

Guia Muito Resumido de PICAT

Tudo¹ de PICAT que você precisa saber para o
Trabalho Final de LMA0001

Miguel Alfredo Nunes

Atualizado² para a versão 2.4 de PICAT

¹ Pode não conter tudo que você precisa saber

² ***Provavelmente*** este guia ficara atualizado para todas as versões 2.x, caso não ocorram mudanças muito drásticas na estrutura linguagem, podendo até ficar atualizado para as versões 3.x.

Sumário

0.	Motivação	3
1.	Introdução	4
1.1	Guia de Instalação	4
1.1.1	Windows.....	5
1.1.2	Ubuntu.....	7
1.2	Como Usar o Sistema do Picat.....	15
1.2.1	Compilação.....	15
1.2.2	Depuração.....	16
1.2.3	Modo Interativo	19
2.	Variáveis e Aritmética.....	21
2.1	Declaração	21
2.2	Particularização e Atribuição	23
2.3	Aritmética e Operadores	25
2.4	Átomos.....	27
2.5	Listas.....	27
2.5.1	Strings	28
2.6	Tuplas	29
3.	Predicados, Funções e index.....	30
3.1	Predicados	30
3.2	Funções	32
3.2.1	Funções de Saída de Dados	33
3.3	index	35
4.	Backtracking e Recursão	37
4.1	Backtracking	37
4.2	Recursão	40
5.	Estrutura de um Programa	42
5.1	Avaliações Lógicas.....	42
5.1.1	Tautologia, Contradição, Conjunção e Disjunção	42
5.2	Corpo de um Programa	43
5.3	Exemplos de Trabalhos Antigos.....	44
5.3.1	Resolução do Exercício	45
6.	Referências.....	54

0. Motivação

Posso dizer que tem dois motivos principais de eu estar montando este manual, o primeiro seria pois eu fui um dos poucos (senão único) que se interessou nesse assunto que muitos outros não dão muita atenção, e tem muitas dificuldades entendendo, por causa dessa dificuldade comum a quase todos eu decidi tentar ajudar da melhor maneira que eu posso.

O segundo foi a falta de algum material assim que fosse acessível a todos, digo isso pois a documentação oficial da linguagem assim como a grande maioria dos recursos similares disponíveis não só estão disponíveis somente em inglês, mas também são muito amplos e técnicos para os intuitos deste trabalho ou simplesmente não abordam quantas coisas que vocês precisariam saber.

Também deveria mencionar que o *Cacic*³ teve uma importante participação na montagem deste guia, dando importantes sugestões, e muito feedback enquanto eu escrevia, que ajudou imensamente a melhorar o guia e a acelerar a escrita dele.

Finalmente, devo agradecer o Professor Cláudio por me dar a oportunidade de poder me aprofundar mais nesse assunto que tanto gostei.

³ Centro Acadêmico da Ciência da Computação, ou simplesmente, o CA da Computação

1. Introdução

Picat é uma linguagem de paradigmas lógico, declarativo e funcional, ou seja, praticamente o oposto polar do que a linguagem C é, o que a torna ideal para resolver problemas computacionais mais simbólicos sem se tornar inútil para aplicações mais “cotidianas”, como *scripting*, por exemplo.

Neste tópico farei uma breve explicação das técnicas da linguagem, como instalá-la e como compilar/depurar programas e utilizar o sistema do Picat.





1.1 Guia de Instalação

Todos os arquivos referentes à linguagem estão no seu site; partindo daqui baixe o arquivo compatível com seu sistema operacional; explicarei como instalar em Windows e Ubuntu.

Aproveitando que ainda está no site, clique na aba *Resources* e baixe os dois primeiros arquivos *.pdf* que são apresentados, respectivamente, *Getting Started* e *User's Guide*, estes arquivos (o segundo mais que o primeiro) são praticamente toda a documentação da linguagem, assim como minha maior referência na escrita deste guia, eles serão de grande ajuda para vocês.

Note também que há dois vídeos feitos pelo professor Cláudio, são uma boa referência, contém resolução de exercícios e explicações e, acima de tudo, estão em português.

Observação: As fotos no tutorial de instalação são todas da versão 2.3, que não é a versão mais recente, porém ainda é válido para a versão mais atual.

 Download Modules Projects Resources

Download

Version 2.4 ([Get Started With Picat](#))

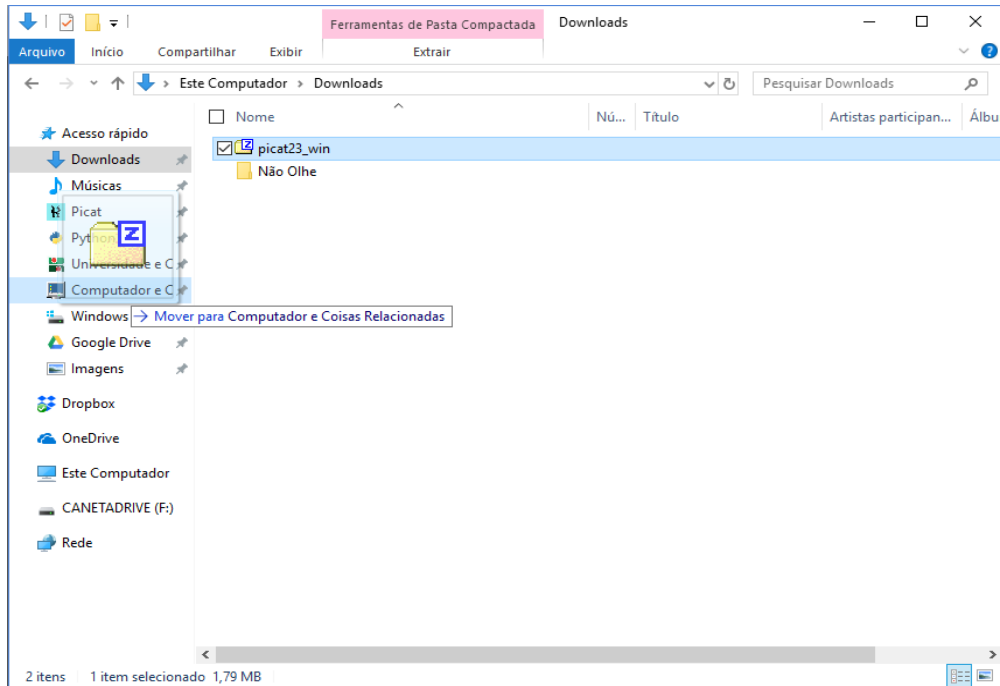
picat24_win.zip	MS Windows (64-bit)
picat24_cygwin64.tar.gz	Cygwin 64-bit
picat24_linux64.tar.gz	Linux 64-bit
picat24_macx.tar.gz	MacOS X (64-bit)
picat24_src.tar.gz	C source code

Foto tirada dia 16/04/2018 as 20:40

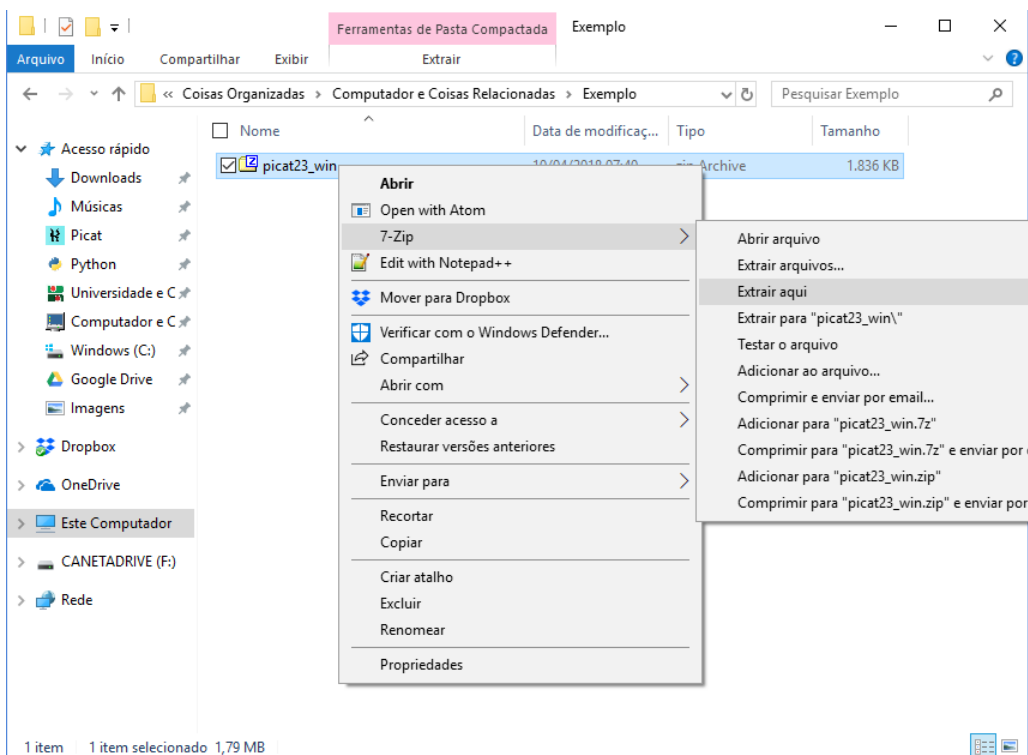
1.1.1 Windows

No Windows, há um método para instalar o sistema do Picat, que é bastante simples, mas você terá que compilar seus programas a partir do próprio console do Windows.

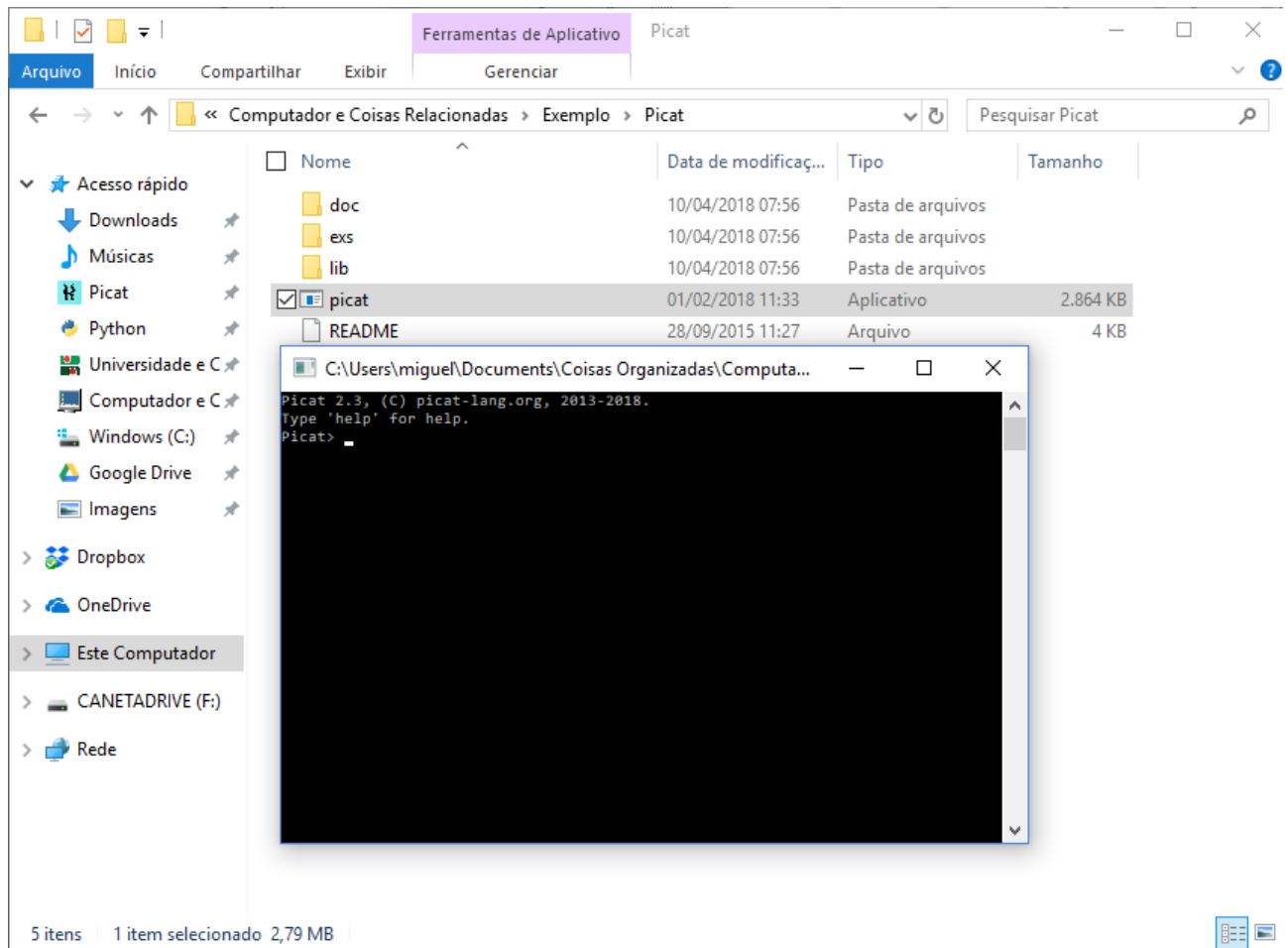
- Primeiramente, baixe o arquivo `.zip` do Picat e mova-o para a pasta em que deseja instalá-lo.



- Agora abra esta pasta e extraia o arquivo nela.



- Agora abra a pasta que foi extraída e execute o arquivo *picat.exe*.



- Agora pode acessar o terminal do Picat a partir deste executável. Como utilizar o sistema do Picat será explicado neste tópico.
- Obs.: Se quiser, você pode criar um atalho para este executável, o que permitirá que você abra o terminal do Picat sem ter que navegar para esta pasta.

1.1.2 Ubuntu

Há dois métodos para instalar em Ubuntu, em um (o mais simples) para você utilizar o sistema do Picat terá que abrir o executável manualmente e terá que compilar seus programas “manualmente”, do mesmo modo que seria feito no Windows; o outro método (o mais complicado) permitirá você acessar o sistema do Picat a partir do terminal, como você poderia fazer com outros comandos do Linux/Ubuntu.

Vou assumir que tens um mínimo entendimento dos comandos do Ubuntu caso queira instalar pelo terminal.

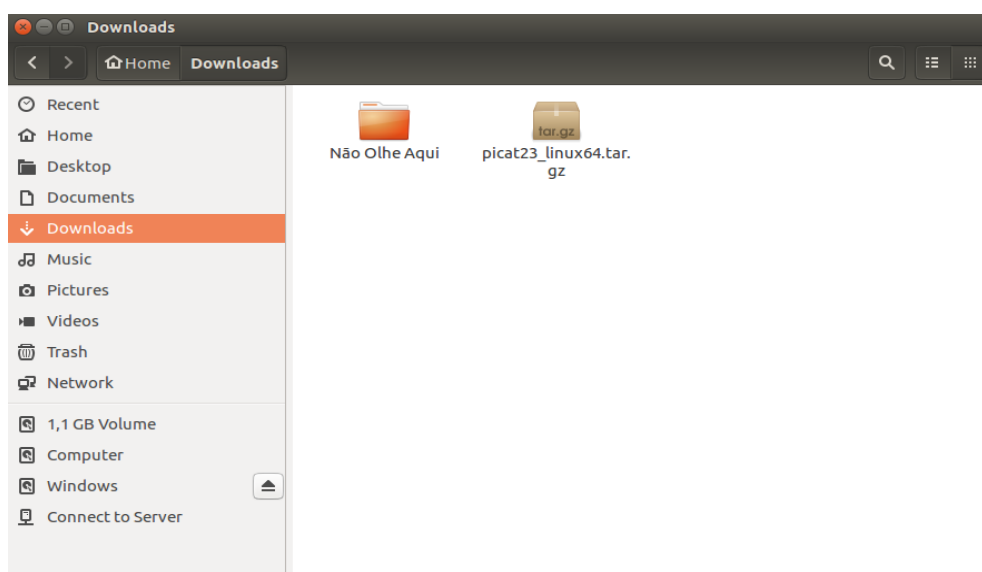
Vão reparar nas fotos que meu terminal é um pouco diferente do terminal padrão, mas a ideia é igual para qualquer computador.

Dica Importante: ao começar a digitar o nome de um arquivo longo no terminal aperte *Tab* para completar automaticamente o nome deste arquivo; caso houverem várias possibilidades para completar o nome, estas serão exibidas e terá que digitar o nome do arquivo que quiser até eles se diferenciarem.

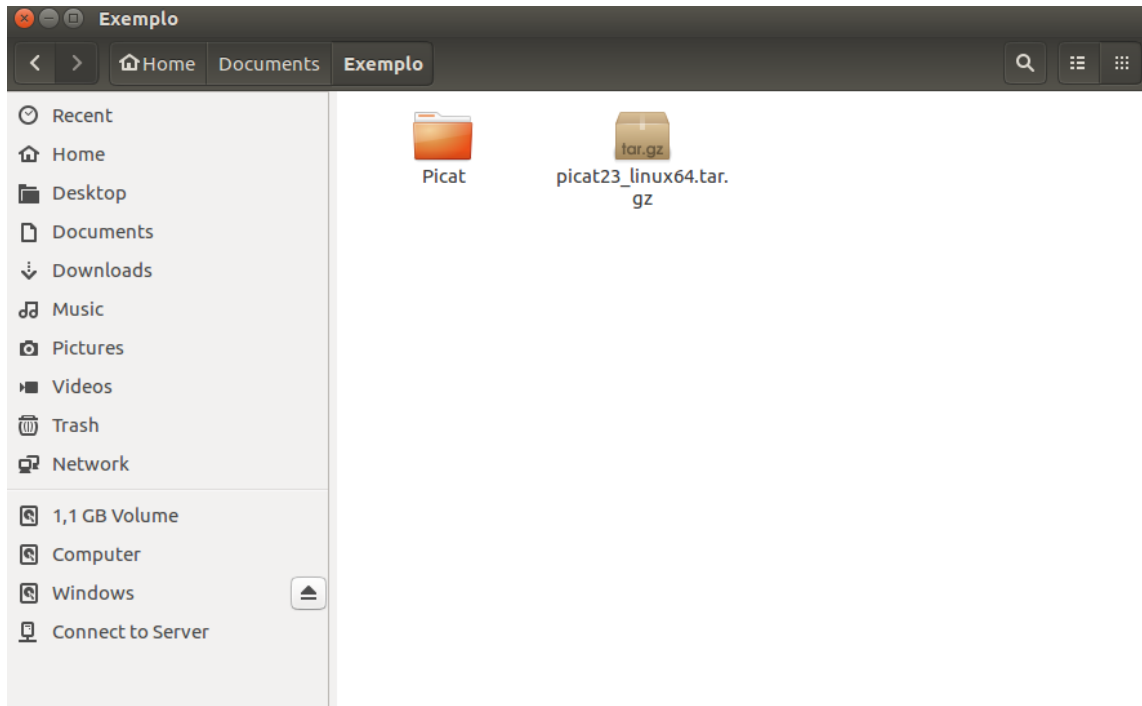
1. Instalando usando o Explorador de Arquivos:

1.1.Primeiro Método:

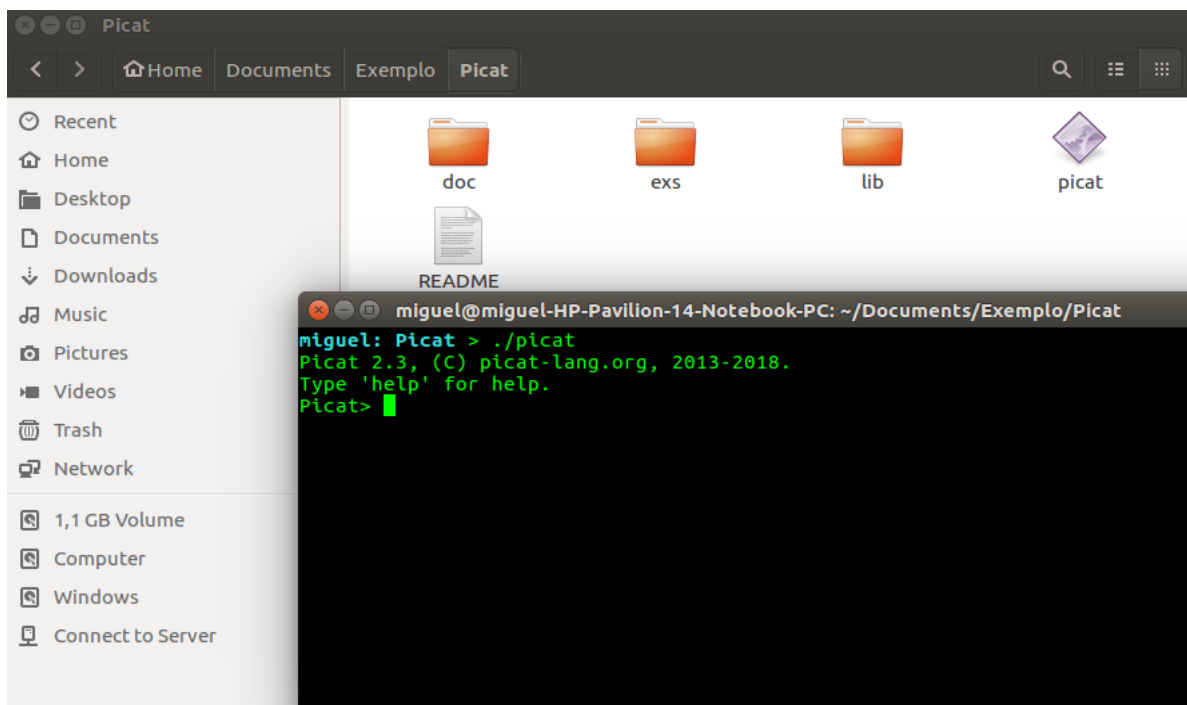
1.1.1. Baixe o arquivo *tar.gz* e mova-o (clique c/ o botão direito sobre o arquivo → *Move to../Mover para..*) para o diretório em que desejar instalá-lo.



1.1.2. Extraia o arquivo neste diretório (clique c/ o botão direito sobre o arquivo → *Extract Here/Extrair Aqui*) e abra o diretório “Picat” que foi extraído.

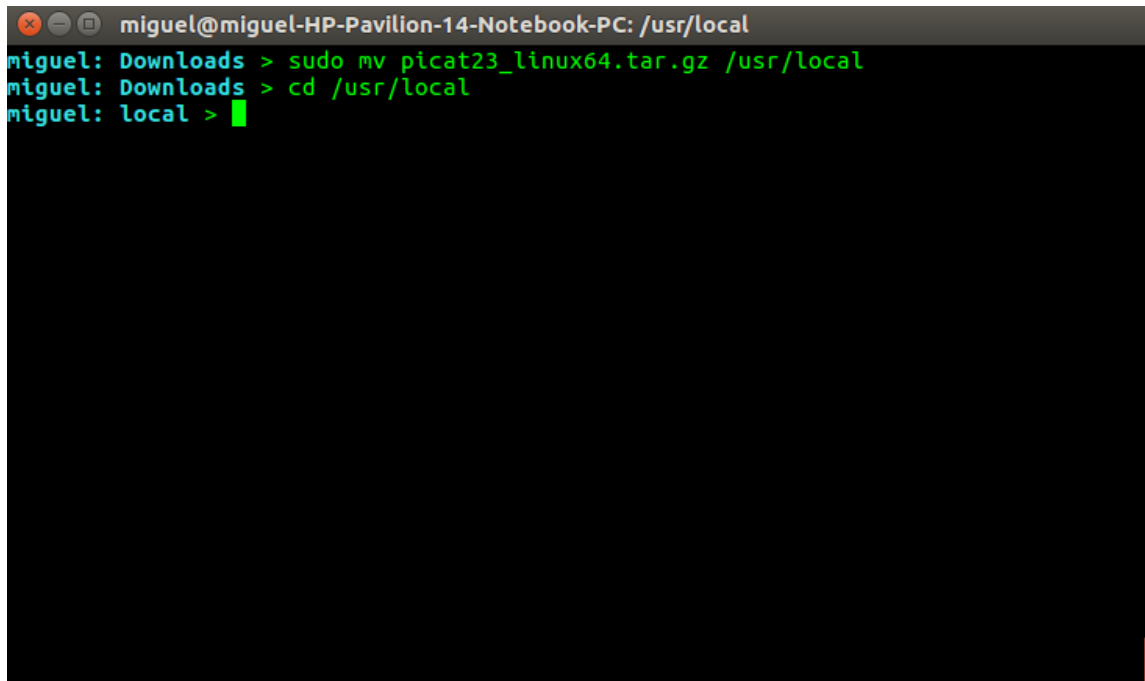


1.1.3. Abra o terminal neste diretório (clique c/ o botão direito → *Open in Terminal/Abrir no Terminal*) e digite o comando “./picat”, isto abrirá o Picat no terminal; como utilizar o console do Picat será abordado neste tópico.



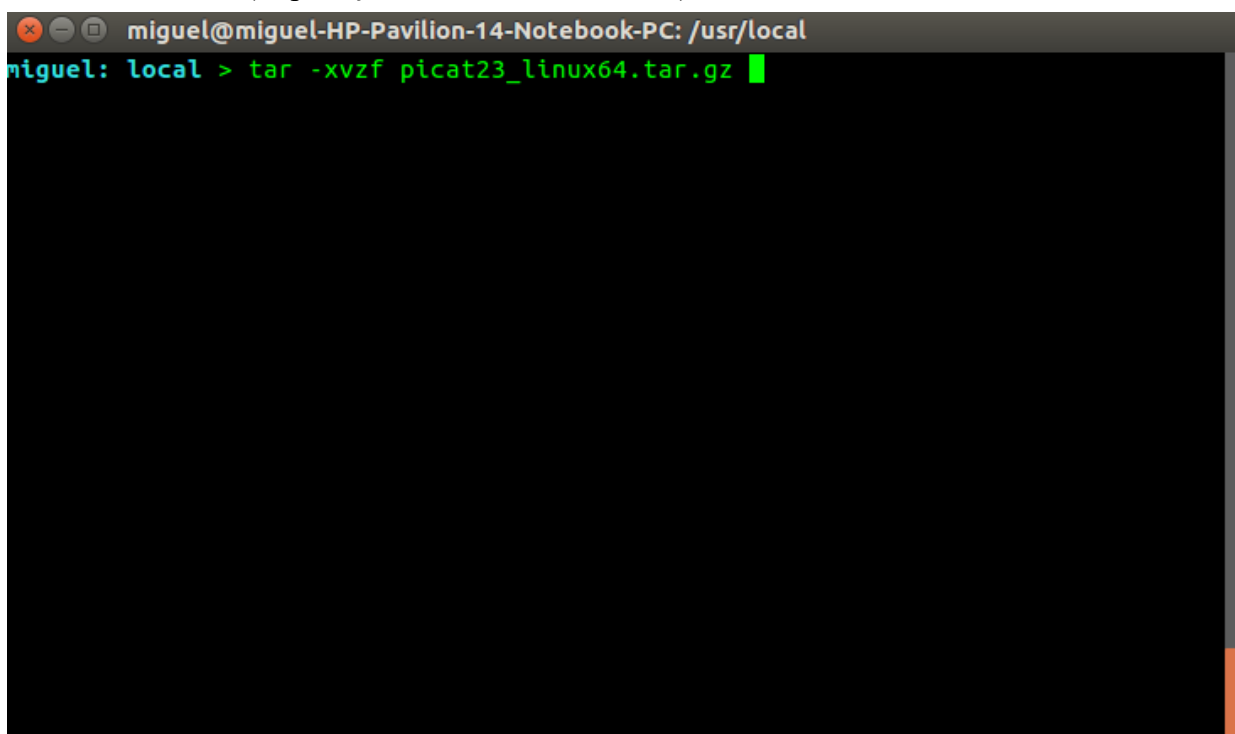
1.2.Segundo Método:

- 1.2.1. Baixe o arquivo *tar.gz*, abra o terminal (clique c/ o botão direito → *Open in Terminal/Abrir no Terminal*) e digite os comandos na imagem para mover o arquivo para a pasta */usr/local* e abrir este diretório no terminal.

A terminal window titled 'miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local'. The prompt is 'miguel: Downloads >'. The user enters 'sudo mv picat23_linux64.tar.gz /usr/local' and presses enter. The prompt changes to 'miguel: Downloads >'. The user enters 'cd /usr/local' and presses enter. The prompt changes to 'miguel: local >'.

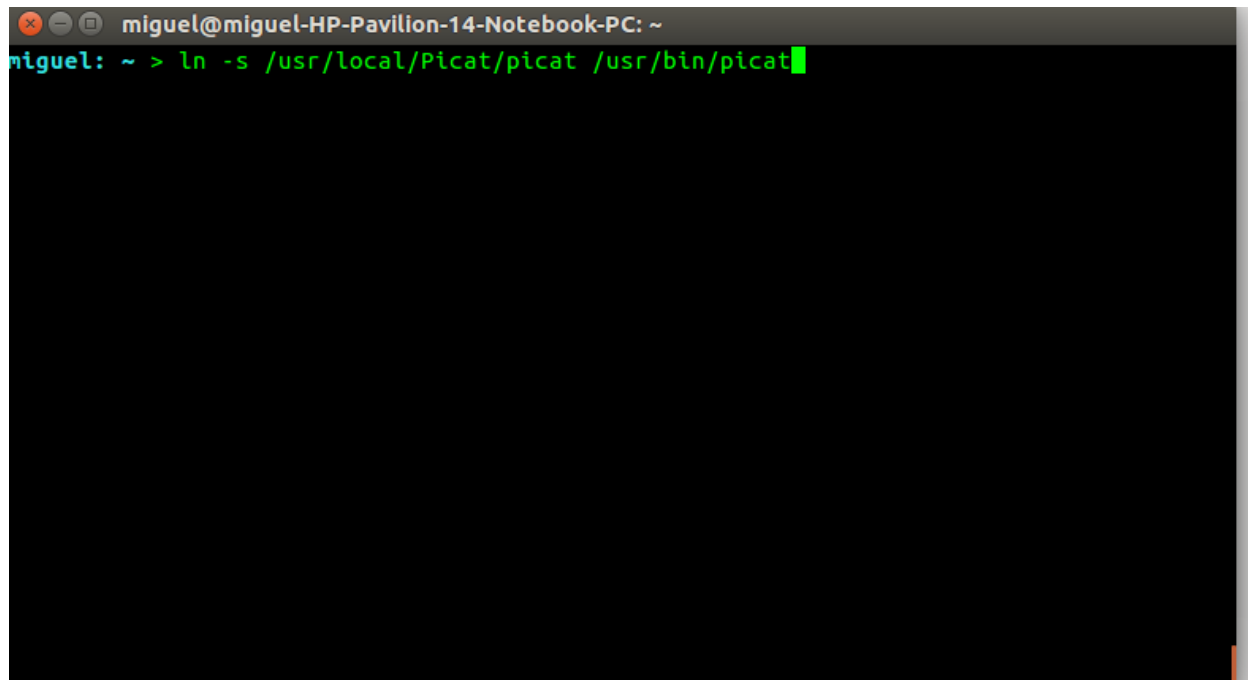
```
miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local
miguel: Downloads > sudo mv picat23_linux64.tar.gz /usr/local
miguel: Downloads > cd /usr/local
miguel: local > █
```

- 1.2.2. Agora ainda no terminal digite o seguinte comando para descompactar o arquivo e abra o diretório que foi extraído **OU** clique c/ o botão direito sobre o arquivo → *Extract Here/Extrair Aqui* e abra o diretório que foi extraído. (Explicação do comando tar -xvzf)

A terminal window titled 'miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local'. The prompt is 'miguel: local >'. The user enters 'tar -xvzf picat23_linux64.tar.gz' and presses enter. The prompt changes to 'miguel: local >'.

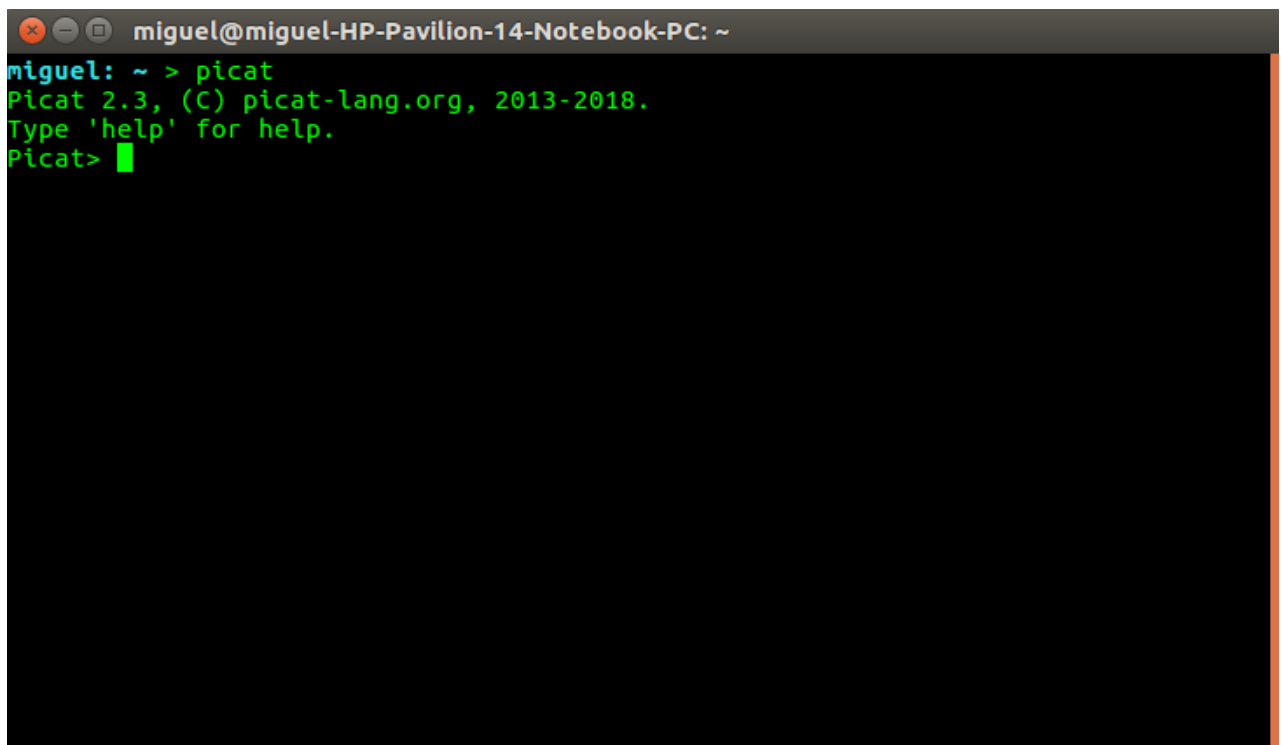
```
miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local
miguel: local > tar -xvzf picat23_linux64.tar.gz █
```

1.2.3. Agora abra o terminal neste diretório e execute o seguinte comando:



```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~  
miguel: ~ > ln -s /usr/local/Picat/picat /usr/bin/picat
```

1.2.4. Isso criará um link simbólico e permitirá que você acesse o sistema do Picat pelo terminal, sem ter que abrir diretamente o executável do Picat; como utilizar o sistema do Picat será abordado neste tópico.

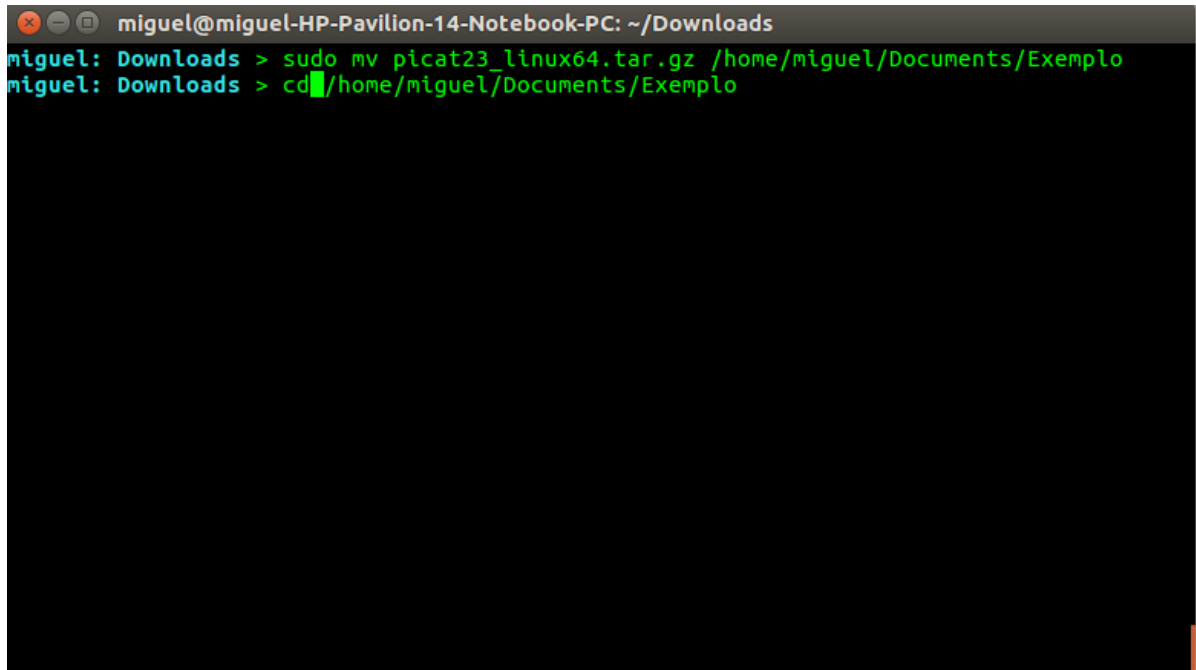


```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~  
miguel: ~ > picat  
Picat 2.3, (C) picat-lang.org, 2013-2018.  
Type 'help' for help.  
Picat>
```

2. Instalando usando o Terminal:

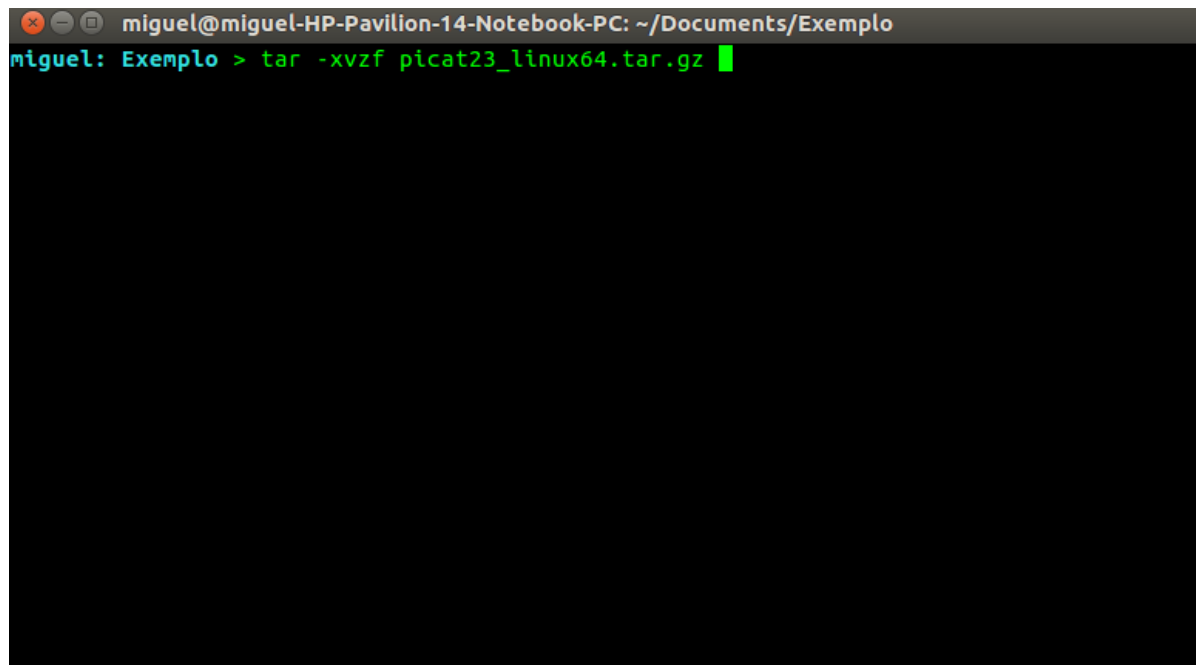
2.1.Primeiro Método:

2.1.1. Baixe o arquivo *tar.gz*, e mova-o para o diretório que desejar instalá-lo:

A terminal window with a dark background and light green text. The title bar at the top reads 'miguel@miguel-HP-Pavilion-14-Notebook-PC: ~/Downloads'. The first command entered is 'sudo mv picat23_linux64.tar.gz /home/miguel/Documents/Exemplo', and the second command is 'cd /home/miguel/Documents/Exemplo'. Both commands are followed by a green cursor.

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~/Downloads
miguel: Downloads > sudo mv picat23_linux64.tar.gz /home/miguel/Documents/Exemplo
miguel: Downloads > cd /home/miguel/Documents/Exemplo
```

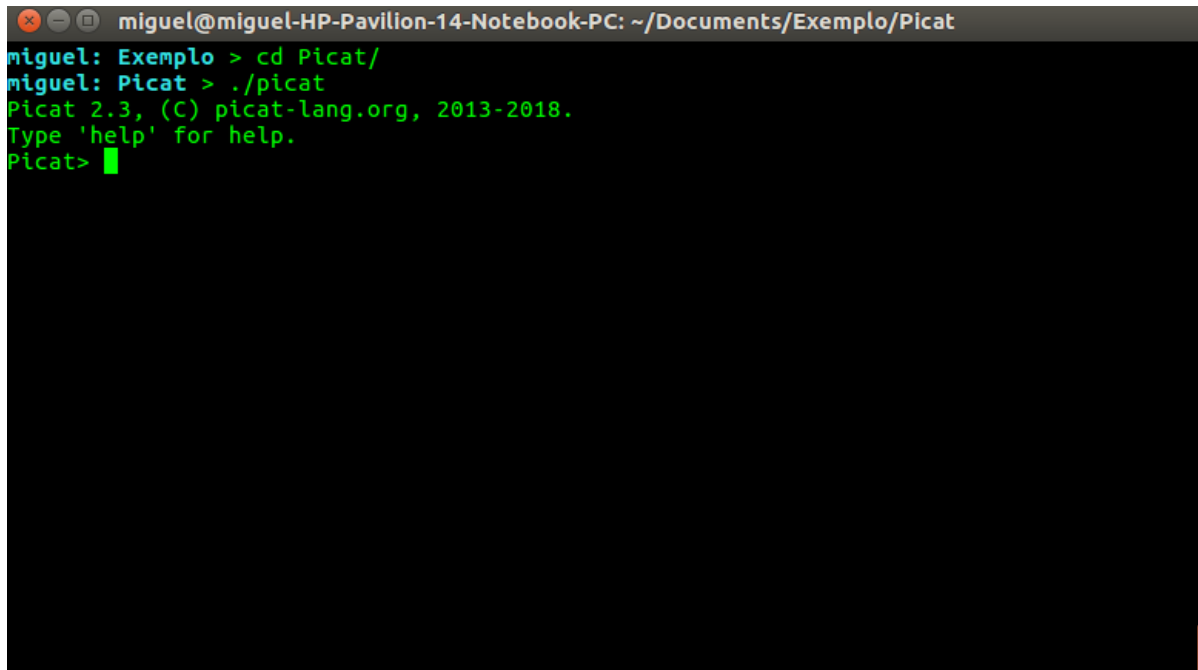
2.1.2. E descompacte-o nesta pasta:

A terminal window with a dark background and light green text. The title bar at the top reads 'miguel@miguel-HP-Pavilion-14-Notebook-PC: ~/Documents/Exemplo'. The command entered is 'tar -xvzf picat23_linux64.tar.gz', followed by a green cursor.

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~/Documents/Exemplo
miguel: Exemplo > tar -xvzf picat23_linux64.tar.gz
```

(Explicação do comando tar -xvzf)

2.1.3. Agora você pode acessar o sistema por essa pasta:

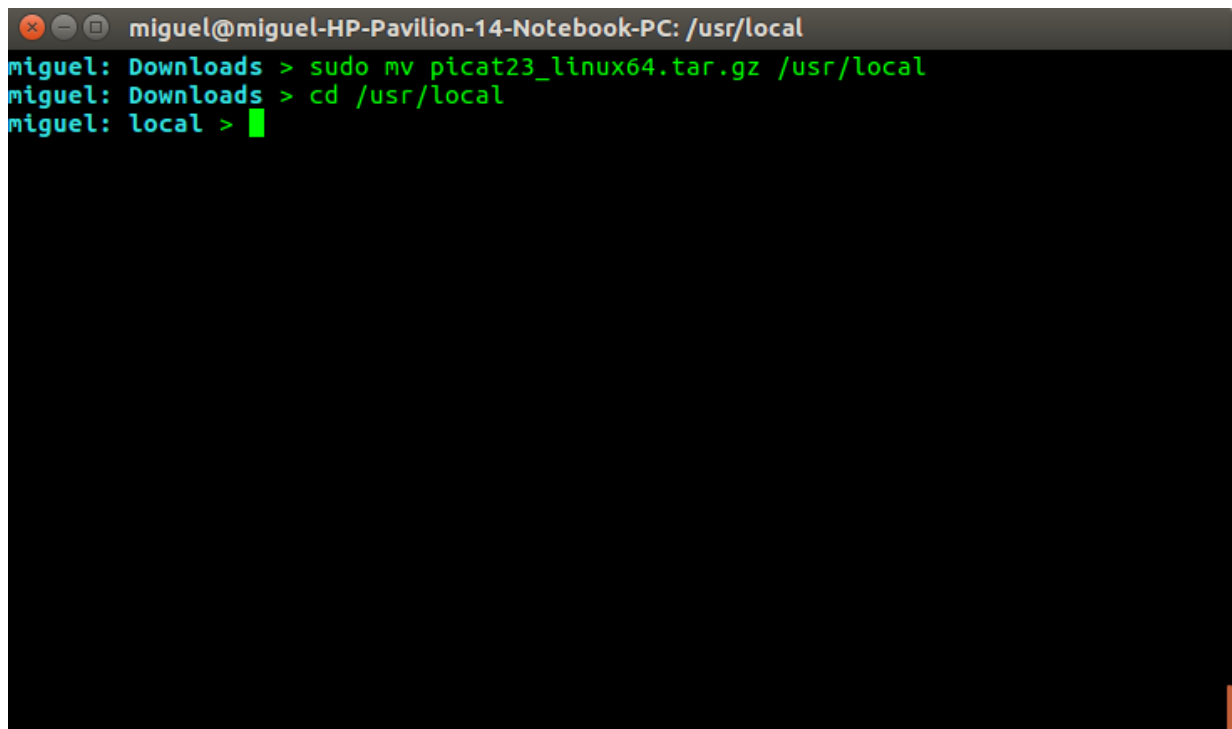
A terminal window with a dark background and light green text. The title bar reads 'miguel@miguel-HP-Pavilion-14-Notebook-PC: ~/Documents/Exemplo/Picat'. The terminal shows the following commands and output: 'miguel: Exemplo > cd Picat/' followed by 'miguel: Picat > ./picat'. The output of the command is 'Picat 2.3, (C) picat-lang.org, 2013-2018. Type 'help' for help.' followed by a new prompt 'Picat>'.

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~/Documents/Exemplo/Picat
miguel: Exemplo > cd Picat/
miguel: Picat > ./picat
Picat 2.3, (C) picat-lang.org, 2013-2018.
Type 'help' for help.
Picat>
```

2.1.4. Como utilizar o console do Picat será abordado neste tópico.

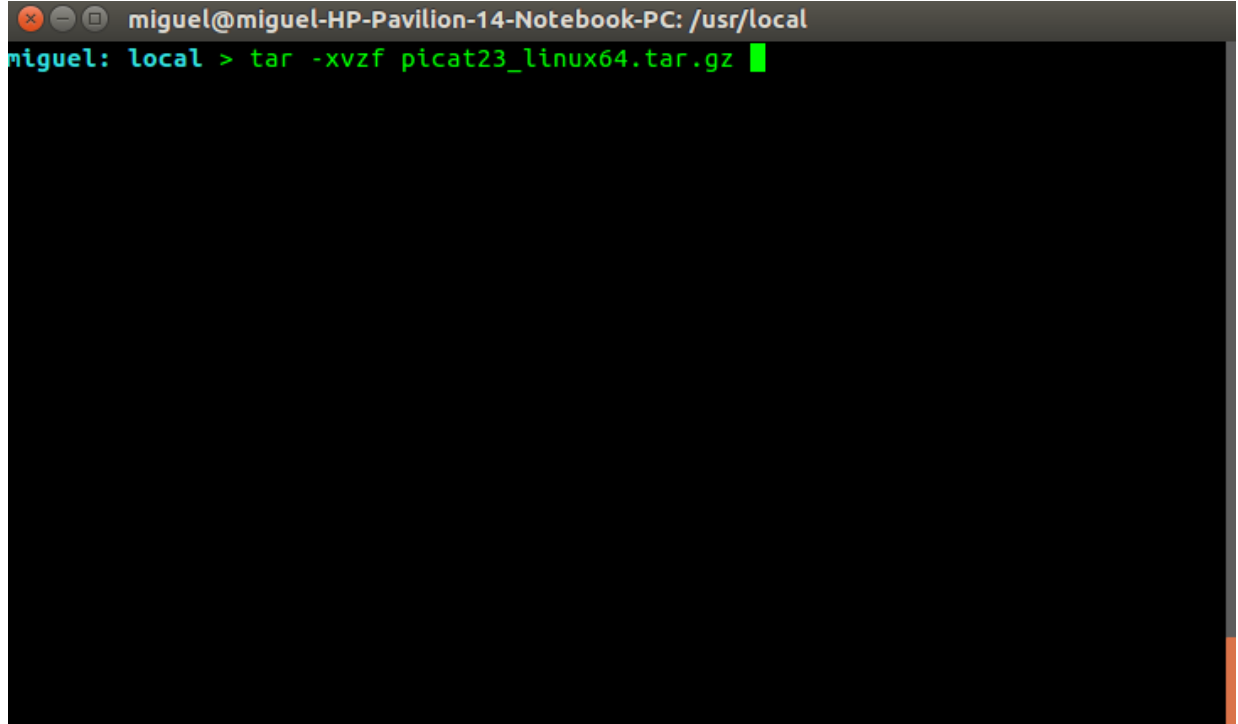
2.2. Segundo Método:

2.2.1. Baixe o arquivo *tar.gz*, e mova-o para o diretório */usr/local/* :

A terminal window with a dark background and light green text. The title bar reads 'miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local'. The terminal shows the following commands and output: 'miguel: Downloads > sudo mv picat23_linux64.tar.gz /usr/local' followed by 'miguel: Downloads > cd /usr/local' and finally 'miguel: local >'.

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local
miguel: Downloads > sudo mv picat23_linux64.tar.gz /usr/local
miguel: Downloads > cd /usr/local
miguel: local >
```

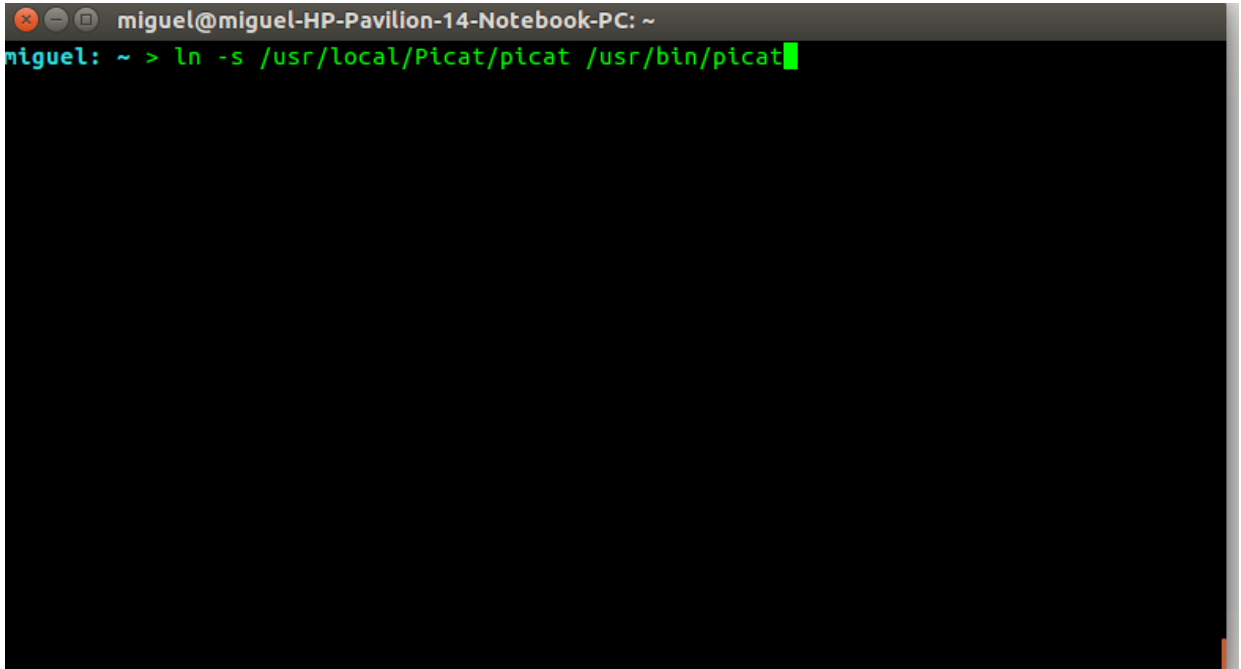
2.2.2. E descompacte-o neste diretório:

A terminal window with a dark background. The title bar shows 'miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local'. The prompt is 'miguel: local >'. The command 'tar -xvzf picat23_linux64.tar.gz' is entered and followed by a green cursor. The rest of the terminal is empty.

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: /usr/local
miguel: local > tar -xvzf picat23_linux64.tar.gz
```

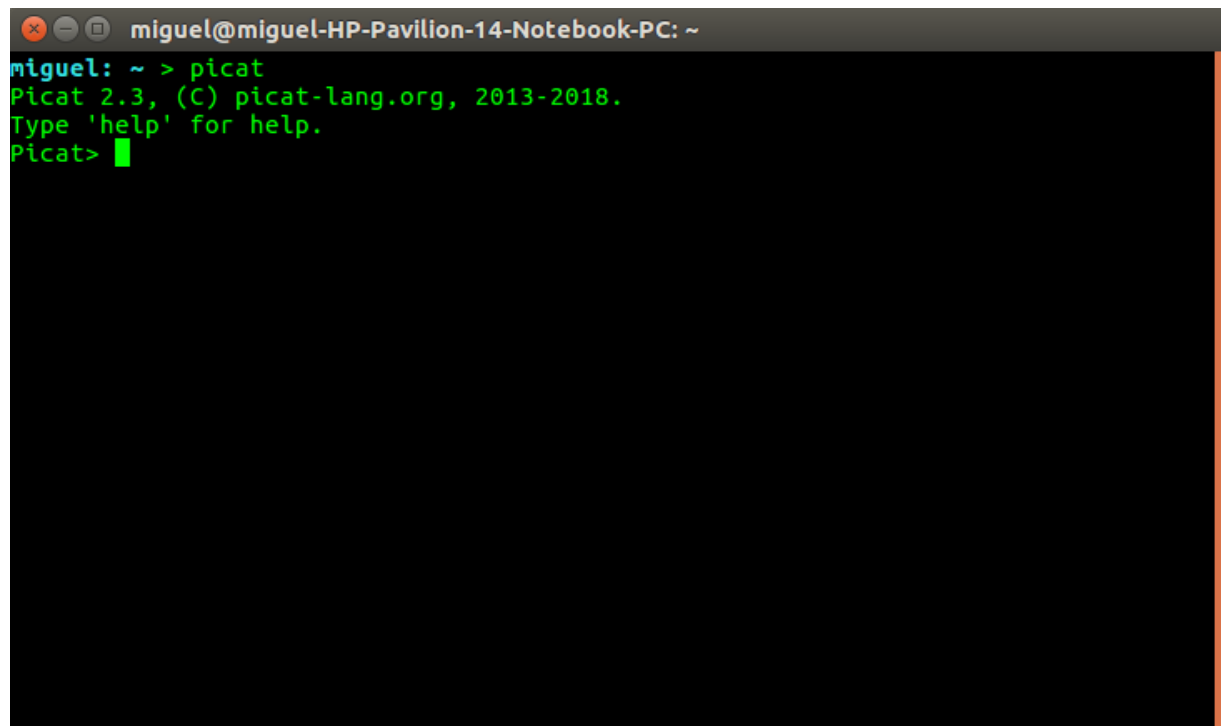
(Explicação do comando tar -xvzf)

2.2.3. Agora crie um link simbólico para o executável do Picat

A terminal window with a dark background. The title bar shows 'miguel@miguel-HP-Pavilion-14-Notebook-PC: ~'. The prompt is 'miguel: ~ >'. The command 'ln -s /usr/local/Picat/picat /usr/bin/picat' is entered and followed by a green cursor. The rest of the terminal is empty.

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~
miguel: ~ > ln -s /usr/local/Picat/picat /usr/bin/picat
```

2.2.4. Isso permitirá que você compile seus programas pelo terminal, sem ter que abrir diretamente o executável do Picat; como utilizar o sistema do Picat será abordado neste tópico.

A terminal window with a dark background and a title bar that reads "miguel@miguel-HP-Pavilion-14-Notebook-PC: ~". The terminal shows the command "picat" being executed, followed by the output "Picat 2.3, (C) picat-lang.org, 2013-2018. Type 'help' for help." and the prompt "Picat>".

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~  
miguel: ~ > picat  
Picat 2.3, (C) picat-lang.org, 2013-2018.  
Type 'help' for help.  
Picat> █
```

1.2 Como Usar o Sistema do Picat

O interpretador do Picat tem um modo interativo, que permite você executar comandos diretamente dentro do próprio interpretador (terminal), estes comandos podem ser desde coisas simples, como comparações até comandos mais complexos, como loops, compreensões de listas e compilação de programas.

1.2.1 Compilação

Há 3 modos de compilar seus programas:

- I. Caso tenha salvo seus arquivos no mesmo diretório em que você instalou o Picat, abra o interpretador do Picat (*picat.exe* no Windows, *picat* no Linux) e execute o seguinte comando:

```
cl(nome_do_seu_arquivo) *
```

- II. Caso não tenha salvo no mesmo diretório, abra o interpretador do Picat (*picat.exe* no Windows, *picat* no Linux) e execute o seguinte comando, o primeiro é o comando para Windows, o segundo é para Ubuntu:

```
cl("c:\\onde_salvou_seu\\arquivo.pi") **
```

```
cl("//home//seu_usuario//onde_salvou_seu//arquivo.pi") **
```

- III. Caso tenha criado o link simbólico no Ubuntu, abra o terminal na pasta em que salvou seu arquivo e digite o seguinte comando NO TERMINAL:

```
picat nome_do_seu_arquivo ***
```

* Este modo só funciona caso o arquivo *nome_do_seu_arquivo.pi* esteja salvo na mesma pasta em que o executável do Picat está salvo, o mesmo se aplica tanto para Ubuntu quanto Windows.

** (Windows) Este modo funciona para arquivos salvos em qualquer diretório do disco C: e também para caminhos relativos para o arquivo *arquivo.pi*, ou seja, caminhos tipo “*..\arquivo.pi*”, para arquivos salvos num diretório “acima” do diretório do Picat, ou “*diretório\arquivo.pi*”, para diretórios “abaixo” do diretório do Picat.

** (Ubuntu) Se aplica o mesmo que foi explicado para Windows.

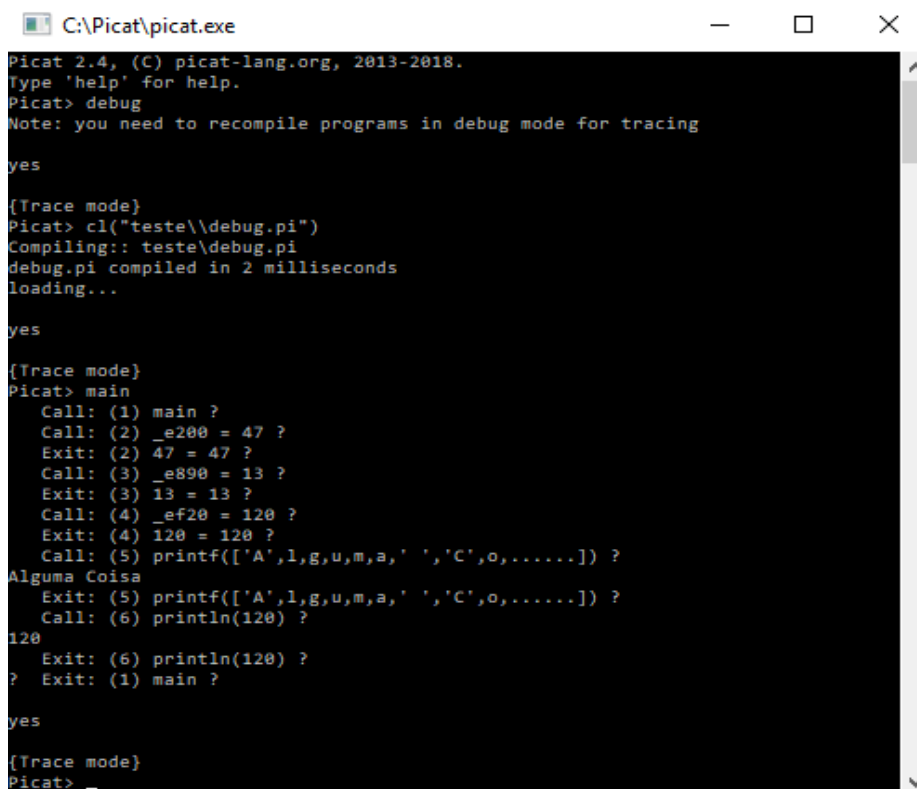
*** Este modo é tão simples quanto parece, ao executar este comando o interpretador do Picat irá compilar e executar o predicado *main* deste arquivo, e o interpretador será imediatamente encerrado e retornará ao terminal.

1.2.2 Depuração

Para depurar seus programas primeiro terá que executar o comando “*debug*” ou “*trace*”, isto iniciará o interpretador em modo de depuração, depois terá que compilar seu programa, como já foi explicado, e então terá que executar o predicado “*main*”, ou qualquer outro que queira testar.

Ao iniciar a depuração o interpretador irá avaliar seu código linha por linha, pausando em cada linha esperando input do usuário (você), a primeira linha sempre será a chamada do predicado que escolheste como o ponto de início da sua depuração, para continuar a depuração aperte *Enter*, isso prosseguirá para a próxima linha, para terminar a depuração aperte *a*.

Quando o depurador chegar a uma linha qualquer, a linha será procedida com “*Call: (Nº da Linha)*” e então o conteúdo da linha como é processada pelo interpretador, a linha seguinte será procedida com “*Exit: (Nº da Linha)*” que é como o interpretador avaliou a expressão nessa linha. Exemplo:



```
C:\Picat\picat.exe
Picat 2.4, (C) picat-lang.org, 2013-2018.
Type 'help' for help.
Picat> debug
Note: you need to recompile programs in debug mode for tracing

yes

{Trace mode}
Picat> cl("teste\debug.pi")
Compiling:: teste\debug.pi
debug.pi compiled in 2 milliseconds
loading...

yes

{Trace mode}
Picat> main
  Call: (1) main ?
  Call: (2) _e200 = 47 ?
  Exit: (2) 47 = 47 ?
  Call: (3) _e890 = 13 ?
  Exit: (3) 13 = 13 ?
  Call: (4) _ef20 = 120 ?
  Exit: (4) 120 = 120 ?
  Call: (5) printf(['A',l,g,u,m,a,' ','C',o,.....]) ?
Alguma Coisa
  Exit: (5) printf(['A',l,g,u,m,a,' ','C',o,.....]) ?
  Call: (6) println(120) ?
120
  Exit: (6) println(120) ?
? Exit: (1) main ?

yes

{Trace mode}
Picat>
```


Este exemplo mostra a depuração do seguinte código:

```
main =>  
    X = 47,  
    Y = 13,  
    Z = (X + Y) * 2,  
    printf("Alguma Coisa\n"),  
    println(Z)
```

.

O depurador do Picat não exibe nomes de variáveis, somente como elas existem para o interpretador, assim como não deixa muito claro quando há uma chamada de uma função/predicado, tanto recursiva quanto normal, ou quando há uma chamada de *backtrack* para um predicado, em suma, o depurador do Picat (na minha opinião) é bastante confuso e bastante difícil de decifrar o que está acontecendo em muitas situações.

Use quando achar necessário, mas saiba que não necessariamente conseguirá descobrir o que pode estar causando erro.

Uma alternativa é utilizar vários *prints*⁴ em partes do código pensar que pode haver algum erro, como no exemplo abaixo:

```
main =>  
    X = 47,  
    Y = 13,  
    printf("Ate aqui tudo bem"),  
    X == Y,  
    printf("Isso talvez printe"),  
    false,  
    printf("Isso não vai printar")
```

.

⁴ Por prints eu me refiro a funções de exibição de dados, como o *printf* do C, algo que é explicado [aqui](#)

Na foto, compilando e depurando este código, e depois, mudando o valor de Y para 47:

```
Picat> main
Ate aqui tudo bem
no

Picat> trace
Note: you need to recompile programs in debug mode for tracing

yes

{Trace mode}
Picat> cl("teste\\debug.pi")
Compiling:: teste\\debug.pi
debug.pi compiled in 2 milliseconds
loading...

yes

{Trace mode}
Picat> main
  Call: (1) main ?
  Call: (2) _e200 = 47 ?
  Exit: (2) 47 = 47 ?
  Call: (3) _e890 = 13 ?
  Exit: (3) 13 = 13 ?
  Call: (4) printf(['A',t,e,' ',a,q,u,i,' ',.....]) ?
Ate aqui tudo bem  Exit: (4) printf(['A',t,e,' ',a,q,u,i,' ',.....]) ?
  Call: (5) 47 == 13 ?
  Fail: (5) 47 == 13 ?
  Fail: (1) main ?

no
```

```
Picat> main
Ate aqui tudo bem
Isso talvez printe

no

Picat> trace
Note: you need to recompile programs in debug mode for tracing

yes

{Trace mode}
Picat> cl("teste\\debug.pi")
Compiling:: teste\\debug.pi
debug.pi compiled in 2 milliseconds
loading...

yes

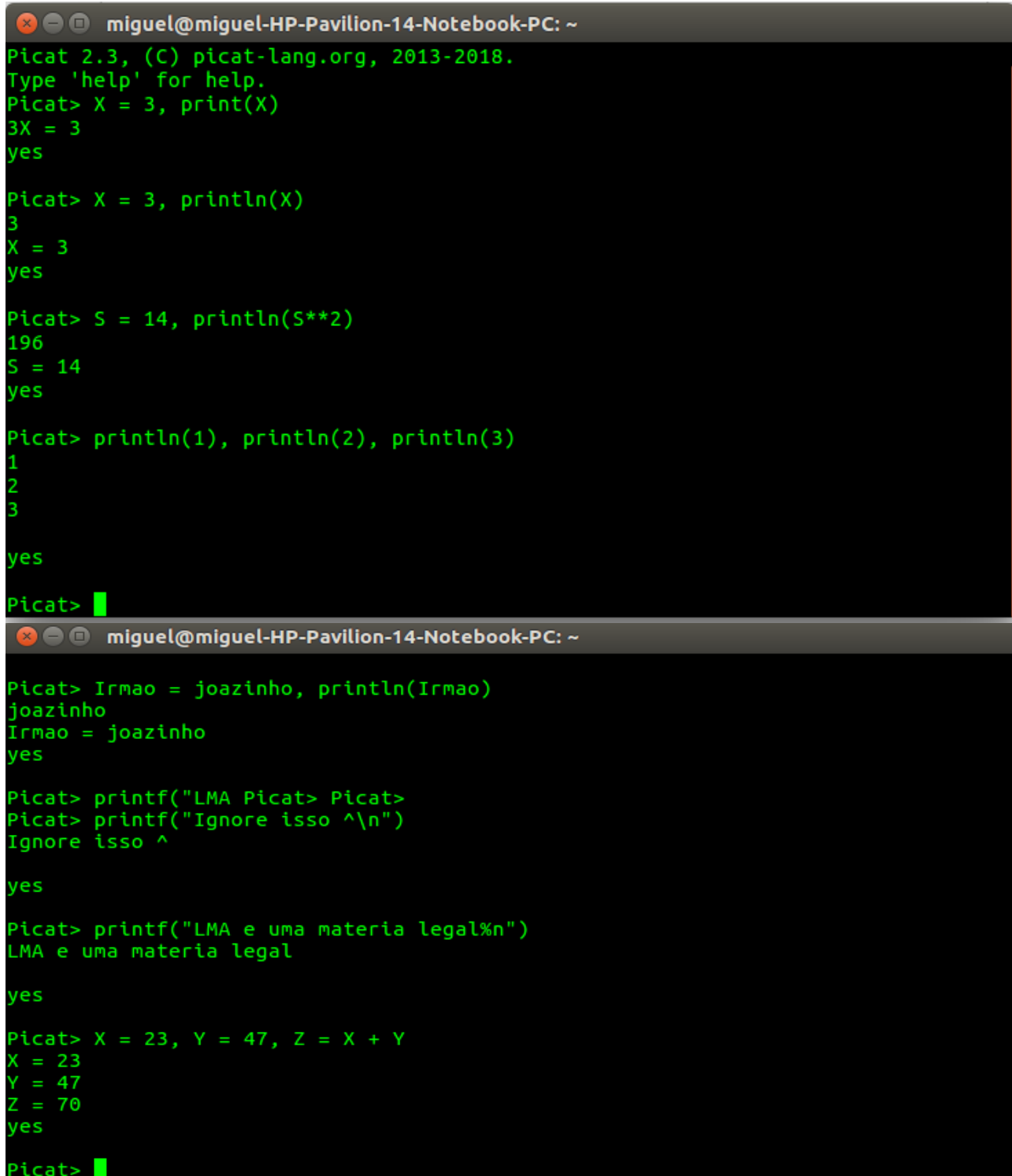
{Trace mode}
Picat> main
  Call: (1) main ?
  Call: (2) _e200 = 47 ?
  Exit: (2) 47 = 47 ?
  Call: (3) _e890 = 47 ?
  Exit: (3) 47 = 47 ?
  Call: (4) printf(['A',t,e,' ',a,q,u,i,' ',.....]) ?
Ate aqui tudo bem  Exit: (4) printf(['A',t,e,' ',a,q,u,i,' ',.....]) ?
  Call: (5) 47 == 47 ?
  Exit: (5) 47 == 47 ?
  Call: (6) printf(['I',s,s,o,' ',t,a,l,v,.....]) ?
Isso talvez printe  Exit: (6) printf(['I',s,s,o,' ',t,a,l,v,.....]) ?
  Fail: (1) main ?

no
```

1.2.3 Modo Interativo

O modo interativo permite que você brinque um pouco com o que a linguagem pode oferecer sem ter que criar uma arquivo *.pi*, para acessá-lo basta abrir o interpretador do Picat (*picat.exe* no Windows, *picat* no Linux), e escrever seu código na linha de comando, importante notar que seus comandos devem todos estar contidos em uma linha e serem separados por vírgula ou um ponto e vírgula.

Exemplos:



```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~
Picat 2.3, (C) picat-lang.org, 2013-2018.
Type 'help' for help.
Picat> X = 3, print(X)
3X = 3
yes

Picat> X = 3, println(X)
3
X = 3
yes

Picat> S = 14, println(S**2)
196
S = 14
yes

Picat> println(1), println(2), println(3)
1
2
3
yes

Picat>

miguel@miguel-HP-Pavilion-14-Notebook-PC: ~
Picat> Irmao = joazinho, println(Irmao)
joazinho
Irmao = joazinho
yes

Picat> printf("LMA Picat> Picat>
Picat> printf("Ignore isso ^\n")
Ignore isso ^
yes

Picat> printf("LMA e uma materia legal%n")
LMA e uma materia legal
yes

Picat> X = 23, Y = 47, Z = X + Y
X = 23
Y = 47
Z = 70
yes

Picat>
```

```
miguel@miguel-HP-Pavilion-14-Notebook-PC: ~
Z = 70
yes

Picat> printf("Aqui e uma bom lugar pra descobrir o que causa erros)
*** error(syntax_error(in_character),b_NEXT_TOKEN_ff)

Picat> printf("Acredite em mim, vcs encontraram %s", "varios")
Acredite em mim, vcs encontraram varios
yes

Picat> printf("Comandos de uma linha nao sao carregados para outra linha\n")
Comandos de uma linha nao sao carregados para outra linha

yes

Picat> printf("Ou seja o X que foi declarado algumas linhas atras nao existe +")
Ou seja o X que foi declarado algumas linhas atras nao existe +
yes

Picat> printf("Prova: %w",X)
Prova: _103d0
yes

Picat> █
```

```
C:\Picat\picat.exe
Picat 2.4, (C) picat-lang.org, 2013-2018.
Type 'help' for help.
Picat> printf("Isso, obviamente, tambem funciona no Windows\n")
Isso, obviamente, tambem funciona no Windows

yes

Picat> printf("Nao por isso que Windows e melhor que Linux\n")
Nao por isso que Windows e melhor que linux

yes

Picat> printf("E so uma observacao mesmo\n")
E so uma observacao mesmo

yes

Picat> Nota = "vcs podem fazer praticamente qualquer coisa aqui"
Nota = [v,c,s,' ',p,o,d,e,m,' ',f,a,z,e,r,' ',p,r,a,t,i,c,a,m,e,n,t,e,' ',q,u,
a,l,q,u,e,r,' ',c,o,i,s,a,' ',a,q,u,i]
yes

Picat> Obs = "So que nem tudo vai ser apresentado de um jeito legivel"
Obs = ['S',o,' ',q,u,e,' ',n,e,m,' ',t,u,d,o,' ',v,a,i,' ',s,e,r,' ',a,p,r,e,s,
e,n,t,a,d,o,' ',d,e,' ',u,m,' ',j,e,i,t,o,' ',l,e,g,i,v,e,l]
yes

Picat> printf("A maioria das coisas, pra falar a verdade\n")
A maioria das coisas, pra falar a verdade

yes

Picat> Obs2 = "O Word nao gosta de colaborar com a formatacao as vezes"
Obs2 = ['O',' ',',','W',o,r,d,' ',n,a,o,' ',g,o,s,t,a,' ',d,e,' ',c,o,l,a,b,o,r,a,
r,' ',c,o,m,' ',a,' ',f,o,r,m,a,t,a,c,a,o,' ',a,s,' ',v,e,z,e,s]
```

2. Variáveis e Aritmética

Variáveis em Picat são drasticamente diferentes do que em C e linguagens parecidas com C, neste tópico tentarei explicar a diferença entre os dois e como trabalhar com as variáveis em Picat.

Picat é uma linguagem de tipagem dinâmica, ou seja, o tipo de uma variável é checado quando o programa está rodando.

Outra coisa a ser notada sobre as variáveis em Picat é que há dois tipos básicos de variáveis que irei abordar aqui; Primitivas que são números e átomos, e Compostas que são listas e *strings* (mais aqui).

2.1 Declaração

Para declarar uma variável em C, você declara o tipo da variável e seu nome, seguindo as regras de nomenclatura da linguagem, como neste exemplo:

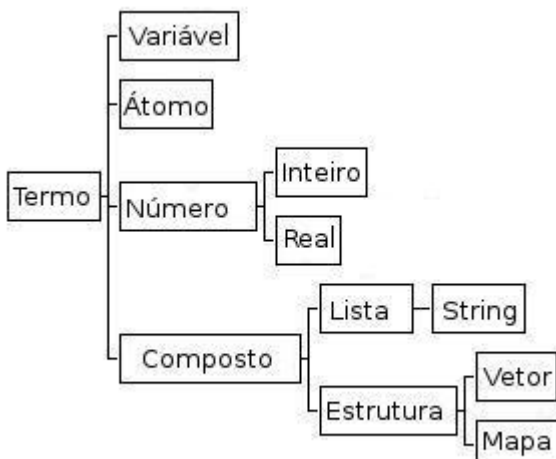
```
int numero1 = 3, numero2 = 5, numero3;  
char Letra = 'a', Tuberculo[] = "Batata";  
float NumVirgula = 3.1415;
```

Em Picat, declarar variáveis é muito diferente, mas muito mais simples que em C, para declarar uma variável basta escrever o nome dela e o valor que quer atribuí-la, como neste exemplo:

```
Nome = "Miguel",  
A = 22,  
B = 8320,  
C = 3.1415,  
Lista = [1,2,3,4],  
String = "Isso e uma string",  
String2 = [o, u, t, r, a, " ", s, t, r, i, n, g]
```

Três coisas devem ser notadas quanto a declaração de variáveis em Picat;

- I. Não se declara o tipo da variável quando uma variável é declarada, o tipo da variável é definido baseado no valor que a ela foi atribuído.
- II. Toda vez que uma variável é declarada, ela deve ser atribuída um valor, você não pode declarar uma variável sem nenhum valor, como foi feito com a variável *numero3* no exemplo de variáveis em C, caso precise de uma variável que não tenha um valor pré-definido, tem duas opções, crie ela antes e atribua-a um valor qualquer e, quando for utiliza-la, atribuir o valor desejado; ou pode cria-la imediatamente antes de usa-la e atribuir o valor neste momento.
- III. O nome de uma variável deve **SEMPRE COMEÇAR** com uma letra **MAIÚSCULA OU _** (sublinhado, *underline*, *underscore*, etc.), nenhum nome de variável pode começar com letras minúsculas ou números (mas podem conter letras minúsculas e números).
 - a. Obs.: Apesar que é possível criar uma variável cujo nome seja somente , você não deveria, pois este nome é usado para *variáveis anônimas*, ou seja, mesmo que você possa atribuir valores à estas variáveis, estes valores não são “guardados” na memória.



Aqui podemos ver como são hierarquizados os tipos de dados em PICAT.

Lembrando que variáveis podem conter outros termos.

2.2 Particularização e Atribuição

Em Picat há duas operações para definir valores a variáveis, a particularização (usa o símbolo =) e a atribuição (usa o símbolo :=), que são muito parecidas, mas causam muita confusão.

A particularização (=) é uma operação reversível⁵, que pode ser feita somente uma vez por variável, ou seja, uma vez que uma variável é particularizada para um valor ela não pode ser particularizada para outro valor; quando uma variável é particularizada ela terá a mesma identidade do valor à qual ela foi particularizada, ou seja, ela terá o mesmo comportamento do tipo deste valor (ou seja, se particularizar uma variável à uma *string*, ela se comportará como *string*, se particularizar uma variável a um número, ela se comportará como um número, etc.).

A atribuição (:=) também é uma operação reversível, que pode ser feita quantas vezes quiser por variável; por questões de simplicidade tudo que precisam saber é que esta operação é equivalente a operação de atribuição do C, e que após uma variável ser particularizada a um valor ela pode ser atribuída a outro valor sem problemas.

Exemplos:

Certo = *“Isso foi uma particularização”,*

Certo := *“Isso foi uma atribuição, nada de errado aqui, posso repetir isso quantas vezes quiser”,*

Errado = *“Particularização só pode ser feita uma vez”,*

Errado = *“Particularização novamente, isso causaria um erro”,*

Certo2 = *“Números e letras minúsculas podem aparecer em nomes de variáveis”,*

Certo2 := *“Desde que não sejam o primeiro caractere no nome”,*

Certo2 := *“Novamente, atribuição pode ser feita quantas vezes eu quiser”,*

C3rto := *“Eu posso declarar uma variável com uma atribuição sem problemas”,*

⁵ Isso significa que caso seja feito *backtrack* para um ponto do programa antes de uma particularização, ela será desfeita e pode ser refeita; backtracking é explicado [aqui](#).

Passando estas declarações para um arquivo *.pi* e depurando o código (com algumas pequenas alterações para fazer com que isso funcione) temos o seguinte resultado:



```
C:\Picat\picat.exe
{Trace mode}
Picat> cl("teste\\debug")
Compiling:: teste\debug.pi
debug.pi compiled in 2 milliseconds
loading...

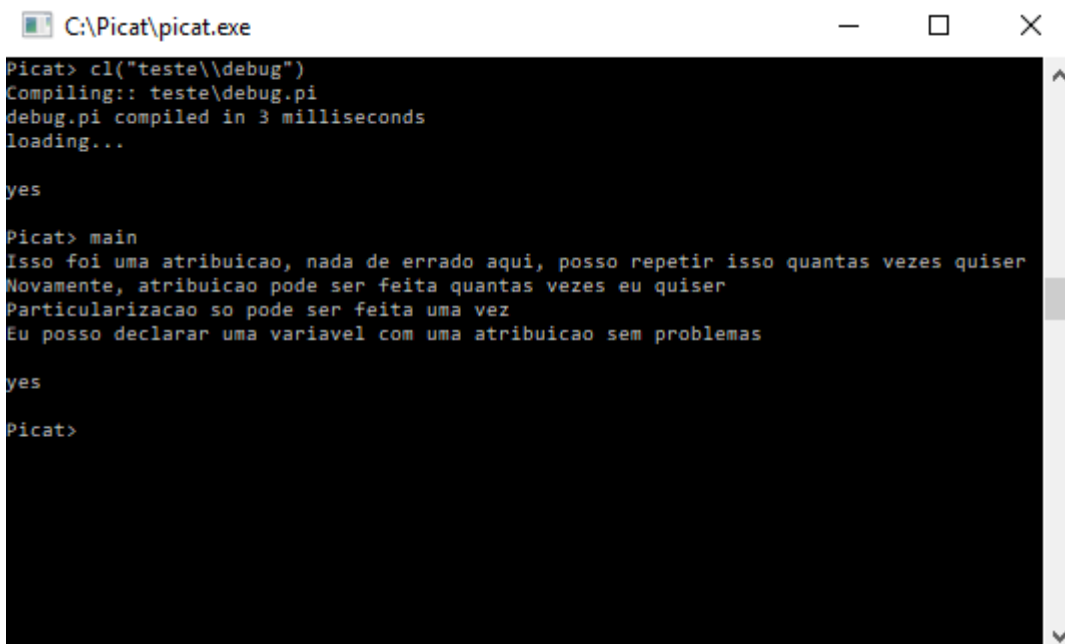
yes

{Trace mode}
Picat> main
Call: (1) main ?
Call: (2) _e3d0 = ['I',s,s,o,' ',f,o,i,' ',.....] ?
Exit: (2) ['I',s,s,o,' ',f,o,i,' ',.....] = ['I',s,s,o,' ',f,o,i,' ',.....] ?
?
Call: (3) _f7f0 = ['P',a,r,t,i,c,u,l,a,.....] ?
Exit: (3) ['P',a,r,t,i,c,u,l,a,.....] = ['P',a,r,t,i,c,u,l,a,.....] ?
Call: (4) ['P',a,r,t,i,c,u,l,a,.....] = ['P',a,r,t,i,c,u,l,a,.....] ?
Fail: (4) ['P',a,r,t,i,c,u,l,a,.....] = ['P',a,r,t,i,c,u,l,a,.....] ?
Fail: (1) main ?

no

{Trace mode}
Picat>
```

Como podem ver, há um erro na 4ª linha, que é onde foi refeita a particularização da variável Errado; removendo este erro, temos o seguinte resultado:



```
C:\Picat\picat.exe
Picat> cl("teste\\debug")
Compiling:: teste\debug.pi
debug.pi compiled in 3 milliseconds
loading...

yes

Picat> main
Isso foi uma atribuicao, nada de errado aqui, posso repetir isso quantas vezes quiser
Novamente, atribuicao pode ser feita quantas vezes eu quiser
Particularizacao so pode ser feita uma vez
Eu posso declarar uma variavel com uma atribuicao sem problemas

yes

Picat>
```

Obs.: Na segunda foto não foi feita a depuração do código, isso foi feito pois na primeira foto era necessária fazer a depuração para exibir qual parte do código está causando erro.

Obs2.: Na segunda foto são exibidas as variáveis no seu estado final, ou seja, após todas as particularizações e atribuições.

2.3 Aritmética e Operadores

Picat possui todas as operações matemáticas básicas presentes no C, assim como muitos operadores lógicos e outros operadores diversos; neste tópico apresentarei os mais relevantes para vocês.

Os operadores estarão em ordem de precedência.

MATEMÁTICOS	
Operador	Significado
$X ** Y$	Potenciação
$X * Y$	Multiplicação
X / Y	Divisão
$X // Y$	Divisão Inteira [*]
$X \bmod Y$	Resto da Divisão ^{**}
$X + Y$	Adição
$X - Y$	Subtração

IMPLICAÇÕES (Predicados e Funções)	
Operador	Significado
\Rightarrow	Implicação
$? \Rightarrow$	Implicação “retrocessível”

Obs.: As implicações não são operadores, tecnicamente, mas estão nesta lista por questões de simplicidade e organização

LÓGICOS	
Operador	Significado
not X	Negação (\sim ou \neg)
X , Y X && Y	Conjunção (E) ⁺
X ; Y X Y	Disjunção (OU) ⁺⁺
true yes	Tautologia [±]
false no	Contradição ^{±±}

OUTROS	
Operador	Significado
L1 ++ L2	Concatenação de Listas ^x
X != Y X !== Y	Diferença
X == Y	Equivalência
X ::= Y	Equivalência Numérica ^{xx}
X in L	Dentro de Lista ^{xxx}
Início..[Passo]..Fim	Alcance de Números ^{xxxx}

Obs. 2: Para a grande maioria dos operadores pode-se usar a notação *operador(OperandoE, OperandoD)*, por exemplo; $\Rightarrow(X, Y)$, $+(X, Y)$, $-(X, Y)$, $** (X, Y)$...

Vamos assumir que X e Y são duas variáveis quaisquer e que L, L1 e L2 são três listas quaisquer.

*: Esta é uma operação de divisão que retorna inteiros, veja o exemplo, em que na esquerda temos a divisão inteira e na direita a divisão normal: $5//2 = 2$ | $5/2 = 2.5$

** : O resto da divisão é o valor que “sobra” da divisão de dois números, muito usado geralmente para determinar se um número é par ou não

⁺: A conjunção de duas coisas é muito importante para a linguagem, além de ter a função de ser um operador lógico para condicionais, é fundamental para a estrutura da linguagem, algo que será explicado aqui.

⁺⁺: A disjunção de duas coisas é muito importante para a linguagem, além de ter a função de ser um operador lógico para condicionais, é fundamental para a estrutura da linguagem, algo que será explicado aqui.

[±]: A tautologia, ou *true/yes*, não é um operador propriamente dito, mas é bastante usado para estruturar programas e controlar fluxo de predicados, algo que é melhor explicado aqui.

^{±±}: A contradição, ou *false/no*, não é um operador propriamente dito, mas é bastante usado para estruturar programas e controlar fluxo de predicados, algo que é melhor explicado aqui.

[×]: A concatenação de duas listas vai juntar o começo da lista L2 com o final da lista L1, listas são explicadas aqui.

^{××}: Na comparação normal ($==$), caso compare um número *float* com um número *int*, mesmo que sejam tecnicamente iguais (1.0 e 1, por exemplo), serão ditos que são diferentes pois o interpretador considera números de tipos diferentes como coisas diferentes, a comparação numérica ignora os tipos, só avaliando o valor numérico.

^{×××}: Este operador vai verificar se X é um termo da lista L, independente do tipo de X ou do tamanho de L.

^{××××}: Um alcance de números é uma função especial que gera uma lista de números começando em Início, adicionando Passo em cada iteração até chegar em Fim. Passo pode ser omitido, caso for, será considerado igual a 1, caso queira que Início seja menor que Fim, podes usar um Passo negativo.

2.4 Átomos

Átomos em Picat são constantes simbólicas que representam caracteres únicos ou *strings* imutáveis, eles podem estar encapsulados entre aspas simples (‘’) ou não, sendo que caso não estejam, o primeiro caractere deve ser uma letra minúscula, caso estejam, o nome pode ser qualquer sequência arbitrária de caracteres que desejar.

Alguns exemplos de átomos e como podem ser usados:

```
W = a,  
X = atomo_qualquer,  
Y = string_imutavel,  
Z = 'Atomo_com_aspas',
```

Átomos não podem ser declarados como variáveis normais, e seu uso é restrito há *strings* imutáveis e caracteres.

2.5 Listas

Listas em Picat são como vetores em C, ou seja, uma variável que “contém” outras variáveis, as principais diferenças entre os vetores de C e as listas de Picat (além do fato de um ser um *array* e outro ser uma lista *linkada*) é que a **indexação das listas** de Picat **começa em 1**, e listas não tem tipo fixo, ou seja, elas podem conter valores de múltiplos tipos dentro de si.

Listas são declaradas como uma sequência arbitrária de caracteres ou números separados por vírgula dentro de colchetes.

Para acessar um termo em uma lista usa-se a notação de subscrito *Lista*[*I*, *J*, ...], *I* e *J* devem ser números inteiros maiores ou iguais que 1 e menores ou iguais que o tamanho (quantidade de termos) da lista, e *Lista* deve, obviamente, ser uma lista.

Exemplos:

```
Lista = [1,2,3,4,5],  
X = Lista[1],  
println(X),  
println(Lista[5])
```

Ao executar este código temos como saída 1 e 5, que os valores que estavam na lista “Lista” nas posições 1 e 5.

2.5.1 *Strings*

Strings são um “subtipo” de listas, ou seja, são listas especiais.

O que torna *strings* diferentes de listas normais é que os únicos valores armazenados em *strings* são átomos de caracteres únicos, ou seja, *strings* são listas de chars, como em C.

Strings são declaradas com uma sequência arbitrária de caracteres quaisquer encapsulados por aspas duplas (“”) ou uma sequência arbitrária de caracteres minúsculos (caso queira usar caracteres maiúsculos ou espaços estes devem estar encapsulados em aspas simples) separados por vírgula dentro de colchetes.

Caso tenhas uma lista em que nem todos os valores não sejam átomos de caracteres únicos ela será tratada como uma lista normal, não como uma *string*.

Exemplos:

```
String1 = “Uma string”,
String2 = [o,u,t,r,a,’ ’,s,t,r,i,n,g],
String3 = [‘M’, ‘A’, ‘I’, ‘S’, ‘ ’, ‘U’, ‘M’, ‘A’],
println(String1),
println(String2),
println(String3)
```

Executando este código, temos a saída “Uma string”, “outra string” e “MAIS UMA”.

2.6 Tuplas

Tuplas são um tipo especial de variável, elas são, basicamente, listas imutáveis, ou seja, uma vez que uma *tupla* é declarada contendo quaisquer valores estes valores não podem ser alterados nem substituídos, mas a variável que contém uma *tupla* pode conter outros tipos de valores.

Assim como listas, você acessa os termos de uma *tupla* usando a notação de subscrito $Tupla[I, \dots, J]$, onde I e J são numero inteiros maiores que 1 e menores ou iguais ao tamanho de *Tupla*.

Uma grande diferença entre *tuplas* e listas é que os valores de uma *tupla* são encapsulados por parênteses, ao invés de colchetes, porém ainda são separados por virgulas e ainda podem conter qualquer valor arbitrário

Exemplos:

```
Tupla = (1,2,3,4,5),  
println(Tupla[1]),  
Tupla2 = (aaaaaaa),  
pritln(Tupla2),  
Tupla := [aa, bb],  
Tupla2[1] := 'AbCdEf',
```

Caso execute este código verá que tem um erro na ultima linha, pois estou tentando alterar o valor de uma *tupla*, algo que é proibido, porém repare que o que esta sendo feito na penúltima linha não é proibido, pois como havia dito antes, uma variável que contém uma *tupla* pode receber outros valores, mas uma *tupla* não pode ser alterada.

3. Predicados, Funções e *index*

Predicados e funções são talvez as mais importantes estruturas de Picat, elas são usadas para criar regras sobre o domínio que esteja trabalhando, e operar sobre este domínio de um modo que não altere ele ou o fluxo do resto do programa.

3.1 Predicados

Predicados são regras que definem relações entre termos sobre o qual eles foram definidos, estas relações podem ter nenhuma, uma, ou várias respostas. Para o interesse do trabalho de LMA, considere somente uma resposta possível para algum predicado que for fazer.

O domínio sobre qual um predicado vai operar é o domínio dos seus argumentos, todas as relações que forem estabelecidas em um predicado serão validas somente dentro deste domínio.

Predicados seguem o formato *predicado*(T_1, \dots, T_n), *Cond*=> *Corpo*. ou *predicado*(T_1, \dots, T_n), *Cond* ?=> *Corpo*.⁶, onde N é a quantidade de termos sobre o qual ele é definido, ou quantidade de argumentos; N é chamado de *Aridade*, *predicado* é o nome do predicado, este nome pode ser qualquer um que desejar, desde que comece com uma letra minúscula e não tenha espaços, *Corpo* é onde é escrito o código em si, onde são feitas as comparações e atribuições e tudo mais, e *Cond* é uma condição sobre a execução desse predicado, algo opcional.

Estes N termos sobre o qual um predicado é definido são chamados de *argumentos* do predicado, por convenção é tomado que o último argumento do predicado é a sua saída, ou seja, o resultado da relação proposta, e todos os outros são sua entrada, ou seja, aquilo que será aplicado a relação.

Dentro do corpo de um predicado é onde vai escrever a maior parte de seu código, o mais importante sobre isso é que cada linha (instrução) deve ser separada por vírgula ou ponto e vírgula e que a última linha deve terminar com um ponto.⁷

⁶ Este seria um predicado backtrackable, ou seja, um predicado que pode fazer backtracking, algo que é explicado [aqui](#)

⁷ É possível também você agrupar todo o corpo do seu predicado em uma só linha, separando cada instrução individual do mesmo modo.

Caso um predicado tenha *aridade* = 0, ele é escrito como *predicado()*=> *Corpo.* ou *predicado*=> *Corpo.*, isto é, ele não recebe nenhum termo para avaliar e nem retorna nenhum termo.

Caso um predicado tenha *aridade* = 1, ele é escrito como *predicado(T)*=> *Corpo.*, isto é, ele recebe ou retorna somente um termo, geralmente este termo é sua saída.

Exemplos:

```
umdoistres()=> println(1), println(2), println(3) .

cria_lista(Lista)=> Lista = [X : X in 1..5].

exibe_numeros(Num1, Num2, Num3)=> printf("1: %d\n2: %d\n3: %d\n",Num1,
Num2, Num3).

compara_coisas(Coisa1, Coisa2, Coisa3, Coisa4)=>
    Coisa1 <= Coisa2,
    Coisa2 >= Coisa3,
    Coisa4 = Coisa2
.

soma_numeros(Num1, Num2, NumS)=> NumS = Num1 + Num2.

chamada_de_predicado()=>
    umdoistres(),
    soma_numeros(12, 33, S),
    println(S),
    exhibe_numeros(12,33,S-1)
.
```

No predicado “*cria_lista(Lista)*” é mostrado um predicado de único argumento que é sua própria saída; os predicados “*compara_coisas*” e “*soma_numeros*” são predicados com N argumentos quaisquer cujo o último é seu retorno; o predicado “*chamada_de_predicado*” mostra como é feita a chamada de um predicado normal e o uso do seu retorno.

3.2 Funções

Funções serão menos usadas para o trabalho de LMA, mas ainda assim são importantes.

Funções, assim como predicados, estabelecem relações entre termos de um domínio sobre os quais foram definidos, a principal diferença é que funções fazem isso de um modo menos abstrato que predicados.

Funções são escritas da forma *funcao*(T_1, \dots, T_n), *Cond* = *Retorno* => *Corpo*., onde *funcao* é o nome da função (segue as mesmas regras de nomenclatura que predicados), *N* é a *aridade*, *Retorno* é a saída da função, o valor que ela retornar à o que chamou-a, e *Corpo* é onde é escrito o código em si (seguindo as mesmas regras de estruturação que predicados), e *Cond* é uma condição opcional sobre a execução dessa função.

Exemplos:

```
eleva_quadrado(X) = Y => Y = X ** 2.  
% Outro modo:  
% eleva_quadrado(X) = X ** 2.  
cria_lista_tamanhoX(X) = Lista => Lista = new_list(X).  
soma_numeros(N1, N2) = S => S = N1 + N2.  
chamada_de_funcao() =>  
    Numero = eleva_quadrado(4),  
    L = cria_lista_tamanhoX(Numero),  
    Numero2 = soma_numeros(Numero, eleva_quadrado(2))  
.
```

As funções “*eleva_quadrado*” e “*cria_lista_tamanhoX*” representam a forma geral de uma função que recebe um argumento e retorna um valor, o mesmo para a função “*soma_numeros*”, neste caso com dois argumentos, o predicado “*chamada_de_funcao*” mostra como é feita a chamada de uma função normal e como seu retorno pode ser usado, assim como mostra a possibilidade de você chamar uma função passando ou função como argumento.

Algo interessante sobre funções em Picat é que elas foram feitas de modo que tenham uma estrutura semelhante com funções em matemática, algo que talvez não tenha ficado muito claro no ultimo exemplo.

O modo alternativo da função “*eleva_quadrado*” exemplifica isso.

3.2.1 Funções de Saída de Dados

Picat tem duas funções principais de saída de dados, *print* e *write*, que são muito semelhantes; cada função também aceita um “sufixo”, (tecnicamente são funções diferentes, mas com o nome quase igual) *f* ou *ln*.

SUFIJO	RESULTADO
Nenhum <i>print(X)</i> <i>write(X)</i>	Exibe a variável, aceita somente um argumento.
<i>ln</i> <i>println(X)</i> <i>writeln(X)</i>	Exibe a variável e pula uma linha (adiciona um ‘\n’ ao fim), aceita somente um argumento.
<i>f</i> <i>printf(“%esp”, X)</i> <i>writeln(“%esp”, X)</i>	Exibe a variável com formatação, permite que seja exibido texto e requer o uso de um especificador de formato para exibir o valor (equivalente ao <i>printf()</i> do C), aceita múltiplos argumentos.
ESPECIFICADOR	SIGNIFICADO
<i>%d %i</i>	Número Inteiro c/ Sinal
<i>%%</i>	Porcentagem
<i>%c</i>	Caractere
<i>%f</i>	Número Real
<i>%n \n</i>	Nova Linha/Quebra Linha
<i>%s</i>	<i>String</i>
<i>%w</i>	Termo, o valor “interno” de uma variável
<i>%u</i>	Número Inteiro s/ Sinal

FUNÇÕES	ENTRADAS			
	“abc”	[a,b,c]	‘a@b’	[a,b,c], “abc”
<i>print</i>	abc	abc	a@b	
<i>println</i>	abc	abc	a@b	
<i>printf</i>	abc Usando %s	abc Usando %w	a@b Usando %w	abc abc Usando %w %s
<i>write</i>	[a,b,c]	[a,b,c]	‘a@b’	
<i>writeln</i>	[a,b,c]	[a,b,c]	‘a@b’	
<i>writeln</i>	abc Usando %s	[a,b,c] Usando %w	a@b Usando %w	[a,b,c] abc Usando %w %s

3.3 index

O predicado *index()* é um predicado especial, que permite você criar domínios (ou fatos) sobre os quais você pode operar.

O *index* é definido deste modo: *index* ($M_1, \dots, M_n, \dots, (M_{m1}, \dots, M_{mn})$), em que cada M_{mn} é um símbolo + ou –, em que + significa que aquela variável já possui um valor e – significa que aquela variável não tem um valor.

Para o interesse de vocês, só trabalharão com fatos do tipo *index(-)*, ou seja, somente uma variável será indexada (atribuída um valor) por vez.

Após ser declarado o *index(-)* ele deve ser seguido por quantos fatos quiser, os fatos devem ser do tipo *fato(Valor)*., em que *fato* é o nome do fato (segue as regras de nomenclatura de predicados normais) e *Valor* é algum valor que pode ser atribuído a uma variável (número, *string*, átomo, etc.) sempre seguidos por um ponto.

Para usar os fatos criados usando *index*, deve fazer uma chamada do fato com quantas variáveis como argumentos quanto há valores possíveis; ou seja, se declarou seu fato como *nome(valor)*, quando for usar o fato *nome* deve fazer uma chamada do tipo *nome(Variavel)*.

Seguem alguns exemplos de *index()*, suas possíveis aplicações e seu uso correto:

```
index(-)
pessoa("Miguel").
pessoa("Antonio").
pessoa("Rodrigo").
pessoa("Maise").
pessoa("Italo").
```

```
main() =>
    pessoa(X),
    println(X)
```

.

Neste exemplo, X será particularizado para o primeiro fato *pessoa* que for encontrado (neste caso, "Miguel").

Neste próximo exemplo X será particularizado para todos os possíveis fatos *pessoa*, usando backtracking.

```
index(-)
pessoa("Miguel").
pessoa("Antonio").
pessoa("Rodrigo").
pessoa("Maise").
pessoa("Italo").
```

```
main() ?=>
    pessoa(X),
    println(X),
    fail
.
```

Neste exemplo, serão exibidos todos os fatos *pessoa*.

Obs.: Quando são criados fatos usando o predicado *index()* estes fatos já são criados como fatos “*retrocessíveis*”, ou seja, eles são do tipo *fato(V)?=> Valor.*, neste caso em uma situação em que faz verificações com os valores que vieram de um fato e esta verificação falhar, o retrocesso ocorrerá primeiro dentro do fato, antes de ocorrer dentro do predicado em que estão trabalhando.

Obs. 2: Quando cria fatos usando o predicado *index()*, tecnicamente está criando predicados especiais que só possuem um valor de retorno.

4. *Backtracking* e Recursão

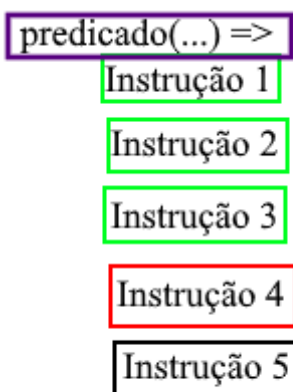
Em Picat *backtracking* e recursão são duas das coisas mais importantes da linguagem, e igualmente importantes para o trabalho, assim como extremamente confusas e que geram muitas dificuldades entre quem está vendo isso pela primeira vez.

4.1 *Backtracking*

Backtracking, em português significa literalmente retrocesso, voltar atrás, é exatamente o que é feito pelo interpretador do Picat quando ele chega em um ponto em que uma regra *backtrackable* (“*retrocessível*”) falha.

Uma regra “*retrocessível*” é um predicado que é do tipo *predicado*(T_1, \dots, T_N) $?=>$ *Corpo*. , ou seja, que tem o “operador” $?=>$ no lugar do “operador” $=>$.

Para poder explicar melhor o que significa uma regra ser *backtrackable* terei que adiantar um pouco uma explicação do próximo tópico.



Mencionarei bastante o sucesso ou não de uma instrução, por sucesso quero dizer como essa instrução foi avaliada pelo interpretador, ou seja, se ela foi avaliada como verdadeiro ou falso.

As instruções em Picat são avaliadas como verdadeiro ou falso pois elas são, de certo modo, proposições lógicas (isso inclui funções e predicados), são os p , q , r e s da Lógica Proposicional, mas saber isso não importa muito agora, só importa saber que elas são avaliadas como verdadeiro ou falso.

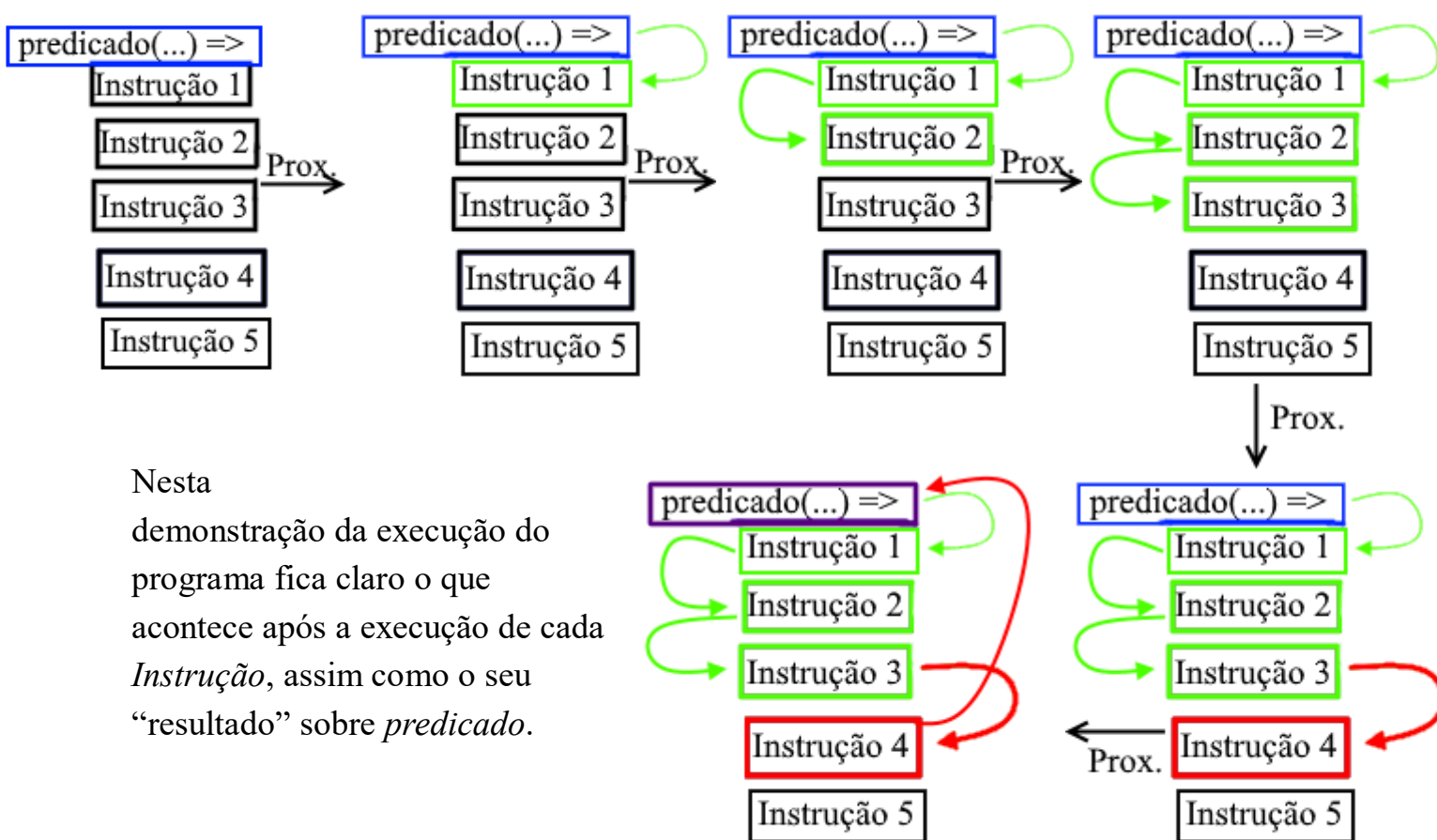
Outra coisa importante, por serem proposições lógicas instruções em Picat estão todas separadas por conjunções (vírgulas) ou disjunções (ponto e vírgula), o que faz com que uma instrução falhar ou suceder afete todas as outras.

Digamos que esta imagem cujo texto está muito bem alinhado representa um predicado qualquer não “*retrocessível*”.

E digamos que as caixas verdes representam uma Instrução que sucedeu, as vermelhas uma que não sucedeu, as pretas são instruções que não foram executadas ainda, as azuis são predicados que estão sendo executados, as roxas são predicados que não sucederam, e verde escuro são predicados que sucederam.

Como podem ver a *Instrução 4* que, digamos, era dependente de um fato não sucedeu, por algum motivo qualquer que não nos interessa, o que causou com que *predicado* também falhasse, já que *predicado* não é “retrocessível” e suas instruções são separadas por conjunções sua execução parou neste ponto.

Analizando desde o início a execução deste predicado, temos algo como o seguinte:



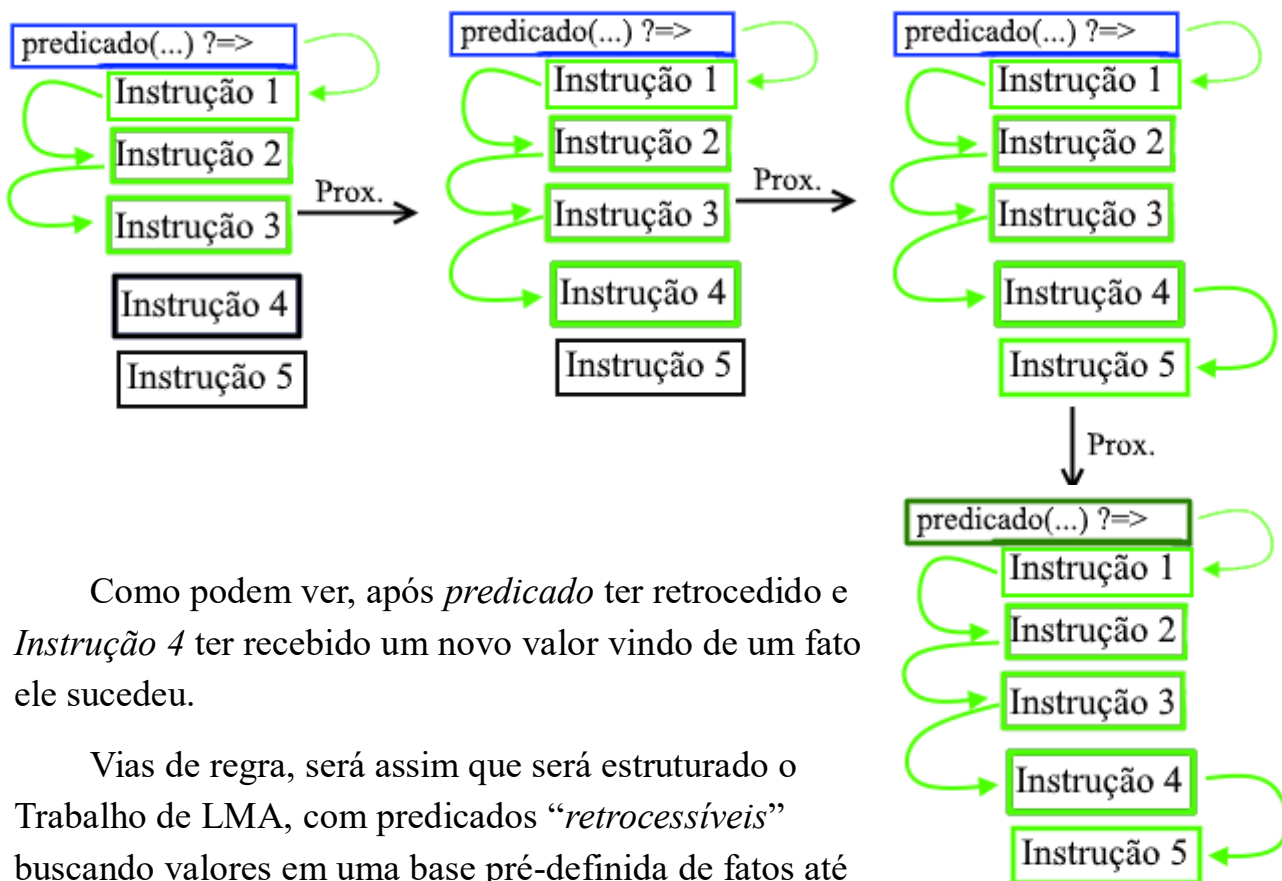
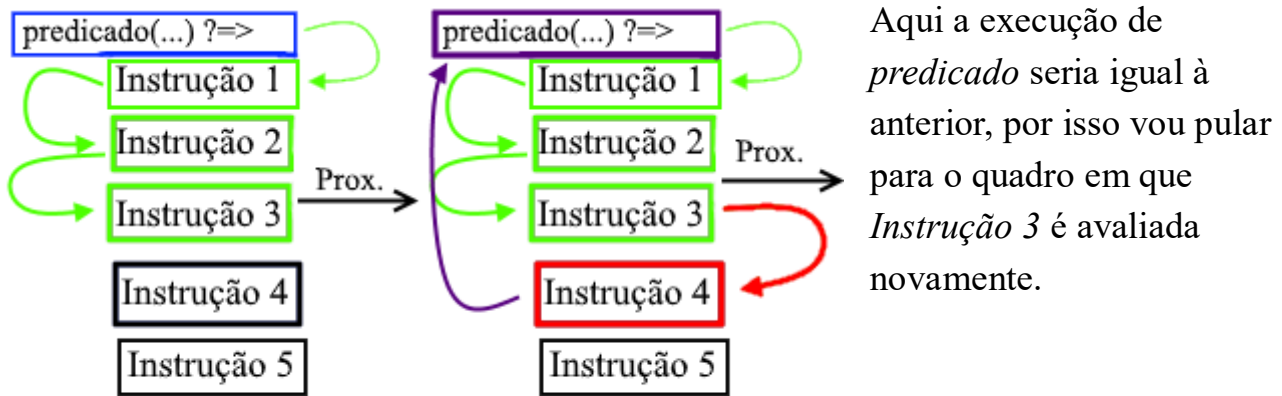
Como queremos que *predicado* não falhe temos que tomar alguma medida para que isso ocorra, esta medida é o *backtracking*.⁸

Caso *predicado* fosse “retrocessível”, ao chegar em *Instrução 4*, *predicado* seria “refeito”, ou seja, sua execução recomençaria a partir de *Instrução 1*, ao ser feito

⁸ Gostaria de deixar claro que backtracking não vai resolver todos os seus problemas, se o código que escreveu está errado ele não vai ser magicamente arrumado porque está usando backtracking.

backtracking a coisa mais importante que ocorrerá é que todas as variáveis que foram instanciadas a partir de um fato serão *reinstanciadas* para um valor diferente, até que não haja mais valores para serem instanciados.

Agora, ainda tomando que *Instrução 4* era dependente de um fato, vamos reavaliar *predicado*, só que desta vez com ele sendo “*retrocessível*”.



Como podem ver, após *predicado* ter retrocedido e *Instrução 4* ter recebido um novo valor vindo de um fato ele sucedeu.

Vias de regra, será assim que será estruturado o Trabalho de LMA, com predicados “*retrocessíveis*” buscando valores em uma base pré-definida de fatos até que seja encontrada a solução desejada.

4.2 Recursão

Recursão, ao contrário de *backtracking*, é algo que está presente em muitas outras linguagens de programação (incluindo C) assim como na própria matemática, e é muito mais simples de entender que *backtracking*.

Em Picat, tanto funções quanto predicados podem ser recursivos.

Uma função ou predicado é considerado recursivo se ele chama a si próprio, quando é feita uma chamada recursiva de uma função/predicado a execução do inicial é “pausada” até a chamada recursiva ser resolvida, sendo que dentro de uma chamada recursiva podem ser feitas outras chamadas recursivas, pois uma chamada de uma função/predicado recursiva é considerada uma chamada normal pelo interpretador.

Uma característica muito importante de funções/predicados recursivos é que eles (quase) sempre possuem uma regra de aterramento (também chamada de condição de parada), ou seja, uma regra que diz quando a recursão deve parar, sem isso serão feitas infinitas chamadas recursivas.

Seguem alguns exemplos de predicados e funções recursivas.

```
fatP(0, NumF) => NumF = 1.
fatP(Num, NumF) =>
    fatP(Num-1, NumF2),
    NumF = Num * NumF2
.
fatF(0) = 1.
fatF(N) = NF => NF = N * fat(N-1).
% <- Isso é o símbolo de um comentário
% Pode ser escrito desta forma mais simples:
% fatF(N) = N * fatF(N-1).
```

Este exemplo mostra como calcular o fatorial de um número N qualquer, o seguinte mostra como calcular o N-ésimo termo da sequência de Fibonacci.

Lembrando que $N! = N * (N-1) * (N-2) * \dots$ onde $0! = 1$, e a sequência de Fibonacci é a seguinte: 0, 1, 1, 2, 3, 5, 8, 13, ... definida pela regra:

$$F(n) = F(n-1) + F(n-2), F(0) = 0 \text{ e } F(1) = 1.$$

fibP(0, *Fb*) => *Fb* = 0.

fibP(1, *Fb*) => *Fb* = 1.

fibP(*X*, *Fb*) =>

fibP(*X*-1, *Fb1*),

fibP(*X*-2, *Fb2*),

Fb = *Fb1* + *Fb2*

.

fibF(0) = 0.

fibF(1) = 1.

fibF(*N*) = *fibF*(*N*-1) + *fibF*(*N*-2).

zero_infinito() => *println*(0), *zero_infinito*().

Este ultimo exemplo mostra um predicado recursivo sem regra de aterramento, neste caso ele ficara imprimindo zeros na tela infinitamente.

5. Estrutura de um Programa

Em Picat um programa é estruturado como uma sequência de proposições lógicas (no caso, as instruções são proposições), porém sem a noção de sequencialidade de instruções que existe em Lógica Proposicional; ao invés disso, há noções de conhecimento declarativo e procedimental, ou seja, domínios que contém conhecimentos base e procedimentos que definem regras sobre os conhecimentos.

Devido a falta da necessidade de as instruções seguirem uma sequência obrigatória para se obter um resultado você pode sequenciar seu programa do modo que quiser, porém saiba que dependendo de como é o problema que está tentando resolver, certas sequências de avaliações serão mais eficientes que outras. Vias de regras, para determinar o que deve ser avaliado primeiro tem que saber quais informações dependem de quais, sabendo isso, aquelas que são pré-requisito comum há muitas outras deveria ser sua prioridade.

5.1 Avaliações Lógicas

Proposições são avaliadas e atribuídas um valor lógico, verdadeiro ou falso caso, respectivamente, tenham sucedido ou falhado, a maioria das coisas que farão deverão sempre retornar verdadeiro (prints, atribuições, particularizações, e coisas assim), mas coisas como comparações e avaliações podem (e em muitos casos vão) falhar, sendo que muitas vezes elas deverão falhar para que você obtenha o resultado desejado.

5.1.1 Tautologia, Contradição, Conjunção e Disjunção

Há fatos especiais que são sempre verdadeiros ou sempre falsos, são eles o *true* ou *yes*, que são sempre verdadeiros e o *false* ou *no*, que são sempre falsos, estes fatos especiais são bastante úteis para controlar o fluxo do programa.

A conjunção é um operador lógico especial que é usado para separar proposições em um programa, como deveriam saber, a conjunção só é verdadeira se ambos os termos avaliados forem verdadeiros.

Um programa comum em Picat é feito de modo que as principais proposições são separadas por conjunções, isso se dá principalmente para tornar claro qualquer erro que exista, pois quando ocorre um erro com uma proposição ela será tida como falsa, o que falharia o seu predicado principal.

A disjunção é outro operador lógico especial que também é usado para separar as proposições de um programa, como deveriam saber, a disjunção é verdadeira se pelo menos um dos seus termos avaliados for verdadeiro.

A disjunção é muito mais utilizada em avaliações do que para separar proposições, por exemplo, para encontrar alguma regra/relação verdadeira sobre termos de um domínio.

5.2 Corpo de um Programa

Seus programas geralmente serão estruturados seguindo uma variação deste modelo seguinte:

```
/* algum comentário */  
  
index(-)  
  
fato(valor).  
fato(valor).  
fato(valor).  
fato(valor).  
  
% outro comentário  
  
predicado(Argumentos) ?=>  
    fato(X),  
    alguma_coisa_usando_X,  
    alguma_coisa  
  
.  
  
main() ?=>  
    predicado(Argumentos),  
    alguma_coisa  
  
.
```

Ou seja, será estabelecido um domínio de fatos sobre os quais irão trabalhar, serão criado(s) predicado(s) para trabalhar sobre este domínio, e (opcionalmente) será criado um predicado principal (geralmente chamado de *main*) onde será feita a chamada dos predicados que estabeleceu.

Sendo que não necessariamente o domínio precisa ser declarado no começo do programa, ele pode muito bem ser declarado no final do programa, contanto que siga as regras de declaração já mencionadas.

Assim como não necessariamente é preciso que sejam criados vários predicados para trabalhar sobre o domínio, um predicado pode muito bem ser o suficiente para seu programa, sendo que muitas vezes esse predicado é o próprio *main*.

Outra coisa que podem (devem) fazer é comentar seus códigos, algo que não só vai ajudar (imensamente) você a entender o que já fez e o que deveria fazer, assim como vai ajudar outros⁹ a entender o que fez até agora.

5.3 Exemplos de Trabalhos Antigos

Neste tópico mostrarei alguns exemplos de trabalhos finais do meu semestre, todos os arquivos que vou mencionar estão no *Github* do professor Cláudio.

Primeiramente, o *pdf* com os enunciados dos problemas que serão mostrados está aqui.

Segundamente, mostrarei a resolução do exercício 1, que se chama Campeonato de Boliche.

Obs.: Vejam todos os trabalhos antigos que estão disponíveis no *Github* do professor; eu não vou poder explicar cada menor detalhe presente neste trabalho que eu peguei de exemplo assim como este trabalho que escolhi para ser exemplo pode não apresentar todos os menores detalhes que possam ser relevantes para o trabalho final.

⁹ Incluindo (Principalmente) os Professores

5.3.1 Resolução do Exercício

Este exercício lhe dá uma série de fatos sobre 5 equipes de boliche e seu trabalho é relacionar estes fatos para descobrir quem são as equipes, algo que soa bastante simples e, de fato, este era o exercício mais fácil dos 2 desse tipo.

Não irei mostrar todo o código, somente as partes relevantes para esta explicação.

É muito importante vocês entenderem que quando há uma chamada do tipo *fato(Var)* a variável *Var* vai receber um valor do domínio *fato*, e que uma comparação do tipo *Var == valor* está verificando se *Var* foi particularizada para *valor*, sendo que *valor* veio de *fato*.

Essas particularizações e comparações serão repetidas iterativamente até que se encontre uma sequência de valores que simultaneamente satisfazem todas as condições propostas, esta sequência deveria ser a resposta do problema.

A primeira coisa a ser notada é que há um comentário no início do programa com a resposta do problema e o tempo que foi necessário para esta resposta ser processada, é muito útil ter a resposta do problema antes de tentar programar uma resolução, pois isso irá tornar mais fácil pensar em um algoritmo para resolver o problema.

```
/* Saída:  
CPU time 1.604 seconds.  
Equipe 1:  
amarelo peraltas arthur 0 27 420  
Equipe 2:  
vermelho fenix adriano 1 23 410  
Equipe 3:  
branco catorzebiz lucas 3 25 400  
Equipe 4:  
verde tratores rogerio 4 26 390  
Equipe 5:  
azul supinos filipe 2 24 380  
*/
```

Após este comentário há a declaração dos fatos e a declaração de um predicado, de *aridade* 5, que contém 5 *tuplas*, cada uma com 6 elementos, que são as características da equipe que serão calculadas nesse predicado.

Segue abaixo uma versão reduzida da declaração dos fatos, e a declaração do predicado.

```
index(-)
uniforme(amarelo).
...
index(-)
equipe(catorzebiz).
...
index(-)
capitao(arthur).
...
index(-)
strikes(0).
...
index(-)
idade(23).
...
index(-)
pontuacao(380).
...

boliche((U1, E1, C1, S1, I1, P1),
        (U2, E2, C2, S2, I2, P2),
        (U3, E3, C3, S3, I3, P3),
        (U4, E4, C4, S4, I4, P4),
        (U5, E5, C5, S5, I5, P5)) ?=>
```

Logo após há uma particularização de valores a partir de um fato, em que as variáveis U1, U2, ... recebem valores vindos do fato *uniforme()*, há uma chamada de um predicado que verifica se todas as variáveis receberam valores diferentes, e, por último, são comparadas com outras variáveis para encontrar um valor que satisfaça a condição proposta.

```
uniforme(U1),uniforme(U2),uniforme(U3),uniforme(U4),uniforme(U5),
all_different([U1, U2, U3, U4, U5]),

% A equipe de Branco está exatamente à esquerda da equipe de Verde.
(
(U1 == branco, U2 == verde);
(U2 == branco, U3 == verde);
(U3 == branco, U4 == verde);
(U4 == branco, U5 == verde)
),

% As equipes verde e azul estão lado a lado.
(
(U1 == verde, U2 == azul);
(U2 == verde, U3 == azul);
(U3 == verde, U4 == azul);
(U4 == verde, U5 == azul);
(U2 == verde, U1 == azul);
(U3 == verde, U2 == azul);
(U4 == verde, U3 == azul);
(U5 == verde, U4 == azul)
),
```

Veja que há uma separação do que está sendo avaliado em “blocos”, onde cada bloco serve para calcular a condição que é explicitada no comentário.

Note também que todas as comparações estão dentro de parênteses separadas entre si por ; (disjunções), e que todos estes “mini blocos” de comparações estão dentro de um parêntese maior, que engloba todas elas.

Isto é deste modo pois o que você está procurando no domínio que está trabalhando é uma série de valores que satisfaça pelo menos uma daquelas condições. Tomando como exemplo a condição “As equipes Verde e Azul estão lado a lado”, temos:

```
(  
  A equipe 1 é a verde e a equipe 2 é a azul  
  OU  
  A equipe 2 é a verde e a equipe 3 é a azul  
  OU  
  A equipe 3 é a verde e a equipe 4 é a azul  
  OU  
  ...  
) , E  
  A próxima coisa que eu vou avaliar
```

Neste bloco você vai avaliar se pelo menos há um par de variáveis (U_n, U_m) que satisfaz a condição de que uma delas recebeu o valor “verde” e outra recebeu o valor “azul” em uma ordem específica (nesse caso a ordem seria $n = m-1$ ou $m = n-1$).

Note que aqui as variáveis são comparadas somente com elas mesmas, pois estas são as únicas variáveis que foram particularizadas até agora, portanto as únicas que possuem algum valor atribuído a elas.

Após isso há novamente uma particularização a partir de um fato, chamada de um predicado para verificar diferença entre as variáveis e comparações entre as variáveis.

```
pontuacao(P1),pontuacao(P2),pontuacao(P3),pontuacao(P4),pontuacao(P5),  
all_different([P1, P2, P3, P4, P5]),
```

```
% Na segunda posição está a equipe com 410 pontos.
```

```
P2 == 410,
```



```
% A equipe com 390 pontos está exatamente à esquerda da equipe Azul.  
(  
  (P1 == 390, U2 == azul);  
  (P2 == 390, U3 == azul);  
  (P3 == 390, U4 == azul);  
  (P4 == 390, U5 == azul)  
)
```

Neste bloco há duas coisas interessantes para serem observadas; primeiro na condição “Na segunda posição está a equipe com 410 pontos” temos que a variável *P2* sempre deverá receber o valor “410”, caso isso não ocorra esta iteração automaticamente não vai gerar a resposta correta.

Segundo vejam que na condição “A equipe com 390 pontos está exatamente à esquerda da equipe Azul” há comparações entre variáveis que são particularizadas a partir de fatos diferentes, e há também um pequeno erro; na segunda comparação deste bloco temos:

```
(P2 == 390, U3 == azul);  
A equipe 2 tem 390 pontos e a equipe 3 é a azul
```

Isso não está tecnicamente errado, mas também não está tecnicamente certo, pois é uma comparação redundante, já que a partir da regra anterior temos o conhecimento de que a equipe 2 é a equipe com 410 pontos, portanto é impossível que esta seja a equipe com 390 pontos.

Além disso podemos inferir um conhecimento bastante útil sobre estas duas regras: Já que a equipe 2 tem 410 pontos e a equipe de uniforme azul está à direita da equipe com 390 sabemos que a equipe 3 não está de uniforme azul.

Como podemos ver pela resposta do problema no início do código vemos que essa inferência está de fato certa.

Agora inserindo esse novo conhecimento no código temos algo tipo isso:

```

pontuacao(P1),pontuacao(P2),pontuacao(P3),pontuacao(P4),pontuacao(P5),
all_different([P1, P2, P3, P4, P5]),

% Na segunda posição está a equipe com 410 pontos.
P2 == 410,

/* Como a segunda equipe tem 410 pontos e a equipe a direita da equipe
com 390 é a equipe de uniforme azul sabemos que a equipe 3 não pode
estar usando uniforme azul*/
U3 != azul,

% A equipe com 390 pontos está exatamente à esquerda da equipe Azul.
(
(P1 == 390, U2 == azul);
(P3 == 390, U4 == azul);
(P4 == 390, U5 == azul)
),

```

Outra possibilidade é agrupar este conhecimento com as regras sobre os uniformes das equipes, o que tornaria o código notavelmente mais eficiente, pois todas as iterações em que U3 recebesse o valor ‘azul’ seriam imediatamente descartadas.

Logo temos o seguinte resultado:

```

uniforme(U1),uniforme(U2),uniforme(U3),uniforme(U4),uniforme(U5),
all_different([U1, U2, U3, U4, U5]),

/* Como a segunda equipe tem 410 pontos e a equipe a direita da equipe
com 390 é a equipe de uniforme azul sabemos que a equipe 3 não pode
estar usando uniforme azul*/
U3 != azul,

```

```
% A equipe de Branco está exatamente à esquerda da equipe de Verde.
```

```
(  
(U1 == branco, U2 == verde);  
(U2 == branco, U3 == verde);  
(U3 == branco, U4 == verde);  
(U4 == branco, U5 == verde)  
) ,
```

```
% As equipes verde e azul estão lado a lado.
```

```
(  
(U1 == verde, U2 == azul);  
(U3 == verde, U4 == azul);  
(U4 == verde, U5 == azul);  
(U2 == verde, U1 == azul);  
(U3 == verde, U2 == azul);  
(U5 == verde, U4 == azul)  
) ,
```

Veja que além de somente adicionar a condição que o valor de U3 seja diferente de ‘azul’, podemos remover duas comparações dentro do bloco “As equipes verde e azul estão lado a lado”, pois são comparações que assumem U3 sendo equivalente a ‘azul’.

Os próximos blocos não apresentam nada que eu já não tenha explicado ainda, por isso vou pular para o próximo bloco que apresente algo que não tenha sido explicado.

```

equipe(E1),equipe(E2),equipe(E3),equipe(E4),equipe(E5),
all_different([E1, E2, E3, E4, E5]),

% A equipe Vermelha está em algum lugar entre a equipe do capitão mais
velho e a equipe com mais Strikes, nessa ordem.

(
(I1 == 27, S3 == 4, (U2 == vermelho));
(I1 == 27, S4 == 4, (U2 == vermelho ; U3 == vermelho));
(I1 == 27, S5 == 4, (U2 == vermelho ; U3 == vermelho ; U4 == vermelho));
(I2 == 27, S4 == 4, (U3 == vermelho));
(I2 == 27, S5 == 4, (U3 == vermelho ; U4 == vermelho));
(I3 == 27, S5 == 4, (U4 == vermelho))
)
.

```

Neste bloco podemos ver que são feitas múltiplas avaliações em uma linha, pois a condição que é avaliada define uma regra que depende de 3 fatos simultaneamente; reescrevendo este bloco usando uma sintaxe mais próxima a linguagem natural temos:

```

% A equipe Vermelha está em algum lugar entre a equipe do capitão mais
velho e a equipe com mais Strikes, nessa ordem.

(
(O primeiro capitão tem 27 anos E a terceira equipe tem 4 strikes E a
segunda equipe está de vermelho) OU
(O primeiro capitão tem 27 anos E a quarta equipe tem 4 strikes E (a
segunda equipe está de vermelho OU a terceira equipe está de vermelho))
OU
(O primeiro capitão tem 27 anos E a quinta equipe tem 4 strikes E (a
segunda equipe está de vermelho OU a terceira equipe está de vermelho OU
a quarta equipe está de vermelho)) OU
(O segundo capitão tem 27 anos E a quarta equipe tem 4 strikes E a
terceira equipe está de vermelho) OU

```

(O segundo capitão tem 27 anos E a quinta equipe tem 4 strikes E (a terceira equipe está de vermelho OU a quarta equipe está de vermelho))
OU

(O terceiro capitão tem 27 anos E a quinta equipe tem 4 strikes E a quarta equipe está de vermelho)

)

.

Percebam que este bloco não é procedido por uma , (conjunção) mas sim por um ponto, pois esta é a última proposição deste predicado.

Este processo de particularizar por fato, checar equivalência com o valor esperado e comparar até achar uma combinação de variáveis que resolva o problema é, em essência, tudo que é feito nesse trabalho.

Não por isso significa que vai ser fácil.

6. Referências

Na montagem desse guia os principais recursos que usei foram retirados do Site da Linguagem, principalmente na Aba de Recursos, e do GitHub do Professor Cláudio, principalmente na Pasta de Picat.

Dentro no site da linguagem, além de serem encontrados vários recursos diversos como problemas resolvidos, na aba de recursos, podem encontrar o Guia Oficial da Linguagem (em inglês), que contém (literalmente) tudo sobre a linguagem, algo interessante para os curiosos, mas muito abrangente e genérico caso somente queira entender a linguagem para terminar o trabalho.

No GitHub do professor, encontrarão muitos e muitos problemas resolvidos, resoluções de trabalhos passados, como o problema Campeonato de Boliche que já havia mostrado aqui, e outros trabalhos diversos que o professor tenha disponibilizado.