Picat: uma Linguagem Multiparadigma

Claudio Cesar de Sá*[1], Paulo Victor de Aguiar†[1], and Neng-Fa Zhou‡[2]

[1]Departamento de Ciência da Computação , Universidade do Estado de Santa Catarina (UDESC), Rua Paulo Malschitzki, 200 - Campus Universitário Prof. Avelino Marcante, 89.219-710 – Joinville – SC – Brazil
[2]Department of Computer and Information Science, Brooklyn College, The City University of New York, 2900 Bedford Avenue, Brooklyn, NY 11210-2889 – USA

14 de junho de 2016

Resumo

This course proposal aims to present Picat, a multi-paradigm programming language, describing its possibilities and experiments for novices in programming. Although the theme of "the first undergraduate programming language course " is a recurrent dilemma, this course presents a new perspective for pedagogy using programming in logic. This course presents some of the classic examples in introductory programming as well as some advanced problem-solving techniques in Picat.

Esta proposta de curso apresenta o Picat, uma linguagem multiparadigma, descrevendo suas possibilidades e experimentos para iniciantes em programação. Embora o tema sobre "a primeira linguagem de programação em cursos de graduação" seja um dilema recorrente, este curso visa apresentar uma nova perspectiva do ensino de programação usando a programação em lógica. A proposta é desenvolver alguns exemplos clássicos das disciplinas introdutórias de programação em Picat, ao final estende-se o seu uso na resolução de problemas avançados de programação.

*claudio.sa@udesc.br
†pavaguiar@gmail.com
‡zhou@sci.brooklyn.cuny.edu

Sumário

1 Introdução **3**
1.1 Contexto das Linguagens de Programação 3
1.2 O Ensino da Primeira Linguagem de Programação 4

2 A Linguagem Picat **5**

3 Usando o Picat **6**
3.1 Como Instalar? 6
3.2 Como Funciona? 6
3.3 Como Usar? 6

4 Elementos da Linguagem **8**
4.1 Tipos de Dados 8
4.2 Variáveis 9
4.3 Átomos 9
4.4 Números 10
4.5 Termos Compostos 10

5 Exemplos **12**
5.1 Atribuição, Igualdade e Unificação 12
5.2 Estruturas de Controle 13
5.3 Entradas e Saídas 13
5.4 Laços de Repetição 14
5.5 Listas e Vetores 15
5.6 Funções e Predicados Recursivos 16

6 Tópicos Avançados **16**
6.1 Programação por Restrições 17
6.2 Programação Dinâmica 17
6.3 Planejamento 18

7 Conclusões **21**

1 Introdução

1.1 Contexto das Linguagens de Programação

Uma linguagem de programação é uma ferramenta computacional destinada a escrever códigos a nível de usuário, conhecido como *código-fonte*, o qual é traduzido é um *código-alvo* ou *código-objeto*. Basicamente é um mecanismo do programador se comunicar com o computador através de um conjunto de instruções e regras. Tanto as instruções quanto as regras são submetidas há uma sintaxe e semântica da linguagem. Ou seja, elas precisam seguir um certo padrão conforme a funcionalidade da linguagem [Sebesta 2003].

As linguagens de programação são onipresentes em ambientes de desenvolvimento de sistemas, e desde os primórdios computadores, estas vem se apresentado sob novas propostas e visões. Assim, surgiram os diversos paradigmas de linguagens, dentre os mais conhecidos são: imperativo ou procedural, funcional, orientação-objeto, lógico, baseado-em-regras, etc. Eis as características de alguns paradigmas citados anteriormente [Sebesta 2003] [Suh 2011]:

Imperativo: A programação imperativa trabalha com uma sequência de ações, chamadas de instruções para executar um programa e nesse processo pode alterar o estado de suas variáveis. Isto é feito pelo conceito de atribuição de valor à variáveis, estruturas de decisão e repetição.

Funcional: As linguagens funcionais trabalham no armazenamento de dados e dão preferência a recursividade do que laços de repetição (*loops*). Nessa linguagem são utilizadas funções cuja a interação entre elas fazem o programa ser executado. O objetivo dessas funções é retornar um valor.

Orientação-objeto: A orientação a objetos trabalha com um conjunto de objetos e suas classes. Os objetos recebem instância de valores via mensagens e a maneira como ele reage à elas é chamado de *método*.

Lógico: Baseado na utilização de sentenças lógicas, utiliza-se a lógica matemática em linguagens de programação. Um dos pontos importantes são os objetos e seus relacionamentos, onde se podem declarar vários fatos, regras e questionamentos, onde as respostas podem ser observadas através das regras inferenciais da lógica [Aguiar 2016, Kowalski 1974].

A linguagem aqui apresentada se apresenta sob uma visão de multiparadigma, com conceitos predominantes do paradigma lógico, com uma sintaxe funcional. Sob estes quatro importantes paradigmas, vamos delinear características de Picat versus outras linguagens de programação. Embora existam centenas de linguagens, foram escolhidas 5 linguagens populares para um comparativo com o Picat. As 5 linguagens bem conhecidas são: C,

Haskell, Java, Prolog e Python [Sestoft 2012]. Este comparativo é apresentado na tabela 1 (ver página 4) onde alguns critérios foram extraídos de [Rosettacode 2016] [Wikipedia 2016].

A tabela 1 (ver página 4) está distante de estar completa e de um consenso sobre características. Esta foi em parte baseada do sítio [Rosettacode 2016]. Um outro comparativo pode ser encontrado no sítio [Wikipedia 2016] onde Picat figura entre muitas outras linguagens.

O texto precisa ser revisado....

1.2 O Ensino da Primeira Linguagem de Programação

Seção de PROBLEMATIZACAO, pretende-se escrever tópicos como:

O perfil do jovem ao ingressar nos cursos de informática e computação:

A baixa atratividade pela área: embora o número de postos de trabalho só tende a aumentar

A evasão dos cursos de informática e computação:

- A dificuldade da base matemática:
- A dificuldade da abstração de problemas:

Tabela 1: Comparativo entre algumas linguagens

	C	**Haskell**	**Java**	**Prolog**	**Python**	**Picat**
Paradigma(s)	procedural	funcional	orientado à objetos	lógico	multi-paradigma	multi-paradigma
Tipagem	fraca	forte	forte	fraca	fraca	fraca
Verificação de tipos	estático	estático	estático	dinâmico	dinâmico	dinâmico
Possui segurança?	não	sim	sim	não	sim	sim
Passagem de parâmetros	valor	-	valor	-	valor	casamento
Compatib. das estruturas (tipos)	nominativo	estrutura polimórfica	nominativo	-	-	-
Coletor de lixo?	não	sim	sim	sim	sim	sim
Legibilidade	baixa	média	média	média	boa	boa
Padronização	ANSI C89	Haskell 2010 Report	Java SE Specifications	ISO	não possui	não possui
Aplicação	diversas	diversas	diversas	IA, etc	diversas	diversas
Arquitetura de SO	diversas	diversas	diversas	diversas	diversas	diversas

- A dificuldade com a primeira linguagem de programação: Aqui se beneficiam linguagens como Python, como um caso de sucesso nos Estados Unidos, tanto nas escolas de ensino médio como nas universidades.

Enfim, muitas são as possibilidade de atrair mais jovens para as áreas exatas, mas novos paradigmas de apresentar um conteúdo e dar uma consecução ao mesmo se faz necessário. Neste viés, sucessos de linguagens como Python, Lua (o maior sucesso do Brasil na área), Rugby on Rails, Rasquet, tem apresentado resultados bem interessantes tando dentro como fora do mundo acadêmico.

Assim, P.I.C.A.T. com suas raízes antigas, apresenta paralelos a estas linguagens em um novo formato de sintaxe, elegância e velocidade. Finalmente, esta resolve problemas complexos como serão apresentados na seção 6. Estes problemas complexos são desafios instigantes a natureza humana, porque não aos jovens?

2 A Linguagem Picat

O Picat é uma linguagem multiparadigma projetada para aplicações gerais de programação [Zhou et al. 2015]. Esta foi criada em 2013 por Neng-Fa Zhou e Jonathan Fruhman utilizando o B-Prolog como base na implementação, ambas as linguagens utilizam regras lógicas na programação. A terminologia do Picat segue as bases teóricas da linguagem Prolog [Zhou et al. 2015]. Isto é lógica de primeira-ordem [Enderton 2001] onde os objetos são chamados por *termos*. Os destaques do Picat é a sua natureza declarativa, funcional, tipagem dinâmica, sintaxe simples, porém poderosa. O nome Picat é um anacrônico onde cada letra representa uma característica marcante de sua funcionalidade:

***P*attern-matching**: Utiliza o conceito de casamento de padrão. Um predicado define uma relação e pode ter zero ou várias respostas. Aqui, uma função é um predicado especial que sempre retorna uma única resposta. Ambos são definidos com regras em Picat, e seus predicados e funções seguem as *regras do casamento*.

***I*ntuitive**: O Picat oferece atribuições e laços de repetição (*loops*) para a programação dos dias de hoje. Uma variável atribuída pode imitar várias variáveis lógicas, alterando seu valor seguindo o estado da computação. As atribuições são úteis para associar os termos, bem como utilizadas nas estruturas de laços repetitivos (exemplos na subseção 5.4).

***C*onstraints**: Picat suporta a programação por restrições. Dado um conjunto de variáveis, cada uma possui um domínio de valores possíveis e restrições para limitar os valores a serem atribuídos nas variáveis. O objetivo é atribuir os valores que satisfaçam todas as restrições (exemplo na subseção 6.1).

***A*ctors**: Atores são chamadas orientadas à eventos. Em Picat, as regras de ação descrevem comportamentos dos atores. Um ator recebe um objeto e dispara uma ação. Os eventos são postados via canais de mensagem e um ator pode ser conectado há um canal, verificar e/ou processar seus eventos postados no canal. Neste ponto, estes atores desempenham características da área de inteligência artificial, especificamente a áreas de planejamento e sistemas multi-agentes.

***Tabling*:** Considerando que operações entre variáveis podem ser armazenados parcialmente em uma tabela na memória, permitindo que um programa acesse valores já calculados. Assim, evita-se a repetição de operações já realizadas. Com esta técnica de *memoization*, o Picat oferece soluções imediatas para problemas de programação dinâmica.

Seguindo o anacrônico de P.I.C.A.T. , as duas primeiras letras se encontram na seção ilustradas 5 e as três últimas letras, seguem para seção 6.

Precisa ser melhorado e estendido .... e ligar os conceitos de LPs a Picat [Sestoft 2012, Sebesta 2003, Tate 2010, Editors 2016b, Editors 2016a]. Como está é superficial.

3 Usando o Picat

Nesta seção é apresentada a instalação e o modo de uso da linguagem.

3.1 Como Instalar?

A linguagem Picat é multiplataforma, haja visto que o seu código primário de construção foi a linguagem C padrão. Logo, esta linguagem é disponibilizada para qualquer arquitetura de processador e sistema operacional.

Para instalar, basicamente faça download da versão desejada do site: `http://picat-lang.org/`, segue-se por uma extração no diretório onde se deseja instalar. O diretório criado é o `Picat/` dentro deste reside o executável `picat`, sua documentação em `.tex` e `.pdf`, módulos e exemplos.

3.2 Como Funciona?

Basicamente, o interpretador lê um código fonte com a extensão `.pi`, este é traduzido para um código intermediário pronto para ser executado. Para compilar para este código intermediário, extensão `.qi`, usa-se o comando `cl`, maiores detalhes ver [Zhou and Fruhman 2016b].

Basicamente há dois modos de usar o Picat, em modo *linha de comando* ou no modo *interativo*. Quanto ao interpretador é executado com o comando `picat` diretamente sobre um arquivo fonte, e este compila para um código intermediário e executá-o sobre a cláusula desejada. O *default* é uma chamada a regra `main` existindo no código-fonte.

No modo do *interativo*, o interpretador é executado em um ambiente próprio, compilando os programas que são carregados na memória, e disponibilizando-os a execução via esta console. Ambos os modos são apresentados a seguir.

3.3 Como Usar?

Modo Console: este é executado pelo interpretador do Picat seguido pelo nome do programa. O comando `picat` tem várias opções, como a escolha de um predicado ou uma função para inicializar a execução. O *default* é que programas nesta linguagem tenham um predicado/função `main` dentro deste programa, contudo, não é obrigatório. Veja o exemplo numa console *(Linux-like*, uma execução é dada como:

```
$ picat alo.pi
  HOJE EH SETXTA!!!
```

Cujo código-fonte do programa `alo.pi` é dado por:

```
main =>
  printf("  HOJE EH SETXTA!!!  ").
```

Modo Interativo: ou interpretado, o ambiente do Picat apresenta uma primeira linha com a expressão `Picat>` . Um comando `help` pode ser executado para ver as opções. Após a compilação e o carregamento na memória, nesta ordem, consultas podem ser feitas de modo interativo. Por exemplo:

```
Picat> X=1+2
X=3
Picat> printf("HOJE ." ++ ". EH ." ++ ". SEXTA")
HOJE .. EH .. SEXTA
```

Para carregar um arquivo com a extensão `.pi`, digite na console do Picat o comando `load(‘‘nome_do_arquivo’’)`, sem extensão, e execute a regra do programa desejada. Vale lembrar que o arquivo `.pi` precisa estar no mesmo diretório do interpretador, ou indicar o caminho do mesmo, via uma variável de ambiente.

Ao executar o comando `load` este vai compilar, se um código intermediário compilado não existir, e o carrega na memória principal. Ou seja, análogo ao comando `cl`. Por questões práticas o uso de `cl` é recomendado.

```
Picat> load(alo)
Compiling:: alo.pi
alo.pi compiled in 0 milliseconds
loading...
yes
Picat> main
  HOJE EH SETXTA!!!
yes
```

O código compilado não é visível no diretório corrente, pois o mesmo é residente na memória, como um *processo-filho* do interpretador. Sob este ambiente interpretado muito sobre a linguagem pode ser explorada. Por exemplo

```
Picat> X := 3 + 4,  write(X), nl.
7
X = 7
yes
```

Observa-se que esta interatividade com o programador, a torna atrativa em seu processo ensino-aprendizagem. Como teste, faça este mesmo exemplo na linguagem C ou C++ e verifique quantos elementos deveriam ser transmitidos ao aluno até que este realizasse a soma de dois números inteiros.

Precisa ser melhorado e estendido a idéia

4 Elementos da Linguagem

Nessa seção são apresentados os elementos da linguagem, iniciando com os tipos de dados (variáveis, átomos, listas, vetores, ..., etc) , ilustrando-os com o uso de exemplos .

4.1 Tipos de Dados

Picat é uma linguagem com tipagem dinâmica, ou seja, a verificação do tipo de dado ocorre durante a execução do programa. Qualquer elemento da linguagem do Picat é chamado de *termo* e ele se divide entre variáveis e valores (genericamente, conhecidos como *átomos*). Felizmente, toda a terminologia do Picat tem heranças do Prolog [Kowalski 1974], o que o torna uma ferramenta útil para disciplinas de lógica matemática, por exemplo. Um conhecimento elementar de lógica [Enderton 2001] e Prolog, obviamente, vão auxiliar significativamente no processo de aprendizagem do aluno.

Neste sentido, alguns experimentos com P.I.C.A.T. tem sido feito em trabalhos finais de disciplinas de lógica da UDESC, em que problemas dedução natural tem sido sistematicamente implementados com sucesso, ver [de Sá 2016].

Na tipagem dinâmica, o tipo de uma variável é determinado quando é instanciada a um valor. No Picat tanto a variável quanto o valor são termos e um valor pode ser primitivo ou composto. Um valor primitivo pode ser um número inteiro, real ou um átomo. Enquanto um valor composto se divide em listas estruturas, *strings*, vetores, mapas e conjuntos. Esta hierarquia dos tipos de dados em Picat é ilustrada na figura 1.

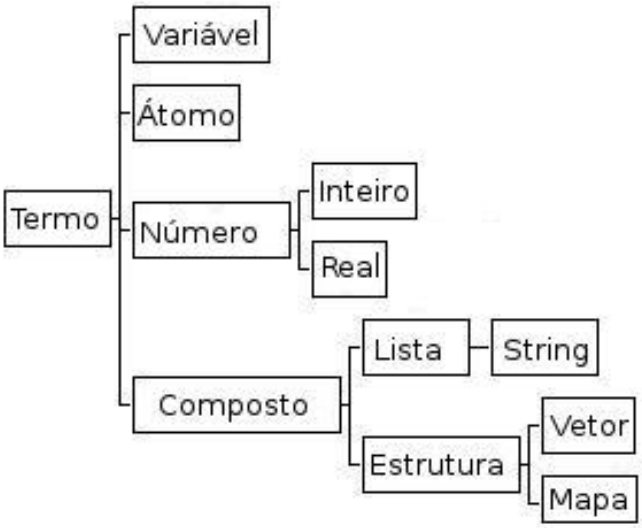

Figura 1: Hierarquia dos Tipos de dados

Quanto aos operadores, alguns deles são ilustrados na tabela 2 (ver página 9), estes são usados em operações aritméticas, lógicas e também para a execução do interpretador. Alguns operadores têm precedência maior do que outros [Zhou and Fruhman 2016b].

Tabela 2: Tabela de Operadores

Precedência	**Símbolo**	**Significado**
Alta	*. @*	teste padrão
	div, mod, rem	divisão, modulo, resto da divisão
	***	potenciação
	» «	conversores binários
	^	disjunção exclusiva (*Xor*)
	..	enumerador de conjunto
	++	concatenação
	not,once	negação, único
	&& ,	conjunção (*And*)
Baixa	*// ;*	disjunção (*Or*)

4.2 Variáveis

As variáveis em Picat são similares às variáveis das matemática, pois ambas guardam valores. Diferentemente das linguagens imperativas, as variáveis em Picat não possuem um endereço simbólico na memória do computador. Quando uma variável ainda não foi instanciada com um valor, ela fica em um estado **livre**. Uma vez quando for **instanciada** com um valor, ela terá a mesma identidade como se fosse um valor até que ela seja liberada de novo [Zhou and Fruhman 2016b].

O nome de uma variável é identificado por iniciar com uma letra maiúscula ou com *underline*, como demonstrado nos seguintes exemplos:
`X, Y, Xa, Y1, _a, _bc`
Algumas funções são usadas para trabalhar com as variáveis:

`var(Termo):` verifica se a variável é livre, se for retorna `true`.

`nonvar(Termo):` verifica se a variável não é livre, se for retorna `true`.

`attr_var(Termo):` verifica se a variável está instanciada, se for retorna `true`.

`dvar(Termo):` verifica se a variável instancia está dentro do domínio, se for retorna `true`.

4.3 Átomos

Um átomo é uma constante simbólica e seu nome pode ser representado tanto com aspas simples ou sem. Um átomo não pode ultrapassar uma linha de comando e seu nome tem um limite de mil caracteres. Eis alguns exemplos de átomos:
`x, x_1, 'a', 'b1'` [Zhou and Fruhman 2016b]. Algumas funções manipulam os átomos, tais como:

`atom(Termo):` verifica se o termo é um átomo.

`atom_chars(Termo):` retorna uma *string* contendo os caracteres do átomo, irá ocorrer um erro se a função não for um átomo.

`atom_codes(Termo):` retorna uma lista de códigos dos caracteres do átomo, irá ocorrer um erro se a função não for um átomo.

`char(Termo):` verifica se o átomo possui um único carácter, se for retorna `True`.

`digit(Termo):` verifica se o átomo possui um único digito, se for retorna `True`.

`len(Termo):` retorna o número de caracteres de um átomo.

4.4 Números

Um número é um átomo inteiro ou real. Um número inteiro pode ser representado na forma decimal, binária, octal ou hexadecimal. Já o número real usa o ponto (`.`) no lugar da virgula para separar os valores depois de zero como: `3.1415`. A tabela 3 mostra as operações aritméticas mais básicas a seguir, funções que trabalham com números e alguns exemplos [Zhou and Fruhman 2016b]:.

`number(Termo):` verifica se o termo é um número.

`float(Termo):` verifica se o termo é um número real.

`int(Termo):` verifica se o termo é um número inteiro.

`max(X, Y):` compara dois termos e retorna o maior deles.

`min(X, Y):` compara dois termos e retorna o menor deles.

```
Picat> A =  5, B = 7, number(A), number(B),
max(A, B) = Maximo, min(A, B) = Minimo.
A = 5
B = 7
Maximo = 7
Minimo = 5
yes
```

Tabela 3: Operadores Aritméticos

Formula	**Operação**
X + Y	*Adição*
X - Y	*Subtração*
*X * Y*	*Multiplicação*
X / Y	*Divisão*
X // Y	*Divisão Truncada*
X mod Y	*Resto da Divisão*

4.5 Termos Compostos

Um termo composto se divide entre listas, *strings*, estruturas e outros tipos compostos são derivados destes: vetores, mapas e conjuntos. Entretanto, ambos têm seus elementos acessados via casamento de padrões de fatos, predicados e funções. Os termos compostos são dados por:

Lista: A forma de uma lista reúne um conjunto de termos e os coloca dentro de colchetes: $[t_1, t_2, ..., t_n]$. Veja o exemplo:

```
Picat> A=[1,2,3], list(A), length(A)=L_A,  B= [4,5,6], list(B),
length(B) = L_B,  A ++ B = C, list(C), length(C) = L_C.
A = [1,2,3]
L_A = 3
B = [4,5,6]
L_B = 3
C = [1,2,3,4,5,6]
L_C = 6
yes
```

***Strings*:** Uma *string* pode ser representada em forma de lista, ou seja, cada carácter de uma *string* é um termo de uma lista. Por exemplo: a palavra `"carro"` pode ser expressa da forma `[c,a,r,r,o]`. Veja o exemplo:

```
Picat> X = "Isto eh uma", string(X), to_uppercase(X) = Y.
X = ['I',s,t,o,' ',e,h,' ',u,m,a]
Y = ['I','S','T','O',' ','E','H',' ','U','M','A']
yes
```

Estruturas: A forma de uma estrutura é definida como `$s`$(t_1,\ t_2,\ \ldots t_n)$, onde `s` é um átomo e $ é usado para diferenciar de uma função. Seus principais elementos são o nome da estrutura, que é o átomo que fica na frente, e a aridade (número de argumentos do predicado). Veja o exemplo:

```
Picat> N = $(1,2,3,4,5), struct(N), arity(N) = Aridade,
to_list(N) = Lista.
N = (1,2,3,4,5)
Aridade = 2
Lista = [1,(2,3,4,5)]
yes
```

Vetores: um vetor ou um *array* tem o formato de $\{t_1, t_2, ...t_n\}$, onde t_i é um termo desta estrutura de aridade n. Veja o exemplo:

```
Picat>  A = {a, b, c}, array(A), length(A) = Comprimento.
A = {a,b,c}
Comprimento = 3
yes
```

Mapas: Os mapas têm a mesma forma de uma estrutura, porém eles possuem um valor especial para ser usado como chave. Veja o exemplo:

```
Picat> new_map(3)=M, put(M,a,3), put(M,b,2), put(M,c,1),
Valor = get(M,b).
M = (map)[a = 3,b = 2,c = 1]
Valor = 2
yes
```

Conjuntos: Um conjunto é um mapa onde todos os elementos da estrutura estão associados a uma chave de valor não-numérico. Todas as funções existentes para os mapas, ver [Zhou and Fruhman 2016b], podem ser aqui utilizadas. Para criá-lo é necessário o comando `new_set`, veja o exemplo:

```
Picat>  new_set(2) = N, put(N,a), put(N,b), put(N,a),
size(N)=Cardinalidade.
N = (map)[a,b]
Cardinalidade = 2
yes
```

Outras funções estão disponíveis para o uso de conjuntos, mas para isto há um módulo para ser importado: `import ordset`. Estas funções podem ser vistas em [Zhou and Fruhman 2016b]. Praticamente, conjunto é um mapa não ordenado e sem a repetição de elementos.

5 Exemplos

Nesta seção são apresentadas questões práticas de uso e ilustrações com exemplos simples.

5.1 Atribuição, Igualdade e Unificação

Causa um pouco de confusão alguns novos conceitos da programação em lógica e que é inexistente em outras linguagens. A novidade do P.I.C.A.T. perante o Prolog é a *atribuição*, que embora a sua implementação não seja uma das mais eficientes [Zhou and Fruhman 2016b], deixa-a muito próxima as linguagens imperativas.

"`==`" este é o predicado de comparação e funciona conforme esperado.

"`:=`" esta é a atribuição e os `:` indica o lado esquerdo. Assim, `X := 77` indica que a variável `X` terá o conteúdo `77` após a execução desta atribuição. Muitas linguagens utilizam esta mesma notação. Para que uma variável do lado esquerdo receba um valor do lado direito, este deve estar instanciado. Isto é, uma variável não recebe uma outra variável condicionalmente, ai é o caso da unificação [Zhou and Fruhman 2016b]. Nos casos inaceitáveis em Prolog, como `X = X + 77`, com o uso da "`:=`"foi resolvido. Seja o exemplo:

```
Picat> X:=7, X := X + 7, X := X + 7.
X = 21
```

"`=`" este é o predicado de igualdade e funciona conforme as regras lógica da unificação. Este predicado é bidirecional, isto é $X = Y \Leftrightarrow Y = X$, e temos condições de sucesso em ambos os lados para que a unificação seja bem sucedida. Como regra geral, uma variável unifica com qualquer objeto [Zhou et al. 2015].

Em resumo, diferentemente do Prolog, a atribuição de valor há uma variável segue a programação imperativa, com uma notação próxima a linguagem Pascal (`'x:= ...'`). Para o caso uma atribuição de um *átomo* há uma variável, `'x= ...'` ou `'x:= ...'` funcionam de modo semelhante. Contudo, caso `'x= ...'` seja um parâmetro de regra (predicado ou função), `'x= ...'` deve ser usado, pois neste caso trata-se de *casamento de padrões* e não há sentido para o conceito de atribuição. Caso contrário, use `'x:=...'` tratando-se de atribuição convencional.

Aos novatos na área, a unificação e a atribuição tendem criar alguma confusão. Os exemplos neste texto procuram ilustrar estas diferenças.

5.2 Estruturas de Controle

As estruturas de controle são bem próximas a maioria das linguagens. Seguem um padrão próximo as linguagens mais usuais [Tate 2010, Scott 2000]. Basicamente P.I.C.A.T. segue um padrão tal como `if-then-else-end`, no caso de duas opções na condicional, e aninhamentos são permitidos com a introdução do `elseif` entre o `then` e o `else` do caso anterior.

Exemplo 1:

```
ex1 =>
   X:=3, Y:=4,
   if(X >= Y)
   then printf("%d", X)
   else printf("%d", Y)
   end.
```

Exemplo 2:

```
main => maior_3(5,4,3), maior_3(3,5,4), maior_3(3,4,5).
maior_3(X,Y,Z) =>
   if ( (X >= Y) && (X >= Z) )
   then printf("\nX eh o MAIOR %d", X)
   elseif ( (Y >= X)&&(Y >= Z) )
   then  printf("\nY eh o MAIOR %d", Y)
   else  printf("\nZ eh o MAIOR %d", Z)
   end.
```

5.3 Entradas e Saídas

Ao contrário de linguagens como Prolog e Haskell, os esquemas de entradas e saídas de dados seguem linguagens como Python, C, etc. Seja o exemplo de calcular a média entre 2 números:

```
main =>
  printf(" Digite dois números:  "),
  N_real_01 = read_real(),
  N_real_02 = read_real(),
  Media = (N_real_01+N_real_02)/2,
  printf(" A média é: %6.2f ", Media ),
  printf("\n ..... FIM ..... \n ").
```

Sua execução na console é dada por:

```
$picat media2.pi
 Digite dois números:  12.34       56.78
 A média é:  34.56
 ...... FIM .......
```

Todas as facilidades de entradas e saídas de linguagem C, bem como junto a chamadas ao sistema operacional para entradas e saídas, seguem as linguagens modernas como Python.

Faltam alguns bom exemplos de I/Os. A parte de entrada e saída sempre são problemáticas nas linguagens declarativas. Para saídas Picat é muito forte com muitas opcoes. Quanto a entrada de dados.... precisamos explorar alguns bons exemplos.

5.4 Laços de Repetição

Algumas estruturas de repetição estão prontas e seguem padrões da maioria das linguagens. A herança aqui veio de linguagens como Eclipse-CLP [Niederliński 214, Apt and Wallace 2007], um Prolog orientado à programação por restrições.

As estruturas mais usuais são: `foreach`, `while-end` e `do-while`, outras podem ser construídas a partir destas, ver [Zhou and Fruhman 2016b]. Os exemplos que seguem são equivalentes entre si, e calculam a soma dos `N` primeiros valores. Os códigos abaixo são para fins didáticos, bem como os demais acima, e não estão comprometidos com eficiência e concisão. Contudo, propositalmente, estas implementações são alternativas as funções apresentadas na sub-seção 5.6.

```
main => soma_01(7), soma_02(7),  soma_03(7).
soma_01(N) =>   %% USING foreach
S := 0,
foreach(Aux in 1 .. N)
  printf(" %d", Aux),
   S := S + Aux,
end,
printf("\n SOMA de 1 ate %d: %d\n", N, S).

soma_02(N) => %% USING do-while
S := 0,
Aux := N,
do
  printf(" %d", N),
   S := S + N,
   N := N - 1
while (N >= 0),
printf("\n SOMA de 1 ate %d: %d\n", Aux, S).

soma_03(N) => %% USING while-end
S := 0,
Aux := N,
while (N >= 0)
  printf(" %d", N),
```

```
   S := S + N,
   N := N - 1
end,
printf("\n SOMA de 1 ate %d: %d\n", Aux, S).
```

Uma execução em modo console para esta chamada de `main`, é dada por:

```
$ picat lacos_soma_N.pi
1 2 3 4 5 6 7
SOMA de 1 ate 7: 28
7 6 5 4 3 2 1 0
SOMA de 1 ate 7: 28
7 6 5 4 3 2 1 0
SOMA de 1 ate 7: 28
```

5.5 Listas e Vetores

Em um curso introdutório de programação o conteúdo de vetores é a estrutura de dados mais avançada neste nível. Como Picat tem nas listas um tratamento nativo, porém abstrato há um iniciante, em geral este tópico é reservado para o curso de estrutura de dados.

Assim, os vetores são estruturas mais rígidas e claras ao iniciante. Nesta direção, alguns exemplos de vetores são apresentados, deixando as listas para um segundo momento, porém a resolução segue a mesma idéia. Vamos há alguns exemplos clássicos:

Leitura de uma lista diretamenta na linha de comando: Esta é uma entrada *default*, a partir desta lista de termos, qualquer tratamento de tipagem pode ser feito. Falta incluir um exemplo didatico

Soma de todos elementos de um vetor de uma dimensão: `soma_vetor_1D =>`

```
Vetor := {1, 5 , 3, 7},
N := length(Vetor),
 S := 0,
foreach(Ind in 1 .. N)
        printf( " %d", Vetor[Ind] ),
        S := S + Vetor[Ind]
end,
printf("\n Soma de %w  eh:  %d\n" , Vetor, S).
```

Saída: `Soma de {1,5,3,7} eh:  16`

Soma de todos elementos de um vetor de duas dimensões: `soma_vetor_2D =>`

```
Vetor := {{1, 5 , 3, 7},
         {7, 3 , 1, 5}},
Linhas := length(Vetor),
Colunas := length(Vetor[1]),
 S := 0,
foreach(Ind_1 in 1 .. Linhas)
  foreach(Ind_2 in 1 .. Colunas)
          S := S + Vetor[Ind_1, Ind_2]
  end,
```

```
end,
printf("\n Soma  de %w  eh:  %d\n" , Vetor, S).
```

Saída: `Soma de {{1,5,3,7},{7,3,1,5}} eh:  32`

A soma para uma matriz tri-dimensional segue a mesma idéia, esta se encontra implementada em [de Sá 2016]. Outras implementações são possíveis, como a encontrada em [Zhou et al. 2015].

5.6 Funções e Predicados Recursivos

Da linhagem de linguagens como Prolog e Haskell [Tate 2010, Scott 2000], a recursividade é embutida tanto em regras (leia-se predicados) e funções. Aqui, Picat segue as linguagens como Prolog e Haskell na estrutura da recursão. Sejam os exemplos das funções e predicados abaixo, cujo objetivo é a soma dos N primeiros naturais.

```
% Soma como predicado
  soma_p(0,S) => S = 0.
  soma_p(N,S), N > 0 =>
                    soma_p(N-1,Parcial),
                    S = N + Parcial.

  % Soma como funcao
  soma_f1(0) = S => S = 0.
  soma_f1(N) = S, N >= 1 => S = N + soma_f1(N-1).

  % Soma como funcao de fatos -- Haskell
  soma_f2(0) = 0.
  soma_f2(N) = N + soma_f2( N-1 ).
```

A principal diferença para a linguagem Prolog, é o *backtracking* que é desabilitado como *default*. Assim, todas as funções acima são determinísticas, embora recursivas. Assim, o funcionamento é semelhante ao Haskell pelo *casamento de padrões*, *verificação de condicional* e *geração de listas*. A *geração de listas* (*list compreenhsion*) falta abordar.

Caso se queira uma computação não-determinística em algum conjunto de cláusulas; algo muito desejável em muitos casos; antes do símbolo de início de regra `=>`, adiciona-se o símbolo ‘‘?’’ ao `=>`. Assim, toda regra que apresentar a sequência `?=>` em sua construção, indica que a mesma pode ser acionada várias vezes sob instâncias de falhas. Leia-se que esta regra é *backtrackable* (leia-se em português:*backtrackável*), isto é, o mecanismo *backtracking* é acionado para regras subsequentes a esta. Assim, diferentemente do Prolog, o *backtracking* é controlável.

6 Tópicos Avançados

Nesta seção apresenta-se três tópicos avançados, os quais ilustram a facilidade com que problemas complexos podem ser resolvidos com P.I.C.A.T. . Estes tópicos, Programação por Restrições, Programação Dinâmica e Planejamento, definem áreas de pesquisas em que P.I.C.A.T. apresenta-se como uma ferramenta candidata a estas implementações. Alguns bons exemplos de problemas, em diversos graus de dificuldades são encontrados em [de Sá 2016, Kjellerstrand 2016, Zhou and Fruhman 2016a].

6.1 Programação por Restrições

A linguagem Picat apresenta uma sintaxe natural ao paradigma da Programação por Restrições [Niederliński 214, Apt and Wallace 2007, Russell and Norvig 2010]. Seja um problema clássico cripto-aritmético[1], tal como:

```
        I  P  A
  +   B  E  E  R
 -----------------
   P  I  C  A  T
```

onde o conjunto da variáveis {I,P,A,B,E,R,C,T} devem receber algum valor entre 0 e 9 **não repetidos**, tal que a soma aritmética acima seja satisfeita.

A solução deste problema é dado pelo seguinte código:

```
import cp.
main =>
    Vars = [I,P,A,B,E,R,C,T],
    Vai_UM = [C1, C2, C3, C4],
    Vars :: 0..9,
    Vai_UM :: 0..1,
    all_different( [I,P,A,B,E,R,C,T] ),
    A + R #= 10*C1 + T,
    P + E + C1 #= 10*C2 + A,
    I + E + C2 #= 10*C3 + C,
        B + C3 #= 10*C4 + I,
        C4 #= P,
    append([I,P,A,B,E,R,C,T], [C1,C2,C3,C4], X),
    solve([ff] , X),
    printf("\nI:%d P:%d A:%d B:%d E:%d R:%d C:%d T:%d ",
            I,P,A,B,E,R,C,T),
    printf("\nC1:%d C2:%d C3:%d C4:%d\n", C1,C2,C3,C4).
```

Uma (sim, há muitas outras) saída deste programa é dada por:

```
$picat puzzle_BEER.pi

I:3 P:0 A:9 B:2 E:8 R:5 C:1 T:4
C1:1 C2:0 C3:1 C4:0
```

Outras explorações sobre este tema e outros fundamentos são encontrados em [Zhou 2014, Zhou et al. 2015].

6.2 Programação Dinâmica

Esta linguagem tem suporte direto a técnica de *memoization* da Programação Dinâmica (PD) usando o predicado `table`. O predicado *table* antecedendo um conjunto de regras recursivas, indica que o mesmo tem uma estrutura de tabela para armazenar seus resultados intermediários.

[1]Detalhes destes tipos de problemas encontram-se em [Zhou et al. 2015], contudo, este exemplo bem como os demais aqui apresentados, são originais.

O exemplo que segue é um clássico da área de grafos e recursividade. Aqui, uma regra recursiva é usada para definição de caminho entre dois pontos `X` e `Y`. Um caminho, neste exemplo, é dado pela existência de uma estrada direta entre os dois pontos `X` e `Y`. Caso contrário, um caminho entre `X` e `Y` é definido pela regra recursiva, o qual é dada por uma estrada de `X` a `Z`, seguido por um caminho de `Z` a `Y`.

Este código é dado por:

```
index  (-,-,-) %% index: sem ponto no final -- fatos NAO ordenados
estrada(itajai , ilhota , 20).
estrada(ilhota , gaspar , 10).
estrada(gaspar , brusque , 30).
estrada(itajai , brusque , 50).
estrada(itajai , navegantes , 10).
estrada(navegantes , gaspar , 30).
%% REGRA DA PD
table           %% table: sem ponto no final
caminho(X,Y,D) ?=> estrada(X,Y,D).   %%% esta regra eh BACKTRACKABLE
caminho(X,Y,D) => caminho(X,Z,D1),
                  estrada(Z,Y,D2),
                  D = D1 + D2.
```

O predicado `index` apresenta uma base de fatos dinâmica; quanto ao sinal - como argumento, indica que estes valores não estão indexados por nenhum argumento. A execução do mesmo para uma saída ilustrar os caminhos que chegam a cidade de Brusque é dado por:

```
Picat> caminho( C, brusque, D),
printf("\n Cidades: %w  Distancia: %d", C, D), fail.
 Cidades: gaspar  Distancia: 30
 Cidades: itajai  Distancia: 50
 Cidades: ilhota  Distancia: 40
 Cidades: navegantes  Distancia: 60
 Cidades: itajai  Distancia: 60
 Cidades: itajai  Distancia: 70
no
Picat>
```

6.3 Planejamento

Todos os códigos apresentados nesta seção se encontram em [de Sá 2016]. Outros códigos do uso desta linguagens em problemas da área de satisfação por restrições podem se encontrados em `http://www.hakank.org/picat/`.

O exemplo de planejamento aqui ilustrado é uma variação de um *jogo da amarelinha* (falta descrever), ver código 1. Falta explicar o jogo ....

```
1 /*************************************************************/
2 /*
3       X2          X5
4    /     \    /     \
5  X1       X4         X7
```

```
6       \     /     \     /
7         X3          X6
8   */
9   /*************************************************************/
10  import datetime.
11  import planner.
12
13  /* pontos iniciais e finais do problema */
14  index(-)
15  pt_origem(  [ b , a , c , o , o , o , o ]  ).
16
17  index(-)
18  pt_final( [o , o, o, o, c, a, b] ).
19
20  %% for the planner
21  final( [o, o, o, o, c, a, b] ) => true .
22
23
24  /* movimentos permitidos  ==> veja o diagrama*/
25  % $ picat amarelinha_planner.pi
26  %[ 'X4 <-> X2', 'X1 <-> X2', 'X4 <-> X6', 'X4 <-> X2', 'X6 <-> X7', 'X4 <->
        X6', 'X4 <-> X3', 'X4 <-> X5']
27
28  /* Describing the possible actions ==> for the planner */
29  action([ X1,X2,X3,X4,X5,X6,X7 ], S1, Action, Action_Cost ) ?=>
30      Action_Cost = 1,
31      Action = 'X4 <-> X2', %% a description
32      ((X2 == o, X4 !=o) ;  (X4 == o, X2 != o)),  %% conditions
33      S1 = [ X1,X4,X3,X2,X5,X6,X7 ].    %% results
34
35  action( [ X1,X2,X3,X4,X5,X6,X7 ] , S1, Action, Action_Cost ) ?=>
36      Action_Cost = 1,
37      Action = 'X4 <-> X3',
38      ((X3 == o, X4 !=o) ;  (X4 == o, X3 != o)),
39      S1 =  [ X1,X2,X4,X3,X5,X6,X7 ] .
40
41  action( [ X1,X2,X3,X4,X5,X6,X7 ] , S1, Action, Action_Cost ) ?=>
42      Action_Cost = 1,
43      Action = 'X1 <-> X2',
44          (X2 == o, X1 !=o),
45      %%%  (X1 == o, X2 != o)),
46      S1 = [ X2,X1,X3,X4,X5,X6,X7 ] .
47
48  action( [ X1,X2,X3,X4,X5,X6,X7 ], S1, Action, Action_Cost ) ?=>
49      Action_Cost = 1,
50      Action = 'X3 <-> X1',
51          X3 == o, (X1 != o),
52          S1 = [ X3,X2,X1,X4,X5,X6,X7 ].
53
54  action( [ X1,X2,X3,X4,X5,X6,X7 ],  S1, Action, Action_Cost ) ?=>
```

```
55      Action_Cost = 1,
56      Action = 'X4 <-> X6',
57      ((X6 == o, X4 !=o) ;  (X4 == o, X6 != o)),
58      S1 =  [ X1,X2,X3,X6,X5,X4,X7 ] .
59
60  action( [ X1,X2,X3,X4,X5,X6,X7 ] , S1, Action, Action_Cost ) ?=>
61      Action_Cost = 1,
62      Action = 'X4 <-> X5',
63      ((X5 == o, X4 !=o) ;  (X4 == o, X5 != o)),
64          S1 =  [ X1,X2,X3,X5,X4,X6,X7 ].
65
66  action( [ X1,X2,X3,X4,X5,X6,X7 ], S1, Action, Action_Cost ) ?=>
67      Action_Cost = 1,
68      Action = 'X5 <-> X7',
69      ((X5 == o, X7 !=o) ;  (X7 == o, X5 != o)),
70      S1 =  [ X1,X2,X3,X4,X7,X6,X5] .
71
72  action( [ X1,X2,X3,X4,X5,X6,X7 ]  , S1, Action, Action_Cost ) =>
73      Action_Cost = 1,
74      Action = 'X6 <-> X7',
75          ((X6 == o, X7 !=o) ;  (X7 == o, X6 != o)),
76           S1 =  [X1,X2,X3,X4,X5,X7,X6].
77
78  %%%%%%%%%%%%%%%% the main call %%%%%%%%%
79  main  ?=>
80      pt_origem(X),     %%%write([X]),
81      pt_final(Y),
82      T1 = current_time(),
83      time(best_plan_unbounded( X , Solucao)), %%% CPU TIME
84      T2 = current_time(),
85
86      write(Solucao), nl,
87      write_L(Solucao),
88
89      Total := length(Solucao) ,
90      Num_Movts := (Total -1) ,
91     % T_CPU :=T2-T1, %%% think in something better
92      printf("\n Initial Position (state): %w ", X),
93      printf("\n Final  Position (state): %w", Y),
94      printf("\n Total of states: %d", Total),
95      printf("\n Total of moviments: %w", Num_Movts),
96      printf("\n CPU TIME INIC: %w  FIM %w",      T1, T2),
97      printf(" \n ==========\n ")
98      .
99
100 main => nl, write('No solution in the model  ... Houston we have a
        trouble!!!!') .
101
102 /*********************************************************/
103  write_L([])  ?=> true.
```

```
104  write_L([X|L]) =>   writeln( X ),   write_L(L).
105  /* ********************************************************** */
```

Listing 1: Jogo da Amarelinha

7 Conclusões

Falta revisar e incompleta, pois falta conectar a questão do ensino em informática

A linguagem Picat é jovem com menos de 3 anos, porém ela oferece um avanço significativo desde seu principal antecessor: o Prolog [Scott 2000]. A linguagem Prolog já existe há mais de 40 anos e tem sido largamente utilizada na indústria e academia. Após o Prolog, muitas outras linguagens se inspiraram em seus conceitos tais como: *backtracking*, casamento de padrões, o não-determinismo, a terminologia da lógica de primeira-ordem, etc. Algumas seguiram o Prolog em sua sintaxe e no paradigma lógico, exemplos : Goedel, Answer Set Programming (ASP), Datalog, etc. Outras estenderam sua sintaxe, especificamente com estruturas de repetição, e aperfeiçoaram seu mecanismo de busca para diversas finalidades, tais como: Eclipse-CLP, Sictus-Prolog, Prolog-III, etc. Outras ainda se inspiraram em Prolog sob paradigmas não-lógicos, por exemplo as linguagens funcionais como: Erlang, Ciao, Clojure, Curry, etc. Finalmente, com algumas características multi-paradigma como o Picat tem-se: Mercury , Ciao e Oz [Sestoft 2012, Sebesta 2003, Tate 2010, Editors 2016b, Editors 2016a].

Cada uma destas linguagens tiveram seus objetivos com propósitos definidos. Quanto a Picat é de propósito geral, tipagem dinâmica, eficiente, portável e multiparadigma quanto a sua sintaxe [Zhou et al. 2015, Wikipedia 2016]. Alguns pontos podem ser destacados:

- Enfatiza uma visão moderna e controlável quanto ao seu mecanismo de *backtracking*, tornando-o mais legível, motivado pela clareza de construir regras declarativas;
- As funções também podem ser disponibilizadas em módulos, com uma sintaxe semelhante a Haskell e sob um ambiente de programação análogo ao Python;

Contudo, Picat possui alguns pontos-negativos:

- Manteve as letras maiúsculas para variáveis como feito no B-Prolog. Esta é uma herança do Prolog que herdou seus conceitos básicos da lógica de primeira-ordem. Logo, não há muito como evoluir nesta direção pesquisa;
- A geração de um código executável ainda não é pura. A geração de código puro *stand-alone* ainda se encontra em desenvolvimento. Há uma compilação para um código intermediário, e este é muito rápido ao ser executado sobre a máquina virtual do Picat. Uma característica análoga a Java e outras linguagens;
- A comunidade de usuários é incipiente, e principalmente de mantenedores da linguagem.

Picat demonstra um potencial de uso, possuindo uma variedade de características dos diversos paradigmas de programação. Enfim, Picat é uma linguagem nova, revolucionária, com um futuro promissor para as áreas de pesquisas e de usos comercial diversos [Zhou et al. 2015].

Um blog sempre atualizado é mantido por Håkan Kjellerstrand em [Kjellerstrand 2016], além da página da linguagem em `http://picat-lang.org` com um fórum de discussões bem ativo [Zhou and Fruhman 2016a].

Referências

[Aguiar 2016] Aguiar, M. (2016). Tipos de paradigmas de programação. Internet. http://tiserviceprovider.blogspot.com.br/2010/11/tipos-de-paradigmas-de-programacao-mais.html, acessada: 14 de junho de 2016.

[Apt and Wallace 2007] Apt, K. R. and Wallace, M. (2007). *Constraint logic programming using Eclipse*. Cambridge University Press.

[de Sá 2016] de Sá, C. C. (2016). Repositório de programas em picat. Internet. https://github.com/claudiosa/CCS/tree/master/picat, acessada: 14 de junho de 2016.

[Editors 2016a] Editors, V. (2016a). Encyclopedia of programming languages. Internet. http://progopedia.com, acessada: 14 de junho de 2016.

[Editors 2016b] Editors, V. (2016b). Programming languages – wikipedia. Internet. https://en.wikipedia.org/wiki/Programming_language, acessada: 14 de junho de 2016.

[Enderton 2001] Enderton, H. (2001). *A Mathematical Introduction to Logic*. Harcourt/Academic Press.

[Kjellerstrand 2016] Kjellerstrand, H. (2016). My picat page. Internet. http://www.hakank.org/picat/, acessada: 14 de junho de 2016.

[Kowalski 1974] Kowalski, R. A. (1974). Predicate logic as programming language. In *IFIP Congress'74*, pages 569–574.

[Niederliński 214] Niederliński, A. (214). *A Gentle Guide to Constraint Logic Programming via ECLiPSe*. Jacek Skalmierski Computer Studio.

[Rosettacode 2016] Rosettacode (2016). Language comparison table. Internet. https://rosettacode.org/wiki/Language_Comparison_Table, acessada: 14 de junho de 2016.

[Russell and Norvig 2010] Russell, S. J. and Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

[Scott 2000] Scott, M. L. (2000). *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Sebesta 2003] Sebesta, R. (2003). *Conceitos de Linguagens de Programacao*. BOOKMAN COMPANHIA ED.

[Sestoft 2012] Sestoft, P. (2012). *Programming Language Concepts*. Undergraduate Topics in Computer Science. Springer London.

[Suh 2011] Suh, E. (2011). The tower of babel – a comparison programming languages. Internet. http://www.cprogramming.com/langs.html, acessada: 14 de junho de 2016.

[Tate 2010] Tate, B. A. (2010). *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition.

[Wikipedia 2016] Wikipedia (2016). Comparison of multi-paradigm programming languages. Internet. `https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages`, acessada: 14 de junho de 2016.

[Zhou 2014] Zhou, N. (2014). Combinatorial search with picat. *CoRR*, abs/1405.2538.

[Zhou and Fruhman 2016a] Zhou, N. and Fruhman, J. (2016a). Official site of picat. Internet. `http://picat-lang.org/`, acessada: 14 de junho de 2016.

[Zhou and Fruhman 2016b] Zhou, N. and Fruhman, J. (2016b). A user's guide to picat. Internet. `http://picat-lang.org/download/picat_guide.pdf`, acessada: 14 de junho de 2016.

[Zhou et al. 2015] Zhou, N., Kjellerstrand, H., and Fruhman, J. (2015). *Constraint Solving and Planning with Picat*. Springer Briefs in Intelligent Systems. Springer.