

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

7 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



1 Introdução

Estrutura da Linguagem

Paradigmas

Usando Picat

2 Tipos de Dados e Variáveis

Tipos de Dados

Variáveis

Unificação e Atribuição

Tabela de Operadores

Operadores Especiais

3 Predicados e Funções

Casamento de Padrões

Funções

Relembrando as Regras

Regras do Tipo Fatos

Exemplos

4 Estruturas de Decisão, Laços e Iteradores



Funções e Predicados Especiais

5 Recursão

Recursão

Backtracking

6 Listas

7 Buscas

8 Programação Dinâmica

9 Planejamento

10 Programação por Restrições

11 Conclusões

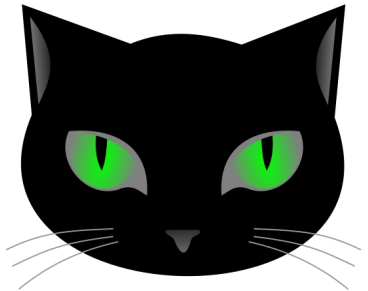
Faltando

Dicas de Programação

Agradecimentos



- Histórico
- Contexto
- Exemplo: *Alo Mundo*
- Como usar
- Site e recursos



- Criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman
- Utiliza o B-Prolog como base de implementação, tendo a Lógica de Primeira-Ordem (LPO) como parte de seu mecanismo programação



- Criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman
- Utiliza o B-Prolog como base de implementação, tendo a Lógica de Primeira-Ordem (LPO) como parte de seu mecanismo programação
- Uma evolução ao Prolog após seus mais de 40 anos de sucesso!



- Criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman
- Utiliza o B-Prolog como base de implementação, tendo a Lógica de Primeira-Ordem (LPO) como parte de seu mecanismo programação
- Uma evolução ao Prolog após seus mais de 40 anos de sucesso!
- Sua atual versão é a 2.x (7 de maio de 2019).



- Criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman
- Utiliza o B-Prolog como base de implementação, tendo a Lógica de Primeira-Ordem (LPO) como parte de seu mecanismo programação
- Uma evolução ao Prolog após seus mais de 40 anos de sucesso!
- Sua atual versão é a 2.x (7 de maio de 2019).
- Código-aberto, segue as regras da FSF



- Picat é uma linguagem de programação simples de usar, poderosa e multi-uso
- Alguma de suas características são associadas com linguagens lógicas, como Prolog, B-Prolog, Goedel, etc



- Picat é uma linguagem de programação simples de usar, poderosa e multi-uso
- Algumas de suas características são associadas com linguagens lógicas, como Prolog, B-Prolog, Goedel, etc
- Picat é uma linguagem essencialmente multiparadigma, abrangendo partes de vários paradigmas de programação: declarativo (lógico e funcional) e imperativo



O que é ser Multiparadigma ?

- Paradigma: um conjunto de características baseado em alguma abordagem teórica



O que é ser Multiparadigma ?

- Paradigma: um conjunto de características baseado em alguma abordagem teórica
- Picat é uma linguagem multiparadigma pois abrange os seguintes paradigmas:
 - Lógico
 - Funcional
 - Procedural



O que é ser Multiparadigma ?

- Paradigma: um conjunto de características baseado em alguma abordagem teórica
- Picat é uma linguagem multiparadigma pois abrange os seguintes paradigmas:
 - Lógico
 - Funcional
 - Procedural
- Em resumo, *uma boa mistura* de: Haskell (Funcional) , Prolog (Lógica) e Python (Procedural e Funcional).



- Uma linguagem lógica é uma onde o programa é expresso como um conjunto de predicados lógicos, escritos por *fatos* e *regras*



- Uma linguagem lógica é uma onde o programa é expresso como um conjunto de predicados lógicos, escritos por *fatos* e *regras*
- Regras são escritas em formas de cláusulas, as quais são interpretadas como implicações lógicas.
Dependem das premissas serem verdadeiras para esta ser verdadeira.



- Uma linguagem lógica é uma onde o programa é expresso como um conjunto de predicados lógicos, escritos por *fatos* e *regras*
- Regras são escritas em formas de cláusulas, as quais são interpretadas como implicações lógicas.
Dependem das premissas serem verdadeiras para esta ser verdadeira.
- Fatos são cláusulas sem premissas, verdades absolutas.



- Uma linguagem lógica é uma onde o programa é expresso como um conjunto de predicados lógicos, escritos por *fatos* e *regras*
- Regras são escritas em formas de cláusulas, as quais são interpretadas como implicações lógicas.
Dependem das premissas serem verdadeiras para esta ser verdadeira.
- Fatos são cláusulas sem premissas, verdades absolutas.
- Este paradigma é a **base** do Picat



- Uma linguagem funcional é uma onde os elementos do programa podem ser avaliados e tratados como funções matemáticas.



- Uma linguagem funcional é uma onde os elementos do programa podem ser avaliados e tratados como funções matemáticas.
- Um dos principais motivos em usar linguagens funcionais é a previsibilidade e facilidade no entendimento do estado atual do programa.



- Uma linguagem funcional é uma onde os elementos do programa podem ser avaliados e tratados como funções matemáticas.
- Um dos principais motivos em usar linguagens funcionais é a previsibilidade e facilidade no entendimento do estado atual do programa.
- Este fato de uma sintaxe simples, torna o Picat intuitivo e legível na funcionalidade de seus códigos.



- Uma linguagem procedural é uma que pode ser subdividida em *procedimentos*, também chamados de rotinas, subrotinas ou funções



- Uma linguagem procedural é uma que pode ser subdividida em *procedimentos*, também chamados de rotinas, subrotinas ou funções
- Em linguagens procedurais há um procedimento principal (em geral é chamado de *Main*) que controla o uso e a chamada de outros procedimentos. Em Picat há tal hierarquia.



- Uma linguagem procedural é uma que pode ser subdividida em *procedimentos*, também chamados de rotinas, subrotinas ou funções
- Em linguagens procedurais há um procedimento principal (em geral é chamado de *Main*) que controla o uso e a chamada de outros procedimentos. Em Picat há tal hierarquia.
- Em Picat, cada premissa é tratada como um procedimento, que é resolvido por meio de métodos de inferência lógica.

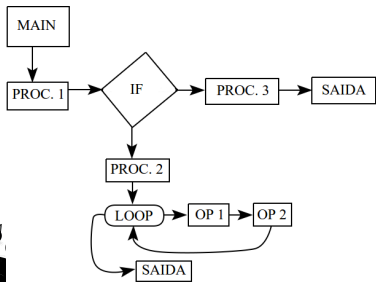


Figura 1: Fluxograma representando a estrutura de um programa Procedural



Algumas Características:



- Sintaxe elegante e simples, facilitando a leitura e entendimento do código



- Sintaxe elegante e simples, facilitando a leitura e entendimento do código
- Velocidade de execução em um ambiente *interpretado* (há uma *máquina virtual* como Python, Java e alguns Prologs)



- Sintaxe elegante e simples, facilitando a leitura e entendimento do código
- Velocidade de execução em um ambiente *interpretado* (há uma *máquina virtual* como Python, Java e alguns Prologs)
- Disponibilidade em vários sistemas operacionais e arquiteturas



- Sintaxe elegante e simples, facilitando a leitura e entendimento do código
- Velocidade de execução em um ambiente *interpretado* (há uma *máquina virtual* como Python, Java e alguns Prologs)
- Disponibilidade em vários sistemas operacionais e arquiteturas
- Análogo a Python, podem ser feitas *queries* ou *consultas* ao terminal de Picat.



- Sintaxe elegante e simples, facilitando a leitura e entendimento do código
- Velocidade de execução em um ambiente *interpretado* (há uma *máquina virtual* como Python, Java e alguns Prologs)
- Disponibilidade em vários sistemas operacionais e arquiteturas
- Análogo a Python, podem ser feitas *queries* ou *consultas* ao terminal de Picat.
- Há várias bibliotecas da própria linguagem, e diversas ferramentas externas permitindo o incremento do poder do Picat.



- P:** *Pattern-matching*: Utiliza o conceito de *casamento de padrões* entre objetos, bem como os conceitos da *unificação* da LPO
- I:** *Intuitive*: Oferece estruturas de decisão, atribuição e laços de repetição, etc. Análogo há outras linguagens de programação mais populares
- C:** *Constraints*: Suporta a programação por restrições (PR) para problemas combinatórios
- A:** *Actors*: Suporte as chamadas a eventos, chamada via *atores*
- T:** *Tabling*: Implementa a técnica de *memoization* com soluções imediatas para problemas de Programação Dinâmica (PD).



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS
- Criar um link simbólico (Linux) ou atalhos (Windows):
`ln -s /usr/local/Picat/picat /usr/bin/picat`



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS
- Criar um link simbólico (Linux) ou atalhos (Windows):
`ln -s /usr/local/Picat/picat /usr/bin/picat`
- Se quiser adicionar (opcional) uma variável de ambiente:
`PICATPATH=/usr/local/Picat/`
`export PICATPATH`



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS
- Criar um link simbólico (Linux) ou atalhos (Windows):
`ln -s /usr/local/Picat/picat /usr/bin/picat`
- Se quiser adicionar (opcional) uma variável de ambiente:
`PICATPATH=/usr/local/Picat/`
`export PICATPATH`
- Ou ainda, adicione o caminho:
`PATH=$PATH:/usr/local/Picat`



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS
- Criar um link simbólico (Linux) ou atalhos (Windows):
`ln -s /usr/local/Picat/picat /usr/bin/picat`
- Se quiser adicionar (opcional) uma variável de ambiente:
`PICATPATH=/usr/local/Picat/`
`export PICATPATH`
- Ou ainda, adicione o caminho:
`PATH=$PATH:/usr/local/Picat`
- Finalmente, tenha um editor de texto apropriado.
Sugestão: *Geany*, *Sublime* ou *VS Code*.



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS
- Criar um link simbólico (Linux) ou atalhos (Windows):
`ln -s /usr/local/Picat/picat /usr/bin/picat`
- Se quiser adicionar (opcional) uma variável de ambiente:
`PICATPATH=/usr/local/Picat/`
`export PICATPATH`
- Ou ainda, adicione o caminho:
`PATH=$PATH:/usr/local/Picat`
- Finalmente, tenha um editor de texto apropriado.
Sugestão: *Geany*, *Sublime* ou *VS Code*.
- Editor on-line mantido pelo Alexandre:
`http://retina.inf.ufsc.br/picat.html`



- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/Picat/` ou `/opt/Picat/` no Linux e IOS
- Criar um link simbólico (Linux) ou atalhos (Windows):
`ln -s /usr/local/Picat/picat /usr/bin/picat`
- Se quiser adicionar (opcional) uma variável de ambiente:
`PICATPATH=/usr/local/Picat/`
`export PICATPATH`
- Ou ainda, adicione o caminho:
`PATH=$PATH:/usr/local/Picat`
- Finalmente, tenha um editor de texto apropriado.
Sugestão: *Geany*, *Sublime* ou *VS Code*.
- Editor on-line mantido pelo Alexandre:
`http://retina.inf.ufsc.br/picat.html`
- Se não tiver *plugin* para Picat, escolha a sintaxe da linguagem *Erlang*.



- Os seus arquivos fontes utilizam a extensão **.pi**. Exemplo: `programa.pi`
- Há dois modos principais de utilização do Picat:
 - Modo interativo, onde seu código é digitado e compilado diretamente na linha de comando;
 - *Modo console* onde o console só é utilizado para compilar seus programas.



- Os seus arquivos fontes utilizam a extensão **.pi**. Exemplo: `programa.pi`
- Há dois modos principais de utilização do Picat:
 - Modo interativo, onde seu código é digitado e compilado diretamente na linha de comando;
 - *Modo console* onde o console só é utilizado para compilar seus programas.
- Códigos executáveis 100% **stand-alone**: ainda não!
- Neste quesito, estamos em igualdade com Java, Prolog e Python



Acompanhar as explicações do código de:

https://github.com/claudiosa/CCS/blob/master/picat/alo_mundo.pi

```
main => msg_01  ,  
        msg_02 .
```

```
msg_01 => printf("  ALO MUNDO!!! ").  
msg_02 => printf("\n  FIM \n").
```



Execução na Console Linux ou Windows

```
$ picat alo_mundo.pi  
  ALO MUNDO!!!  
  FIM  
$
```



```
$ picat alo_mundo.pi  
  ALO MUNDO!!!  
  FIM  
$
```

Análogo ao desenvolvimento com Python!



Execução no Ambiente do Interpretador

```
$ picat
Picat 2.0, (C) picat-lang.org, 2013-2016.
Type 'help' for help.
Picat> cl(álo_mundo.pi').
Compiling:: alo_mundo.pi
alo_mundo.pi compiled in 0 milliseconds
loading...
```

yes

```
Picat> main
    ALO MUNDO!!!
    FIM
```

yes

```
Picat> msg_02
```

FIM



yes

Ambiente do Interpretador – Uso do `getline`

- Inicialmente, aqui o código foi carregado com o comando '`c1`' (digite `help` na console), o qual **compila** o seu código e **carrega** em um código intermediário pronto para ser executado e testado



Ambiente do Interpretador – Uso do `getline`

- Inicialmente, aqui o código foi carregado com o comando '`c1`' (digite `help` na console), o qual **compila** o seu código e **carrega** em um código intermediário pronto para ser executado e testado
- Neste ambiente interpretado há comandos básicos de teclado (mouse não funciona aqui) do programa `getline` do Linux. Os mais importantes são:



Ambiente do Interpretador – Uso do getline

- Inicialmente, aqui o código foi carregado com o comando 'c1' (digite `help` na console), o qual **compila** o seu código e **carrega** em um código intermediário pronto para ser executado e testado
- Neste ambiente interpretado há comandos básicos de teclado (mouse não funciona aqui) do programa `getline` do Linux. Os mais importantes são:
 - **Ctrl-a**: move o cursor para o início da linha
 - **Ctrl-e**: move o cursor para o final da linha (*end*)
 - **Ctrl-f**: move o cursor de uma posição a frente (*forward*)
 - **Ctrl-b**: move o cursor de uma posição para trás (*backward*)
 - **Ctrl-d**: exclui o carácter sob o cursor (a 2a. vez – sai do ambiente)
 - **Ctrl-u**: exclui a linha inteira
 - As flechas ... repetem os últimos comandos



- Use um editor externo de sua preferência. Por exemplo: geany com plugin do Picat



- Use um editor externo de sua preferência. Por exemplo: geany com plugin do Picat
- Mantenha duas janelas de terminais abertas
 - Uma para o ambiente interpretado
 - Outra para usá-lo diretamente: `$console$ picat seu_programa.pi`



- Use um editor externo de sua preferência. Por exemplo: geany com plugin do Picat
- Mantenha duas janelas de terminais abertas
 - Uma para o ambiente interpretado
 - Outra para usá-lo diretamente: `$console$ picat seu_programa.pi`
- Os dois modos são importantes de se trabalhar simultaneamente



- Use um editor externo de sua preferência. Por exemplo: geany com plugin do Picat
- Mantenha duas janelas de terminais abertas
 - Uma para o ambiente interpretado
 - Outra para usá-lo diretamente: `$console$ picat seu_programa.pi`
- Os dois modos são importantes de se trabalhar simultaneamente
- Em dúvidas, digite: `Picat> help .`



- O conteúdo desta parte do curso pode ser complementado com a **Videoaula 01: Introdução ao PICAT**, disponível no Youtube:
<https://www.youtube.com/watch?v=0DmTyFFQPK8>



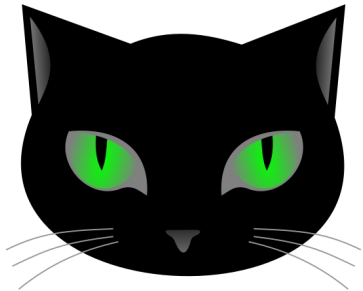
- O conteúdo desta parte do curso pode ser complementado com a **Videoaula 01: Introdução ao PICAT**, disponível no Youtube:
<https://www.youtube.com/watch?v=0DmTyFFQPK8>
- Para próxima seção esteja com o Picat instalado em seu computador para um melhor aproveitamento.



- O conteúdo desta parte do curso pode ser complementado com a **Videoaula 01: Introdução ao PICAT**, disponível no Youtube:
<https://www.youtube.com/watch?v=0DmTyFFQPK8>
- Para próxima seção esteja com o Picat instalado em seu computador para um melhor aproveitamento.
- Em códigos fontes: o símbolo '%' no início de linha, comenta a linha corrente



- Contextualizar
- Definições
- Exemplos



- Em projetos de linguagens de programação há dois tipos de verificação do tipo de dados: estática e dinâmica



- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: estática e dinâmica
- A verificação de tipos dados estática em *tempo de compilação*.



- Em projetos de linguagens de programação há dois tipos verificação do tipo de dados: estática e dinâmica
- A verificação de tipos dados estática em *tempo de compilação*.
- Enquanto a dinâmica em *tempo de execução*.



- Em projetos de linguagens de programação há dois tipos de verificação do tipo de dados: estática e dinâmica
- A verificação de tipos dados estática em *tempo de compilação*.
- Enquanto a dinâmica em *tempo de execução*.
- Linguagens fortemente tipadas, tais como Java, Haskell e Pascal, exigem que o tipo do dado (conteúdo) seja do mesmo tipo da variável ao qual este valor será atribuído. Tudo isto é pré-definido durante a fase da *compilação*.



- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa. Exemplo: Java é uma linguagem com tipagem estática que é executada em uma máquina virtual.



- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa. Exemplo: Java é uma linguagem com tipagem estática que é executada em uma máquina virtual.
- Prós e contras sobre a melhor tipagem de dados, esta é uma discussão que fica de lado neste momento, neste contexto



- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa. Exemplo: Java é uma linguagem com tipagem estática que é executada em uma máquina virtual.
- Prós e contras sobre a melhor tipagem de dados, esta é uma discussão que fica de lado neste momento, neste contexto
- Em resumo Picat, tem uma tipagem **dinâmica** e é compilado para uma máquina virtual própria.



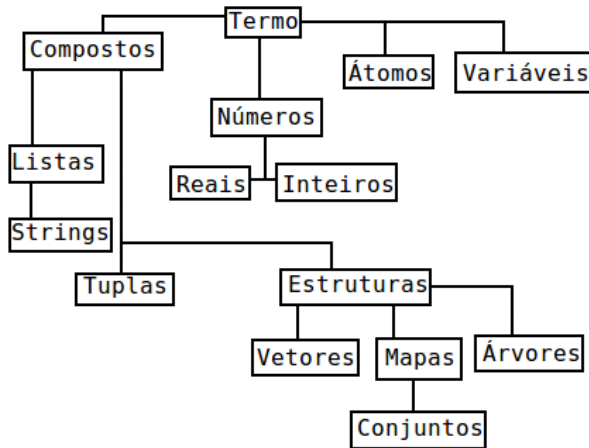


Figura 2: Hierarquia dos Tipos de Dados



- Em Picat, variáveis e valores são *genericamente* chamados de *termos*



- Em Picat, variáveis e valores são *genericamente* chamados de *termos*
- Os valores são subdivididos em duas categorias, números e valores compostos
- Os números, por suas vez, podem ser inteiros ou reais, e valores compostos podem ser listas e estruturas



- Átomos são constantes simbólicas, podendo ser delimitados ou não, por aspas simples.
- Carácteres são representados por átomos de comprimento 1.
- Átomos não delimitados por aspas simples, nunca começam com uma letra maiúscula, nem número ou *underscore*.



- Átomos são constantes simbólicas, podendo ser delimitados ou não, por aspas simples.
- Carácteres são representados por átomos de comprimento 1.
- Átomos não delimitados por aspas simples, nunca começam com uma letra maiúscula, nem número ou *underscore*.

Exemplos

x x_1 ' _ ' '\\ ' 'a\'b\n' '_ab' '\$%'



Números se dividem em:

- **Inteiro:** Inteiros podem ser representados por números binários, octais, decimais ou hexadecimais.

Exemplos

12_345	12345 em notação decimal, usando _ como separador
0b100	4 em notação binária
0o73	59 em notação octal
0xf7	247 em notação hexadecimal

O **underscore** é ignorado pelo compilador e o interpretador.



- **Real:** Números reais são compostos por um parte inteira, um ponto, seguido por uma fração decimal, ou um expoente.
- Se existe uma parte inteira em um número real então ela deve ser seguida por uma fração ou um expoente. Isso é necessário para distinguir um número real de um número inteiro.

Exemplos

12.345 0.123 12-e10 0.12E10



- Termos compostos podem conter mais de um valor ao mesmo tempo.



- Termos compostos podem conter mais de um valor ao mesmo tempo.
- Termos compostos são acessados pela notação de índice, começando a partir de 1 e indo até N , onde N é o tamanho deste termo.



- Termos compostos podem conter mais de um valor ao mesmo tempo.
- Termos compostos são acessados pela notação de índice, começando a partir de 1 e indo até N , onde N é o tamanho deste termo.
- Se dividem em Listas e Estruturas.



Listas são agrupamentos de valores quaisquer sem ordem e sem tamanho pré-definido. Seu tamanho não é armazenado na memória, sendo necessário recalculá-lo sempre que necessário seu uso. Listas são encapsuladas por colchetes.

Exemplos

[1,2,3,4,5] [a,b,32,1.5,aaac] ["string",14,22]

Há uma seção dedicada a esta poderosa estrutura de dados!



Strings são listas especiais que contém somente carácteres. *Strings* podem ser inicializadas como uma sequência de carácteres encapsulados por aspas duplas, ou como uma sequência de carácteres dentro colchetes separados por vírgulas.

Exemplos

```
"Hello" "World!" "\n" [o,l,a," ",m,u,n,d,o]
```



- Tuplas é um conjunto de termos não-ordenados, podendo ser acessados por notação de índice assim como listas.
- Tuplas são estáticas, ou seja, os termos contidos em uma tupla não podem ser alterados, assim como não podem ser adicionados ou removidos termos de tuplas.
- Tuplas são encapsuladas por parênteses e seus termos são separados por vírgulas.

Exemplos

(1,2,3,4,5) (a,b,32,1.5,aaac) ("string",14,22)

Em geral, usamos as tuplas dentro de listas.



Estruturas são termos especiais que podem ser definidos pelo usuário. Estruturas tomam a seguinte forma:

$$s(t_1, \dots, t_n)$$

Onde ' s ' é um átomo que denomina a estrutura, cada ' t_i ' é um de seus termos, e ' n ' é a aridade ou tamanho da estrutura.

Exemplo

`$ponto(1,2)` `$pessoa(jose, "123.456.789.00", "1.234.567")`



Estruturas são termos especiais que podem ser definidos pelo usuário. Estruturas tomam a seguinte forma:

$$\$s(t_1, \dots, t_n)$$

Onde 's' é um átomo que denomina a estrutura, cada 't_i' é um de seus termos, e 'n' é a aridade ou tamanho da estrutura.

Exemplo

\$ponto(1,2) \$pessoa(jose, "123.456.789.00", "1.234.567")

Temos 4 outras estruturas que não usam o símbolo \$, são elas:



Vetores ou *arrays* são estruturas especiais do tipo:

$$\{t_1, \dots, t_n\}$$



Vetores ou *arrays* são estruturas especiais do tipo:

$$\{t_1, \dots, t_n\}$$

- Vetor é um conjunto ordenado de tamanho n , delimitado por ' $\{\}$ '.
- Vetores tem comportamentos análogo às listas, tanto é que quase todas as funções de listas são sobrecarregadas para vetores.
- A diferença entre vetores e listas é que vetores tem um tamanho constante.
- Vetores são muito práticos quando se manipula matrizes na entrada



Exemplos

$\{1,2,3,4,5\}$ $\{a,b,32,1.5,aaac\}$ $\{"string",14,22\}$

- **Mapas** são estruturas especiais que são conjuntos de relações do tipo chave-valor.
- **Conjuntos** são sub-tipos de mapas onde todas as chaves estão relacionadas com o átomo `not_a_value`.
- *Heaps* são árvores binárias completas representadas como vetores. Árvores podem ser do tipo *máximo*, onde o maior valor está na raiz, ou *mínimo*, onde o menor valor está na raiz.



- Picat é uma linguagem de Tipagem Dinâmica, ou seja, o tipo de uma variável é validado durante a execução do programa
- Isto é, quando uma variável é criada, seu tipo não é instanciado
- Variáveis são análogas as da matemática, são símbolos que *seguram* ou representam um valor
- Ao contrário de variáveis em linguagens imperativas, variáveis em Picat não são endereços simbólicos de locais na memória
- Uma variável é dita *livre* (*free*) se não contém nenhum valor, e dita *instanciada* (*bound*) se ela contém um valor
- Uma vez que uma variável é instanciada, ela permanece com este valor na execução atual
- Por isso, diz-se que variáveis em Picat são de *atribuição única*



- O nome de variáveis devem sempre ser iniciado com letras maiúsculas ou com o carácter *underscore* (`_`), porém;
 - Variáveis cujo nome é unicamente um caractere `_` são chamadas de *variáveis anônimas*.
 - As *variáveis anônimas* podem receber qualquer valor não os guardam durante a execução do programa;
 - Num mesmo programa, podem existir diversas variáveis anônimas, instanciadas durante a execução do mesmo



Há dois modos de definir valores às variáveis:



Há dois modos de definir valores às variáveis:

- Unificação usa o operador '='
- Atribuição usa o operador ':='



- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor



- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.



- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.
- Uma instanciação é indefinida até que se encontre um valor que possa ser unificada a uma variável.



- A Unificação é uma operação que instancia uma variável a um termo, substituindo toda ocorrência dessa variável pelo valor
- Caso ocorra uma instância que não falhe nenhuma situação a variável é unificada à este termo ou padrão.
- Uma instanciação é indefinida até que se encontre um valor que possa ser unificada a uma variável.
- Termos são ditos unificáveis se são idênticos ou podem ser tornados idênticos instanciando variáveis nos termos.



Exemplo

```
Picat> X = 1
X = 1
Picat> $f(a,b) = $f(a,b)
yes
Picat> [H|T] = [a,b,c]
H = a
T = [b,c]
Picat> $f(X,b) = $f(a,Y)
X = a
Y = b
Picat> bind_vars({X,Y,Z},a)
Picat> X = $f(X)
```

Cuidar neste último caso, há um laço infinito nesta chamada!



- A atribuição simula a atribuição em linguagens imperativas



- A atribuição simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa



- A atribuição simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa
- O escopo da atribuição da variável é local e volátil



- A atribuição simula a atribuição em linguagens imperativas
- Permite que variáveis assumam novos valores durante a execução do programa
- O escopo da atribuição da variável é local e volátil
- Na unificação, uma nova variável temporária é criada afim de substituir um valor atribuído ou outra variável.



Exemplo

`teste => X = 0, X := X + 1, X := X + 2, write(X).`

- Neste exemplo X é unificado a 0.
- Em seguida, há uma atribuição X a $X + 1$, porém X já foi unificado a um termo.
- Então, outras operações devem ser feitas para que esta atribuição seja possível.
- Nesse caso, o compilador cria uma variável temporária, $X1$ por exemplo, e unifica com $X + 1$. Cada vez que X for instanciado, o compilador/programa atualiza em $X1$.
- O mesmo ocorre na atribuição $X1 := X1 + 2$, neste caso uma outra variável temporária é criada, por exemplo $X2$, e o processo se repetido.



Exemplo

Portanto, estas atribuições sucessivas são compiladas como:



Exemplo

Portanto, estas atribuições sucessivas são compiladas como:

```
test => X = 0, X1 = X + 1, X2 = X1 + 2, write(X2).
```



Exemplos de Variáveis Válidas

X1	_	_ab
X	A	Variavel
_invalido	_correto	_aa

⇒ Relembrando, um nome de variável é válido se começa com letra **maiúscula** ou **_**



Exemplos de Variáveis Inválidas

1_Var	variável	valida
23	"correto	'termo
!numero	\$valor	#comum



Tabela 1: Operadores Aritméticos em Ordem de Precedência

$X ** Y$	Potenciação
$X * Y$	Multiplicação
X / Y	Divisão, resulta em um real
$X // Y$	Divisão de Inteiros, resulta em um inteiro
$X \text{ mod } Y$	Resto da Divisão
$X + Y$	Adição
$X - Y$	Subtração
<i>Inicio .. Passo .. Fim</i>	Uma série (lista) de números com um passo
<i>Inicio .. Fim</i>	Uma série (lista) de números com passo 1



Tabela 2: Tabela de Operadores Completa em Ordem de Precedência

Ops Aritméticos	Ver Tabela ??
++	Concatenação de Listas/Vetores
= :=	Unificação e Atribuição
== ==:=	Equivalência e Equivalência Numérica
!= !===	Não Unificável e Diferença
< =< <=	Menor que
> >=	Maior que
in	Contido em
not	Negação Lógica
, &&	Conjunção Lógica
;	Disjunção Lógica



Operadores de Termos Não-Compostos

- **Equivalência(==)**: compara se dois termos são iguais.
No caso de termos compostos, eles são ditos equivalentes se todos os termos contidos em si são equivalentes. O compilador considera termos de tipos diferentes como totalmente diferentes, portanto a comparação $1.0 == 1$ seria avaliada como falsa, mesmo que os valores sejam iguais. Nesses casos, usa-se a *Equivalência Numérica*.



Operadores de Termos Não-Compostos

- **Equivalência(==)**: compara se dois termos são iguais.
No caso de termos compostos, eles são ditos equivalentes se todos os termos contidos em si são equivalentes. O compilador considera termos de tipos diferentes como totalmente diferentes, portanto a comparação $1.0 == 1$ seria avaliada como falsa, mesmo que os valores sejam iguais. Nesses casos, usa-se a *Equivalência Numérica*.
- **Equivalência Numérica(==)**: Compara se dois números são o mesmo valor. Deve ser usada com termos que sejam números.



Operadores de Termos Não-Compostos

- **Diferença(!=)**: compara se dois termos são diferentes, isto é, a negação da equivalência.



Operadores de Termos Não-Compostos

- **Diferença(\neq):** compara se dois termos são diferentes, isto é, a negação da equivalência.
- **Não-Unificável(\neq):** Verifica se dois termos não são unificáveis. Termos são ditos unificáveis se são idênticos ou podem ser tornados idênticos instanciando variáveis destes termos.



Exemplos

① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no
- ③ $1.0 ::= 1$, $1.2 ::= 1$



Exemplos

- 1 $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão no)
- 2 $1.0 == 1$
no
- 3 $1.0 ::= 1$, $1.2 ::= 1$
yes, no



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no
- ③ $1.0 := 1$, $1.2 := 1$
yes, *no*
- ④ $1.0 != 1$, $Var3 != Var4$



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no
- ③ $1.0 := 1$, $1.2 := 1$
yes, *no*
- ④ $1.0 !== 1$, $Var3 !== Var4$
yes, Depende dos valores (padrão *yes*)



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no
- ③ $1.0 == 1$, $1.2 == 1$
yes, *no*
- ④ $1.0 != 1$, $Var3 != Var4$
yes, Depende dos valores (padrão *yes*)
- ⑤ $1.0 != 1$, $aa != bb$, $Var1 != Var5$



Exemplos

- ① $a == a$, $[1, 2, 3] == [1, 2, 3]$, $Var1 == Var2$
yes, yes, Depende dos valores (padrão *no*)
- ② $1.0 == 1$
no
- ③ $1.0 ::= 1$, $1.2 ::= 1$
yes, *no*
- ④ $1.0 !== 1$, $Var3 !== Var4$
yes, Depende dos valores (padrão *yes*)
- ⑤ $1.0 != 1$, $aa != bb$, $Var1 != Var5$
yes, yes, *no*



- **Concatenação (++)**: concatena duas listas ou vetores. O termo da esquerda é a primeira parte lista e a segundo a parte final da lista resultante.



- **Concatenação** ($++$): concatena duas listas ou vetores. O termo da esquerda é a primeira parte lista e a segundo a parte final da lista resultante.
- **Separador** ($H \mid T$): separa uma lista L em seu primeiro termo H , chamado de *cabeça* (em inglês *Head*), e o resto da lista T , chamado de *cauda* (em inglês *Tail*).

Na seção de listas este assunto é retomado



- **Iterador** (X in L): itera X no termo composto L , instanciando um termo não-composto X aos termos contidos em L .



- **Iterador** (X in L): itera X no termo composto L , instanciando um termo não-composto X aos termos contidos em L .
- **Sequência** (*Inicio..Passo..Fim*): gera uma lista ou vetor, começando (inclusivamente) em *Inicio* incrementando por *Passo* e parando (inclusivamente) em *Fim*. Se *Passo* for omitido, este é automaticamente atribuído 1.

Na seção de listas este assunto é retomado



Operadores de Termos Compostos – Exemplos

① $[1, 2, 3] ++ [4, 5, 6], \quad [] ++ [1, 2, 3], \quad [] ++ []$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*



Operadores de Termos Compostos – Exemplos

- 1 $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- 2 $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- 3 *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3



Operadores de Termos Compostos – Exemplos

- 1 $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- 2 $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- 3 *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- 4 $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$



Operadores de Termos Compostos – Exemplos

- 1 $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- 2 $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- 3 *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- 4 $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$



Operadores de Termos Compostos – Exemplos

- 1 $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- 2 $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- 3 *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- 4 $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
 $Y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$



Operadores de Termos Compostos – Exemplos

- ① $[1, 2, 3] ++ [4, 5, 6], [] ++ [1, 2, 3], [] ++ []$
 $[1, 2, 3, 4, 5, 6], [1, 2, 3], []$
- ② $L = [1, 2, 3], [H|T] = L$
 $L = [1, 2, 3]$
 $H = 1$
 $T = [2, 3]$
- ③ *foreach*(X in $[1, 2, 3]$) *printf*(" %w ", X) *end*
1 2 3
- ④ $X = 1..10, Y = 0..2..20, Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
 $Y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$
 $Z = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$



- Picat: tipagem dinâmica. O erro entre variáveis e dados, ocorre no meio da execução do código



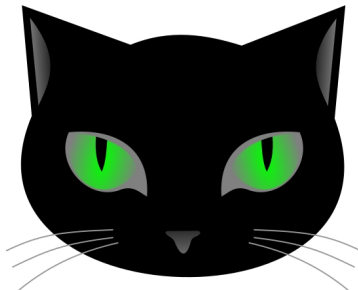
- Picat: tipagem dinâmica. O erro entre variáveis e dados, ocorre no meio da execução do código
- As estruturas de dados compostos são as clássicas: listas, vetores, mapas, conjuntos, etc



- Picat: tipagem dinâmica. O erro entre variáveis e dados, ocorre no meio da execução do código
- As estruturas de dados compostos são as clássicas: listas, vetores, mapas, conjuntos, etc
- O conteúdo desta parte do curso pode ser complementado com a **Videoaula 02: Tipos de Dados do PICAT**
<https://www.youtube.com/watch?v=7fPKPd0ZDnc>



- Definições
- Predicados (Cláusulas)
- Funções
- Contexto de uso
- Exemplos



- *Predicados lógicos* definem a base teórica do Picat



- *Predicados lógicos* definem a base teórica do Picat
- *Predicados lógicos* são também conhecidos como *cláusulas lógicas*, ora como simplesmente *regras lógicas* (no contexto de linguagens de programação)



- *Predicados lógicos* definem a base teórica do Picat
- *Predicados lógicos* são também conhecidos como *cláusulas lógicas*, ora como simplesmente *regras lógicas* (no contexto de linguagens de programação)
- Estas regras lógicas são de 04 tipos:



- *Predicados lógicos* definem a base teórica do Picat
- *Predicados lógicos* são também conhecidos como *cláusulas lógicas*, ora como simplesmente *regras lógicas* (no contexto de linguagens de programação)
- Estas regras lógicas são de 04 tipos:
 - Regras condicionais (ou completas)
 - Regras do tipo **fatos** ou sempre verdadeiras
 - Regras do tipo **metas**
 - Regras tipo **funções**



- *Predicados lógicos* definem a base teórica do Picat
- *Predicados lógicos* são também conhecidos como *cláusulas lógicas*, ora como simplesmente *regras lógicas* (no contexto de linguagens de programação)
- Estas regras lógicas são de 04 tipos:
 - Regras condicionais (ou completas)
 - Regras do tipo **fatos** ou sempre verdadeiras
 - Regras do tipo **metas**
 - Regras tipo **funções**
- Excetuando-se as funções, as regras sempre assumem os seguintes valores lógicos:
true (1 ou yes ...) ou false (0 ou no ...).



- Nos argumentos do **predicados** podem-se passar n -termos e receber outros m -termos, tal que $n \geq m$. Exemplo:

`raizes(A, B, C, X1, X2)`



- Nos argumentos do **predicados** podem-se passar n -termos e receber outros m -termos, tal que $n \geq m$. Exemplo:

`raizes(A, B, C, X1, X2)`

- As **funções** seguem as regras de funções matemáticas, logo, retornam um único valor



- Nos argumentos do **predicados** podem-se passar n -termos e receber outros m -termos, tal que $n \geq m$. Exemplo:

`raizes(A, B, C, X1, X2)`

- As **funções** seguem as regras de funções matemáticas, logo, retornam um único valor
- Ou ainda, as **funções** são um tipo particular de **regras lógicas**



- Nos argumentos do **predicados** podem-se passar n -termos e receber outros m -termos, tal que $n \geq m$. Exemplo:

`raizes(A, B, C, X1, X2)`

- As **funções** seguem as regras de funções matemáticas, logo, retornam um único valor
- Ou ainda, as **funções** são um tipo particular de **regras lógicas**
- Predicados e funções são inferenciados via *casamento de padrões*

Este conceito é ilustrado pelos exemplos deste curso.



As regras lógicas são de dois tipos:

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

Cabeça , Condicional \Rightarrow Corpo .



As regras lógicas são de dois tipos:

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

Cabeça , Condicional \Rightarrow Corpo .

- Regras **com** *backtracking*:

Cabeça , Condicional \Rightarrow Corpo .



A identificação da sintaxe é dada por:

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

Forma geral:

$$\text{regra}(\text{termo}_1, \dots, \text{termo}_n)$$

Onde:

- *regra* é um átomo que define o nome da regra
- *n* é a aridade da regra (*i.e.* número 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.
- **Regras e funções** são ilustradas pelos exemplos deste curso.



Regras Com e Sem Backtracking – Exemplo

```
main => regra_01(7) , regra_01(-4) ,  
        regra_01( 0 ) , regra_01( 1) ,  
        regra_01(44) .
```

```
regra_01(0)           ?=> printf("\n CHEGOU A 0 !!!").  
regra_01(N) , N < 0  ?=> printf("\n EH UM NEGATIVO !!!").  
regra_01(N) , N > 10 ?=> printf("\n EH MAIOR QUE 10 !!!").  
regra_01(N) , N <= 10 => printf("\n NUM de 0 a 10 :%d", N).
```

```
%% =< EH IGUAL a <= :: sobrecarga  
%% >= EH IGUAL => :: sobrecarga  
%%% $ picat regras_ex_04.pi
```



```
$ picat regras_ex_04.pi
```

```
NUM de 0 a 10 :7  
EH UM NEGATIVO !!!  
CHEGOU A 0 !!!  
NUM de 0 a 10 :1  
EH MAIOR QUE 10 !!!
```

Aqui, o main foi um tipo função e uma regra tipo meta!



- O *casamento de padrões* é o mecanismo de encontrar regras que casem com a instância corrente de execução do programa



- O *casamento de padrões* é o mecanismo de encontrar regras que casem com a instância corrente de execução do programa
- O objetivo é encontrar dois ou mais padrões que possam ser unificados para se inferir alguma nova ação



- O *casamento de padrões* é o mecanismo de encontrar regras que casem com a instância corrente de execução do programa
- O objetivo é encontrar dois ou mais padrões que possam ser unificados para se inferir alguma nova ação
- Muitos exemplos de uso ao longo do curso



Procedimento de *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 $p_2(u_1, \dots, u_n)$.



- 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*



- 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



- 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, uma operação lógica, um termo ...etc



- 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, uma operação lógica, um termo ...etc
- Exemplo: a cláusula `main` é uma meta a ser provada!



- 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, uma operação lógica, um termo ...etc
- 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:
$$\text{Cabeça} = X \Rightarrow \text{Corpo}.$$
- Caso tenhamos uma condição *Cond*::
$$\text{Cabeça} = X, \text{Cond} \Rightarrow \text{Corpo}.$$
- Funções **não** admitem *backtracking*.



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



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



```
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").
```

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

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



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,Zeze), write(Zeze).  %% goal
7Zeze = 7
yes
Picat> uma_funcao(3,4) = Ratu, write(Ratu).  %% goal
7Ratu = 7
yes
%%%%%%%%%% na console Linux
$ picat predicados_funcoes.pi
Z: 7   R: 7
FIM
```



Algumas linhas em **branco** na saída foram omitidas!

- Forma geral de um predicado do tipo Regra:

Cabeça , Condicional \Rightarrow Corpo .



- Forma geral de um predicado do tipo Regra:
 $\text{Cabeça} , \text{Condicional} \Rightarrow \text{Corpo} .$
- Forma geral de um predicado com *backtracking*:
 $\text{Cabeça} , \text{Condicional} \textcolor{red}{?}\Rightarrow \text{Corpo} .$



- Forma geral de um predicado do tipo Regra:

Cabeça , Condicional \Rightarrow Corpo .

- Forma geral de um predicado com *backtracking*:

Cabeça , Condicional $\textcolor{red}{?}\Rightarrow$ Corpo .

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



- Forma geral de um predicado do tipo Regra:

Cabeça , Condicional \Rightarrow Corpo .

- Forma geral de um predicado com *backtracking*:

Cabeça , Condicional $\textcolor{red}{?}\Rightarrow$ Corpo .

- Em Prolog, esta condicional (Cond) entra no corpo da regra.
Picat é mais flexível!
- Na prática, o estilo de programação usado é:

$\textcolor{violet}{\text{Cabeça}} \Rightarrow \textcolor{violet}{\text{Condicional}} , \textcolor{violet}{\text{Corpo}} .$



- Forma geral de um predicado do tipo Regra:

Cabeça , Condicional \Rightarrow Corpo .

- Forma geral de um predicado com *backtracking*:

Cabeça , Condicional $\color{red}{?}\Rightarrow$ Corpo .

- Em Prolog, esta condicional (Cond) entra no corpo da regra.
Picat é mais flexível!
- Na prática, o estilo de programação usado é:

$\color{magenta}{\text{Cabeça} \Rightarrow \text{Condicional} , \text{Corpo} .}$

- Porquê? Reuso de código de Prolog ...



- Dentro de uma regra, *Cond* só é avaliada uma vez (**cuidado**), exceto em caso falha e a regra for habilitada para *backtracking*



- Dentro de uma regra, *Cond* só é avaliada uma vez (**cuidado**), exceto em caso falha e a regra for habilitada para *backtracking*
- Estar atento que: regras são **sempre avaliados com valores lógicos** (*true* ou *false*)



- Dentro de uma regra, *Cond* só é avaliada uma vez (**cuidado**), exceto em caso falha e a regra for habilitada para *backtracking*
- 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.



- As regras que não tem condicionais e nem corpo são conhecidas como: *fatos* ou *regras sem-corpo*
- Estes *fatos* são regras *sempre verdadeiras*



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

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

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



- A declaração de um fato é precedida pela declaração *index*:

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



- A declaração de um fato é precedida pela declaração *index*:
$$\text{index } (M_{11}, M_{12}, \dots, M_{1n}) \dots (M_{m1}, M_{m2}, \dots, M_{mn})$$
- Onde um M_{ij} com o simbolo '+', significa que este termo já foi indexado.
- Logo, o '-' significa que este termo deve ser indexado
- Ou seja, quando ocorre um simbolo '+' em um grupo do *index*, o compilador avalia este como um valor constante a ser casado
- 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



- A declaração de um fato é precedida pela declaração *index*:
$$\text{index } (M_{11}, M_{12}, \dots, M_{1n}) \dots (M_{m1}, M_{m2}, \dots, M_{mn})$$
- Onde um M_{ij} com o simbolo '+', significa que este termo já foi indexado.
- Logo, o '-' significa que este termo deve ser indexado
- Ou seja, quando ocorre um simbolo '+' em um grupo do *index*, o compilador avalia este como um valor constante a ser casado
- 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



- A declaração de um fato é precedida pela declaração *index*:
 $\text{index } (M_{11}, M_{12}, \dots, M_{1n}) \dots (M_{m1}, M_{m2}, \dots, M_{mn})$
- Onde um M_{ij} com o simbolo '+', significa que este termo já foi indexado.
- Logo, o '-' significa que este termo deve ser indexado
- Ou seja, quando ocorre um simbolo '+' em um grupo do *index*, o compilador avalia este como um valor constante a ser casado
- 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
- Cuidado: **Não pode haver um predicado e um predicado fato com mesmo nome.**



```
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), % AND2 pois and eh reservado
    printf("\n X: %w \t Y: %w \t Z: %w", X, Y, Z),
    false.
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_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 Função

```
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!



Exemplos de Fatos

```
1
2 index (-,-) (+,-) (-,+)      %%% NAO HA PONTO FINAL
3 pai (salomao, rogerio).
4 pai (salomao, fabio).
5 pai (rogerio, miguel).
6 pai (rogerio, henrique).
7 .....
```



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.
```



- Esta seção trata dos elementos do Picat: regras (ou cláusulas) lógicas



- Esta seção trata dos elementos do Picat: regras (ou cláusulas) lógicas
- *Regras* é um nome genérico a predicados ou cláusulas e funções



- Esta seção trata dos elementos do Picat: regras (ou cláusulas) lógicas
- *Regras* é um nome genérico a predicados ou cláusulas e funções
- Predicados ou cláusulas são de 4 tipos: **regras**, **fatos**, **metas** e **funções**



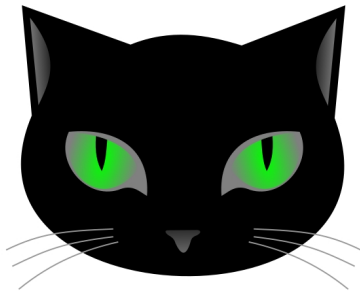
- Esta seção trata dos elementos do Picat: regras (ou cláusulas) lógicas
- *Regras* é um nome genérico a predicados ou cláusulas e funções
- Predicados ou cláusulas são de 4 tipos: **regras**, **fatos**, **metas** e **funções**
- *Regras* sem corpo são conhecidas como verdades e chamadas de **fatos**
Uso obrigatório do meta-predicado: *index*



- Esta seção trata dos elementos do Picat: regras (ou cláusulas) lógicas
- *Regras* é um nome genérico a predicados ou cláusulas e funções
- Predicados ou cláusulas são de 4 tipos: **regras**, **fatos**, **metas** 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



- Definições
- Contexto de uso
- Estruturas de decisão
- Estruturas de repetição
- Iteradores
- Entradas e saídas
- 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



- 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
```



- 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.



- 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.



- 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



- 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



- Um laço foreach tem a seguinte forma:

```
foreach ( $E_1$  in  $D_1$ ,  $Cond_1$ , ...,  $E_n$  in  $D_n$ ,  $Cond_n$ )  
    Metas  
end
```



- 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.
7  .....
8  Picat> cl('backtracking_ex_02').
9  Compiling:: backtracking_ex_02.pi
10 ** Warning: singleton variables (backtracking_ex_02.pi, 50-55): I
11 backtracking_ex_02.pi compiled in 3 milliseconds
12 loading...
13
14 yes
15
16 Picat> imp_tracejados(30).
17
18 =====
```



O I é iterado com valores de 1 a N

- 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.
```



- 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:



- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado:
 - Geração (compreensão) de listas e vetores
 - Entradas e saídas de dados



- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado:
 - Geração (compreensão) de listas e vetores
 - Entradas e saídas de dados
- Na verdade, já fizemos uso delas, porém sem a ênfase de que são funções ora predicados.



- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado:
 - Geração (compreensão) de listas e vetores
 - Entradas e saídas de dados
- Na verdade, já fizemos uso delas, porém sem a ênfase de que são funções ora predicados.
- Um **atalho** considerável na programação com Picat!



- A função de compreensão de listas e vetores é uma função especial que permite a fácil criação de listas ou vetores.
- Sua notação é:

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

- 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.



- Esta função pode gerar um vetor também, a notação é um pouco diferente:

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

- 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("\nLista: %w \nVetor: %w\n", L, V),
  imp_vetor(V),
  imp_vetor(L).
```

```
imp_vetor (M) =>
  Tam = M.length,  %% tamanho de M
  nl,
  foreach(I in 1 .. Tam )
    printf("Vou_L(%d):%w \t" , I, M[I] )
  end,
  nl.
%%%% $picat vetor_exemplo_01.pi
```



Saída – Compreensão de Listas e Vetores

```
$ picat vetor_exemplo_01.pi
```

```
Lista: [(a,1),(a,2),(b,1),(b,2)]
```

```
Vetor: {(1,a),(2,a),(1,b),(2,b)}
```

```
V_ou_L(1):1,a V_ou_L(2):2,a V_ou_L(3):1,b V_ou_L(4):2,b
```

```
V_ou_L(1):a,1 V_ou_L(2):a,2 V_ou_L(3):b,1 V_ou_L(4):b,2
```

```
[hulk@SAGUACU picat]$
```



- 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.



- 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) ⇒` 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.



- 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`, 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	<i>String</i>
%u	Número Inteiro Sem Sinal
%w	Termo qualquer



Comparação entre write e print

Dados \Rightarrow	"abc"	[a,b,c]	'a@b'
write	[a,b,c]	[a,b,c]	'a@b'
writeln	[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
```



```
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 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



```
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 ).
```



```
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.




```
$ 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
```



- Esta seção avança na sintaxe do Picat e precisa ser praticada



- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.



- Esta seção avança na sintaxe do Picat e 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 *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas



- Esta seção avança na sintaxe do Picat e 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 *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado:** as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false**!



- Esta seção avança na sintaxe do Picat e 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 *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado:** as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false!**
- Felizmente os erros e problemas de sintaxe são prontamente reportados



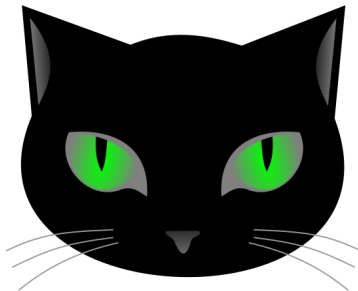
- Esta seção avança na sintaxe do Picat e precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, voce está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado para *entradas* e *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado:** as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false!**
- Felizmente os erros e problemas de sintaxe são prontamente reportados
- Em https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos tem vários exemplos avançados de entradas e saídas



- Esta seção avança na sintaxe do Picat e 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 *saídas*, com Picat, tudo ficou semelhante as demais linguagens imperativas
- **Cuidado:** as funções de *entradas* e *saídas*, e outras do sistema como *time*, etc, **nunca** falham e nem aceitam *backtracking* definidas para elas. Em caso de falha do predicado que as contém, estas são **silenciosas**, apresentando um simples **no** ou **false!**
- Felizmente os erros e problemas de sintaxe são prontamente reportados
- Em https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos tem vários exemplos avançados de entradas e saídas
- Mãos à obra!



- 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.

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

Portanto:

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



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



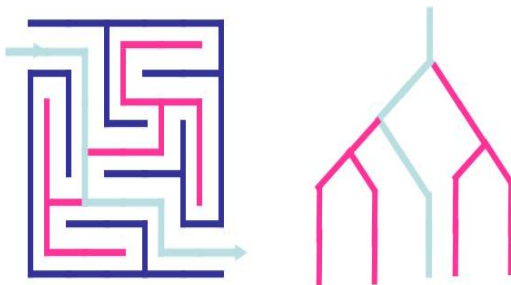
- O mecanismo de *backtracking* é bem conhecido por algumas linguagens de programação
- Em Picat, o *backtracking* é controlável e é habilitado pelo símbolo `?=>` no escopo da regra.



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):

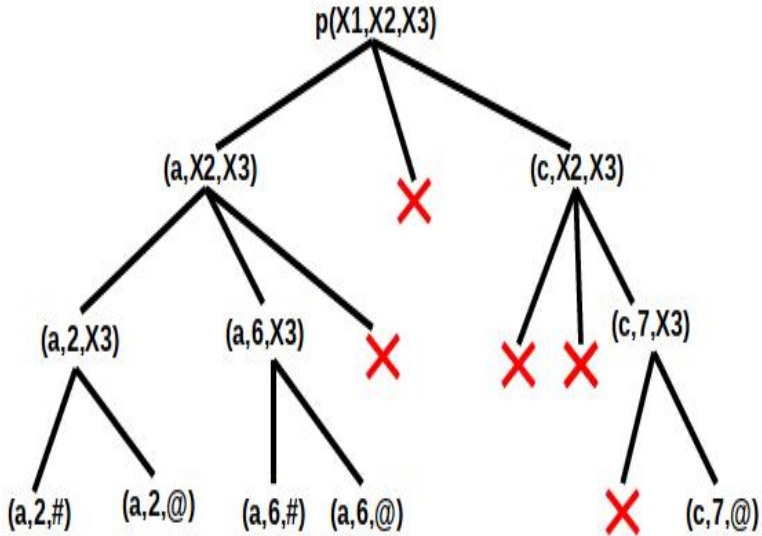
$$p(X_1, X_2, X_3, \dots, X_n) \Rightarrow q_1(X_1), q_2(X_2), \dots, q_n(X_n).$$
- 3 Caso ocorra uma falha durante a execução da regra p , o compilador busca re-instanciar as variáveis do corpo de p que falharem. Esta tentativa segue uma ordem:
 $q_1(X_1) \rightarrow q_2(X_2) \rightarrow \dots \rightarrow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

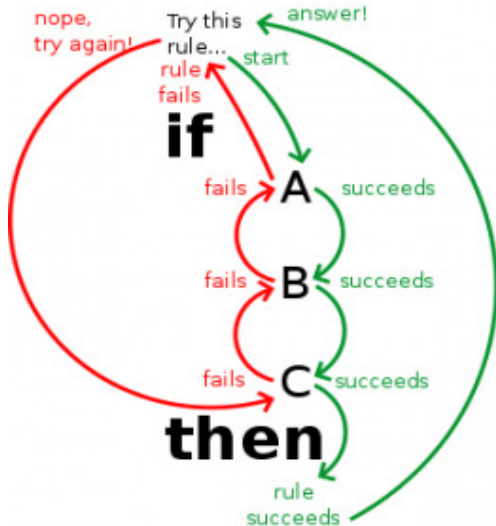


Ilustra o *Backtracking* – Árvore de Busca



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").
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
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 (**?=>**)
- A avaliação destas regras **são sempre da esquerda para direita**, ocorrendo o *backtracking* em caso de falha ou de uma nova resposta



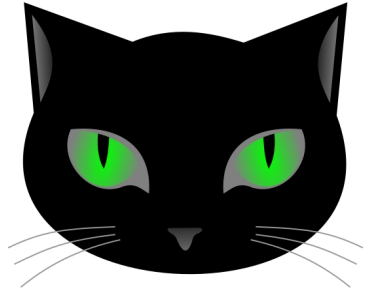
- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que *sempre vem antes* das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados (*?=>*)
- A avaliação destas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



- Definição de listas
- Representação
- Operadores
- Geração de listas
- Exemplos



- Requisito: conceito de recursividade, *aterramento* etc, dominados!



- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais



- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (**cada nó sempre tem duas ramificações**)



- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (*cada nó sempre tem duas ramificações*)
- Lembrando que uma estrutura binária de árvore tem uma equivalência com uma árvore n-ária (ver livro de Estrutura de Dados)



- Requisito: conceito de recursividade, *aterramento* etc, dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (*cada nó sempre tem duas ramificações*)
- Lembrando que uma estrutura binária de árvore tem uma equivalência com uma árvore n-ária (ver livro de Estrutura de Dados)
- Logo, listas são estruturas flexíveis e poderosas!



Ilustrando uma Lista em Formato Binário

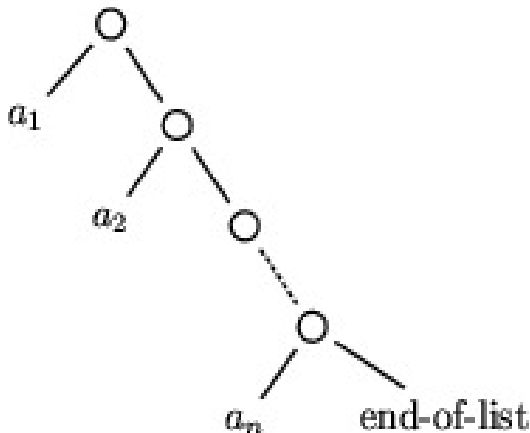


Figura 3: Uma estrutura Lista – Homogênea



Ilustrando Listas e o Operador '|' (ou ':' da figura)

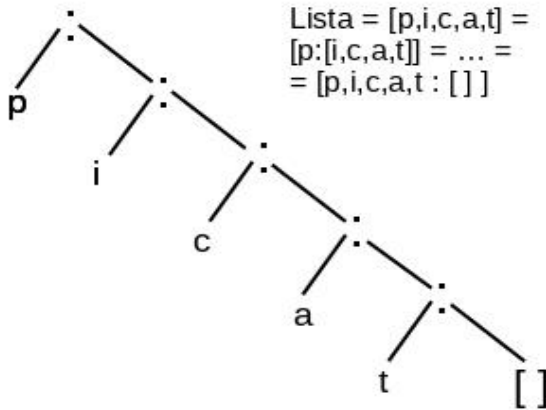


Figura 4: Listas são inerentemente **recursivas**!



lista: [a,b,c,d]
cabeça: a
cauda: [b,c,d]

lista: [[a,b],c,[d,e,f],g]
cabeça: [a,b]
cauda: [c,[d,e,f],g]

lista: [[A11,A12],[A21,A22]]
cabeça: [A11,A12]
cauda: [[A21,A22]]



Definições iniciais (e recursivas)

- Uma lista é uma sequência de termos (objetos)



Notação:

- O símbolo “[” é usado para descrever o início de uma lista, e “]” para o final da mesma;
- Exemplo: seja a lista [a, b, c, d], logo um predicado cujo argumento seja algumas letras, tem-se uma lista do tipo:
 - letras([a, b, c, d])
 - Onde ‘a’ é o *cabeça* (primeiro elemento) da lista
 - e [b, c, d] é uma *sub-lista* que é uma lista!
- Os elementos de uma lista são lidos da esquerda para direita;
- A “*sub-lista*” [b, c, d] é conhecida como *resto* ou “*cauda*” da lista;
- Esta sub-lista é uma lista e toda definição segue-se recursivamente.



Operador “|”:

- “*Como vamos distinguir de onde se encontra a cabeça da cauda da lista?*”
- Com as listas novos símbolos foram introduzidos, isto é, além dos delimitadores [...], há um novo operador que **separa** ou **define** quem é a elemento cabeça da lista e cauda.
- Este operador é conhecido como “*pipe*” (ou *barra vertical*), simbolizado por “|”, que separa o lado esquerdo da direita da lista.
- Esta separação é necessário para se realizar os *casamentos de padrões* nas linguagens lógicas.



Exemplos de *casamentos*:

```
[ a, b, c, d ] = X
[ X | b, c, d ] = [ a, b, c, d ]
[ a | b, c, d ] = [ a, b, c, d ]
[ a , b | c, d ] = [ a, b, c, d ]
[ a , b , c | d ] = [ a, b, c, d ]
[ a , b , c , d | [] ] = [ a, b, c, d ]
[] = X
[ [ a | b , c , d ] ] = [ [ a , b , c , d ] ]
[ a | b , c , [ d ] ] = [ a , b , c , [ d ] ]
[ _ | b , c , [ d ] ] = [ a , b , c , [ d ] ]
[ a | Y ] = [ a , b , c , d ]
[ a | _ ] = [ a , b , c , d ]
[ a , b | c , d ] = [ X , Y | Z ]
```



Contra-exemplos de *casamentos*:

```
[ a , b | [c, d] ] != [ a, b, c, d ]
[ [ a , b , c , d] ] != [ a, b, c, d ]
[ a , b , [ c ] , d, e ] != [ a, b, c, d, e ]
[ [ [ a ] | b , c , d] ] != [ [ a , b , c , d] ]
```



- Estes casamentos de termos de uma lista são também conhecidos por *matching*
- Devido ao fato de listas modelarem qualquer estrutura de dados, invariavelmente, seu uso é extensivo há problemas em geral (dos simples a complexos)
- Porém, alguns cuidados no uso de predicados com *backtracking*. Acompanhe os exemplos.
- Os próximos exemplos encontram-se no arquivo:
`../picat/listas.pi`



Exemplos sobre Listas (e Implementações)

- 1 Comprimento de uma lista: retorna um valor numérico



Exemplos sobre Listas (e Implementações)

- 1 Comprimento de uma lista: retorna um valor numérico
- 2 Se um elemento x pertence a lista: retorna um valor binário (*true* ou *false*)



Exemplos sobre Listas (e Implementações)

- 1 Comprimento de uma lista: retorna um valor numérico
- 2 Se um elemento x pertence a lista: retorna um valor binário (*true* ou *false*)
- 3 Adicionar um elemento x em uma lista: retorna uma nova lista, o x inserido nesta lista, se x já estiver presente, não insira. Em resumo: insere x em L sem repetição.



Exemplos sobre Listas (e Implementações)

- 1 Comprimento de uma lista: retorna um valor numérico
- 2 Se um elemento x pertence a lista: retorna um valor binário (*true* ou *false*)
- 3 Adicionar um elemento x em uma lista: retorna uma nova lista, o x inserido nesta lista, se x já estiver presente, não insira. Em resumo: insere x em L sem repetição.
- 4 Concatena duas listas: retorna uma terceira lista

Em resumo: 4 métodos clássicos!



Exemplo 01: encontrar o comprimento de uma lista l

- O comprimento de uma lista é o comprimento de sua **sub-lista**, mais **um**
- O comprimento de uma lista vazia (`[]`) é zero.

Em Picat, sob uma visão funcional, este enunciado é escrito por:

```
comprimento_02( [ ] ) = 0.  
comprimento_02([ _ | L ]) = N =>  
    N = 1 + comprimento_02( L ).
```



Exemplo 01: encontrar o comprimento de uma lista II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
comprimento_01([],N) ?=> N = 0.  
%%% em PROLOG, apenas comprimento_01([],0). PORQUÊ?  
comprimento_01(_|L,N) =>  
    comprimento_01( L , Parcial ),  
    N = 1 + Parcial.
```



Exemplo 01: encontrar o comprimento de uma lista III

Um *mapa de memória* é dado por:

	Regra	X	T	N	$N = N+1$
<code>compto([a,b,c,d],N)</code>	#2	a	<code>[b,c,d]</code>	$3 \rightarrow$	$3+1=4$
<code>compto([b,c,d],N)</code>	#2	b	<code>[c,d]</code>	$2 \rightarrow$	$\swarrow 2+1$
<code>compto([c,d],N)</code>	#2	c	<code>[d]</code>	$1 \rightarrow$	$\swarrow 1+1$
<code>compto([d],N)</code>	#2	d	<code>[]</code>	$0 \rightarrow$	$\swarrow 0+1$
<code>compto([],N)</code>	#1	—	—	—	$\swarrow 0$



Exemplo 02: verificar a pertinência de um objeto na lista I

- Verifica se um dado objeto pertence há uma lista
- Um método clássico – muito usado
- Tem embutido no Picat: o *member*

Em Picat, sob uma visão funcional, esta função é escrita por:

```
pertence_02( _ , [ ]) = false.  
pertence_02( A, [A|_] ) = true.  
% CUIDAR ... em funcoes nao hah ? em ?=> ...  
% sem backtracking em funcoes  
pertence_02(A, [B|L]) = X =>  
    A != B,  
    X = pertence_02(A,L).
```



Exemplo 02: verificar a pertinência de um objeto na lista II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
pertence_01( A, [A|_] ) ?=> true.  
% Again, backtracking CONTROLADO ... diferente do Prolog  
pertence_01(A,[B|L]) =>  
    A != B ,  
    pertence_01(A,L).
```



Exemplo 03: adicionar um elemento em uma lista I

- Um objeto é adicionado no início da lista (sem repetição) caso este já esteja contido na lista, a lista original é a retornada:

Em Picat, sob uma visão funcional, esta função é escrita por:

```
add_X_lista_02(X, [ ]) = [X].  
add_X_lista_02(X, Y) = Z =>  
    pertence_02(X, Y) = true,  
    Z = Y ;  
    Z = [ X | Y ].
```



Exemplo 03: adicionar um elemento em uma lista II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
add_X_lista_01(X, [ ], Z )  ?=> Z = [X] .  
add_X_lista_01(X, Y, Z) ?=>  
    pertence_01(X, Y),  
    Z = Y .  
add_X_lista_01(X, Y, Z) =>  
    Z = [ X | Y ] .
```



Exemplo 04: união de duas listas I

- O método de união ou concatenação entre duas listas, resultando em uma terceira lista
- Este predicado é conhecido como *append* ou *concatena*. O *append* está pronto na biblioteca default do Picat
- Há uma versão simplificada: $L3 = L1 ++ L2$

Em Picat, sob uma visão funcional, esta função é escrita por:

```
uniao_02( [], X ) = X.
```

```
uniao_02( [X|L1], L2 ) = L3 =>
```

```
    L3 = [X | uniao_02( L1, L2 )].
```



Exemplo 04: união de duas listas II

Em Picat, sob uma visão lógica, este predicado pode ser construído como:

```
uniao_01( [], X, Y ) ?=> Y = X.  
uniao_01( A , L2, R ) =>  A = [X|L1] ,  
                           R = [X|L3] ,  
                           uniao_01( L1, L2, L3 ).
```



- O conceito de *list comprehension* veio da programação funcional
- Basicamente serve para criarmos ou gerarmos listas



- O conceito de *list comprehension* veio da programação funcional
- Basicamente serve para criarmos ou gerarmos listas
- Bastante útil e pode ser usada em qualquer parte de um código



Um *list comprehension* tem o seguinte formato na criação de listas:

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

- T é uma termo (uma expressão num caso genérico)
- E_i é um padrão de iteração
- D_i é uma expressão de um valor composto, em geral um intervalo de domínio
- Opcionalmente, condições $Cond_1, \dots, Cond_n$ são chamados de *termos*
- Esta geração de lista tem a seguinte interpretação: *toda tupla de valores $E_1 \in D_1, \dots, E_n \in D_n$, se as condições $Cond_i$ forem verdades, então o valor do termo T é adicionado na lista em construção*



Um vetor ou matrizes também pode ser construídos com um *array comprehension* e tem o seguinte formato:

$$\{T : E_1 \text{ in } D_1, \text{ Cond}_1, \dots, E_n \text{ in } D_n, \text{ Cond}_n\}$$



Um vetor ou matrizes também pode ser construídos com um *array comprehension* e tem o seguinte formato:

$$\{T : E_1 \text{ in } D_1, \text{ Cond}_1, \dots, E_n \text{ in } D_n, \text{ Cond}_n\}$$

Isto é o mesmo como

```
to_array([T : E_1 in D_1, Cond_1, ..., E_n in D_n,  
           Cond_n])
```



Exemplos de *list comprehension*

```
main => Status = command("clear") ,
printf("===== %d", Status),
    L0 = [I : I in 10..20],
    L1 = [I : I in 10..2..20],
    L2 = [I : I in 1..20, I>10, I<20],
    L3 = [(A,I) : A in [a,b], I in 1..10, I mod 2 == 0],
    L4 = [(I,J,K) : I in 1..2, J in 3..7, K in 1..10, I+J < K],
    printf("\n L0 : %w " , L0),
    printf("\n L1 : %w " , L1),
    printf("\n L2 : %w " , L2),
    printf("\n L3 : %w " , L3),
    printf("\n L4 : %w " , L4),
    printf("\n FIM\n").
```

```
% $ picat geracao_listas.pi
```



```
===== 0
L0 : [10,11,12,13,14,15,16,17,18,19,20]
L1 : [10,12,14,16,18,20]
L2 : [11,12,13,14,15,16,17,18,19]
L3 : [(a,2),(a,4),(a,6),(a,8),(a,10),(b,2),(b,4),(b,6),(b,8),(b,10)]
L4 : [(1,3,5),(1,3,6),(1,3,7),(1,3,8),(1,3,9),(1,3,10),(1,4,6),
      (1,4,7),(1,4,8),(1,4,9),(1,4,10),(1,5,7),(1,5,8),(1,5,9),(1,5,10),(1,6,
      (1,6,9),(1,6,10),(1,7,9),(1,7,10),(2,3,6),(2,3,7),(2,3,8),(2,3,9),(2,3,
      (2,4,7),(2,4,8),(2,4,9),(2,4,10),(2,5,8),(2,5,9),(2,5,10),
      (2,6,9),(2,6,10),(2,7,10)]
FIM
```

L4 ... *cortada*



- Há muitos predicados e funções prontas sobre listas nos módulos do Picat



- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos



- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a **metodologia de se programar em lógica**



- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a **metodologia de se programar em lógica**
- Exercitar-se para aprender os detalhes!



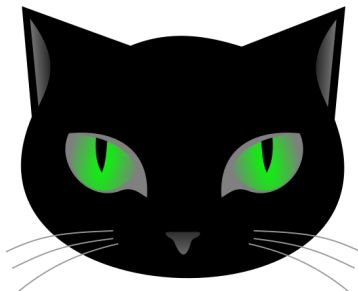
- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a **metodologia de se programar em lógica**
- Exercitar-se para aprender os detalhes!
- Usar as listas como estrutura base em problemas complexos



- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- Contudo, se aprende sobre listas, fazendo **muitos** métodos
- A recursividade em sua modelagem, define a **metodologia de se programar em lógica**
- Exercitar-se para aprender os detalhes!
- Usar as listas como estrutura base em problemas complexos
- Próxima na aula: buscas (**uso extensivo de listas**)



- O que é uma *busca*?
- Problemas \Rightarrow buscar ...
- Buscas em estruturas quaisquer
- Listas são o suficiente!
- Núcleo das buscas
- Exemplo



- Requisito: conceitos de listas e recursividade dominados!



- Requisito: conceitos de listas e recursividade dominados!
- Além destes: noções sobre grafos, árvores, nós, etc



- Requisito: conceitos de listas e recursividade dominados!
- Além destes: noções sobre grafos, árvores, nós, etc
- Solucionar problemas implicar em percorrer estados (um caminho) que levem há um estado-solução



- Requisito: conceitos de listas e recursividade dominados!
- Além destes: noções sobre grafos, árvores, nós, etc
- Solucionar problemas implicar em percorrer estados (um caminho) que levem há um estado-solução
- Grosseiramente: **estados de problemas** \Leftrightarrow **estruturas abstratas**



- Requisito: conceitos de listas e recursividade dominados!
- Além destes: noções sobre grafos, árvores, nós, etc
- Solucionar problemas implicar em percorrer estados (um caminho) que levem há um estado-solução
- Grosseiramente: **estados de problemas** \Leftrightarrow **estruturas abstratas**
- Pois, problemas em geral se apresentam como uma conexão complexa tipo um *grafo*, e a varredura sob este grafo é sistemática sob uma *árvore de busca*



- Requisito: conceitos de listas e recursividade dominados!
- Além destes: noções sobre grafos, árvores, nós, etc
- Solucionar problemas implicar em percorrer estados (um caminho) que levem há um estado-solução
- Grosseiramente: **estados de problemas** \Leftrightarrow **estruturas abstratas**
- Pois, problemas em geral se apresentam como uma conexão complexa tipo um *grafo*, e a varredura sob este grafo é sistemática sob uma *árvore de busca*
- Então, computar listas em Picat é uma estratégia de resolver problemas!



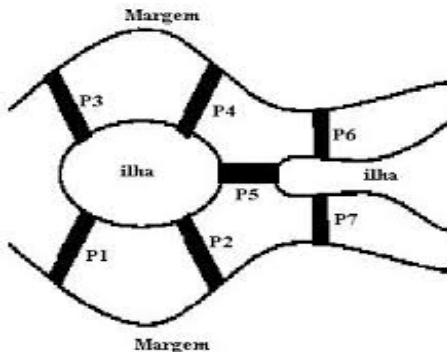


Figura 5: Ciclo Euleriano – Problema das Pontes de Königsberg



- No século 18 havia na cidade de Königsberg (antiga Prússia) um conjunto de sete pontes (identificadas pelas letras de P1 até P7 na figura ao lado) que cruzavam o rio Prególia. Elas conectavam duas ilhas entre si e as ilhas com as margens esquerda e direita.
- Os habitantes daquela cidade perguntavam-se se era possível cruzar as sete pontes numa caminhada contínua sem que se passasse duas vezes por qualquer uma das pontes.
- Embora intrigante, este problema foi atacado por Leonard Euler (1736) e demonstrou que isto não era possível para um grafo qualquer
- Curiosamente, este problema é fácil de resolver. Euler demonstrou uma relação entre vértices e arestas para que isto fosse possível.



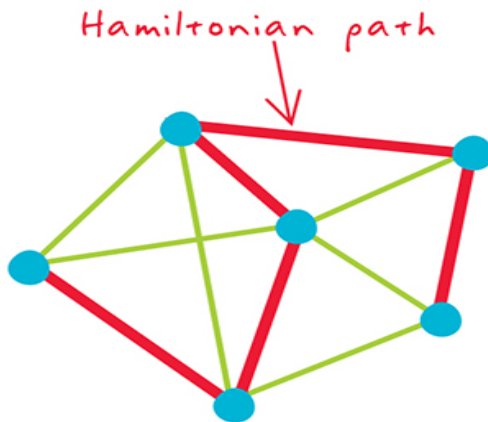


Figura 6: Caminho Hamiltoniano – Há um caminho que passe por todas cidades uma única vez?



- **Diferente** do ciclo Euleriano, o caminho Hamiltoniano, **origem e destino são diferentes**
- Todos os nós precisam ser visitados uma única vez sem repetição
- Num grafo pode haver muitos caminhos Hamiltonianos, mas, pode não existir nenhum!
- Ao contrário do ciclo Euleriano, este problema, computacionalmente é difícil de resolver!
- Vamos usar o caminho Hamiltoniano como exemplo, para construir um algoritmo ingênuo, mas que funciona bem!



Contextualizando estes termos:

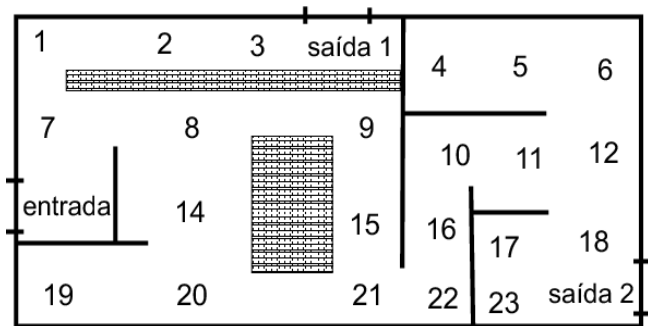
- Em geral, problemas podem ser vistos como *fotografias instantâneas* de uma situação, isto é, **um estado discreto**
- Uma *sucessão* destes estados, compõem *um caminho* de um estado i ao estado j
- Assim, estes *estados* são representados pelos *nós dos grafos*, e a ligação entre estes, são resultados de *uma ação*, mudança ou evolução do problema
- Há um estado particular chamado *inicial*, vários outros de estados *intermediários*, e outros estados *finais*
- Se o problema tiver várias soluções, o mesmo apresenta vários caminhos do estado inicial ao final.



- Assim uma sucessão ou transição válida entre estados, é conhecido como uma *solução* ou *prova* do problema
- Essencialmente vamos varrer uma estrutura entre estados ou nós, de modo sistemático até encontrarmos uma solução aceitável/desejável.
- Logo, vamos empregar alguns conceitos da teoria dos grafos, em modelar problemas e resolvê-los por um esquema de busca computacional



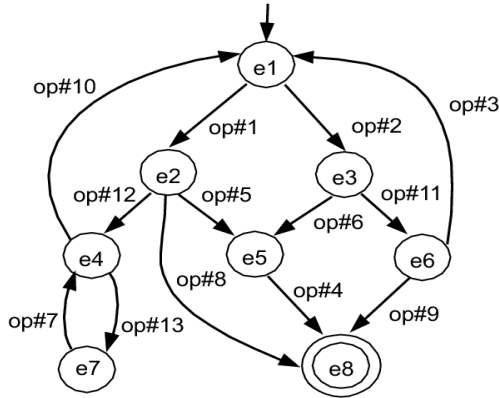
Problema do Robô no Labirinto



Imagine qualquer problema: uma busca na WEB, atomicidade de transações de BD, máquina de café, carro autônomo, etc



Problemas de Grafos se Transformam em Árvores de Buscas



Resumindo, os problemas são modelados em estruturas complexas, tais como grafos, mas o processo de solução se mantém: **realizar uma busca, tal como uma estrutura de uma árvore (listas) !**



Pseudo-código já em Picat

```
resolve(P) =>  
    inicio(Start),  
    busca(Start,[Start],Qsol),  
    imprime_saida(Qsol,P).
```

```
busca(S,P,P) ?=> objetivo(S).      % objetivo alcançado : FIM  
busca(S,Visited,P) =>  
    proximo_estado(S,Nxt),          % gera um proximo estado  
    estado_seguro(Nxt),             % verifica se este estado  
    sem_loop(Nxt,Visited),         % verifica se está em loop  
    busca(Nxt,[Nxt|Visited],P).    % continue a busca recursiva
```



```
sem_loop(Nxt,Visited) :-  
    \+member(Nxt,Visited).
```

```
proximo_estado(S,Nxt) =>    < fill in here >.  
estado_seguro(Nxt) =>      < fill in here >.  
sem_loop(Nxt,Visited) =>    < fill in here >.
```

```
inicio(...).  
objetivo(...).
```

Vamos reescrever este pseudo-código em um problema!



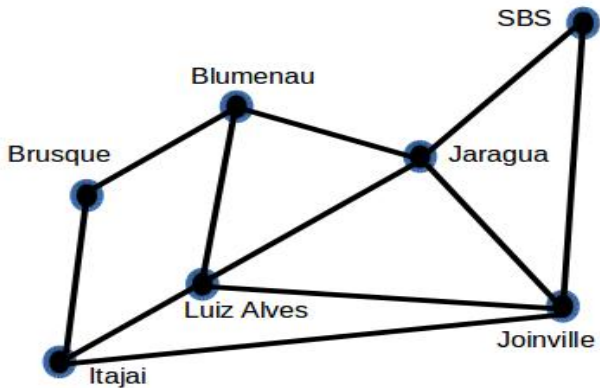
Caminho Hamiltoniano Aplicado



Seja um viajante que sai cedo de Joinville, e chegar a noite em Blumenau, passando por algumas destas cidades uma única vez!



Cidades Escolhidas pelo Viajante



Modelagem do Problema – *O nosso viajante do Vale do Itajaí I*

- Em nosso problema temos 7 cidades pré-escolhidas
- A lista de cidades são:

```
index(-) %% lista de todas cidade do mapa  
as_cidades( [ brusque, blumenau, itajai, luiz_alves,  
             jaragua, sao_bento, joinville ] ).
```

- Duas cidades em particular:

```
index(-)  
destino( blumenau ).
```

```
index(-)  
origem( joinville ).
```

- As estradas transitáveis entre as cidades definem o nosso mapa, conseqüentemente um grafo entre cidades:



Modelagem do Problema – *O nosso viajante do Vale do Itajaí II*

```
%% MAPA da região
index(-,-)
arco(joinville, sao_bento) .
arco(joinville, itajai) .
arco(joinville, jaragua) .
arco(joinville, luiz_alves) .
arco(jaragua, sao_bento) .
arco(jaragua, blumenau) .
arco(jaragua, luiz_alves) .
arco(itajai, luiz_alves) .
arco(blumenau, luiz_alves) .
arco(blumenau, itajai) .
arco(brusque, itajai) .
arco(brusque, blumenau) .
```

- As estradas entre as cidades são bidirecionais. Se há estrada para ir da cidade X a cidade Y então na outra direção é verdadeiro. Em regras isto é escrito por:



Modelagem do Problema – *O nosso viajante do Vale do Itajaí III*

```
/* BI-DIRECIONALIDADE DOS ARCOS */  
move_no(X,Y) ?=> arco(X,Y).  
move_no(X,Y) => arco(Y,X).
```

- Claro, este problema é pequeno e construindo o grafo dá para constatar que existe mais uma solução para o nosso viajante
- Para resolver este problema vamos utilizar uma **busca em profundidade**
- Esta **busca em profundidade** (do inglês. *depth first search* – *DFS*), encontra-se inserida no contexto *buscas em geral*, visto anteriormente.



```
busca_DFS ( [ No_corrente | Caminho] , L_sol) ?=>
    destino(No_final),          %%% condicao de parada 1
    No_corrente == No_final,
    L_sol = [ No_corrente | Caminho ],
    as_cidades(L_Todas_Cidades), %%% condicao de parada 2
    %% TODAS CIDADES FORAM VISITADAS
    length (L_sol) == length(L_Todas_Cidades),
    write(L_sol),
    printf(" \n UMA SOLUCAO ....: OK\n ==>").
```

```
busca_DFS ( [NoH | Caminho], Solucao) =>
    %%% explorar um novo movimento ou um novo noh
    move_no(NoH , Novo_NoH),
    %% testar se este novo noh nao foi visitado ainda
    %% ou novo_NOH eh permitido
    not( member(Novo_NoH, [NoH|Caminho]) ),
    busca_DFS( [Novo_NoH , NoH | Caminho ] , Solucao).
```



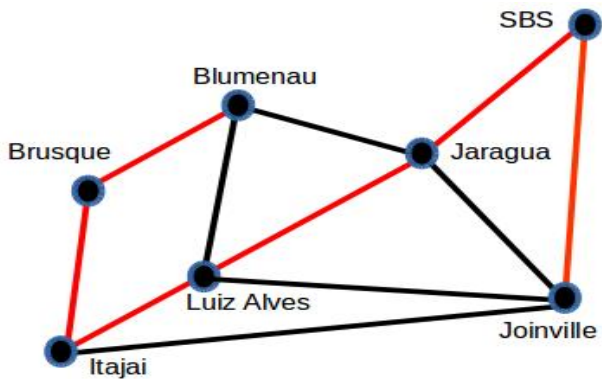
- Acompanhar as explicações do código de:
`https://github.com/claudiosa/CCS/blob/master/picat/hamiltoniano_DFS.pi`
- Muitos elementos da linguagem neste código
- Confira a execução



```
$ picat hamiltoniano_DFS.pi
.....
[blumenau,brusque,itajai,luiz_alves,jaragua,sao_bento,joinville]
UMA SOLUCAO ....: OK
==>joinville : sao_bento : jaragua : luiz_alves : itajai : brusque
: blumenau :
Cidade Inicial: joinville
Cidade Final: blumenau
Total de cidades visitadas: 7
CPU time 0.000000 em SEGUNDOS
OVERALL PICAT CPU time 0.013000 em SEGUNDOS
Backtrackings total 0
=====
[ccs@gerzat picat]$
```

Existe uma função chamada **findall**, cujo valor de retorno é uma lista com todas soluções possíveis de um dado predicado!





- A recursividade na modelagem das buscas, define **uma metodologia** de se *programar em lógica* e resolver problemas



- A recursividade na modelagem das buscas, define **uma metodologia** de se *programar em lógica* e resolver problemas
- A área de buscas é ampla e apresenta muitas variações. Apresentamos nesta seção um **núcleo mágico**, que pode ser utilizado em muitas outras estratégias–métodos de buscas.



- A recursividade na modelagem das buscas, define **uma metodologia** de se *programar em lógica* e resolver problemas
- A área de buscas é ampla e apresenta muitas variações. Apresentamos nesta seção um **núcleo mágico**, que pode ser utilizado em muitas outras estratégias–métodos de buscas.
- Praticamente todos os métodos de buscas fazem o uso extensivo das **listas** em problemas complexos



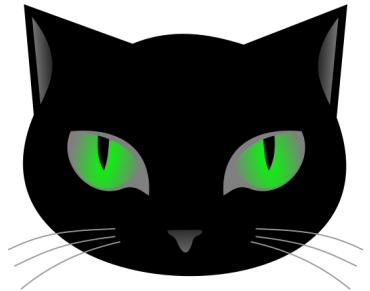
- A recursividade na modelagem das buscas, define **uma metodologia** de se *programar em lógica* e resolver problemas
- A área de buscas é ampla e apresenta muitas variações. Apresentamos nesta seção um **núcleo mágico**, que pode ser utilizado em muitas outras estratégias–métodos de buscas.
- Praticamente todos os métodos de buscas fazem o uso extensivo das **listas** em problemas complexos
- Aos problemas complexos, há outras técnicas de programação para resolvê-los.



- A recursividade na modelagem das buscas, define **uma metodologia** de se *programar em lógica* e resolver problemas
- A área de buscas é ampla e apresenta muitas variações. Apresentamos nesta seção um **núcleo mágico**, que pode ser utilizado em muitas outras estratégias–métodos de buscas.
- Praticamente todos os métodos de buscas fazem o uso extensivo das **listas** em problemas complexos
- Aos problemas complexos, há outras técnicas de programação para resolvê-los.
- Assunto das próximas seções: PD, Planejamento e CP



- O que é PD?
- Importância
- Exemplo



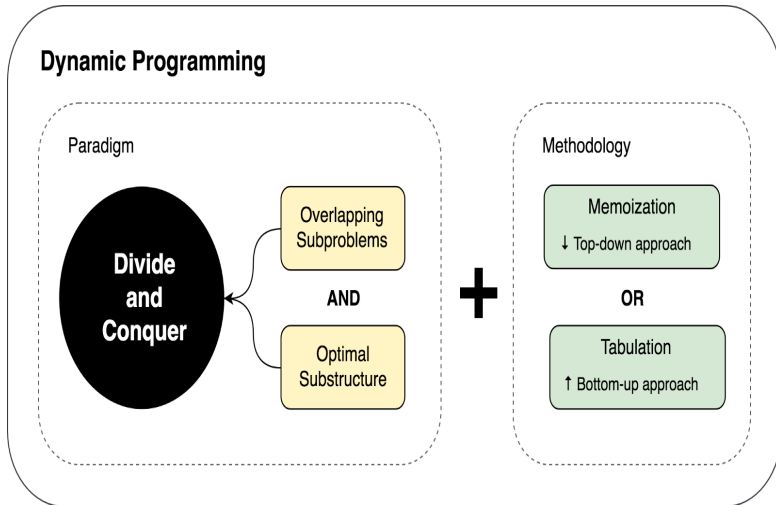


Figura 7: Conceitos da Programação Dinâmica – (PD)



- Uma poderosa técnica de programação que contorna a complexidade de certos problemas exponenciais



- Uma poderosa técnica de programação que contorna a complexidade de certos problemas exponenciais
- O problema **deve** apresentar uma regra de recorrência



- Uma poderosa técnica de programação que contorna a complexidade de certos problemas exponenciais
- O problema **deve** apresentar uma regra de recorrência
- A idéia é que todos os cálculos feitos a partir desta *regra de recorrência*, são consultados e armazenados numa *tabela dinâmica*



- Uma poderosa técnica de programação que contorna a complexidade de certos problemas exponenciais
- O problema **deve** apresentar uma regra de recorrência
- A idéia é que todos os cálculos feitos a partir desta *regra de recorrência*, são consultados e armazenados numa *tabela dinâmica*
- Esta técnica de utilizar uma *tabela dinâmica* nos cálculos intermediários, evitando a repetição do que já foi calculado anteriormente, é conhecida como: Programação Dinâmica, ou simplesmente: PD



- Como Picat usa a recursão, na programação em lógica, nada mais natural do que esta ter a PD disponível



- Como Picat usa a recursão, na programação em lógica, nada mais natural do que esta ter a PD disponível
- O comando que cria uma tabela para um determinado predicado é o *tabling*



- Como Picat usa a recursão, na programação em lógica, nada mais natural do que esta ter a PD disponível
- O comando que cria uma tabela para um determinado predicado é o *tabling*
- O *tabling* é um dos elementos fortes do planejador do Picat (módulo *planner*)



- Como Picat usa a recursão, na programação em lógica, nada mais natural do que esta ter a PD disponível
- O comando que cria uma tabela para um determinado predicado é o *tabling*
- O *tabling* é um dos elementos fortes do planejador do Picat (módulo *planner*)
- O exemplo escolhido para ilustrar a PD em Picat, veio do texto *Modeling and Solving AI Problems in Picat*, de Roman Barták e Neng-Fa



Exemplo de Uso da Programação Dinâmica – (PD)

- Seja o binômio $(x + y)^n$, conhecido *Binômio de Newton*



Exemplo de Uso da Programação Dinâmica – (PD)

- Seja o binômio $(x + y)^n$, conhecido *Binômio de Newton*
- Casos particulares são:
- $(x + y)^0 = 1$
- $(x + y)^1 = x + y$
- $(x + y)^2 = x^2 + 2xy + y^2$



Exemplo de Uso da Programação Dinâmica – (PD)

- Seja o binômio $(x + y)^n$, conhecido *Binômio de Newton*
- Casos particulares são:
- $(x + y)^0 = 1$
- $(x + y)^1 = x + y$
- $(x + y)^2 = x^2 + 2xy + y^2$
- $(x + y)^2 = x^2y^0 + 2x^1y^1 + x^0y^2$
- $(x + y)^3 = x^3y^0 + 3x^2y^1 + 3x^1y^2 + x^0y^3$
- $(x + y)^4 = x^4y^0 + 4x^3y^1 + 6x^2y^2 + 4x^1y^3 + x^0y^4.$
-



Exemplo de Uso da Programação Dinâmica – (PD)

- Seja o binômio $(x + y)^n$, conhecido *Binômio de Newton*
- Casos particulares são:
 - $(x + y)^0 = 1$
 - $(x + y)^1 = x + y$
 - $(x + y)^2 = x^2 + 2xy + y^2$
 - $(x + y)^2 = x^2y^0 + 2x^1y^1 + x^0y^2$
 - $(x + y)^3 = x^3y^0 + 3x^2y^1 + 3x^1y^2 + x^0y^3$
 - $(x + y)^4 = x^4y^0 + 4x^3y^1 + 6x^2y^2 + 4x^1y^3 + x^0y^4.$
 -
- Como obter estes coeficientes polinômios?



Exemplo de Uso da Programação Dinâmica – (PD)

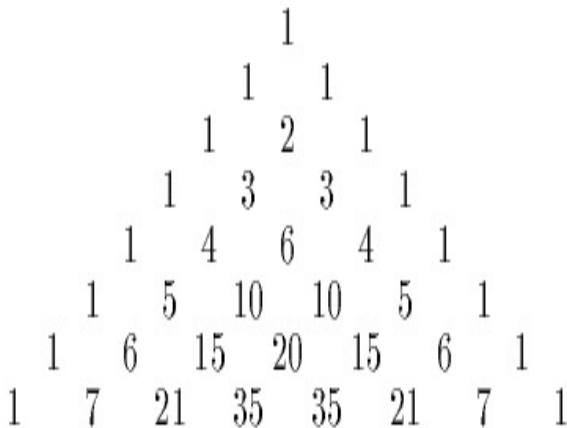


Figura 8: O triângulo de Pascal



Exemplo de Uso da Programação Dinâmica – (PD)

A Triângulo de Pascal

	p_0	1	2	3	4	5	6
N							
0	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$						1
1	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \end{pmatrix}$					$\underline{1} \quad \underline{1}$
2	$\begin{pmatrix} 2 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 2 \end{pmatrix}$				$\underline{1} \quad \underline{2} \quad 1$
3	$\begin{pmatrix} 3 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 3 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 3 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 3 \\ 3 \end{pmatrix}$			$\underline{1} \quad \underline{3} \quad \underline{3} \quad 1$
4	$\begin{pmatrix} 4 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 3 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 4 \end{pmatrix}$		$\underline{1} \quad \underline{4} \quad 6 \quad 4 \quad 1$
5	$\begin{pmatrix} 5 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 5 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 5 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 5 \\ 3 \end{pmatrix}$	$\begin{pmatrix} 5 \\ 4 \end{pmatrix}$	$\begin{pmatrix} 5 \\ 5 \end{pmatrix}$	$\underline{1} \quad \underline{5} \quad 10 \quad \underline{10} \quad 5 \quad 1$
6	$\begin{pmatrix} 6 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 6 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 6 \\ 2 \end{pmatrix}$	$\begin{pmatrix} 6 \\ 3 \end{pmatrix}$	$\begin{pmatrix} 6 \\ 4 \end{pmatrix}$	$\begin{pmatrix} 6 \\ 5 \end{pmatrix}$	$\begin{pmatrix} 6 \\ 6 \end{pmatrix}$
							1 6 15 20 15 6 1

Figura 9: O triângulo de Pascal – Coeficientes Binomiais



- O *coeficiente binomial*, também chamado de *número binomial*, de um número n , na classe k , consiste no número de combinações de n termos, k a k .



- O *coeficiente binomial*, também chamado de *número binomial*, de um número n , na classe k , consiste no número de combinações de n termos, k a k .
- O número binomial de um número n , na classe k , pode ser escrito como:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$



- Alternativa ao cálculo do fatorial, tem-se a relação de Stiffel:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



- Alternativa ao cálculo do fatorial, tem-se a relação de Stiffel:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- O coeficiente binomial é muito utilizado no Triângulo de Pascal, onde o termo na linha n e coluna k é dado por: $\binom{n-1}{k-1}$



- Alternativa ao cálculo do fatorial, tem-se a relação de Stiffel:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

- O coeficiente binomial é muito utilizado no Triângulo de Pascal, onde o termo na linha n e coluna k é dado por: $\binom{n-1}{k-1}$
- A fórmula de Stiffel é recorrente, e diretamente escrita em Picat.
Veja os códigos.



```
import datetime.    %%% para o statistics
import util.
```

```
table
c(_, 0) = 1.
c(N, N) = 1.
c(N,K) = c(N-1, K-1) + c(N-1, K).
```



```

main ?=>
    statistics(runtime,_), % faz uma marca do 1o. statistics
    N = 10, %% ateh uns 30 ... são números grandes ... fatorial
    foreach(I in 0 .. N)
        foreach(J in 0 .. I)
            printf("  %d", c(I,J))
            end,
            printf(" \n"),
        end,
    statistics(runtime, [T_Picat_ON, T_final]),
    T = (T_final) / 1000.0, %%% está em milisegundos
    printf("\n CPU time %f em SEGUNDOS ", T),
    printf("\n OVERALL PICAT CPU time %f em SEGUNDOS ", T_Picat_ON/1000
    printf(" \n =====\n ")
    %%% , fail descomente para multiplas solucoes
.
main => printf("\n Para uma solução .... !!!!" ) .

```



- Acompanhar as explicações do código de:
`https://github.com/claudiosa/CCS/blob/master/picat/coeficiente_binomial_PD.pi`
- Confira a execução



```
[ccs@gerzat picat]$ picat coeficiente_binomial_PD.pi
```

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

```
CPU time 0.000000 em SEGUNDOS
```

```
OVERALL PICAT CPU time 0.009000 em SEGUNDOS
```

```
=====
```



- Há outros métodos para se resolver estes problemas



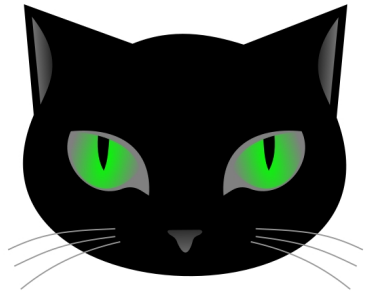
- Há outros métodos para se resolver estes problemas
- O comando *tabling* é a base do módulo *planner*



- Há outros métodos para se resolver estes problemas
- O comando *tabling* é a base do módulo *planner*
- Uma estratégia de programação bem poderosa



- O que é Planejamento?
- Importância da área
- Muitas definições
- Exemplo



- Requisitos: conceitos de listas e recursividade dominados!



- Requisitos: conceitos de listas e recursividade dominados!
- Além destes: conceitos grafos, árvores de busca, nós, etc



- Requisitos: conceitos de listas e recursividade dominados!
- Além destes: conceitos grafos, árvores de busca, nós, etc
- *Planejamento* é um termo amplo e em vários domínios



- Requisitos: conceitos de listas e recursividade dominados!
- Além destes: conceitos grafos, árvores de busca, nós, etc
- *Planejamento* é um termo amplo e em vários domínios
- O que **não** é o nosso contexto de *planejamento* ?
Exemplo: planejamento estratégico das empresas, planejar como distribuir os dividendos da empresa, orçamento familiar, etc



- Requisitos: conceitos de listas e recursividade dominados!
- Além destes: conceitos grafos, árvores de busca, nós, etc
- *Planejamento* é um termo amplo e em vários domínios
- O que **não** é o nosso contexto de *planejamento* ?
Exemplo: planejamento estratégico das empresas, planejar como distribuir os dividendos da empresa, orçamento familiar, etc
- O que é o nosso contexto de *planejamento* ?



- Requisitos: conceitos de listas e recursividade dominados!
- Além destes: conceitos grafos, árvores de busca, nós, etc
- *Planejamento* é um termo amplo e em vários domínios
- O que **não** é o nosso contexto de *planejamento* ?
Exemplo: planejamento estratégico das empresas, planejar como distribuir os dividendos da empresa, orçamento familiar, etc
- O que é o nosso contexto de *planejamento* ? Questões que envolvam um ambiente, um agente (um programa, um robô, etc), sensores, e ações que modifiquem estados.
Exemplo clássico: robótica em geral



- Problemas em geral necessitam de um **plano** para serem solucionados, assim, há uma visão que encontrar um plano para um problema, é ter uma solução!



- Problemas em geral necessitam de um **plano** para serem solucionados, assim, há uma visão que encontrar um plano para um problema, é ter uma solução!
- Em resumo, a área de planejamento é bem complexa, antiga na área da IA e robótica (1970 – STRIPS), efervescente, e de muito interesse na indústria.



- Problemas em geral necessitam de um **plano** para serem solucionados, assim, há uma visão que encontrar um plano para um problema, é ter uma solução!
- Em resumo, a área de planejamento é bem complexa, antiga na área da IA e robótica (1970 – STRIPS), efervescente, e de muito interesse na indústria.
- Várias abordagens sobre a visão clássica da IA. Mas temos evoluções significativas ...



- Problemas em geral necessitam de um **plano** para serem solucionados, assim, há uma visão que encontrar um plano para um problema, é ter uma solução!
- Em resumo, a área de planejamento é bem complexa, antiga na área da IA e robótica (1970 – STRIPS), efervescente, e de muito interesse na indústria.
- Várias abordagens sobre a visão clássica da IA. Mas temos evoluções significativas ...
- PDDL (*Planning Domain Definition Language*): unanimidade (ou próxima a esta) entre os pesquisadores de planejamento, como linguagem descritora de problemas de planejamento.



- Problemas em geral necessitam de um **plano** para serem solucionados, assim, há uma visão que encontrar um plano para um problema, é ter uma solução!
- Em resumo, a área de planejamento é bem complexa, antiga na área da IA e robótica (1970 – STRIPS), efervescente, e de muito interesse na indústria.
- Várias abordagens sobre a visão clássica da IA. Mas temos evoluções significativas ...
- PDDL (*Planning Domain Definition Language*): unanimidade (ou próxima a esta) entre os pesquisadores de planejamento, como linguagem descritora de problemas de planejamento.
- Vários problemas ainda sem solução, pois a complexidade é exponencial



- Plano: seqüência ordenada de ações



- Plano: seqüência ordenada de ações
 - problema: obter banana, leite e uma furadeira (nesta ordem)



- Plano: seqüência ordenada de ações
 - problema: obter banana, leite e uma furadeira (nesta ordem)
 - plano: ir ao supermercado, ir à seção de frutas, pegar as bananas, ir à seção de leite, pegar uma caixa de leite, ir ao caixa, pagar tudo, ir a uma loja de ferramentas, ..., voltar para casa.



- Plano: seqüência ordenada de ações
 - problema: obter banana, leite e uma furadeira (nesta ordem)
 - plano: ir ao supermercado, ir à seção de frutas, pegar as bananas, ir à seção de leite, pegar uma caixa de leite, ir ao caixa, pagar tudo, ir a uma loja de ferramentas, ..., voltar para casa.
- Um Planejador: Combina conhecimento de um ambiente, um agente e suas ações possíveis, entradas (luz, cor, cheiro, sensor, etc), um estado corrente e/ou inicial, e com isto resolve de problemas planejar seqüência de ações, que mudam de estados a cada ação, até atingir um estado final.



Exemplos do que é planejamento ...

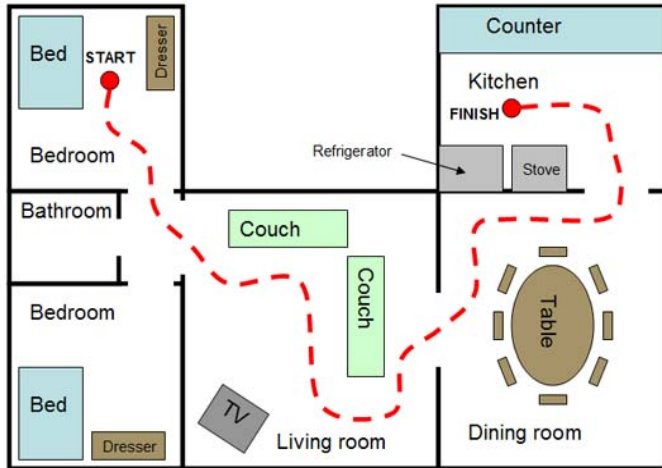


Figura 10: A fome no meio da noite!



Exemplos do que é planejamento ...

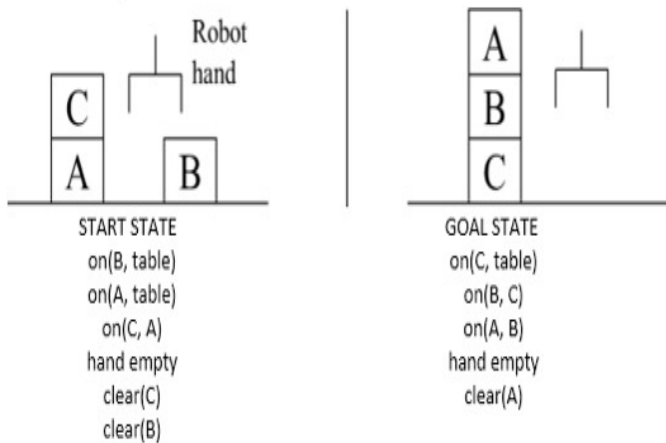


Figura 11: O mundo dos blocos



Graph of state space

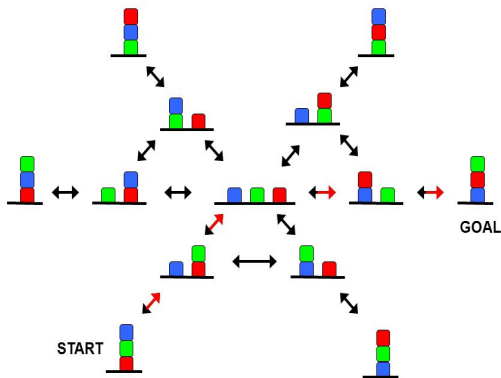


Figura 12: O espaço de estados do *mundo dos blocos* \times ações



- Plano: uma sequência ordenada de ações, criada incrementalmente a partir do estado inicial
Ex. posições das peças de um jogo

$$S_1 < S_2 < \dots < S_n$$

- Ambiente: onde um programa–agente vai receber entradas em um determinado estado e atuar com uma ação apropriada
- Estados: descrição completa de possíveis estados atingíveis
Problema: quanto aos estados não-previstos, inacessíveis?
- Estado inicial: um estado particular onde nosso programa–agente inicia a sua busca
- Objetivos: estados desejados que o programa–agente precisa alcançar, isto é, um dos *estados finais* desejados



- Percepções: cheiro, brisa, luz, choque, som, posições ou coordenadas, vizinhanças, etc
- Ações: provocam modificações entre os estados corrente e sucessor
Exemplos: avançar para próxima célula, girar 90 graus à direita ou à esquerda pegar um objeto, atirar na direção do alvo, etc
- Operadores: vocabulário ou repertório de atuações atômicas do que o agente pode fazer.
Exemplos: *pegar*(X), *mover_de*(X, Y), *levantar*(X), *livre*(X), etc
- Uma eventual confusão para iniciantes: um ação é um conjunto de um ou mais operadores, e ainda, **a ação é condicional**. A ação só é disparada se as condições de pré-requisitos forem satisfeitas.



- Heurística: alguma função que indica o progresso sobre os estados não visitados e sua convergência para uma finalização do plano



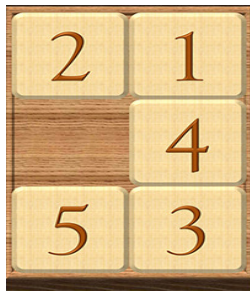


Figura 13: Um quebra-cabeça (2×3 ou 3×2) *simplificado* do conhecido 3×3



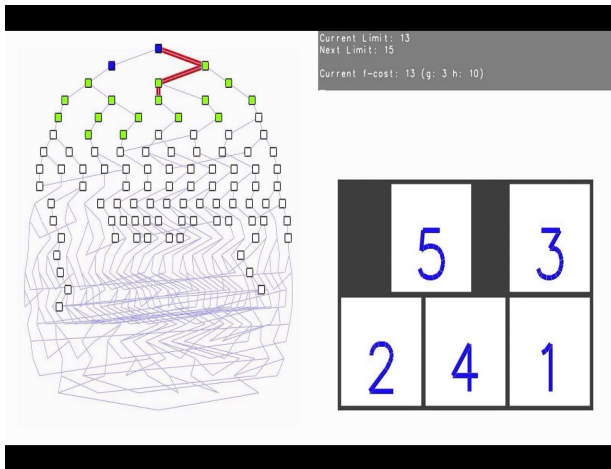


Figura 14: Sim, *simplificado* mas não muito!




```
/*
```

```
A  B  C
```

```
D  E  F
```

```
*/
```

```
%%%%%%%%%
```

```
%  1 5 %
```

```
% 4 3 2 %
```

```
%%%%%%%%%
```

```
import datetime.
```

```
import planner.
```



```
index(-)  
estado_inicial( [0,1,5,4,3,2] ).
```

```
%% funcao final do planner  
final( [1,2,3,4,5,0] ) => true .
```



```
% Up <-> Down
/* Descrevendo as possiveis acoes para o planner */
action([A,B,C, D,E,F], S1, Acao, Custo_Acao ) ?=>
    Custo_Acao = 1,
    ( A == 0 ), %% conj. condicoes
    S1 = [D,B,C, 0,E,F],
    Acao = ($up(D),S1). %%a acao + estado modificado

action([A,B,C, D,E,F], S1, Acao, Custo_Acao ) ?=>
    Custo_Acao = 1,
    (A == 0 ), %% conj. condicoes
    S1 = [0,B,C, A,E,F],
    Acao = ($dow(A),S1). %%a acao + estado modificado
```

.....



```
% Left <-> Right
action([A,B,C, D,E,F], S1, Acao, Custo_Acao ) ?=>
    Custo_Acao = 1,
    (A == 0), %% conj. condicoes
    S1 = [B,0,C, D,E,F],
    Acao = ($left(B), S1). %%a acao + estado modificado

action([A,B,C, D,E,F], S1, Acao, Custo_Acao ) ?=>
    Custo_Acao = 1,
    (B == 0), %% conj. condicoes
    S1 = [0,A,C, D,E,F],
    Acao = ($right(A), S1). %%a acao + estado modificado
.....
```



```
main ?=>
    estado_inicial( Q ),

    best_plan_unbounded( Q , Sol_Acoes),
    println(sol=Sol_Acoes),

    printf("\n Estado Inicial: "),
    w_Quadro( Q ),
    w_L_Estado( Sol_Acoes ),

    Total := length(Sol_Acoes) ,
    Num_Movts := (Total -1) ,
    printf("\n Inicial (estado): %w ", Q),
    printf("\n Total de acoes: %d", Total),
    printf(" \n =====\n ")
    %%% , fail descomente para multiplas solucoes
    .
```



```
main => printf("\n Para uma solução .... !!!!" ) .
```

```
.....
```



- Acompanhar as explicações do código de:
`https://github.com/claudiosa/CCS/blob/master/picat/puzzle_2x3_planner.pi`
- Confira a execução



```
[ccs@gerzat picat]$ picat puzzle_2x3_planner.pi
sol = [(left(1),[1,0,5,4,3,2]),(left(5),[1,5,0,4,3,2]),
(up(2),[1,5,2,4,3,0]),(right(3),[1,5,2,4,0,3]),(dow(5),[1,0,2,4,5,3]),
(left(2),[1,2,0,4,5,3]),(up(3),[1,2,3,4,5,0])]
```

Estado Inicial:

```
0 1 5
4 3 2
```

Acao: left(1)

```
1 0 5
4 3 2
```

Acao: left(5)

```
1 5 0
4 3 2
```




```
.....  
Acao: left(2)  
  1 2 0  
  4 5 3  
  
Acao: up(3)  
  1 2 3  
  4 5 0  
  
Inicial  (estado): [0,1,5,4,3,2]  
Total de acoes: 7  
=====
```



- O que efetivamente voce precisa saber



- O que efetivamente voce precisa saber
- Importar um módulo
`import planner.`



- O que efetivamente voce precisa saber
- Importar um módulo
`import planner.`
- O predicado: *final*
`final(S,Plan,Cost) => Plan=[], Cost=0, final(S).`



- O que efetivamente voce precisa saber
- Importar um módulo
`import planner.`
- O predicado: *final*
`final(S,Plan,Cost) => Plan=[], Cost=0, final(S).`
- O predicado *action*
`action(S,NextS,Action,ActionCost)`



- A eficiência do planner do Picat se dá devido a sua combinação de técnicas: busca em profundidade (e variações) e Programação Dinâmica (PD), com o *tabling*



- A eficiência do planner do Picat se dá devido a sua combinação de técnicas: busca em profundidade (e variações) e Programação Dinâmica (PD), com o *tabling*
- O núcleo de busca dos planejadores disponíveis no Picat são de 2 tipos:
 - ① Usam um busca em profundidade com limites (*Depth-Bounded Search*)
 - ② Usam um busca em profundidade ilimitada de recursos (*Depth-Unbounded Search*)
- Contudo, estes 2 tipos apresentam muitas variações e opções:



- A eficiência do planner do Picat se dá devido a sua combinação de técnicas: busca em profundidade (e variações) e Programação Dinâmica (PD), com o *tabling*
- O núcleo de busca dos planejadores disponíveis no Picat são de 2 tipos:
 - ① Usam um busca em profundidade com limites (*Depth-Bounded Search*)
 - ② Usam um busca em profundidade ilimitada de recursos (*Depth-Unbounded Search*)
- Contudo, estes 2 tipos apresentam muitas variações e opções: Sem escapatória \Rightarrow consultar o manual do Picat (*User Guide to Picat*)
- No exemplo aqui apresentado:
`best_plan_unbounded(S,Plan)`



- Outros métodos para se resolver estes problemas



- Outros métodos para se resolver estes problemas
- Mas perdemos na portabilidade de usar em outros planejadores



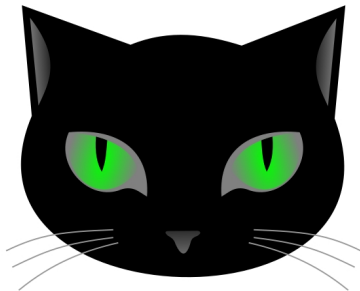
- Outros métodos para se resolver estes problemas
- Mas perdemos na portabilidade de usar em outros planejadores
- Os modelos escritos em PDDL (*Planning Domain Definition Language*) facilmente portáveis para Picat



- Outros métodos para se resolver estes problemas
- Mas perdemos na portabilidade de usar em outros planejadores
- Os modelos escritos em PDDL (*Planning Domain Definition Language*) facilmente portáveis para Picat
- Sob um uso mais restrito, um modelo em PDDL é executado diretamente em Picat



- Conceituar a PR
- Princípios
- 03 exemplos
- 03 técnicas



- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**



- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**
- Uma poderosa teoria (e técnica) que contorna a complexidade de certos problemas exponenciais



- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**
- Uma poderosa teoria (e técnica) que contorna a complexidade de certos problemas exponenciais
- A PR encontrava-se inicialmente dentro da IA e PO, mas como várias outras, tornaram-se fortes e autônomas. Atualmente uma área de pesquisa bem forte em alguns países.



- A Programação por Restrições (PR) é conhecida por *Constraint Programming* ou simplesmente **CP**
- Uma poderosa teoria (e técnica) que contorna a complexidade de certos problemas exponenciais
- A PR encontrava-se inicialmente dentro da IA e PO, mas como várias outras, tornaram-se fortes e autônomas. Atualmente uma área de pesquisa bem forte em alguns países.
- Nesta seção, temos 3 exemplos ilustrar conceitos da PR



- Aproximadamente o algoritmo da PR é dado:



- Aproximadamente o algoritmo da PR é dado:
 - ① Avaliar algebricamente os domínios das variáveis com suas restrições
 - ② Intercala iterativamente a propagação de restrições com um algoritmo de busca
 - ③ A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
 - ④ Volte ao passo inicial até que os domínios permaneçam estáticos e que as variáveis apresentem instâncias consistentes



- Aproximadamente o algoritmo da PR é dado:
 - ① Avaliar algebricamente os domínios das variáveis com suas restrições
 - ② Intercala iterativamente a propagação de restrições com um algoritmo de busca
 - ③ A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
 - ④ Volte ao passo inicial até que os domínios permaneçam estáticos e que as variáveis apresentem instâncias consistentes
- Este núcleo é uma busca por constantes otimizações



- Aproximadamente o algoritmo da PR é dado:
 - ① Avaliar algebricamente os domínios das variáveis com suas restrições
 - ② Intercala iterativamente a propagação de restrições com um algoritmo de busca
 - ③ A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
 - ④ Volte ao passo inicial até que os domínios permaneçam estáticos e que as variáveis apresentem instâncias consistentes
- Este núcleo é uma busca por constantes otimizações
- Uma das virtudes da PR: a legibilidade e clareza de suas soluções



- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR



- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR
- Quando temos problemas que precisamos conhecer **todas** as respostas, não apenas a melhor resposta



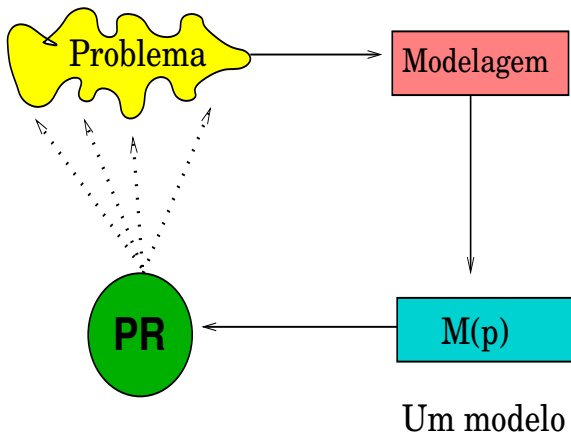
- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR
- Quando temos problemas que precisamos conhecer **todas** as respostas, não apenas a melhor resposta
- Quando necessitamos de respostas *precisas* e não apenas as aproximadas. Há um custo computacional a ser pago aqui!

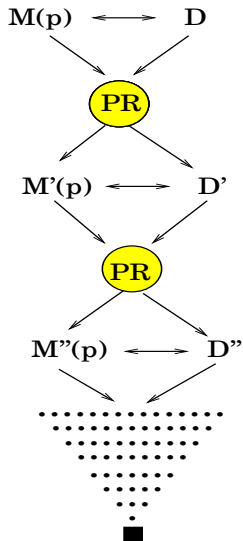


- Problemas combinatoriais com domínio nos inteiros são bons candidatos a serem resolvidos por PR
- Quando temos problemas que precisamos conhecer **todas** as respostas, não apenas a melhor resposta
- Quando necessitamos de respostas *precisas* e não apenas as aproximadas. Há um custo computacional a ser pago aqui!
-



Metodologia da Construção de Modelos





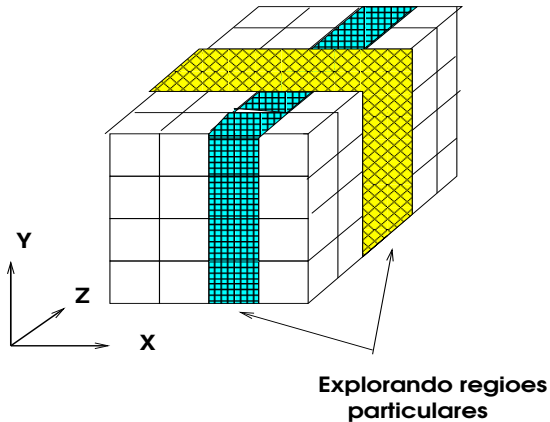


Figura 15: Realizar buscas com regiões reduzidas – promissoras (regiões factíveis de soluções)



Redução Iterativa em Sub-problemas

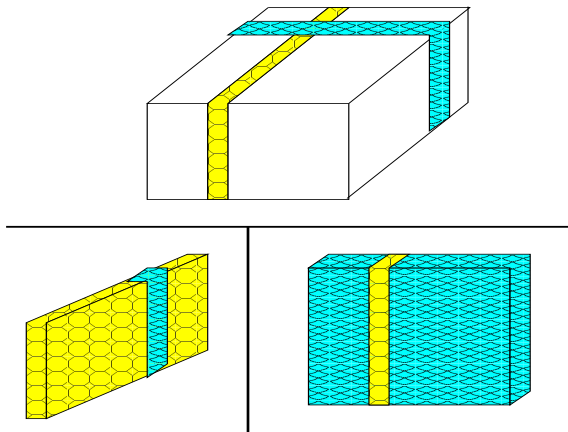


Figura 16: Redução de um CP em outros sub-problemas CPs equivalentes



A PR tem os seguintes elementos:



A PR tem os seguintes elementos:

- Um conjunto de **variáveis**: $X_1, X_2, X_3, \dots, X_n$



A PR tem os seguintes elementos:

- Um conjunto de **variáveis**: $X_1, X_2, X_3, \dots, X_n$
- Um conjunto de **domínios** dessas variáveis: $D_{X_1}, D_{X_2}, D_{X_3}, \dots, D_{X_n}$



A PR tem os seguintes elementos:

- Um conjunto de **variáveis**: $X_1, X_2, X_3, \dots, X_n$
- Um conjunto de **domínios** dessas variáveis: $D_{X_1}, D_{X_2}, D_{X_3}, \dots, D_{X_n}$
- Finalmente, as **restrições**, que são relações n-árias entre estas variáveis



A PR tem os seguintes elementos:

- Um conjunto de **variáveis**: $X_1, X_2, X_3, \dots, X_n$
- Um conjunto de **domínios** dessas variáveis: $D_{X_1}, D_{X_2}, D_{X_3}, \dots, D_{X_n}$
- Finalmente, as **restrições**, que são relações n-árias entre estas variáveis
- Exemplo: $D_{X_1} = D_{X_2} = \{3, 4\}$ e $X_1 \neq X_2$



- Para o exemplo anterior um código em Picat é dado por:



- Para o exemplo anterior um código em Picat é dado por:
 - `[X1, X2] :: 3..4`
 - `X1 #!= X2`



- Para o exemplo anterior um código em Picat é dado por:
 - `[X1, X2] :: 3..4`
 - `X1 #!= X2`
- Em resumo, as relações da PR tem o símbolo '#'



- Para o exemplo anterior um código em Picat é dado por:
 - `[X1, X2] :: 3..4`
 - `X1 #!= X2`
- Em resumo, as relações da PR tem o símbolo `'#'`
- Para tornar toda esta sintaxe da PR disponível, Picat tem um módulo para suporte da PR: `import cp`



Exemplo – 01 – Soma de Números Primos

- Dado um número par qualquer, encontre dois de números primos, N_1 e N_2 , diferentes entre si, que somados dêem este número par.



- Dado um número par qualquer, encontre dois de números primos, N_1 e N_2 , diferentes entre si, que somados dêem este número par.

- Exemplo:

Seja o PAR = 18

Uma solução:

$$N_1 = 7 \text{ e } N_2 = 11$$

pois

$$N_1 + N_2 = 18$$



- N_1 e N_2 assumem valores no domínio dos números primos. Logo, é importante ter os números primos prontos!



- N_1 e N_2 assumem valores no domínio dos números primos. Logo, é importante ter os números primos prontos!
- A soma destes números é o par fornecido como entrada, N_{PAR} :
$$N_1 + N_2 = N_{PAR}$$



- N_1 e N_2 assumem valores no domínio dos números primos. Logo, é importante ter os números primos prontos!
- A soma destes números é o par fornecido como entrada, N_{PAR} :
$$N_1 + N_2 = N_{PAR}$$
- N_1 e N_2 são diferentes entre si
$$N_1 \neq N_2$$



- N_1 e N_2 assumem valores no domínio dos números primos.
Logo, é importante ter os números primos prontos!
- A soma destes números é o par fornecido como entrada, N_{PAR} :
$$N_1 + N_2 = N_{PAR}$$
- N_1 e N_2 são diferentes entre si
$$N_1 \neq N_2$$
- Como são inteiros: $N_1 < N_{PAR}$ e $N_2 < N_{PAR}$
Sim, é óbvio, mas isto faz uma redução significativa de domínio!



- Acompanhar as explicações do código de:
`https://github.com/claudiosa/CCS/blob/master/picat/soma_N1_N2_primos_CP.pi`
- Confira a execução e testes



```
modelo =>  
    PAR = 382,  
    Variaveis = [N1,N2],  
    % Gerando um domino soh de primos  
    % L_dom = [I : I in 1..1000, eh_primo(I) == true],    %0U  
    L_dom = [I : I in 1..1000, prime(I)],  
    Variaveis :: L_dom,
```

Uma ótima estratégia: sair com um domínio de números candidatos!



```
% RESTRICOES
N1 #!= N2,
N1 #< PAR,
N2 #< PAR,
N1 + N2 #= PAR,

% A BUSCA
solve([ff], Variaveis),
    % UMA SAIDA
printf("\n  N1: %d\t N2: %d", N1,N2),
printf("\n.....")
.
```



```
import cp.  
  
% main => modelo .  
% main ?=> modelo, fail.  
% main => true.  
  
main =>  
    L = findall(_, $modelo),  
    writef("\n Total de solucoes:  %d \n", length(L)) .
```




```
Picat> cl('soma_N1_N2_primos_CP').  
Compiling:: soma_N1_N2_primos_CP.pi  
** Warning   : redefine_preimported_symbol(math): prime / 1  
soma_N1_N2_primos_CP.pi compiled in 7 milliseconds  
loading...
```

yes

```
Picat> main.
```

```
    N1: 3   N2: 379
```

```
.....
```

```
    N1: 23  N2: 359
```

```
.....
```

```
    N1: 29  N2: 353
```

```
.....
```



```
.....  
N1: 353  N2: 29  
.....  
N1: 359  N2: 23  
.....  
N1: 379  N2: 3  
.....  
Total de solucoes:  18  
  
yes  
  
Picat>
```



- Seja um Posto Atendimento Médico, um PA, com 4 consultórios e 7 especialidades médicas



- Seja um Posto Atendimento Médico, um PA, com 4 consultórios e 7 especialidades médicas
- O problema é distribuir estes médicos nestes 4 consultórios tal que alguns requisitos sejam atendidos (restrições satisfeitas)



- Seja um Posto Atendimento Médico, um PA, com 4 consultórios e 7 especialidades médicas
- O problema é distribuir estes médicos nestes 4 consultórios tal que alguns requisitos sejam atendidos (restrições satisfeitas)
- A abordagem aqui é ingênua e sem muitos critérios



- Vamos usar uma matriz bi-dimensional para representar o problema. Linhas \leftrightarrow consultórios (1 a 4), e as colunas \leftrightarrow dias da semana (1 a 5)



- Vamos usar uma matriz bi-dimensional para representar o problema. Linhas \leftrightarrow consultórios (1 a 4), e as colunas \leftrightarrow dias da semana (1 a 5)
- Esta matriz será preenchida com valores/códigos de 1 a 7, de acordo com a especialidade médica.



- Vamos usar uma matriz bi-dimensional para representar o problema. Linhas \leftrightarrow consultórios (1 a 4), e as colunas \leftrightarrow dias da semana (1 a 5)
- Esta matriz será preenchida com valores/códigos de 1 a 7, de acordo com a especialidade médica.
- Assim o domínio da matriz Quadro (4×5) será preenchida com um destes códigos.



- Vamos usar uma matriz bi-dimensional para representar o problema. Linhas \leftrightarrow consultórios (1 a 4), e as colunas \leftrightarrow dias da semana (1 a 5)
- Esta matriz será preenchida com valores/códigos de 1 a 7, de acordo com a especialidade médica.
- Assim o domínio da matriz Quadro (4×5) será preenchida com um destes códigos.
- Vamos utilizar restrições globais: `member` e `all_different`



- Vamos usar uma matriz bi-dimensional para representar o problema. Linhas \leftrightarrow consultórios (1 a 4), e as colunas \leftrightarrow dias da semana (1 a 5)
- Esta matriz será preenchida com valores/códigos de 1 a 7, de acordo com a especialidade médica.
- Assim o domínio da matriz Quadro (4×5) será preenchida com um destes códigos.
- Vamos utilizar restrições globais: `member` e `all_different`
- As restrições globais se aplicam sobre um conjunto de variáveis.



- A fase de busca e propagação do comando `solve(Critérios, Variáveis)`, há dezenas de combinações possíveis: consultar o guia do usuário



- A fase de busca e propagação do comando `solve(Critérios, Variáveis)`, há dezenas de combinações possíveis: consultar o guia do usuário
- Tem-se os predicados extras ... são muitos, todos os da CP



- A fase de busca e propagação do comando `solve(Critérios, Variáveis)`, há dezenas de combinações possíveis: consultar o guia do usuário
- Tem-se os predicados extras ... são muitos, todos os da CP
- Finalmente, exemplos sofisticados– de PR com PICAT:
<http://www.hakank.org/picat/> – ***My Picat page*** – por Hakan Kjellerstrand



- Acompanhar as explicações do código de:
`https://github.com/claudiosa/CCS/blob/master/picat/horario_medico_CP.pi`
- Confira a execução e testes



```
modelo =>
    Dias = 5, % segunda= 1, ...., sexta-feira = 5
    Consultorio = 4,
    L_dom = [ oftalmo, otorrino, pediatria,  gineco,
%           1         2         3         4
            cardio, dermato, clin_geral ],
%           5         6         7
    Quadro = new_array(Consultorio, Dias ), %% Lin x Col
    Quadro :: 1 .. L_dom.len , %% operador len . "eh colado"
    ...
```



```
% 0 medico 2 NUNCA trabalha no consultorio 1  
foreach ( J in 1 .. Dias )  
    Quadro[1,J] #!= 2  
end,
```

```
% 0 medico 5 NUNCA trabalha no consultorio 4  
foreach ( J in 1 .. Dias )  
    Quadro[4,J] #!= 5  
end,
```

...




```
%% O Clin Geral deve vir o maior numero de dias ...
%% Esta restricao eh matematicamente é HARD
foreach ( I in 1 .. Consultorio )
    member(7,[Quadro[I,J] : J in 1..Dias])
end,

%% Ninguém trabalha no mesmo consultorio em dias seguidos
foreach ( J in 1 .. Dias )
    all_different( [Quadro[I,J] : I in 1..Consultorio] )
end,

%% Ninguém trabalha no mesmo dia em mais de um consultorio
foreach ( I in 1 .. Consultorio )
    all_different( [Quadro[I,J] : J in 1..Dias] )
end,
```



```
% A BUSCA
solve([ff], Quadro),
    % UMA SAIDA

    printf("\n Uma escolha:"),
    print_matrix( Quadro ),
    print_matrix_NAMES( Quadro , L_dom ),
    printf(".....\n") .
```



```

print_matrix_NAMES( M, Lista ) =>
  L = M.length,
  C = M[1].length,
  nl,
  foreach(I in 1 .. L)
    foreach(J in 1 .. C)
      printf(":%w \t" , print_n_lista( M[I,J], Lista) )
    % printf("(%d,%d): %w " , I, J, M[I,J] ) -- FINE
  end,
  nl
end.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
print_n_lista( _, [] ) = [].
print_n_lista( 1, [A|_] ) = A.
print_n_lista( N, [_|B] ) = print_n_lista( (N-1), B ) .
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```



```
Picat> cl('horario_medico_CP.pi').  
Compiling:: horario_medico_CP.pi  
horario_medico_CP.pi compiled in 10 milliseconds  
loading...
```

yes

```
Picat> main
```

Uma escolha:

```
7 1 3 4 5  
4 7 2 3 1  
1 3 7 5 2  
3 2 1 7 4
```



```
:clin_geral :oftalmo :pediatria :gineco :cardio  
:gineco :clin_geral :otorrino :pediatria :oftalmo  
:oftalmo :pediatria :clin_geral :cardio :otorrino  
:pediatria :otorrino :oftalmo :clin_geral :gineco  
.....  
yes
```



```
$ time(picat horario_medico_CP.pi )
```

```
Uma escolha:
```

```
7 1 3 4 5
```

```
4 7 2 3 1
```

```
1 3 7 5 2
```

```
3 2 1 7 4
```

```
:clin_geral :oftalmo :pediatria :gineco :cardio
```

```
:gineco :clin_geral :otorrino :pediatria :oftalmo
```

```
:oftalmo :pediatria :clin_geral :cardio :otorrino
```

```
:pediatria :otorrino :oftalmo :clin_geral :gineco
```

```
.....
```

```
real 0m0,023s
```

```
user 0m0,007s
```

```
sys 0m0,013s
```

```
[ccs@gerzat picat]$
```



- Este é um exemplo clássico \Rightarrow um NP-Completo \Rightarrow boas soluções apenas com Colônia de Formigas (técnica da Computação Evolucionária)



- Este é um exemplo clássico \Rightarrow um NP-Completo \Rightarrow boas soluções apenas com Colônia de Formigas (técnica da Computação Evolucionária)
- Neste exemplo o Problema do Caixeiro-Viajante (do inglês: TSP – *Travelling Salesman Problem*) é discutido com **dois modelos da PR** destacando dois pontos desta técnica:
 - Uso de outras restrições globais: `element` e `circuit`
 - Uso de variáveis de decisão binária. No exemplo: uma matriz binária.



- Este é um exemplo clássico \Rightarrow um NP-Completo \Rightarrow boas soluções apenas com Colônia de Formigas (técnica da Computação Evolucionária)
- Neste exemplo o Problema do Caixeiro-Viajante (do inglês: TSP – *Travelling Salesman Problem*) é discutido com **dois modelos da PR** destacando dois pontos desta técnica:
 - Uso de outras restrições globais: `element` e `circuit`
 - Uso de variáveis de decisão binária. No exemplo: uma matriz binária.
- Tecnicamente um problema com muitas aplicações similares!



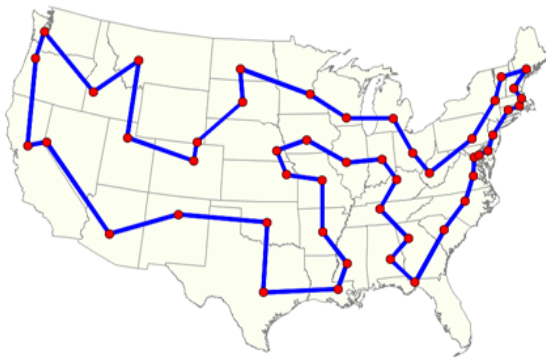


Figura 17: Passar por algumas cidades uma única vez e retornar a cidade de origem



- Usaremos a modelagem matemática de uma matriz binária de decisão



- Usaremos a modelagem matemática de uma matriz binária de decisão
- Esta abordagem é bem conhecida, porém, é eficiente para alguns tipos de problemas semelhantes



- Usaremos a modelagem matemática de uma matriz binária de decisão
- Esta abordagem é bem conhecida, porém, é eficiente para alguns tipos de problemas semelhantes
- Idéia da matriz binária de decisão: N cidades, logo, há possíveis conexões entre elas.



- Usaremos a modelagem matemática de uma matriz binária de decisão
- Esta abordagem é bem conhecida, porém, é eficiente para alguns tipos de problemas semelhantes
- Idéia da matriz binária de decisão: N cidades, logo, há possíveis conexões entre elas.
- Comentários desta modelagem ao longo do código e detalhes do formalismo matemático em qualquer livro da área de combinatória, PO, e modelagem matemática



- Usaremos a modelagem matemática de uma matriz binária de decisão
- Esta abordagem é bem conhecida, porém, é eficiente para alguns tipos de problemas semelhantes
- Idéia da matriz binária de decisão: N cidades, logo, há possíveis conexões entre elas.
- Comentários desta modelagem ao longo do código e detalhes do formalismo matemático em qualquer livro da área de combinatória, PO, e modelagem matemática
- Contudo, duas novas restrições globais são usadas e precisam ser entendidas: `element` e `circuit`



```
element_ex(Vars) =>  
    X :: 1..4, %% NUM de indices da lista  
    element(X , [22, 33, 44, 55], Index),  
    Vars = [X , Index],  
    solve(Vars).
```




```
exe04 =>  
  Todas_Sol = findall(Uma_Sol , $element_ex(Uma_Sol)),  
  foreach( X in Todas_Sol )  
    printf("\n Sol %w", X)  
  end,  
  printf("\n Total de SOL: %d", Todas_Sol.len).
```



```
exe04 =>
  Todas_Sol = findall(Uma_Sol , $element_ex(Uma_Sol)),
  foreach( X in Todas_Sol )
    printf("\n Sol %w", X)
  end,
  printf("\n Total de SOL: %d", Todas_Sol.len).
```

Saída:

```
Sol [1,22]
Sol [2,33]
Sol [3,44]
Sol [4,55]
Total de SOL: 4
```



```
circ_ex(L) =>  
    L = [X1,X2,X3,X4],  
    L :: 1..4, %% NUM de indices da lista  
    circuit(L),  
    solve(L).
```



```
exe02 =>  
  Todas_Sol = findall( Uma_Sol , $circ_ex(Uma_Sol)),  
  foreach( X in Todas_Sol )  
    printf("\n Sol %w", X)  
  end,  
  printf("\n Total de SOL: %d", Todas_Sol.len).
```



```
exe02 =>  
    Todas_Sol = findall( Uma_Sol , $circ_ex(Uma_Sol)),  
    foreach( X in Todas_Sol )  
        printf("\n Sol %w", X)  
    end,  
    printf("\n Total de SOL: %d", Todas_Sol.len).
```

Saída:

```
Sol [2,3,4,1]  
Sol [2,4,1,3]  
Sol [3,1,4,2]  
Sol [3,4,2,1]  
Sol [4,1,2,3]  
Sol [4,3,1,2]  
Total de SOL: 6
```



- Acompanhar as explicações do 1º modelo:
https://github.com/claudiosa/CCS/blob/master/picat/tsp_ESTUDO_hakan.pi
- Acompanhar as explicações do 2º modelo:
https://github.com/claudiosa/CCS/blob/master/picat/tsp_CP.pi
- Confira a execução e testes



1º. Modelo para o TSP – Nilsson e Hakan

```
% Original formulation from Nilsson cited above.
% Codificado por HAKAN e CCS
tsp_test(nilsson, Cidades, Custo) =>
    Cidades    = [X1,X2,X3,X4,X5,X6,X7],
    %% a matriz adjacencia - do mapa - 7 cidades
    element(X1,[ 0, 4, 8,10, 7,14,15],C1),
    element(X2,[ 4, 0, 7, 7,10,12, 5],C2),
    element(X3,[ 8, 7, 0, 4, 6, 8,10],C3),
    element(X4,[10, 7, 4, 0, 2, 5, 8],C4),
    element(X5,[ 7,10, 6, 2, 0, 6, 7],C5),
    element(X6,[14,12, 8, 5, 6, 0, 5],C6),
    element(X7,[15, 5,10, 8, 7, 5, 0],C7),
    Custo #= C1+C2+C3+C4+C5+C6+C7 ,
    circuit( Cidades ) ,
    solve([$min(Custo)], Cidades).
```



```
$ picat tsp_ESTUDO_hakan.pi
Cidades: [2,7,1,3,4,5,6] Custo: 34
  A viagem:
Da cidade 1 --> 2 custa: 4  Acumulado: 4
Da cidade 2 --> 7 custa: 5  Acumulado: 9
Da cidade 7 --> 6 custa: 5  Acumulado: 14
Da cidade 6 --> 5 custa: 6  Acumulado: 20
Da cidade 5 --> 4 custa: 2  Acumulado: 22
Da cidade 4 --> 3 custa: 4  Acumulado: 26
Da cidade 3 --> 1 custa: 8  Acumulado: 34
```




```
$ picat tsp_ESTUDO_hakan.pi
Cidades: [2,7,1,3,4,5,6] Custo: 34
  A viagem:
Da cidade 1 --> 2 custa: 4  Acumulado: 4
Da cidade 2 --> 7 custa: 5  Acumulado: 9
Da cidade 7 --> 6 custa: 5  Acumulado: 14
Da cidade 6 --> 5 custa: 6  Acumulado: 20
Da cidade 5 --> 4 custa: 2  Acumulado: 22
Da cidade 4 --> 3 custa: 4  Acumulado: 26
Da cidade 3 --> 1 custa: 8  Acumulado: 34
```

- A importância deste modelo: facilmente se entende o TSP



```
$ picat tsp_ESTUDO_hakan.pi
Cidades: [2,7,1,3,4,5,6] Custo: 34
A viagem:
Da cidade 1 --> 2 custa: 4   Acumulado: 4
Da cidade 2 --> 7 custa: 5   Acumulado: 9
Da cidade 7 --> 6 custa: 5   Acumulado: 14
Da cidade 6 --> 5 custa: 6   Acumulado: 20
Da cidade 5 --> 4 custa: 2   Acumulado: 22
Da cidade 4 --> 3 custa: 4   Acumulado: 26
Da cidade 3 --> 1 custa: 8   Acumulado: 34
```

- A importância deste modelo: facilmente se entende o TSP
- Hakan fez uma versão genérica para este modelo, de bom desempenho!



```
$ picat tsp_ESTUDO_hakan.pi
Cidades: [2,7,1,3,4,5,6] Custo: 34
A viagem:
Da cidade 1 --> 2 custa: 4 Acumulado: 4
Da cidade 2 --> 7 custa: 5 Acumulado: 9
Da cidade 7 --> 6 custa: 5 Acumulado: 14
Da cidade 6 --> 5 custa: 6 Acumulado: 20
Da cidade 5 --> 4 custa: 2 Acumulado: 22
Da cidade 4 --> 3 custa: 4 Acumulado: 26
Da cidade 3 --> 1 custa: 8 Acumulado: 34
```

- A importância deste modelo: facilmente se entende o TSP
- Hakan fez uma versão genérica para este modelo, de bom desempenho!
- O 2º modelo tem importância como técnica para PR!



2º Modelo para o TSP – Usando Matriz de Decisão

```
import cp,util.  
matriz_adj(Matrix) =>  
    Matrix =  
        [[ 0, 4, 8,10, 7,14,15],  
         [ 4, 0, 7, 7,10,12, 5],  
         [ 8, 7, 0, 4, 6, 8,10],  
         [10, 7, 4, 0, 2, 5, 8],  
         [ 7,10, 6, 2, 0, 6, 7],  
         [14,12, 8, 5, 6, 0, 5],  
         [15, 5,10, 8, 7, 5, 0]].
```

- Os dados são os mesmos do exemplo anterior
- Poderia ser feita leitura via arquivos: ver exemplos de entrada e saída no github
- Comentários no código e áudio



```
tsp_D(Matriz, Cidades, M_Decisao, Custo) =>  
  Len = Matriz.length, %% N x N cidades  
  Cidades = new_list(Len), %%% 1a. dimensao  
  Cidades :: 1..Len,  
  % grafo de DECISAO que representa o resultado dos nos escolhidos  
  M_Decisao = new_array (Len, Len),  
  M_Decisao :: 0..1 ,
```



```
tsp_D(Matriz, Cidades, M_Decisao, Custo) =>  
  Len = Matriz.length, %% N x N cidades  
  Cidades = new_list(Len), %%% 1a. dimensao  
  Cidades :: 1..Len,  
  % grafo de DECISAO que representa o resultado dos nos escolhidos  
  M_Decisao = new_array (Len, Len),  
  M_Decisao :: 0..1 ,  
  
% calculate upper and lower bounds of the Costs list -- HAKAN  
% repensar MELHORAR .....  
SOMA_Dists = sum([Matriz[I,J] : I in 1..Len,  
                  J in 1..Len, Matriz[I,J] > 0]),  
MinDist = 0,  
MaxDist = SOMA_Dists,  
Custo :: 0..MaxDist,
```



```
% Se NAO HOUVER CONEXAO ou ARCO = 0 entao não há conexão  
foreach(I in 1..Len , J in 1..Len)  
    (Matriz[I,J] != 0) ==> (M_Decisao[I,J] != 0)  
end,
```



```
% Se NAO HOUVER CONEXAO ou ARCO = 0 entao não há conexão
foreach(I in 1..Len , J in 1..Len)
    (Matriz[I,J] != 0) ==> (M_Decisao[I,J] != 0)
end,

% Para todas linhas, a soma das colunas é igual a 1
% UMA: uma saída como caminho a ser traçado e somente UMA
foreach(I in 1..Len)
    sum([M_Decisao[I,J] : J in 1..Len, I != J]) != 1
end,

% Para todas colunas, a soma das linhas é igual a 1
% UMA: uma chegada ao nó de destino e somente UMA chegada
foreach(J in 1..Len)
    sum([M_Decisao[I,J] : I in 1..Len, I != J]) != 1
end,
```




```
%% Relacionar as escolhas da M_Decisao com a
%% ... sequencia das Cidades.
foreach(I in 1..Len , J in 1..Len)
    ( M_Decisao[I,J] #= 1 ) #<=> ( Cidades[I] #= J )
end,

%% garante o circuito entre os nós selecionados
circuit(Cidades),
```



```
%% Relacionar as escolhas da M_Decisao com a
%% ... sequencia das Cidades.
foreach(I in 1..Len , J in 1..Len)
    ( M_Decisao[I,J] #= 1 ) #<=> ( Cidades[I] #= J )
end,

%% garante o circuito entre os nós seleccionados
circuit(Cidades),

%% Função custo a ser minimizada
Custo #= sum([M_Decisao[I,J] * Matriz[I,J] :
              I in 1..Len , J in 1..Len]),

%% Vars para BUSCA
Vars = [Cidades, M_Decisao], %% OU Cidades ++ M_Decisao
solve([$min(Custo)], Vars ).
```



```
$ picat tsp_CP.pi
M_Decisao: {{0,1,0,0,0,0,0},{0,0,0,0,0,0,1},{1,0,0,0,0,0,0},{0,0,1,0,0,0,0},
DESTINOS:
```

```
      1 2 3 4 5 6 7
1 -> 0 1 0 0 0 0 0
2 -> 0 0 0 0 0 0 1
3 -> 1 0 0 0 0 0 0
4 -> 0 0 1 0 0 0 0
5 -> 0 0 0 1 0 0 0
6 -> 0 0 0 0 1 0 0
7 -> 0 0 0 0 0 1 0
```

```
Sequência das Cidades: [2,7,1,3,4,5,6]
```

```
Custo: 34
```

```
A viagem:
```

```
Da cidade 1 --> 2 custa: 4   Acumulado: 4
.....
Da cidade 3 --> 1 custa: 8   Acumulado: 34
```



As funções/regras de saída não foram apresentadas!

- Há outros métodos para se resolver estes problemas.
Exemplo: Programação Linear, Buscas Heurísticas, etc



- Há outros métodos para se resolver estes problemas.
Exemplo: Programação Linear, Buscas Heurísticas, etc
- As restrições globais se aplicam sobre um conjunto de variáveis e há muitas outras importantes disponíveis no Picat



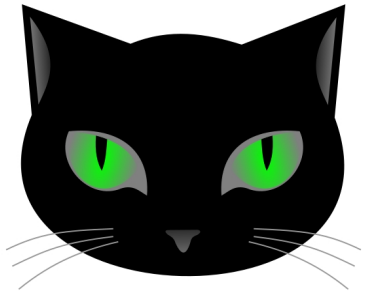
- Há outros métodos para se resolver estes problemas.
Exemplo: Programação Linear, Buscas Heurísticas, etc
- As restrições globais se aplicam sobre um conjunto de variáveis e há muitas outras importantes disponíveis no Picat
- A área é extensa, contudo, Picat adere há todos requisitos da PR



- Há outros métodos para se resolver estes problemas.
Exemplo: Programação Linear, Buscas Heurísticas, etc
- As restrições globais se aplicam sobre um conjunto de variáveis e há muitas outras importantes disponíveis no Picat
- A área é extensa, contudo, Picat adere há todos requisitos da PR
- Resumo da PR: segue por uma notação/manipulação algébrica restrita, simplificar e bissecionar as restrições, instanciar variáveis, verificar inconsistências, avançar sobre as demais variáveis, até que todas estejam instanciadas.



- 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 concomitadamente. 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`

