# A User's Guide to Picat

**Version 1.X**

**Neng-Fa Zhou and Jonathan Fruhman**

**Last updated Dec. 12, 2015**

# Chapter 1

# How to Use the Picat System

## 1.1 How to Use the Picat Interpreter

The Picat system is written in both C and Picat. The Picat interpreter is provided as a single standalone executable file, named `picat.exe` for Windows and `picat` for Unix. The Picat interpreter provides an interactive programming environment for users to compile, load, debug, and execute programs. In order to start the Picat interpreter, users first need to open an OS terminal. In Windows, this can be done by selecting `Start->Run` and typing `cmd` or selecting `Start->Programs->Accessories->Command Prompt`. In order to start the Picat interpreter in any working directory, the environment variable `path` must be properly set to contain the directory where the executable is located.

### 1.1.1 How to Enter and Quit the Picat Interpreter

The Picat interpreter is started with the OS command `picat`.

*OSPrompt* `picat`

where *OSPrompt* is the OS prompt. After the interpreter is started, it responds with the prompt `Picat>`, and is ready to accept queries.

Once the interpreter is started, users can type a query after the prompt. For example,

```
Picat> X=1+1
X=2
Picat> printf("hello"++" picat")
hello picat
```

The `halt` predicate, or the `exit` predicate, terminates the Picat interpreter. An alternative way to terminate the interpreter is to enter `ctrl-d` (control-d) when the cursor is located at the beginning of an empty line.

### 1.1.2 How to Run a Program Directly

The `picat` command can also be used to run a program directly. Consider the following program:

```
main =>
    printf("hello picat%n").

main(Args) =>
    printf("hello picat"),
    foreach(Arg in Args)
        printf(" %s",Arg)
    end,
    nl.
```

Assume the program is stored in a file named `hello.pi` in the directory `C:\work`. The following shows two runs, one executing `main/0` and the other executing `main/1`.

```
C:\work> picat hello.pi
hello picat

C:\work> picat hello.pi 6 12 2013
hello picat 6 12 2013
```

If the command line contains arguments after the file name, then `main/1` is executed and all the arguments are passed to the predicate as a list of strings.

In general, the `picat` command takes the following form:

`picat [-path` *Directories*`][-log] [-g` *InitGoal*`]` *File* $Arg_1$ $Arg_2$ $\ldots$ $Arg_n$

*Directories* is a semicolon-separated and double-quoted list of directories, and will be set as the value of the environment variable `PICATPATH` before the execution of the program. The Picat system will look for the file and the related modules in these directories. The option `-log` makes the system print log information and warning messages. The option `-g` makes Picat execute a specified initial query *InitGoal* rather than the default `main` predicate. The file name *File* can have the extension `.pi`, and can contain a path of directories.

### 1.1.3 How to Use the Command-line Editor

The Picat interpreter uses the `getline` program written by Chris Thewalt. The `getline` program memorizes up to 100 of the most recent queries that the users have typed, and allows users to recall past queries and edit the current query by using Emacs editing commands. The following gives the editing commands:

`ctrl-f` Move the cursor one position forward.
`ctrl-b` Move the cursor one position backward.
`ctrl-a` Move the cursor to the beginning of the line.
`ctrl-e` Move the cursor to the end of the line.
`ctrl-d` Delete the character under the cursor.
`ctrl-h` Delete the character to the left of the cursor.
`ctrl-k` Delete the characters to the right of the cursor.
`ctrl-u` Delete the whole line.
`ctrl-p` Load the previous query in the buffer.
`ctrl-n` Load the next query in the buffer.

Note that the command `ctrl-d` terminates the interpreter if the line is empty and the cursor is located in the beginning of the line.

Neng-Fa: I REARRANGED JUST THIS SECTION ...it is signaled the the end!!!

### 1.1.4 How to Compile and Load Programs

A Picat program is stored in one or more text files with the extension name `pi`. A file name is a string of characters. Picat treats both `'/'` and `'\'` as file name separators. Nevertheless, since `'\'` is used as the escape character in quoted strings, two consecutive backslashes must be used, as in `"c:\\work\\myfile.pi"`, if `'\'` is used as the separator.

A program first needs to be compiled and loaded into the system before it can be executed. So, there are some predicates to do these taskies and others such:

**Predicate `cl`:** The built-in predicate `cl(`$FileName$`)` compiles and loads (**I think that should be more *natural* and logic: loads and compiles**) the source file named $FileName$`.pi`. Note that if the full path of the file name is not given, then the file is assumed to be in the current working directory. Also note that users do not need to give the extension name.

Consider the following source-code as example, where its name is $wellcome$`.pi`:

```
main =>
       printf(" Wellcome to PICAT's world! \n ").
```

Once in Picat's environment has started, we have:

```
Picat> cl(wellcome).
Compiling:: wellcome.pi
wellcome.pi compiled in 4 milliseconds
loading...

yes

Picat> main.
 Wellcome to PICAT's world!

yes

Picat>
```

The system compiles and loads (**Again, I think that should be more *natural* and logic: loads and compiles**) not only the source file $FileName$`.pi`, but also all of the module files that are either directly imported or indirectly imported by the source file. The system searches for such dependent files in the directory in which $FileName$`.pi` resides or the directories that are stored in the environment variable `PICATPATH`. For $FileName$`.pi`, the `cl` command loads the generated byte-codes without creating a byte-code file.

The built-in predicate `cl` (with no argument) compiles and loads a program from the console, ending when the end-of-file character (`ctrl-z` for Windows and `ctrl-d` for Unix) is typed.

(**Proposition: change the name cl to lc, again, I think that should be more *natural* and logic: loads and compiles**)

**Predicate `compile`:** The built-in predicate `compile(`*FileName*`)` compiles the file *FileName*`.pi` and all of its dependent module files without loading the generated byte-code files. The destination directory for the byte-code file is the same as the source file's directory. If the Picat interpreter does not have permission to write into the directory in which a source file resides, then this built-in throws an exception.

Example:

```
Picat> compile(wellcome).
Compiling::wellcome.pi
wellcome.pi compiled in 4 milliseconds

yes

Picat>
```

In the sequence, please check in current directory by the *FileName*`.qi` file. Again, in Unix and Linux systems:

```
$ ls -al
-rwxr-xr-x  1 *** ***   67 Jan 31 18:59 wellcome.exe
-rw-r--r--  1 *** ***   57 Jan 30 17:31 wellcome.pi
-rw-r--r--  1 *** ***  449 Jan 31 19:10 wellcome.qi
```

The file *wellcome*`.qi` generated is in byte-code.

**Predicate `load`:** The built-in predicate `load(`*FileName*`)` loads the byte-code file *FileName*`.qi` and all of its dependent byte-code files. For *FileName* and its dependent file names, the system searches for a byte-code file in the directory in which *FileName*`.qi` resides or the directories that are stored in the environment variable `PICATPATH`.

Summarizing, where the byte-code is loaded istead of source-code, example :

```
Picat> load(wellcome).
loading...wellcome.qi

yes

Picat>
```

If the byte-code file *FileName*`.qi` does not exist but the source file *FileName*`.pi` exists, then this built-in compiles the source file and loads the byte codes without creating a `qi` file.

In a short sequence of steps:

```
Picat> compile(wellcome).
Compiling::wellcome.pi
wellcome.pi compiled in 0 milliseconds
```

```
yes

Picat> load(wellcome).
loading...wellcome.qi

yes

Picat> main.
 Wellcome to PICAT's world!

yes

Picat>
```

### Setting up the `PICATPATH` variable

For Unix and Linux users the variable PICATPATH can be defined in settings system files such: `profile`, `.profile` (local user), `bashrc`, etc. For this task, please add these two lines in one of these files:

```
PICATPATH=/usr/local/share/Picat/
export PICATPATH
```

To verify if this variable has been defined in your system, check it:

```
$echo $PICATPATH
/usr/local/share/Picat/
```

PS: the '`$`' is a prompt defined of a console in Unix and Linux systems.

In this case, the Picat was installed in /usr/local/share/Picat/ and a symbolic link was created in /usr/bin. For this case, see the example:

```
$ln -s /usr/local/share/Picat/picat /usr/bin/picat
$ls -al /usr/bin/picat
lrwxrwxrwx 1 root root 28 Jan 30 17:20 /usr/bin/picat -> /usr/local/share/Pi
```

### Generating a standalone code

The generation of a runtime code is not ready yet. For Unix and Linux users, this details can be solved using a small script. For OS and Windows systems something similar can be done. Please, consider a script named *wellcome*`.exe` as a example and its contents:

```
#!/bin/bash
picat wellcome.pi
echo "Finished!!!"
```

Now, you should give a execution permission for this script:

```
$ chmod 755 wellcome.exe
```

Finally, its execution in console command such:

```
$ ./wellcome.exe
 Wellcome to PICAT’s world!
 Finished!!!
```

The similar idea is found it in others operating systems.

Neng-Fa: ............UP TO HERE.......
finish!

### 1.1.5 How to Run Programs

After a program is loaded, users can query the program. For each query, the system executes the program, and reports yes when the query succeeds and no when the query fails. When a query that contains variables succeeds, the system also reports the bindings for the variables. Users can ask the system to find the next solution by typing ';' after a solution. For example,

```
Picat> member(X,[1,2,3])
X=1;
X=2;
X=3;
no
```

Users can force a program to terminate by typing ctrl-c, or by letting it execute the built-in predicate abort. Note that when the system is engaged in certain tasks, such as garbage collection, users may need to wait for a while in order to see the termination after they type ctrl-c.

## 1.2 How to Use the Debugger

The Picat system has three execution modes: *non-trace mode*, *trace mode*, and *spy mode*. In trace mode, it is possible to trace the execution of a program, showing every call in every possible stage. In order to trace the execution, the program must be recompiled while the system is in trace mode. In spy mode, it is possible to trace the execution of individual functions and predicates that are *spy points*. When the Picat interpreter is started, it runs in non-trace mode. The predicate debug or trace changes the mode to trace. The predicate nodebug or notrace changes the mode to non-trace.

In trace mode, the debugger displays execution traces of queries. An *execution trace* consists of a sequence of call traces. Each *call trace* is a line that consists of a stage, the number of the call, and the information about the call itself. For a function call, there are two possible stages: Call, meaning the time at which the function is entered, and Exit, meaning the time at which the call is completed with an answer. For a predicate call, there are two additional possible stages: Redo, meaning a time at which execution backtracks to the call, and Fail, meaning the time at which the call is completed with a failure. The information about a call includes the name of the call, and the arguments. If the call is a function, then the call is followed by = and ? at the Call stage, and followed by = $Value$ at the Exit stage, where $Value$ is the return value of the call. Consider, for example, the following program:

```
p(X) ?=> X=a.
p(X) => X=b.
q(X) ?=> X=1.
q(X) => X=2.
```

Assume the program is stored in a file named myprog.pi. The following shows a trace for a query:

```
Picat> debug

{Trace mode}
Picat> cl(myprog)

{Trace mode}
```

```
Picat> p(X),q(Y)
   Call: (1) p(_328) ?
   Exit: (1) p(a)
   Call: (2) q(_378) ?
   Exit: (2) q(1)
X = a
Y = 1 ?;
   Redo: (2) q(1) ?
   Exit: (2) q(2)
X = a
Y = 2 ?;
   Redo: (1) p(a) ?
   Exit: (1) p(b)
   Call: (3) q(_378) ?
   Exit: (3) q(1)
X = b
Y = 1 ?;
   Redo: (3) q(1) ?
   Exit: (3) q(2)
X = b
Y = 2 ?;
no
```

In trace mode, the debugger displays every call in every possible stage. Users can set *spy points* so that the debugger only shows information about calls of the symbols that users are spying. Users can use the predicate

spy $\$Name/N$

to set the functor $Name$/$N$ as a spy point, where the arity $N$ is optional. If the functor is defined in multiple loaded modules, then all these definitions will be treated as spy points. If no arity is given, then any functor of $Name$ is treated as a spy point, regardless of the arity.

After displaying a call trace, if the trace is for stage `Call` or stage `Redo`, then the debugger waits for a command from the users. A command is either a single letter followed by a carriage-return, or just a carriage-return. See Appendix **??** for the debugging commands.