



PICAT: Uma Linguagem de Programação Multiparadigma

, Claudio Cesar de Sá, Miguel Alfredo Nunes, Jeferson L. R.
Souza

`miguel.nunes@edu.udesc.br`

`jeferson.souza@udesc.br`

`claudio.sa@udesc.br`

Departamento de Ciência da Computação – DCC
Centro de Ciências e Tecnologias – CCT



Contribuições

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



Apresentação ao Curso de PICAT – I

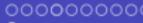
- ### ■ O que é o PICAT?

Apresentação ao Curso de PICAT – I

- O que é o PICAT?

- Uma linguagem de programação de propósitos gerais
- Uma evolução do PROLOG (consagrada linguagem dos primórdios da IA)
- Tem elementos das linguagens Python, Prolog e Haskell

- Uso e finalidades do PICAT:



Apresentação ao Curso de PICAT – I

■ O que é o PICAT?

- Uma linguagem de programação de propósitos gerais
- Uma evolução do PROLOG (consagrada linguagem dos primórdios da IA)
- Tem elementos das linguagens Python, Prolog e Haskell

■ Uso e finalidades do PICAT:

- Uso de programas gerais: de simples à complexos (uma reflexão)
- Provê suporte há vários *solvers* na área de Pesquisa Operacional
- Área: IA, programação por restrições, programação inteira, planejamento, combinatória, etc





Apresentação ao Curso de PICAT – II

- Este curso é dirigido a você?
- Requisitos:

Apresentação ao Curso de PICAT – II

- Este curso é dirigido a você?
 - Requisitos:
 - Conhecimento: noções de lógica matemática (proposicional e primeira-ordem), matemática elementar, e alguma outra linguagem de programação
 - Dedição: depende de você



Apresentação ao Curso de PICAT – II

- Este curso é dirigido a você?
- Requisitos:
 - Conhecimento: noções de lógica matemática (proposicional e primeira-ordem), matemática elementar, e alguma outra linguagem de programação
 - Dedicação: depende de você
- Motivação:



Apresentação ao Curso de PICAT – II

- Este curso é dirigido a você?
- Requisitos:
 - Conhecimento: noções de lógica matemática (proposicional e primeira-ordem), matemática elementar, e alguma outra linguagem de programação
 - Dedicação: depende de você
- Motivação:
 - Dependendo de sua dedicação, ao final você vai estar apto a resolver problemas computacionais de simples à difíceis
 - Difícil: muitas linhas de código e muito conhecimento de algoritmos seriam necessários
 - Com Picat, há sofisticados esquemas prontos para se construir programas.



Apresentação ao Curso de PICAT – III

- #### ■ Requisitos computacionais:

Apresentação ao Curso de PICAT – III

- Requisitos computacionais:
um computador qualquer (arquitetura 16, 32 ou 64 bits), com Linux, Mac ou Windows, que tenha um compilador C instalado completo, preferencialmente.
- Comunidade e ações: <http://picat-lang.org>



Apresentação ao Curso de PICAT – III

- Requisitos computacionais:
um computador qualquer (arquitetura 16, 32 ou 64 bits), com Linux, Mac ou Windows, que tenha um compilador C instalado completo, preferencialmente.
- Comunidade e ações: <http://picat-lang.org>
- Códigos e este material, sempre atualizados em:



Apresentação ao Curso de PICAT – III

- Requisitos computacionais:
um computador qualquer (arquitetura 16, 32 ou 64 bits), com Linux, Mac ou Windows, que tenha um compilador C instalado completo, preferencialmente.
- Comunidade e ações: <http://picat-lang.org>
- Códigos e este material, sempre atualizados em:
 - Este PDF e seu texto original:
http://github.com/claudiosa/Slides_Picat (em código L^AT_EX)
 - Os códigos fontes dos programas:
<http://github.com/claudiosa/CCS/picat>
- Além do material aqui disponível em PDF, o mais importante do curso vai estar na interatividade da minha **apresentação oral**



Apresentação ao Curso de PICAT – III

- Requisitos computacionais:
um computador qualquer (arquitetura 16, 32 ou 64 bits), com Linux, Mac ou Windows, que tenha um compilador C instalado completo, preferencialmente.
- Comunidade e ações: <http://picat-lang.org>
- Códigos e este material, sempre atualizados em:
 - Este PDF e seu texto original:
http://github.com/claudiosa/Slides_Picat (em código L^AT_EX)
 - Os códigos fontes dos programas:
<http://github.com/claudiosa/CCS/picat>
- Além do material aqui disponível em PDF, o mais importante do curso vai estar na interatividade da minha **apresentação oral**





Apresentação ao Curso de PICAT – IV

- Há alguns pontos do curso que estão repetidos:
propositadamente!
Reforça os erros que cometí um dia!

Apresentação ao Curso de PICAT – IV

- Há alguns pontos do curso que estão repetidos:
propositadamente!
Reforça os erros que cometi um dia!
 - As aulas aqui apresentadas **não** serão regravadas!



Apresentação ao Curso de PICAT – IV

- Há alguns pontos do curso que estão repetidos:
propositadamente!
Reforça os erros que cometí um dia!
- As aulas aqui apresentadas **não** serão regravadas!
- Contudo, o texto completo, incluindo os fontes dos programas:
SIM
Pois sempre há perguntas, melhoramentos, etc, que elucidam os pontos aqui abordados.



Apresentação ao Curso de PICAT – IV

- Há alguns pontos do curso que estão repetidos:
propositadamente!
Reforça os erros que cometí um dia!
- As aulas aqui apresentadas **não** serão regravadas!
- Contudo, o texto completo, incluindo os fontes dos programas:
SIM
Pois sempre há perguntas, melhoramentos, etc, que elucidam os pontos aqui abordados.
- Na parte teórica da definição do Picat, mantive os padrões descritos no manual da linguagem
(<http://picat-lang.org>).

Apresentação ao Curso de PICAT – V

- Além desta apresentação do curso, você pode assistir uma parte deste curso em aulas que fiz para o Youtube, há alguns anos atrás:

Apresentação ao Curso de PICAT – V

- Além desta apresentação do curso, você pode assistir uma parte deste curso em aulas que fiz para o Youtube, há alguns anos atrás:
 - Videoaula 01: Introdução ao PICAT
<https://www.youtube.com/watch?v=0DmTyFFQPK8>



Apresentação ao Curso de PICAT – V

- Além desta apresentação do curso, você pode assistir uma parte deste curso em aulas que fiz para o Youtube, há alguns anos atrás:
- Videoaula 01: Introdução ao PICAT
<https://www.youtube.com/watch?v=0DmTyFFQPK8>
- Videoaula 02: Tipos de Dados do PICAT
<https://www.youtube.com/watch?v=7fPKPd0ZDnc>
- Estas videoaulas forem refeitas e encontram-se com uma outra abordagem neste curso.

Apresentação ao Curso de PICAT – VI

- Assim, ao final deste curso terás uma sólida visão de uma ferramenta computacional, utilizada em várias áreas tais como: modelagem matemática, IA, Pesquisa Operacional, etc



Apresentação ao Curso de PICAT – VI

- Assim, ao final deste curso terás uma sólida visão de uma ferramenta computacional, utilizada em várias áreas tais como: modelagem matemática, IA, Pesquisa Operacional, etc
- Ao final você vai conseguir resolver problemas com alguma complexidade e ler códigos de grandes programadores da área: Barták, Neng-Fa, Hakank, Dymichenko, etc



Apresentação ao Curso de PICAT – VI

- Assim, ao final deste curso terás uma sólida visão de uma ferramenta computacional, utilizada em várias áreas tais como: modelagem matemática, IA, Pesquisa Operacional, etc
- Ao final voce vai conseguir resolver problemas com alguma complexidade e ler códigos de grandes programadores da área: Barták, Neng-Fa, Hakank, Dymichenko, etc
- Tópicos cobertos no curso: ver índice



Sumário I

1 Apresentação ao Curso de PICAT

2 Introdução

- Estrutura da Linguagem
 - Paradigmas
 - Usando Picat

3 Tipos de Dados e Variáveis

- Tipos de Dados
- Variáveis
- Unificação e Atribuição
- Tabela de Operadores
 - Operadores Especiais

4 Predicados e Funções

- Casamento de Padrões



Sumário II

- Funções
- Relembrando as Regras
- Regras do Tipo Fatos
- Exemplos

5 Comandos Condicionais, Laços e Repetições

- Funções e Predicados Especiais

6 Recursão

- Recursão
- *Backtracking*

7 Listas

8 Buscas

9 Programação Dinâmica

oooooo●

○

oooooooooooo

○

oooooooooooo

○

oooooooooooo

○

○○

○○○

○○○○

○

○○

○○○○

oooooooooooo

oooooooooooo

Sumário III

10 Planejamento

11 Programação por Restrições

12 Conclusão





Histórico

- 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!



Histórico

- 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 (22 de abril de 2019).



Histórico

- 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 (22 de abril de 2019).
- Código-aberto, segue as regras da FSF



Estrutura da Linguagem

Conhecendo PICAT

- 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



Estrutura da Linguagem

Conhecendo PICAT

- 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 essencialmente multiparadigma, abrangendo partes de vários paradigmas de programação: declarativo (lógico e funcional) e imperativo



Estrutura da Linguagem

O que é ser Multiparadigma ?

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



Estrutura da Linguagem

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



Estrutura da Linguagem

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).



Paradigma Lógico

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



Estrutura da Linguagem

Paradigma Lógico

- 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.



Paradigma Lógico

- 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.



Paradigma Lógico

- 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



Paradigma Funcional

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



Estrutura da Linguagem

Paradigma Funcional

- 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.



Paradigma Funcional

- 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.



Estrutura da Linguagem

Paradigma Procedural

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



Paradigma Procedural

- 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.



Estrutura da Linguagem

Paradigma Procedural

- 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.

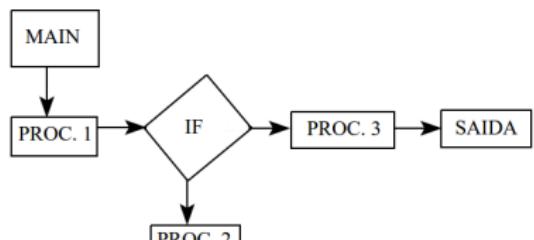


Figura 1: Fluxograma representando a estrutura de



Estrutura da Linguagem

Algumas Características:



Estrutura da Linguagem

Algumas Características:

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



Estrutura da Linguagem

Algumas Características:

- 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)



Algumas Características:

- 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



Algumas Características:

- 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.



Estrutura da Linguagem

Algumas Características:

- 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.



Estrutura da Linguagem

Acrônimo de P.I.C.A.T. I

- 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 a outras linguagens de programação mais populares
- C:** *Constraints*: Suporta a programação por restrições (PR) para problemas combinatórios
- A:** *Actors*: Suporte as chamadas a eventos, os atores;



Estrutura da Linguagem

Acrônimo de P.I.C.A.T. II

T: *Tabling*: Implementa a técnica de *memoization*, com soluções imediatas para problemas de Programação Dinâmica (PD).



Estrutura da Linguagem

Instalação do PICAT

- Baixar a versão desejada de:

<http://picat-lang.org/download.html>



Estrutura da Linguagem

Instalação do PICAT

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



Estrutura da Linguagem

Instalação do PICAT

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



Estrutura da Linguagem

Instalação do PICAT

- Baixar a versão desejada de:

<http://picat-lang.org/download.html>

- Descompactar. Em geral em: **/usr/local/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
```



Estrutura da Linguagem

Instalação do PICAT

- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/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`



Estrutura da Linguagem

Instalação do PICAT

- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/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 Atom*.



Estrutura da Linguagem

Instalação do PICAT

- Baixar a versão desejada de:
`http://picat-lang.org/download.html`
- Descompactar. Em geral em: `/usr/local/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 Atom*.



Estrutura da Linguagem

Usando Picat

- Picat é uma linguagem disponível em qualquer arquitetura de processamento.



Estrutura da Linguagem

Usando Picat

- Picat é uma linguagem disponível em qualquer arquitetura de processamento.
- Qualquer emergência, o ambiente completo de execução do Picat pode ser reconstruído a partir da linguagem C padrão



Estrutura da Linguagem

Usando Picat

- Picat é uma linguagem disponível em qualquer arquitetura de processamento.
- Qualquer emergência, o ambiente completo de execução do Picat pode ser reconstruído a partir da linguagem C padrão
- 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.

Estrutura da Linguagem

Usando Picat

- Picat é uma linguagem disponível em qualquer arquitetura de processamento.
- Qualquer emergência, o ambiente completo de execução do Picat pode ser reconstruído a partir da linguagem C padrão
- 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



Estrutura da Linguagem

Exemplo – *Alô Mundo!*

Acompanhar as explicações do código de:

[https://github.com/claudiosa/CCS/blob/master/picat/
alo_mundo.pi](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").
```



Estrutura da Linguagem

Execução na Console Linux ou Windows

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



Estrutura da Linguagem

Execução na Console Linux ou Windows

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

Análogo ao desenvolvimento com Python!



Estrutura da Linguagem

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
```



Estrutura da Linguagem

Reflexões

- 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>



Estrutura da Linguagem

Reflexões

- 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.



Tipos de Dados e Variáveis – Introdução – I

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



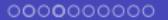
Tipos de Dados e Variáveis – Introdução – I

- 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*.



Tipos de Dados e Variáveis – Introdução – I

- 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*.



Tipos de Dados e Variáveis – Introdução – I

- 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*.
- Linguagens fortemente tipadas, tais como C, Java 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*.



Tipos de Dados e Variáveis – Introdução – II

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a execução do programa



Tipos de Dados e Variáveis – Introdução – II

- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a execução do programa
- Prós e contras para o que é melhor, a discussão fica de lado neste momento



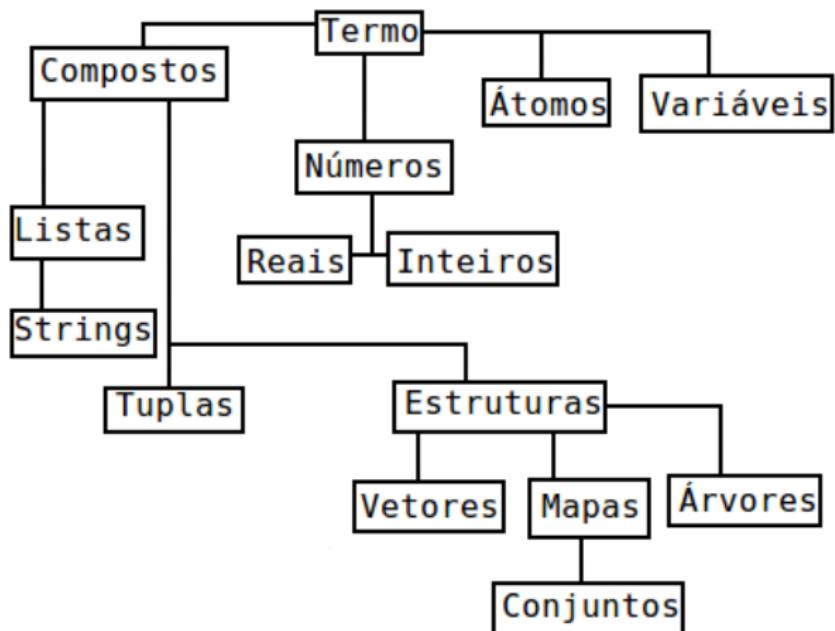
Tipos de Dados e Variáveis – Introdução – II

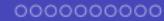
- Nas linguagens interpretadas, com uma máquina virtual, esta definição é feita durante a *execução* do programa
- Prós e contras para o que é melhor, a discussão fica de lado neste momento
- Picat até o momento tem a tipagem **dinâmica**



Tipos de Dados

Tipos de Dados





Tipos de Dados

Termos

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



Tipos de Dados

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



Tipos de Dados

Átomos

- 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*.



Tipos de Dados

Átomos

- 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'    '$%',
```



Tipos de Dados

Números I

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



Tipos de Dados

Números II

O underscore é ignorado pelo compilador e o interpretador.



Números III

- **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



Tipos de Dados

Compostos

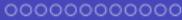
- Termos compostos podem conter mais de um valor ao mesmo tempo.



Tipos de Dados

Compostos

- 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.



Tipos de Dados

Compostos

- 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.



Tipos de Dados

Listas

Listas são agrupamentos de valores quaisquer sem ordem e sem tamanho pré-definido. Seu tamanho não é armazenado na memória, sendo necessário recalcular 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!



Tipos de Dados

Strings – Lista de Carácteres

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

Exemplos

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



Tipos de Dados

Tuplas

- 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.



Tipos de Dados

Estruturas (*Functores*)

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")
```



Tipos de Dados

Estruturas (*Functores*)

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

[fragile, allowframebreaks=0.9]

Vetores ou *arrays* são estruturas especiais do tipo:

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



Tipos de Dados

Vetores

[fragile, allowframebreaks=0.9]

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 sobrearcadas 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



Tipos de Dados

Mapas, Conjuntos e *Heaps*

- **Mapas** são estruturas especiais que são conjuntos de relações do tipo chave–valor.
- **Conjuntos** são sub-tipos de mapas onde todos 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 esta na raiz.



Variáveis I

- 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



Variáveis II

- 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*



Variáveis III

- 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



Unificação e Atribuição

Unificação e Atribuição

Há dois modos de definir valores às variáveis:



Unificação e Atribuição

Unificação e Atribuição

Há dois modos de definir valores às variáveis:

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



Unificação

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



Unificação e Atribuição

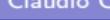
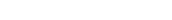
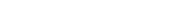
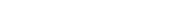
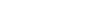
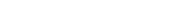
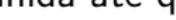
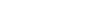
Unificação

- 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.



Unificação

- 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 unificado a uma variável.





Unificação e Atribuição

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)
```





Atribuição

- A atribuição simula a atribuição em linguagens imperativas



Atribuição

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



Atribuição

- 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



Atribuição

- 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.



Unificação e Atribuição

Exemplo

```
test => 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





Unificação e Atribuição

Exemplo

Portanto, estas atribuições sucessivas são compiladas como:



Unificação e Atribuição

Exemplo

Portanto, estas atribuições sucessivas são compiladas como:

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



Unificação e Atribuição

Exemplos de Variáveis Válidas

X1		_ab
X	Ā	Variavel
_invalido	_correto	_aa

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



Unificação e Atribuição

Exemplos de Variáveis Inválidas

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



Tabela de Operadores

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
$X \bmod Y$	Resto da Divisão
$X + Y$	Adição
$X - Y$	Subtração
<i>Inicio ..</i>	Uma série (lista) de números com
<i>Passo ..</i>	um passo
<i>Fim</i>	
<i>Inicio ..</i>	Uma série (lista) de números com
<i>Fim</i>	passo 1



Tabela de Operadores

Tabela 2: Tabela de Operadores Completa em Ordem de Precedência

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



Tabela de Operadores

Operadores Especiais – I

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*.



Tabela de Operadores

Operadores Especiais – I

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.



Tabela de Operadores

Operadores Especiais – I

Operadores de Termos Não-Compostos

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



Tabela de Operadores

Operadores Especiais – I

Operadores de Termos Não-Compostos

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



Tabela de Operadores

Operadores Especiais – II

Exemplos

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



Tabela de Operadores

Operadores Especiais – II

Exemplos

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



Tabela de Operadores

Operadores Especiais – II

Exemplos

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



Tabela de Operadores

Operadores Especiais – II

Exemplos

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



Tabela de Operadores

Operadores Especiais – II

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 $1.0 =:= 1$, $1.2 =:= 1$



Tabela de Operadores

Operadores Especiais – II

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



Tabela de Operadores

Operadores Especiais – II

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 $1.0 != 1$, $Var3 != Var4$





Tabela de Operadores

Operadores Especiais – II

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)



Tabela de Operadores

Operadores Especiais – II

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** $1.0 != 1$, $aa != bb$, $Var1 != Var5$





Tabela de Operadores

Operadores Especiais – II

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





Tabela de Operadores

Alguns Operadores de Termos Compostos

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



Tabela de Operadores

Alguns Operadores de Termos Compostos

- **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 | 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



Tabela de Operadores

Alguns Operadores de Termos Compostos

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



Tabela de Operadores

Alguns Operadores de Termos Compostos

- **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



Tabela de Operadores

Operadores de Termos Compostos – Exemplos

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



Tabela de Operadores

Operadores de Termos Compostos – Exemplos

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



Tabela de Operadores

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 $L = [1, 2, 3]$, $[H|T] = L$



Tabela de Operadores

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 $L = [1, 2, 3]$, $[H|T] = L$
 $L = [1, 2, 3]$



Tabela de Operadores

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 $L = [1, 2, 3]$, $[H | T] = L$
 $L = [1, 2, 3]$
 $H = 1$



Tabela de Operadores

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 $L = [1, 2, 3]$, $[H|T] = L$

$L = [1, 2, 3]$

$H = 1$

$T = [2, 3]$

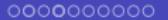


Tabela de Operadores

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 $\text{foreach}(X \text{ in } [1, 2, 3]) \text{ printf}(\text{" \%w "}, X) \text{ end}$

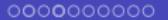


Tabela de Operadores

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 $\text{foreach}(X \text{ in } [1, 2, 3]) \text{ printf}(\text{" \%w "}, X) \text{ end}$
 1 2 3

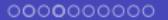


Tabela de Operadores

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 $\text{foreach}(X \text{ in } [1, 2, 3]) \text{ printf}("%w", X) \text{ end}$
 $1 \ 2 \ 3$
- 4 $X = 1..10$, $Y = 0..2..20$, $Z = 10.. - 1..1$



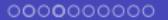


Tabela de Operadores

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 $X = 1..10$, $Y = 0..2..20$, $Z = 10.. - 1..1$
 $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$





Tabela de Operadores

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 $X = 1..10$, $Y = 0..2..20$, $Z = 10.. - 1..1$

$X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

$Y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$





Tabela de Operadores

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 $X = 1..10$, $Y = 0..2..20$, $Z = 10.. - 1..1$

$X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

$Y = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]$





Tabela de Operadores

Reflexões

- 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>



Predicados e Funções em Picat I

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



Predicados e Funções em Picat II

- Regras **sem** *backtracking* (*non-backtrackable*):
Cabeça , Condicional \Rightarrow Corpo .
- Regras **com** *backtracking*:
Cabeça , Condicional $? \Rightarrow$ Corpo .



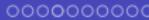
Predicados e Funções em Picat III

- A identificação da sintaxe é dada por:
 - *Cabeça*: indica um padrão de regra a ser casada.
Forma geral:
$$\textit{regra}(\textit{termo}_1, \dots, \textit{termo}_n)$$
Onde:
 - *regra* é um átomo que define o nome da regra.
 - *n* é a aridade da regra (*i.e.* o total de argumentos)
 - Cada *termo_i* é um argumento da regra.
 - *Cond*: é uma ou várias condições sobre a execução desta regra.
 - *Corpo*: define as ações da regra



Predicados e Funções em Picat IV

- Todas as regras são finalizadas por um ponto final (.), seguido por um espaço em branco ou nova linha.
- Ao longo dos exemplos, detalhes a mais sobre esta construção de **predicados e funções**



Regras com e sem Backtracking – Exemplo

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

```
regra_01(0)          ?=> printf("\n  CHEGOU A 0 !!!\n").
regra_01(N) , N < 0  ?=> printf("\n  EH UM NEGATIVO !!!\n").
regra_01(N) , N > 10 ?=> printf("\n  EH MAIOR QUE 10 !!!\n").
regra_01(N) , N <= 10 =>
    printf("\t :%d ", N),
    regra_01(N-1).
```

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

Há uma recursão aqui. Algumas seções a frente.



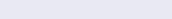
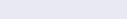
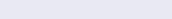
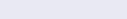
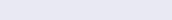
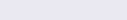
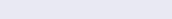
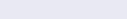
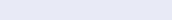
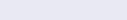
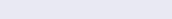
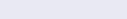
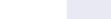
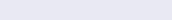
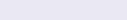
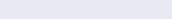
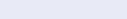
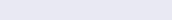
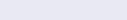
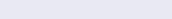
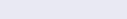
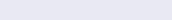
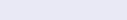
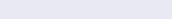
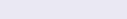
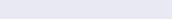
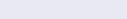
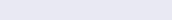
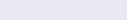
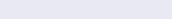
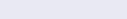
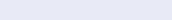
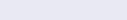
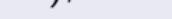
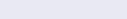
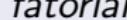
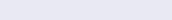
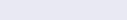
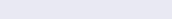
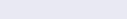
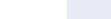
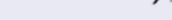
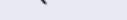
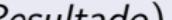
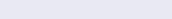
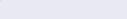
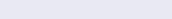
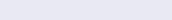
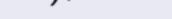
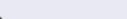
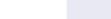
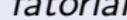
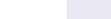
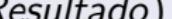
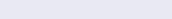
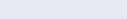
Casamento de Padrões

- O algoritmo de *casamento de padrões* para regras é análogo ao algoritmo de unificação para variáveis.
- O objetivo é encontrar dois padrões que possam ser unificados para se inferir alguma ação.
- Quanto ao *casamento de padrões*:
 - Dado um padrão $p_1(t_1, \dots, t_m)$, este *casa* com um padrão semelhante $p_2(u_1, \dots, u_n)$ se:
 - p_1 e p_2 forem átomos equivalentes;
 - O número de termos (chamado de aridade) em (t_1, \dots, t_m) e (u_1, \dots, u_n) for equivalente.
 - Os termos (t_1, \dots, t_m) e (u_1, \dots, u_n) são equivalentes, ou tornaram-se equivalentes pela unificação de variáveis em qualquer um dos dois termos;
 - Caso essas condições forem satisfeitas, o padrão $p_1(t_1, \dots, t_m)$ *casa* com o padrão $p_2(u_1, \dots, u_n)$.



Exemplos de casamento

- 1 A regra *fatorial(Termo, Resultado)* pode casar com:
fatorial(1, 1), *fatorial(5, 120)*, *fatorial(abc, 25)*,
fatorial(X, Y), etc.



Casamento de Padrões

Exemplos de casamento

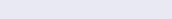
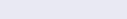
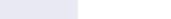
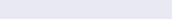
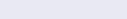
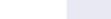
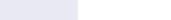
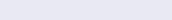
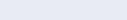
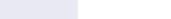
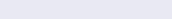
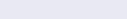
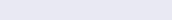
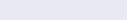
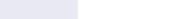
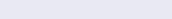
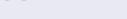
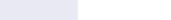
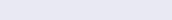
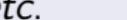
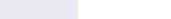
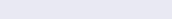
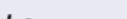
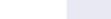
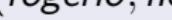
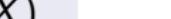
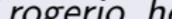
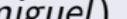
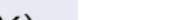
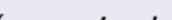
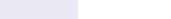
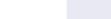
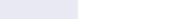
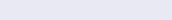
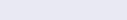
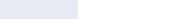
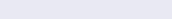
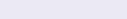
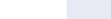
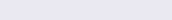
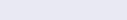
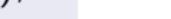
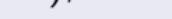
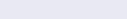
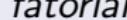
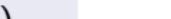
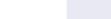
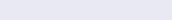
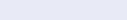
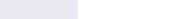
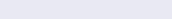
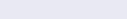
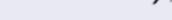
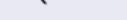
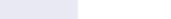
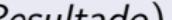
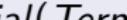
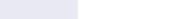
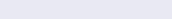
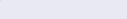
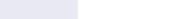
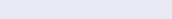
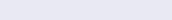
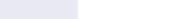
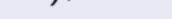
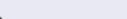
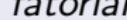
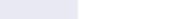
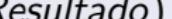
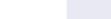
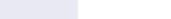
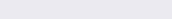
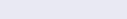
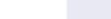
1 A regra *fatorial(Termo, Resultado)* pode casar com:

fatorial(1, 1), *fatorial(5, 120)*, *fatorial(abc, 25)*,

fatorial(X, Y), etc.

2 A regra *fatorial(Termo, Resultado)*, *Termo* ≥ 0 pode casar com:

fatorial(1, 1), *fatorial(5, 120)*, *fatorial(X, Y)*, *fatorial(Z, Z)*, etc.





Exemplos de casamento

- 1 A regra *fatorial(Termo, Resultado)* pode casar com:
 $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(abc, 25)$,
 $fatorial(X, Y)$, etc.
- 2 A regra *fatorial(Termo, Resultado)*, $Termo \geq 0$ pode casar com:
 $fatorial(1, 1)$, $fatorial(5, 120)$, $fatorial(X, Y)$, $fatorial(Z, Z)$,
etc.
- 3 A regra *pai(X, Y)* pode casar com:
 $pai(rogerio, miguel)$, $pai(rogerio, henrique)$, $pai(salomao, X)$,
 $pai(12, 24)$, etc.
- 4 A regra *pai(salomao, X)* pode casar com:
 $pai(salomao, rogerio)$, $pai(salomao, fabio)$.





Exemplos de casamento

- 1 A regra *fatorial(Termo, Resultado)* pode casar com:
fatorial(1, 1), *fatorial(5, 120)*, *fatorial(abc, 25)*,
fatorial(X, Y), etc.
- 2 A regra *fatorial(Termo, Resultado)*, $Termo \geq 0$ pode casar com:
fatorial(1, 1), *fatorial(5, 120)*, *fatorial(X, Y)*, *fatorial(Z, Z)*,
etc.
- 3 A regra *pai(X, Y)* pode casar com:
pai(rogerio, miguel), *pai(rogerio, henrique)*, *pai(salomao, X)*,
pai(12, 24), etc.
- 4 A regra *pai(salomao, X)* pode casar com:
pai(salomao, rogerio), *pai(salomao, fabio)*.





Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* eventualmente é chamado de *prova do programa*



Metas ou Provas – (*goals*)

- Na matemática ao se deduzir um valor de um teorema, tem-se uma *prova*. Assim, o termo *goal* eventualmente é chamado de *prova do programa*
- Metas ou Provas (do inglês: *goal*) são estados que definem o final da execução.



Metas ou Provas – (*goals*)

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



Funções

Funções – I

- A forma geral de uma função é:

Cabeça = X => Corpo.



Funções

Funções – I

- A forma geral de uma função é:

$$\text{Cabeça} = X \Rightarrow \text{Corpo}.$$

- Caso tenhamos uma condição $\text{Cond}::$

$$\text{Cabeça} = X, \text{Cond} \Rightarrow \text{Corpo}.$$

- Funções não admitem **backtracking**.



Funções

Funções – II

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



Funções

Funções – II

- Funções são tipos especiais de regras que sempre sucedem com *uma* resposta.
- Funções em Picat tem como intuito serem sintaticamente semelhantes a funções matemáticas (vide *Haskell*).



Funções

Funções – II

- Funções são tipos especiais de regras que sempre sucedem com *uma* resposta.
- Funções em Picat tem como intuito serem sintaticamente semelhantes a funções matemáticas (vide *Haskell*).
- Em uma função a *Cabeça* é uma equação do tipo $f(t_1, \dots, t_n) = X$, onde f é um átomo que é o nome da função, n é a aridade da função, e cada termo t_i é um argumento da função.
- X é uma expressão que é o retorno da função.



Funções

Funções – III

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



Funções

Regras, Metas e Funções – Exemplo

```
main =>
    X = 3,
    Y = 4,
    um_predicado(X,Y,Z),
    R = uma_funcao(X,Y),
    printf("\n Z: %d \t R: %d", Z, R),
    println("\n FIM").
```

um_predicado(X,Y,Z) => Z = X + Y.

uma_funcao(X,Y) = R => R = X + Y.



Funções

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,Z), write(Z).
```

```
7Z = 7
```

```
yes
```

```
Picat> uma_funcao(3,4) = R, write(R).
```

```
7R = 7
```

```
yes
```



Relembrando as Regras

Relembrando as Regras I

- Forma geral de um predicado do tipo Regra:

Cabeça , Condicional => Corpo .

- Forma geral de um predicado com *backtracking*:

Cabeça , Condicional ?=> Corpo .

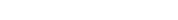
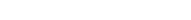
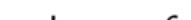
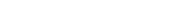
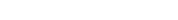
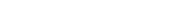
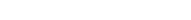
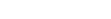
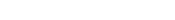
Em Prolog, esta condicional (Cond) entra no corpo da regra. Picat é flexível!



Relembrando as Regras

Relembrando as Regras II

- Dentro de uma regra, *Cond* só pode ser avaliado uma vez, acessando somente termos dentro do escopo do predicado.
- Sempre 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.



















































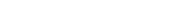




























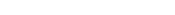


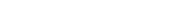






























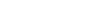




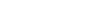




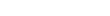










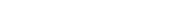








































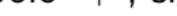




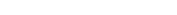




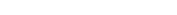


























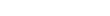




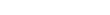




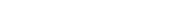


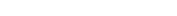









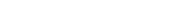












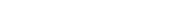














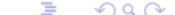































Regras do Tipo Fatos

Regras do Tipo Fatos III

- Ou seja, quando ocorre um simbolo ‘+’ em um grupo do *index*, é avaliado pelo compilador como um valor constante a ser casada
- Quanto ao ‘-’, este é avaliado pelo compilador com uma variável que deverá ser instanciada à um valor. Ou seja, quando se deseja unificar um valor a esta variável



Regras do Tipo Fatos

Regras do Tipo Fatos IV

- Dica: o parâmetro ‘–’ no *index* é quase como regra geral
- **Não** pode haver um **predicado** e um **predicado fato** com **mesmo nome**.



Regras do Tipo Fatos

Exemplo – Função e Regras

```

index (+,+,-) (+,-,-) (-,+,-) (-,-,-)
(+,-,+)
(+,-,-)
%(-,-,-) %% NENHUM argumento instanciao -- UTIL
%(+,-,+)
%(+,-,-)
%(-,+,-)
%(-,-,+)
(-,-,-)
(+,-,+)
(+,-,-)
(+,-,-)
(+,-,-)

and2(true,true,true).
and2(true,false,false).
and2(false,true,false).
and2(false,false,false).

main ?=>
    and2(X,Y,Z), % and eh reservado
    printf("\n X: %w \t Y: %w \t Z: %w", X, Y, Z),
    !.

```



Exemplos

Exemplos

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.
```



Exemplos

Exemplo de Funções

```
1 contas_F0(X1, X2, X3) = Z, (number(X1),  
2                               number(X2),  
3                               number(X3)) =>  
4     if (X1 < X2 && X2 < X3) then  
5         Z = (X2 + X3)  
6     else  
7         Z = X1  
8     end.
```

Aperitivo à próxima seção: condicionais e laços!



Exemplos

Mais Exemplos (Fatos e Regras)

```
1 index(-,-) (+,-) (-,+)
2 pai(salomao, rogerio).
3 pai(salomao, fabio).
4 pai(rogerio, miguel).
5 pai(rogerio, henrique).
6
7 avo(X,Y) ?=> pai(X,Z), pai(Z,Y).
8 irmao(X,Y) ?=> pai(Z,X), pai(Z,Y).
9 tio(X,Y) ?=> pai(Z,Y), irmao(X,Z).
```



Exemplos

Exemplos de Funções – Equivalentes

```
1 eleva_cubo(1) = 1.  
2 eleva_cubo(X) = X**3.  
3 eleva_cubo(X) = X*X*X.  
4 eleva_cubo(X) = X1 => X1 = X**3.  
5 eleva_cubo(X) = X1 => X1 = X*X*X.
```

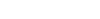
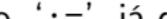
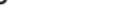
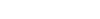
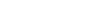
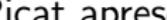
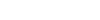
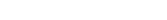
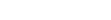
Comandos Condicionais, Laços e Repetições

- Ao contrário do Prolog, Picat apresenta conceitos e comandos da programação imperativa



Comandos Condicionais, Laços e Repetições

- 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





Comandos Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, você faz isto implicitamente)



Comandos Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, você faz isto implicitamente)
- Sua notação é:

```
if (Exp) then
    Ações
else
    Ações
:
end
```



Comandos Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, você faz isto implicitamente)
- Sua notação é:

```
if (Exp) then
    Ações
else
    Ações
:
end
```

- Onde *Exp* é uma expressão lógica avaliada como verdadeira ou falsa.



Comandos Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, você 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 no ponto e vírgula no final da linha



Comandos Condicionais

- Picat implementa uma estrutura condicional explícita (na programação em lógica, você 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 no ponto e vírgula no final da linha



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 ,
```

Estruturas de Repetição

- Picat também implementa 3 estruturas de repetição, são elas: `foreach`, `while`, e `do-while`.



Estruturas de Repetições

- Picat também implementa 3 estruturas de repetição, são elas: `foreach`, `while`, e `do-while`.
- O laço do `foreach` itera sobre termos simples e compostos.



Estruturas de Repetições

- Picat também implementa 3 estruturas de repetição, são elas: `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.



Estruturas de Repetições

- Picat também implementa 3 estruturas de repetição, são elas: `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



Estruturas de Repetições

- Picat também implementa 3 estruturas de repetição, são elas: `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
- O laço `do-while` é análogo ao `while`, porém ele sempre executa pelo menos uma vez.



Estruturas de Repetições: `foreach`

- Um laço `foreach` tem a seguinte forma:

```
foreach (E1 in D1, Cond1, ..., En in Dn, Condn)
    Metas
end
```



Estruturas de Repetições: `foreach`

- Um laço `foreach` tem a seguinte forma:

```
foreach (E1 in D1, Cond1, ..., En in Dn, Condn)
    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 .
- Laços do `foreach` podem conter múltiplos iteradores. Caso isso ocorra, o compilador interpreta isso como diversos laços aninhados.

Exemplo: foreach

```
1
2 laco_01 =>
3     L = [17, 3, 41, 25, 8, 1, 6, 40],
4     foreach (E in L)
5         println(E)
6 end.
```



Estruturas de Repetições: while

- 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.
```



Estruturas de Repetições: do-while

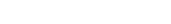
- O laço do-while tem a seguinte forma:

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).
```



Funções e Predicados Especiais

Funções e Predicados Especiais

- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado.



Funções e Predicados Especiais

- Há algumas funções e predicados especiais em Picat que necessitam de algum cuidado.
- São elas: compreensão de listas/vetores, entrada de dados e saída de dados.
- Na verdade, já fizemos uso delas, porém sem a ênfase de que são funções ora predicados.



Compreensão de Listas e Vetores I

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

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

- Onde, T é uma expressão adicionada a lista, cada E_i é um iterador, cada D_i é um termo composto ou expressão que gera um termo composto, e cada $Cond_i$ é uma condição sobre cada iterador de E_1 até E_i .
- Há uma seção dedicada a listas. Voltaremos ao assunto.



Compreensão de Listas e Vetores II

- 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("\nL: %w\nV: %w\n", L, V),
imp_vetor(V).
```

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

nl.

%%%% \$picat vetor_exemplo_01.pi



Leitura e Escrita I

- Picat tem diversas variações 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



Leitura e Escrita II

- As mais importantes são:

- `read_int(FD)` = *Int* ⇒ Lê um *Int* do arquivo *FD*.
- `read_real(FD)` = *Real* ⇒ Lê um *Float* do arquivo *FD*.
- `read_char(FD)` = *Char* ⇒ Lê um *Char* do arquivo *FD*.
- `read_line(FD)` = *String* ⇒ Lê uma *Linha* do arquivo *FD*.

- Caso se deseja ler da console, padrão `stdin`, *FD*, o nome do descriptor de arquivo, pode ser omitido.



Leitura e Escrita III

- 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.
 - `writef(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 $A\dots$. O número de argumentos não pode exceder 10.



Leitura e Escrita IV

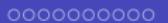
- Analogamente, para o predicado `print`, temos:
 - `print(FD, T)` ⇒ Escreve um termo T no arquivo FD .
 - `println(FD, T)` ⇒ Escreve um termo T no arquivo FD , e pula uma linha ao final do termo.
 - `printf(FD, F, A...)` ⇒ Este predicado é usado para escrita formatada para um arquivo FD , onde F indica uma série de formatos para cada termo contido no argumento $A\dots$. 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



Funções e Predicados Especiais

Comparação entre write e print

Dados ⇒	"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)













































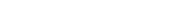












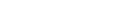
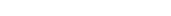








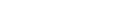










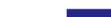




















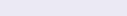
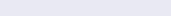




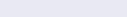
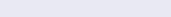


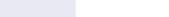


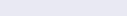
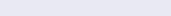


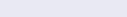
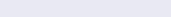


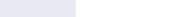



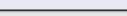
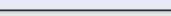


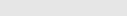
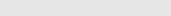


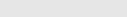
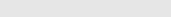


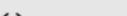
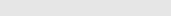


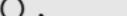
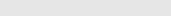


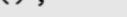
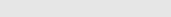


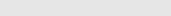


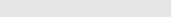








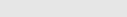
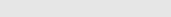




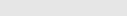
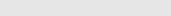










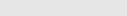




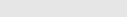
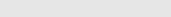




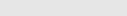
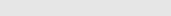




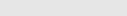
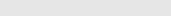


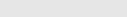
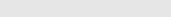


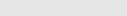
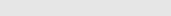


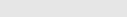
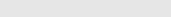


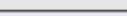
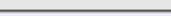


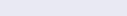
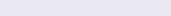




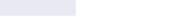





























































Exemplos – Repetções

```

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

```

```

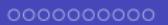
1 main =>
2     X = read_int(),
3     Y = X..X*3,
4     foreach(A in Y)
5         println(A)

```





































































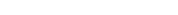










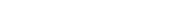



















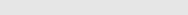
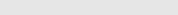




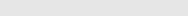
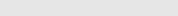




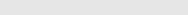
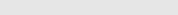


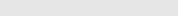


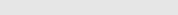


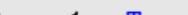
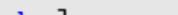




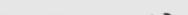




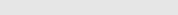




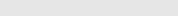





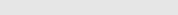




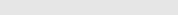


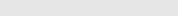


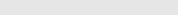


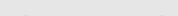


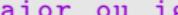


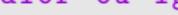


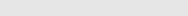
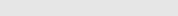


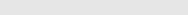
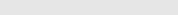


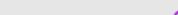




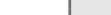
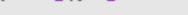
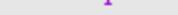


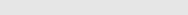
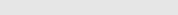




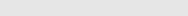
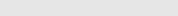


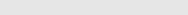
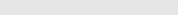


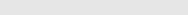
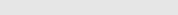








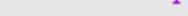
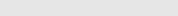




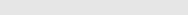
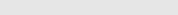




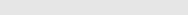
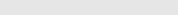






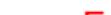

















































Reflexões

- Esta seção trata da sintaxe do Picat



Reflexões

- Esta seção trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser praticada

























































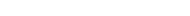




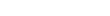






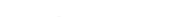








































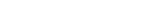








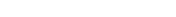
































































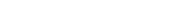




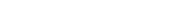








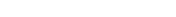




















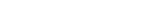






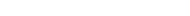
































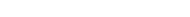








































Reflexões

- Esta seção trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, você está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado, com Picat, tudo ficou análogo a Python e a linguagem C



Reflexões

- Esta seção trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, você está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado, com Picat, tudo ficou análogo a Python e a linguagem C
- Em https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos tem vários exemplos avançados de entradas e saídas



Reflexões

- Esta seção trata da sintaxe do Picat
- Embora sua sintaxe não seja muito extensa, ela precisa ser praticada
- Como este conteúdo se assemelha as LPs clássicas, como exercício, você está apto a fazer alguns algoritmos de outras linguagens.
- Se antes o Prolog era complicado, com Picat, tudo ficou análogo a Python e a linguagem C
- Em https://github.com/claudiosa/CCS/tree/master/picat/input_output_exemplos tem vários exemplos avançados de entradas e saídas
- Mão à obra!



Recursão

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



Recursão

Recursão

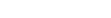
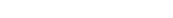
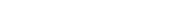
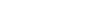
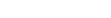
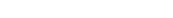
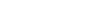
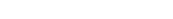
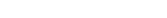
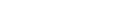
- A *recursão* é um importante conceito da matemática e presente em muitas linguagens de programação. Exemplo: LISP, Haskell, etc
- Permite expressar conceitos complexos em uma sintaxe abstrata, mas simples de ler.

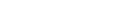
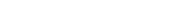
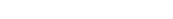
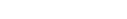
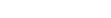
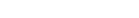
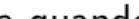
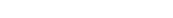
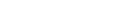
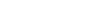


Recursão

Recursão

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







Conceitos de Recursividade via Exemplos – I

Somatório dos N naturais

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

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

Ou seja:

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



Conceitos de Recursividade via Exemplos – II

Fatorial

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

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





Conceitos de Recursividade via Exemplos – III

Sequência Fibonacci

A sequência Fibonacci é uma sequência de números calculada a partir da soma dos dois últimos números anteriores desta. Ou seja o n -esimo termo da Sequência Fibonacci é definido como a soma dos termos $n - 1$ e $n - 2$. Como fato ou definição: os dois primeiros termos, $n = 0$ e $n = 1$, são respectivamente, 0 e 1.

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



Conceitos de Recursividade via Exemplos – IV

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



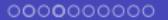
Conceitos de Recursividade via Exemplos – IV

- Podemos perceber algo em comum entre estas três regras, 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).



Conceitos de Recursividade via Exemplos – IV

- Podemos perceber algo em comum entre estas três regras, 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 *regra de aterramento*, poderá ocorrer uma recursão infinita deste regra, ou seja, são feitas infinitas chamadas recursivas da regra.



Exemplos

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

```
1 factorial(0) = 1.  
2 factorial(1) = 1.  
3 factorial(n) = n * factorial(n-1).  
4
```

```
1 fibonacci(0) = 0.  
2 fibonacci(1) = 1.  
3 fibonacci(n) = fibonacci(n-1) + fibonacci(n-2).  
4
```



Recursão Infinita

- Caso a definição do fatorial fosse modificada para:

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



Recursão Infinita

- Caso a definição do fatorial fosse modificada para:

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

- 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.



Exercício

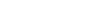
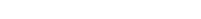
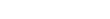
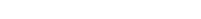
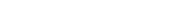
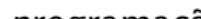
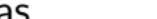
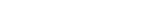
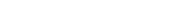
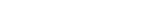
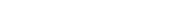
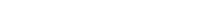
Para os exemplos anteriores, reescreva-os as formulações sob uma visão *lógica e procedural*.

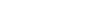
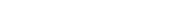
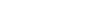
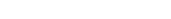
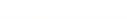
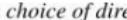
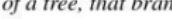
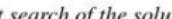
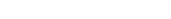
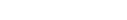
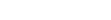


Backtracking

Backtracking

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







Backtracking

Backtracking I

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):
 $p(X_1, X_2, X_3, \dots, X_n) \ ?=> q_1(X_1), q_2(X_2), \dots, q_n(X_n)$.



Backtracking II

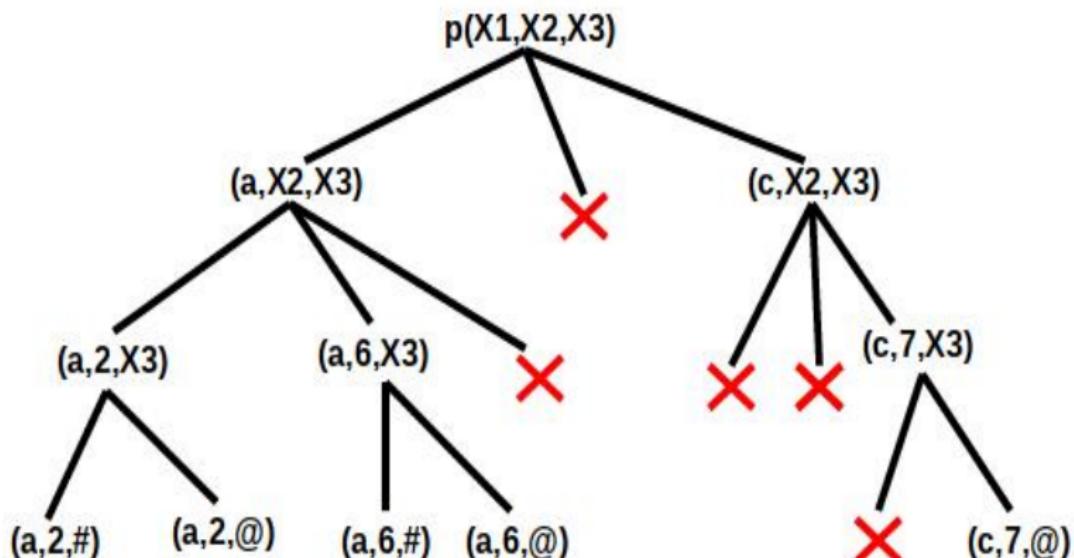
- 3 Caso ocorra uma falha durante a execução da regra p , o compilador busca re-instanciar as variáveis do corpo de p que falharem. Esta tentativa segue uma ordem:
 $q_1(X_1) \rightarrow q_2(X_2) \rightarrow \dots \rightarrow q_n(X_n)$, até a variável X_n
- 4 Caso X_n seja instanciada com sucesso, tem-se uma resposta consistente para p



Backtracking

Backtracking III

- 5 No caso de uma falha completa na regra corrente p , segue-se para uma próxima regra p ($p \ldots . ?=> \ldots$), 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.

*Backtracking*Ilustrando o *Backtracking* – 02

Exercício: descubra os domínios possíveis de x_1 , x_2 e x_3

oooooooo

o

oooooooooooo

oo

oooooooooooo

oo

oooo

oo

oooooooooooo

o

oooooooooooo

o

oo

oooooooooooo

o

oo

oooooooooooo

o

oo

oooooooooooo

o

oo

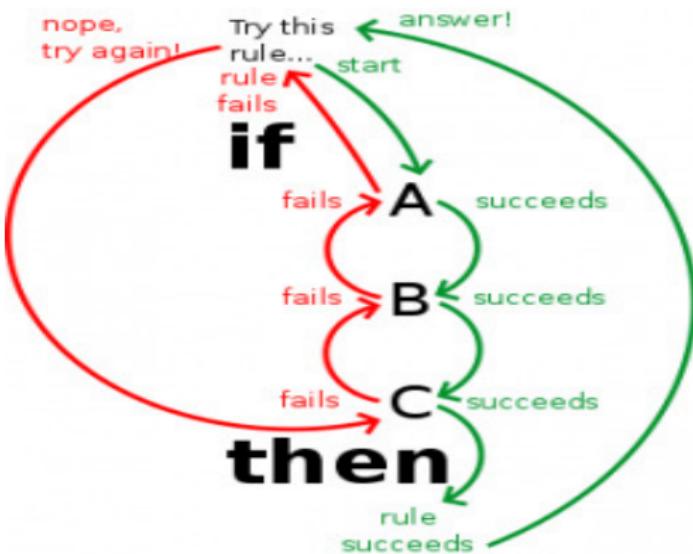
Backtracking

S



Backtracking

Ilustrando o *Backtracking* – 03





Backtracking

Exemplos

Tomando como exemplo uma relação de parentesco, como a seguinte:

```

1 index(-,-) (+,-) (-,+)
2 antecedente(ana,maria).
3 antecedente(pedro,maria).
4 antecedente(maria,paula).
5 antecedente(paula,lucas).
6 antecedente(lucas, eduarda).

7

8 index(-)
9 mulher(ana).
10 mulher(maria).
11 mulher(paula).
12 mulher(eduarda).
13 homem(pedro).
14 homem(lucas).
15

```





Backtracking

Exercícios

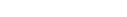
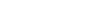
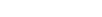
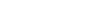
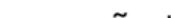
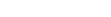
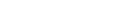
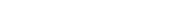
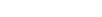
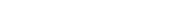
- Uma chamada do tipo $mae(maria, X)$, seria como perguntar ao compilador "Maria é mãe de quem ?".
- Nesse caso o compilador iria testar cada possível valor que pudesse ser unificado com X que pudesse satisfazer a regra $mae(maria, X)$.
- Ou seja, seria como se estivéssemos perguntando:
 - "Maria é mãe de Ana ?".
 - "Maria é mãe de Paula ?".
 - "Maria é mãe de Pedro ?".



Backtracking

Reflexões

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



Backtracking

Reflexões

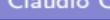
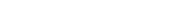
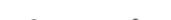
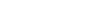
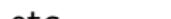
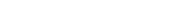
- A recursão é o paradigma das linguagens declarativas como Haskell, Prolog, Picat, ... etc
- As regras recursivas são construídas com uma ou mais *regras aterradas*, que **sempre vem antes** das demais regras recursivas, as quais podem ou não terem o *backtracking* habilitados ($?=>$)



Backtracking

Reflexões

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





Backtracking

Reflexões

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



Backtracking

Reflexões

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



Listas

- Requisito: conceitos de recursividade e functores dominados!
- Os conceitos são os próximos os das LPs convencionais
- Essencialmente vamos computar sob uma árvore binária (cada nó 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

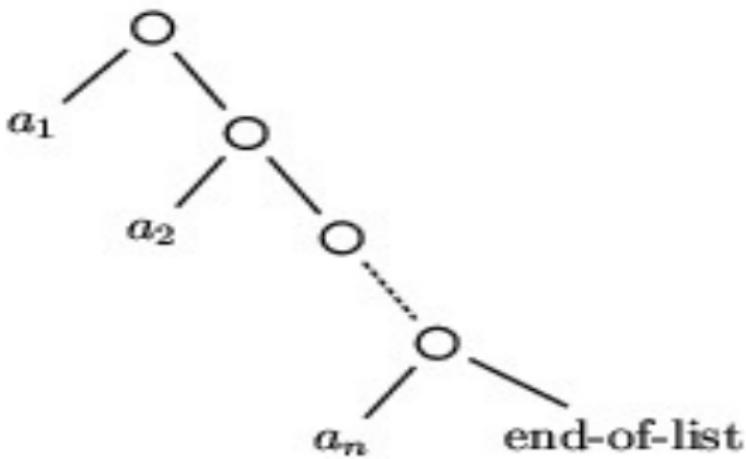


Figura 3: Uma estrutura Lista – Homogênea

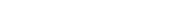
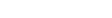
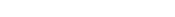
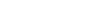
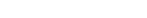
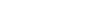
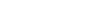
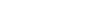
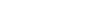
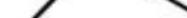
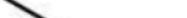
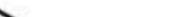
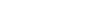
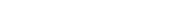


Figura 4: Listas são inherentemente **recursivas!**



Exemplos de Listas

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]]



Sintaxe das Listas I

Definições iniciais (e recursivas)

- Uma lista é uma sequência de objetos;
- Uma lista é uma estrutura de dados que representa uma coleção de objetos homogêneos;
- Uma lista apresenta uma hierarquia natural, internamente, em *cabeça* de lista e sub-lista, até o fim da lista.



Sintaxe das Listas II

Notação:



Sintaxe das Listas III

- 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.





Exemplo: encontrar o comprimento de uma lista I

- 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: 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: encontrar o comprimento de uma lista III

Um “*mapa de memória*” é dado por:

	Regra	X	T	N	$N = N+1$
compto([a,b,c,d],N)	#2	a	[b,c,d]	$3 \rightarrow$	$3+1=4$
compto([b,c,d],N)	#2	b	[c,d]	$2 \rightarrow$	$\nwarrow 2+1$
compto([c,d],N)	#2	c	[d]	$1 \rightarrow$	$\nwarrow 1+1$
compto([d],N)	#2	d	[]	$0 \rightarrow$	$\nwarrow 0+1$
compto([],N)	#1	-	-	-	$\nwarrow 0$



Exemplo: 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:



Exemplo: verificar a pertinência de um objeto na lista II

```
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: verificar a pertinência de um objeto na lista III

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: adicionar um elemento em uma lista |

- 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: 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: 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: 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 ).
```



Geração de Listas – *list comprehension* I

- 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



Geração de Listas – *list comprehension* II

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*



Geração de Listas – *list comprehension III*

- 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; forem verdadeiras, então o valor do termo T é adicionado na lista em construção*



Geração de Listas – *list comprehension* IV

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

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

Isto é o mesmo como

```
to_array([T : E1 in D1, Cond1, ..., En in Dn,  
          Condn])
```



Exemplos de *list comprehension* I

```
L1 = [I : I in 1..20]
```

```
L2 = [I : I in 1..20, I>10, I<20]
```

```
L3 = [(A,I) : A in [a,b], I in 1..2]
```

```
V1 = {I : I in 1 .. 7}
```

```
V2 = {(I,J) : I in 1 .. 2, J in 11 .. 12}
```

Concluindo Listas

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



Concluindo Listas

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



Concluindo Listas

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



Concluindo Listas

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



Concluindo Listas

- Há muitos predicados e funções prontas sobre listas nos módulos do Picat
- 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 em problemas complexos, como na aula de aplicações de buscas.

Buscas

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



Buscas

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



Buscas

- Requisito: conceitos de listas e recursividade dominados!
- Além destes: conceitos grafos, árvores de busca, nós, etc
- 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 a nossa estratégia de resolver problemas!



Ciclo Euleriano I

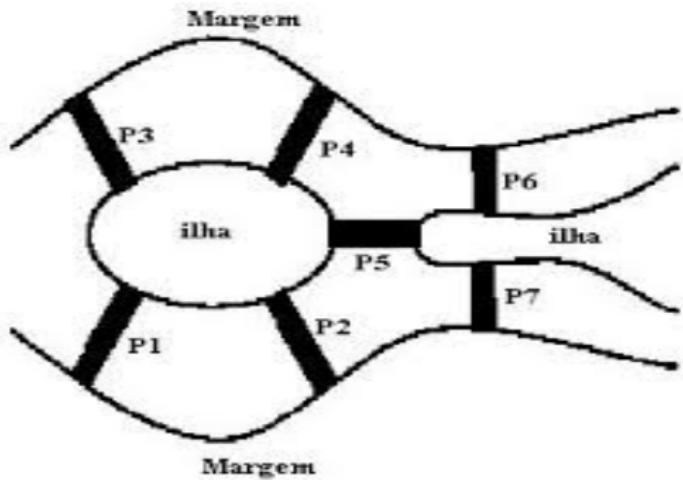


Figura 5: Ciclo Euleriano – Problema das Pontes de Königsberg



Ciclo Euleriano II

- 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



Ciclo Euleriano III

- Curiosamente, este problema, computacionalmente é fácil de resolver!



Caminho Hamiltoniano I

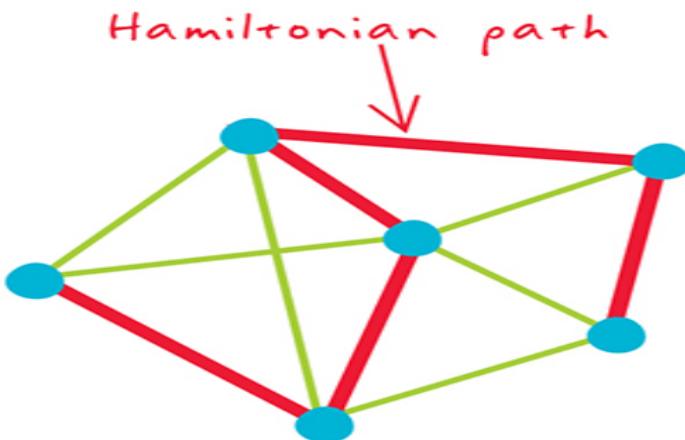


Figura 6: Caminho Hamiltoniano – Há um caminho que passe por todas cidades uma única vez?



Caminho Hamiltoniano II

- 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!
- Mas é este que vamos usar como exemplo, com um algoritmo ingênuo.



Problemas, Estados, Grafos e Árvores de Buscas I

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



Problemas, Estados, Grafos e Árvores de Buscas II

- 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.

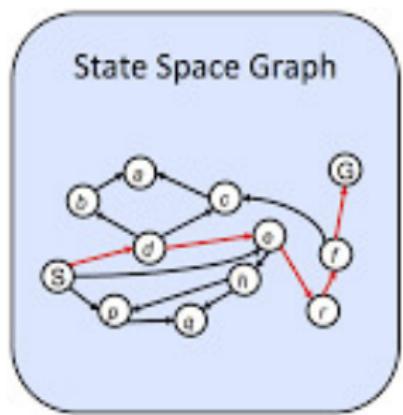


Problemas, Estados, Grafos e Árvores de Buscas III

- Logo, vamos empregar alguns conceitos da teoria dos grafos, em modelar problemas e resolvê-los por um esquema de busca computacional



Problemas de Grafos se Transformam em Árvores de Buscas



Each NODE in in
the search tree is
an entire PATH in
the state space
graph.

We construct both
on demand – and
we construct as
little as possible.

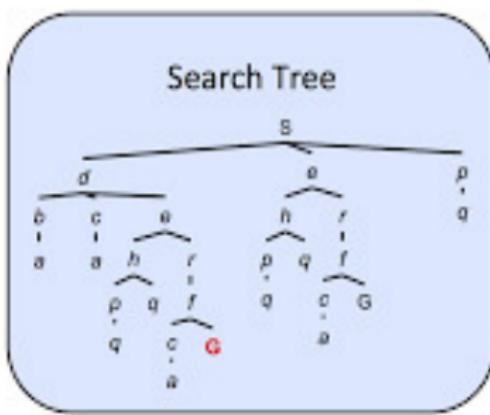


Figura 7: Google ...

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



Núcleo Geral de Buscas I

Pseudo-código já em Picat

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

busca(S,P,P) ?=> objetivo(S).

busca(S,Visited,P) =>

proximo_estado(S,Nxt),
 estado_seguro(Nxt),

% objetivo alcançado : FIM

% gera um proximo estado
% verifica se este estado

Núcleo Geral de Buscas II

```
sem_loop(Nxt, Visited),           % verifica se está em loop
busca(Nxt, [Nxt|Visited], P).    % continue a busca recursiva
```



Núcleo Geral de Buscas III

```
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(...).
```



Núcleo Geral de Buscas IV

Vamos reescrever este pseudo-código em um problema!



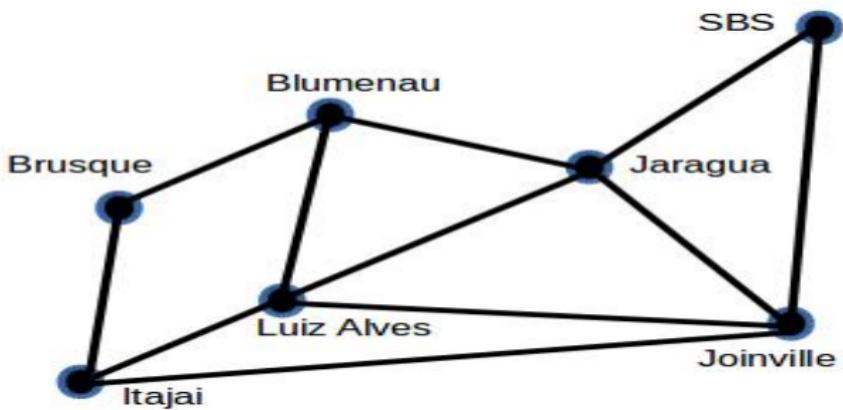
Caminho Hamiltoniano Aplicado



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





O nosso viajante 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



O nosso viajante II

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

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

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



O nosso viajante III

```
%% 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) .
```



O nosso viajante IV

arco(blumenau, itajai) .

arco(brusque, itajai) .

arco(brusque, blumenau) .

- Claro, este problema é pequeno e construindo o grafo dá para perceber que existe algumas soluções
- Para resolver este problema vamos utilizar uma *busca em profundidade*
- Esta *busca em profundidade*, encontra-se inserida no contexto buscas em geral, visto anteriormente



O Miolo ou Núcleo da Busca

```

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]) ).
```

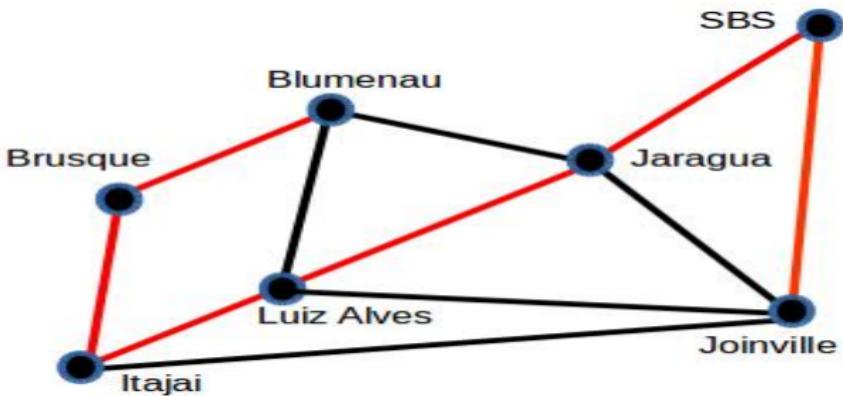
O Código Completo

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

oooooooooooo

o
oooooooooooooooooooooo
oooooooooooo
o
oooooooooooo
oooooooooooooo
ooo
oooo
o
oo
oooooooooooooooooooo
oooooooooooo

Uma Solução





Concluindo Buscas

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



Concluindo Buscas

- A recursividade na modelagem das buscas, define uma metodologia de se *programar em lógica* e resolver problemas
- Exercitar-se para aprender os detalhes!



Concluindo Buscas

- A recursividade na modelagem das buscas, define uma metodologia de se *programar em lógica* e resolver problemas
- Exercitar-se para aprender os detalhes!
- Há o uso extensivo de listas em problemas complexos



Concluindo Buscas

- A recursividade na modelagem das buscas, define uma metodologia de se *programar em lógica* e resolver problemas
- Exercitar-se para aprender os detalhes!
- Há o uso extensivo de listas em problemas complexos
- Aos problemas complexos, há outras técnicas de programação para resolvê-los.



Concluindo Buscas

- A recursividade na modelagem das buscas, define uma metodologia de se *programar em lógica* e resolver problemas
- Exercitar-se para aprender os detalhes!
- Há o uso extensivo de listas em problemas complexos
- Aos problemas complexos, há outras técnicas de programação para resolvê-los.
- Assunto das próximas seções!



Programação Dinâmica

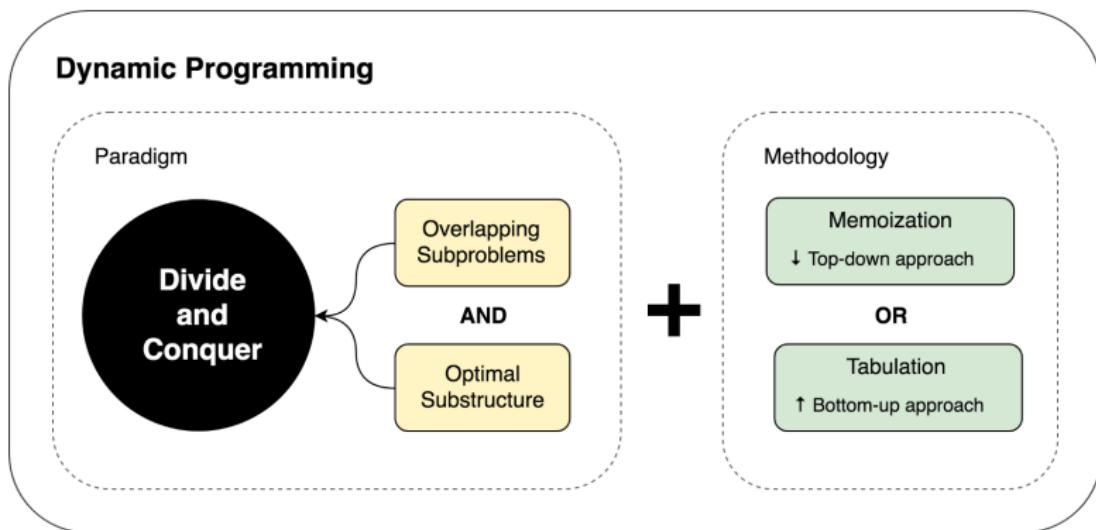


Figura 8: Conceitos da Programação Dinâmica – (PD)



Programação Dinâmica (PD) – I

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



Programação Dinâmica (PD) – I

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



Programação Dinâmica (PD) – I

- 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*



Programação Dinâmica (PD) – I

- 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

Programação Dinâmica (PD) – II

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



Programação Dinâmica (PD) – II

- 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*



Programação Dinâmica (PD) – II

- 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*)



Programação Dinâmica (PD) – II

- 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:
- $(x + y)^0 = 1$
- $(x + y)^1 = x + y$
- $(x + y)^2 = x^2 + 2xy + y^2$
- $(x + y)^3 = x^3y^0 + 3x^2y^1 + 3x^1y^2 + x^0y^3$
- $(x + y)^4 = x^4y^0 + 4x^3y^1 + 6x^2y^2 + 4x^1y^3 + x^0y^4.$
-



Exemplo de Uso da Programação Dinâmica – (PD)

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



Exemplo de Uso da Programação Dinâmica – (PD)

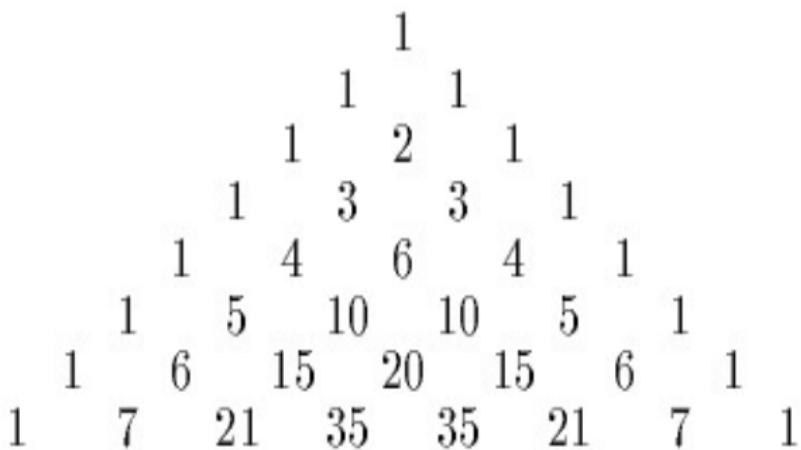


Figura 9: O triângulo de Pascal



Exemplo de Uso da Programação Dinâmica – (PD)

		A Triângulo de Pascal							
		P	0	1	2	3	4	5	6
N									
0		$\binom{0}{0}$							1
1		$\binom{1}{0}$	$\binom{1}{1}$						<u>1</u> <u>1</u>
2		$\binom{2}{0}$	$\binom{2}{1}$	$\binom{2}{2}$					<u>1</u> <u>2</u> <u>1</u>
3		$\binom{3}{0}$	$\binom{3}{1}$	$\binom{3}{2}$	$\binom{3}{3}$				<u>1</u> <u>3</u> <u>3</u> <u>1</u>
4		$\binom{4}{0}$	$\binom{4}{1}$	$\binom{4}{2}$	$\binom{4}{3}$	$\binom{4}{4}$			<u>1</u> <u>4</u> <u>6</u> <u>4</u> <u>1</u>
5		$\binom{5}{0}$	$\binom{5}{1}$	$\binom{5}{2}$	$\binom{5}{3}$	$\binom{5}{4}$	$\binom{5}{5}$		<u>1</u> <u>5</u> <u>10</u> <u>10</u> <u>5</u> <u>1</u>
6		$\binom{6}{0}$	$\binom{6}{1}$	$\binom{6}{2}$	$\binom{6}{3}$	$\binom{6}{4}$	$\binom{6}{5}$	$\binom{6}{6}$	1 <u>6</u> <u>15</u> <u>20</u> <u>15</u> <u>6</u> <u>1</u>

Figura 10: O triângulo de Pascal – Coeficientes Binomiais



Formulação Matemática – I

- 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 .



Formulação Matemática – I

- 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:

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



Formulação Matemática – II

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

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



Formulação Matemática – II

- Alternativa ao cálculo do factorial, 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: $\binom{n-1}{k-1}$



Formulação Matemática – II

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

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

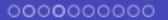
- O coeficiente binomial é muito utilizado no Triângulo de Pascal, onde o termo na linha n e coluna k é dado por: $\binom{n-1}{k-1}$
- A fórmula de Stiffel é recorrente, e diretamente escrita em Picat.
Veja os códigos.



Código em Partes

```
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).
```



Código em Partes

```

main ?=>
    statistics(runtime,_), % faz uma marca do 1o. statistics
    N = 10, %% ate h 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_
    printf("\n ======\n ")

```

Código Completo

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



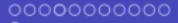
Saída

```
[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 0.000000





Reflexões sobre PD

- Há outros métodos para se resolver estes problemas



Reflexões sobre PD

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



Planejamento

- Requisitos: conceitos de listas e recursividade dominados!



Planejamento

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



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



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



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* ?



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





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!



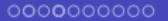
Planejamento

- 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.



Planejamento

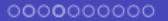
- Problemas em geral necessitam de um **plano** para serem solucionados, assim, há uma visão que encontrar um plano para um problema, é ter uma solução!
- Em resumo, a área de planejamento é bem complexa, antiga na área da IA e robótica (1970 – STRIPS), efervescente, e de muito interesse na indústria.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



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.





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.





Definições

- Plano: seqüência ordenada de ações

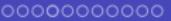
Definições

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



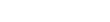
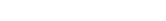
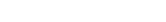
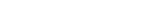
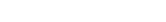
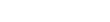
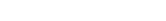
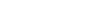
Definições

- 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.



Definições

- 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 sequência de ações, que mudam de estados a cada ação, até atingir um estado final.





Exemplos do que é planejamento ...

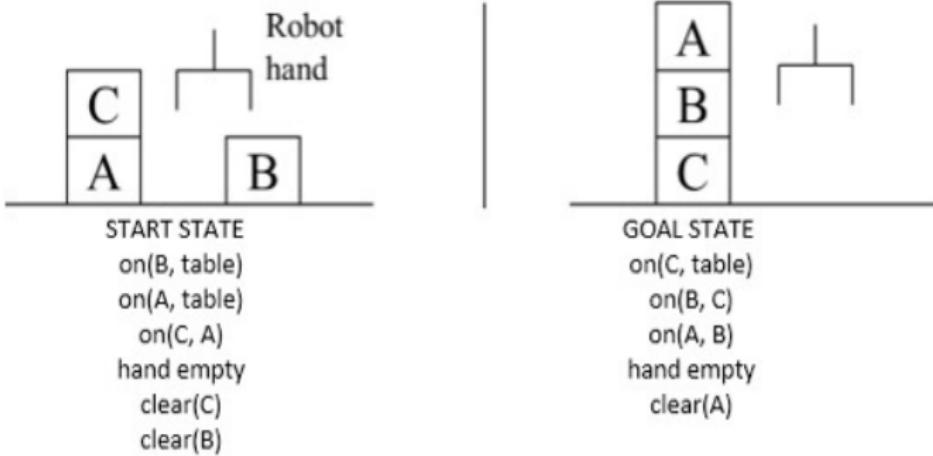


Figura 12: O mundo dos blocos



Espaço de Estados

Graph of state space

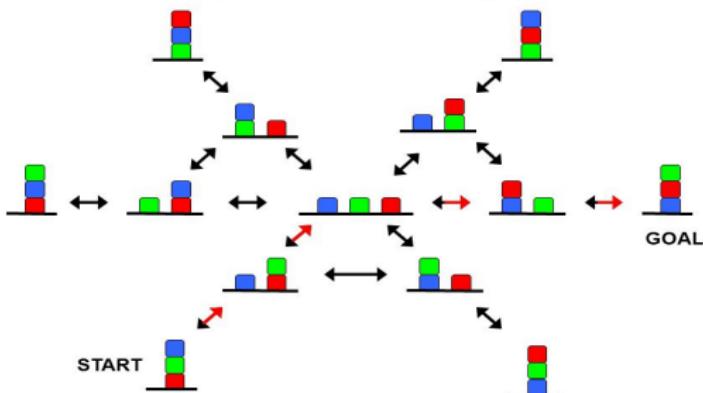


Figura 13: O espaço de estados do *mundo dos blocos* × ações



Elementos de um Planejador – Vocabulário I

- 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?



Elementos de um Planejador – Vocabulário II

- 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



Elementos de um Planejador – Vocabulário III

- 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



O Problema Exemplo I

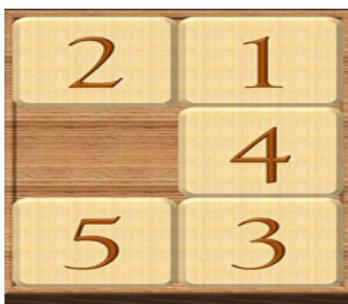


Figura 14: Um quebra-cabeça (2×3 ou 3×2) *simplificado* do conhecido 3×3



























































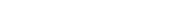


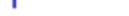




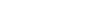














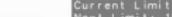
























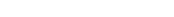


















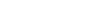
























































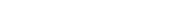




























































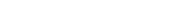









































Partes do código comentado I

```
/*
A  B  C
D  E  F
*/
```

```
%%%%%%%%
%    1 5 %
% 4 3 2 %
%%%%%%%
```



Partes do código comentado II

```
import datetime.  
import planner.
```



Partes do código comentado III

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

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



Partes do código comentado IV

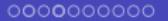
```
% Up <-> Down
/* Descrevendo as possíveis ações para o planner */
action([A,B,C, D,E,F], S1, Acao, Custo_Acao ) ?=>
    Custo_Acao = 1,
    ( A == 0 ), %% conj. condições
    S1 = [D,B,C, 0,E,F],
    Acao = ($up(D),S1). %% a ação + estado modificado
```

```
action([A,B,C, D,E,F], S1, Acao, Custo_Acao ) ?=>
    Custo_Acao = 1,
    (A == 0 ), %% conj. condições
```

Partes do código comentado V

```
S1 = [0,B,C, A,E,F],  
Acao = ($dow(A),S1). %%a acao + estado modificado
```

.....



Partes do código comentado VI

```
% 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],
```



Partes do código comentado VII

```
Acao = ($right(A), S1). %%a acao + estado modificado
```

.....



Partes do código comentado VIII

```
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) ,
```



Partes do código comentado IX

```
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 .... !!!!") .
```

```
.....
```



O código

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

Parte da Saída I

```
[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,
(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
```

oooooooooooo

o

oooooooooooooooooooo

oo

oooooooooooooooooooo

oo

ooooooo

oo

oooooooooooooooo

o

o

oooooooooooo

o

o

ooooo

Parte da Saída II

Acao: left(5)

1 5 0

4 3 2

Parte da Saída III

.....
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 módulo do *planner* do Picat

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



O módulo do *planner* do Picat

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



O módulo do *planner* do Picat

- 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)`



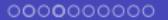
O módulo do *planner* do Picat

- 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 módulo do *planner* do Picat

- 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:



O módulo do *planner* do Picat

- 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)`



Reflexões

- Outros métodos para se resolver estes problemas



Reflexões

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

Reflexões

- 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



Reflexões

- 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



Programação por Restrições (PR) – I

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



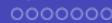
Programação por Restrições (PR) – I

- 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



Programação por Restrições (PR) – I

- 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.







































































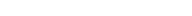


















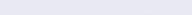
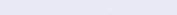


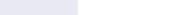


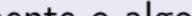


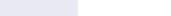




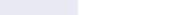



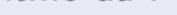


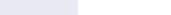


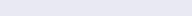
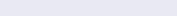



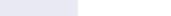


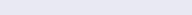
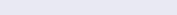



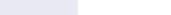



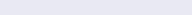
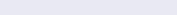


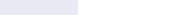



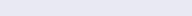
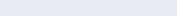


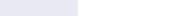



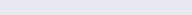
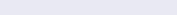


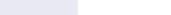



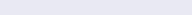
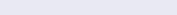


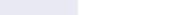



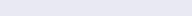
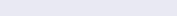


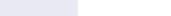



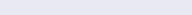
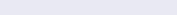


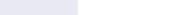



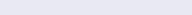
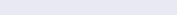


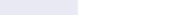



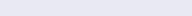
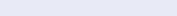


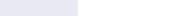



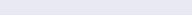
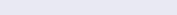


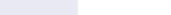



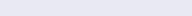
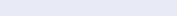


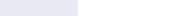



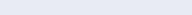
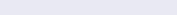


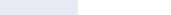



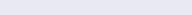
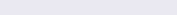


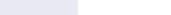



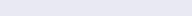
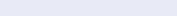


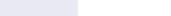



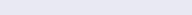
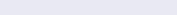


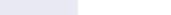



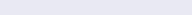
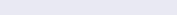


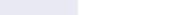



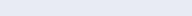
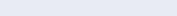


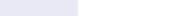



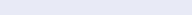
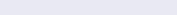


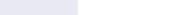



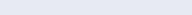
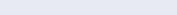


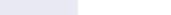



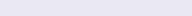
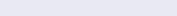


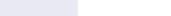



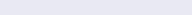
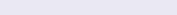


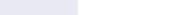



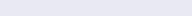
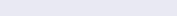


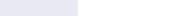



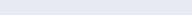
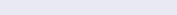


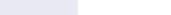



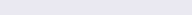
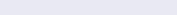


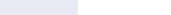
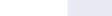
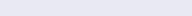
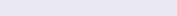



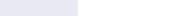


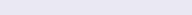
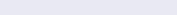



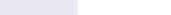


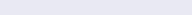
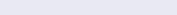



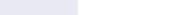





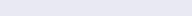
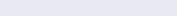


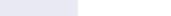



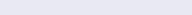
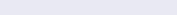


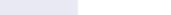



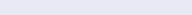
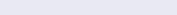


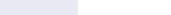



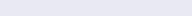
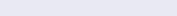


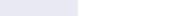



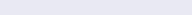
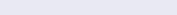


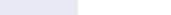



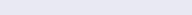
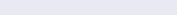












Programação por Restrições (PR) – II

- Aproximadamente o algoritmo da PR é dado:

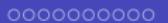
- 1 Avaliar algebraicamente os domínios das variáveis com suas restrições
- 2 Intercala iterativamente a propagação de restrições com um algoritmo de busca
- 3 A cada variável instanciada, o processo é repetido sobre as demais variáveis, reduzindo progressivamente o espaço de busca
- 4 Volte ao passo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



Programação por Restrições (PR) – II

- 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





Programação por Restrições (PR) – II

- 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





Programação por Restrições (PR) – III

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



Programação por Restrições (PR) – III

- 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



Programação por Restrições (PR) – III

- 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!

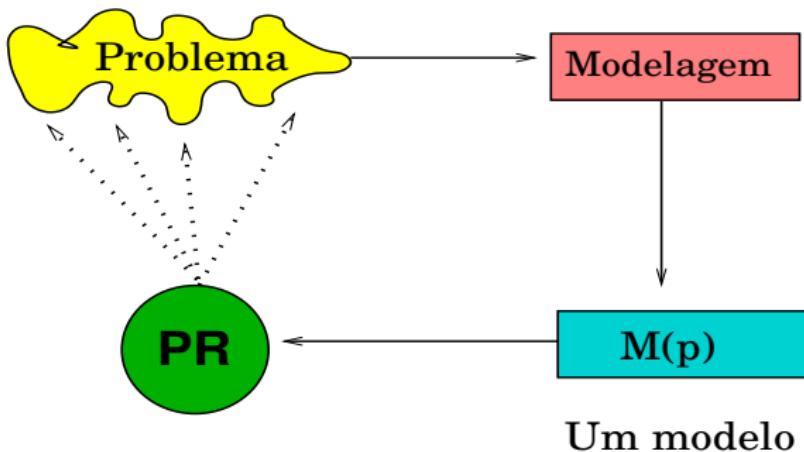


Programação por Restrições (PR) – III

- 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!
-

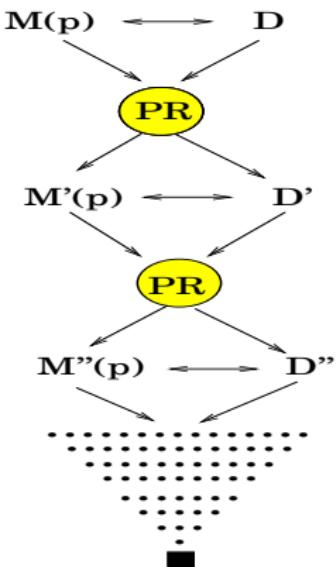


Metodologia da Construção de Modelos





Fluxo de Cálculo da PR





Onde o objetivo da PR é:

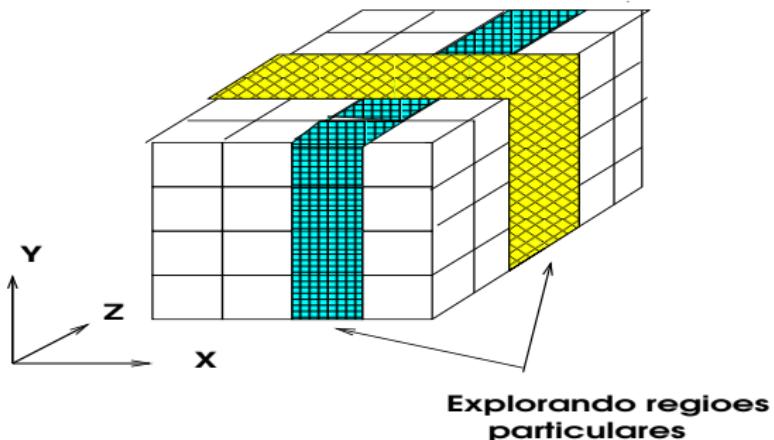


Figura 16: Realizar buscas com regiões reduzidas – promissoras (regiões factíveis de soluções)



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 deêm este número par.



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 deêm este número par.
- Exemplo:

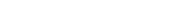
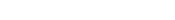
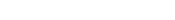
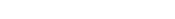
Seja o PAR = 18

Uma solução:

$N_1 = 7$ e $N_2 = 11$

pois

$N_1 + N_2 = 18$





Modelagem do Problema

- 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} :
$$N_1 + N_2 = N_{PAR}$$



Modelagem do Problema

- 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
$$N_1 \neq N_2$$



Modelagem do Problema

- 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!



Código Completo

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

Código em Partes

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

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



Código em Partes

```
% 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.....")
```

```
.
```

Código em Partes

```
import cp.

% main => modelo .
% main ?=> modelo, fail.
% main => true.

main =>
    L = findall(_, $modelo),
    writef("\n Total de solucoes: ~d \n", length(L)) .
```



Saída – I

```
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: 99 N2: 859
```

oooooooooooo

o
oooooooooooooooooooooo
oooooooooooo
o
oooooooooooo
oooooooooooooo
ooo
oooooo
o
oo
oooooooooooooooooooo
oooooooooooooooo

Saída – II

.....
N1: 353 N2: 29

.....
N1: 359 N2: 23

.....
N1: 379 N2: 3

.....
Total de solucoes: 18

yes

Picat>



Exemplo – 02 – Escala de Consultórios

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



Exemplo – 02 – Escala de Consultórios

- 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)



Exemplo – 02 – Escala de Consultórios

- 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 ingênuia e sem muitos critérios



Modelagem do Problema

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



Modelagem do Problema

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



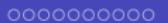
Modelagem do Problema

- 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.



Modelagem do Problema

- 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`



Modelagem do Problema

- 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.



Modelagem – Comentários

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

Modelagem – Comentários

- 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



Modelagem – Comentários

- 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

Código Completo

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



Código em Partes

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

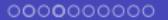
```

Código em Partes

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

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

...



Código em Partes

```
%% 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])
```



Código em Partes

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

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



Código em Partes

```

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 .

```



Saída - I

```
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

Saída - II

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

Saída - III

```
$ 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 :pediatra :gineco :cardio  
:gineco :clin_geral :otorrino :pediatra :oftalmo  
:oftalmo :pediatra :clin_geral :cardio :otorrino  
:pediatra :otorrino :oftalmo :clin_geral :gineco
```



Reflexões

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



Reflexões

- 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



Reflexões

- 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



Reflexões

- 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.



Resumindo

- Picat é jovem (nascida em 2013);
- Uma evolução ao Prolog após seus mais de 40 anos de existência e sucesso!
- Sua sintaxe é moderna;
- Código aberto, multi-plataforma, e repleta de possibilidades;
- Uso para fins diversos;
- Muitas bibliotecas específicas prontas: CP, SAT, Planner, etc;
- .