    Call: (2) writeln(x = 20) ?
x = 20
    Exit: (2) writeln(x = 20) ?
    Redo: (1) q(20) ?          >>>> DEVIDO o false
    Exit: (1) q(22) ?
    Call: (3) writeln(x = 22) ?
x = 22
    Exit: (3) writeln(x = 22) ?
no
{Trace mode}

```



Considerações de Uso: *trace* e *spy*

- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro



Considerações de Uso: *trace* e *spy*

- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro
- Os números após a chamadas, representam a sequência na pilha de cada predicado!



- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro
- Os números após a chamadas, representam a sequência na pilha de cada predicado!
- A idéia de se programar com Picat e outras linguagens declarativas é:



- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro
- Os números após a chamadas, representam a sequência na pilha de cada predicado!
- A idéia de se programar com Picat e outras linguagens declarativas é:
a cada linha de código escrita segue-se por um teste isolado!



- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro
- Os números após a chamadas, representam a sequência na pilha de cada predicado!
- A idéia de se programar com Picat e outras linguagens declarativas é:
a cada linha de código escrita segue-se por um teste isolado!
- Ou seja, use o *trace* e o *spy* em partes do código!



- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro
- Os números após a chamadas, representam a sequência na pilha de cada predicado!
- A idéia de se programar com Picat e outras linguagens declarativas é:
a cada linha de código escrita segue-se por um teste isolado!
- Ou seja, use o *trace* e o *spy* em partes do código!
- Nos modos *trace* e *spy*, veja as opções do *help* ou (h). Experimente-as sem receios. Em caso de dúvida: a (*abort*)!



- Estes comandos são úteis e se encontram em modo *beta* de uso
- Mas, tem **TUDO** que se precisa para descobrir um erro
- Os números após a chamadas, representam a sequência na pilha de cada predicado!
- A idéia de se programar com Picat e outras linguagens declarativas é:
a cada linha de código escrita segue-se por um teste isolado!
- Ou seja, use o *trace* e o *spy* em partes do código!
- Nos modos *trace* e *spy*, veja as opções do *help* ou (h). Experimente-as sem receios. Em caso de dúvida: a (*abort*)!
- Ao se ganhar confiança com a linguagem, voce vai omitindo alguns pontos!



- Se aprende os detalhes da linguagem com um *trace*



- Se aprende os detalhes da linguagem com um *trace*
- Não se assuste com modo trace na console



- Se aprende os detalhes da linguagem com um *trace*
- Não se assuste com modo trace na console
- Atenção na sequência de uso:
 - 1 Ative o modo *trace*
 - 2 Compile e carregue o seu fonte no modo trace, com o comando `c1('.....')`.
 - 3 Execute o que desejares do código ou ative alguns *spy*
 - 4 Para desativar *notrace*



- Se aprende os detalhes da linguagem com um *trace*
- Não se assuste com modo trace na console
- Atenção na sequência de uso:
 - 1 Ative o modo *trace*
 - 2 Compile e carregue o seu fonte no modo trace, com o comando `c1('.....')`.
 - 3 Execute o que desejares do código ou ative alguns *spy*
 - 4 Para desativar *notrace*

Em caso de mudança no código, repita os passos acima



- Se aprende os detalhes da linguagem com um *trace*
- Não se assuste com modo *trace* na console
- Atenção na sequência de uso:
 - 1 Ative o modo *trace*
 - 2 Compile e carregue o seu fonte no modo *trace*, com o comando `c1('.....')`.
 - 3 Execute o que desejares do código ou ative alguns *spy*
 - 4 Para desativar *notrace*

Em caso de mudança no código, repita os passos acima

- Outra alternativa ao *trace* \Rightarrow modo *spy*, para desativar *nospy*



- Se aprende os detalhes da linguagem com um *trace*
- Não se assuste com modo *trace* na console
- Atenção na sequência de uso:
 - ① Ative o modo *trace*
 - ② Compile e carregue o seu fonte no modo *trace*, com o comando `c1('.....')`.
 - ③ Execute o que desejares do código ou ative alguns *spy*
 - ④ Para desativar *notrace*

Em caso de mudança no código, repita os passos acima

- Outra alternativa ao *trace* \Rightarrow modo *spy*, para desativar *nospy*
- Finalmente, *boas depurações!*

