



Picat Tutorial

Neng-Fa Zhou & Jonathan Fruhman



What Is Picat?

- Why the name “PICAT”?
 - Pattern-matching, Intuitive, Constraints, Actors, Tabling
- Core logic programming concepts
 - Logic variables (arrays and maps are terms)
 - Implicit pattern-matching and explicit unification
 - Explicit non-determinism
- Language constructs for scripting and modeling
 - Functions, loops, and list comprehension
- Modules for combinatorial search
 - The `cp`, `sat`, and `mip` modules for CSPs
 - The `planner` module for planning



Niche Applications

- Scripting and Modeling
 - ☐ Constraint solving and optimization
 - ☐ Planning
 - ☐ NLP
 - ☐ Knowledge engineering
 - ☐ Complex data processing
 - ☐ Web services
 - ☐ ...



Data Types

- Variables – plain and attributed

`x1 _ _ab`

- Primitive values

- ☐ Integer and float

- ☐ Atom

`x1 ' _ ' '_ab' '$%' '你好'`

- Compound values

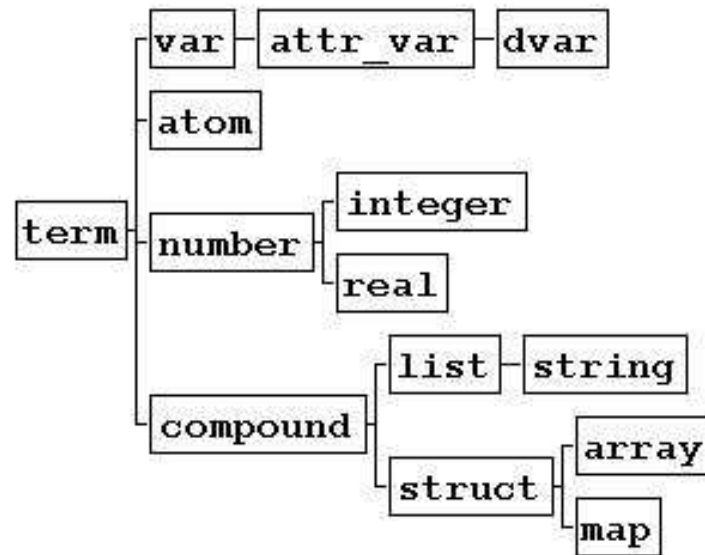
- ☐ List

`[17, 3, 1, 6, 40]`

- ☐ Structure

`$triangle(0.0, 13.5, 19.2)`

The Type Hierarchy





Creating Structures and Lists

- **Generic Structure**

```
Picat> P = new_struct(point, 3)
P = point(_3b0, _3b4, _3b8)
Picat> S = $student(marry,cs,3.8)
```

- **List Comprehension**

```
Picat> L = [E : E in 1..10, E mod 2 != 0]
L = [1,3,5,7,9]
```

- **Range**

```
Picat> L = 1..2..10
L = [1,3,5,7,9]
```

- **String**

```
Picat> write("hello "++"world")
[h,e,l,l,o,', ',w,o,r,l,d]
```

- **Array**

```
Picat> A = new_array(2,3)
A = [{_3d0, _3d4, _3d8}, {_3e0, _3e4, _3e8}]
```

- **Map**

```
Picat> M = new_map([alpha= 1, beta=2])
M = (map)[alpha = 1,beta = 2]
```



Special Structures

- These structures do not need to be preceded by a \$ symbol

- Patterns

- $$p(X+Y) \Rightarrow p(X), p(Y).$$

- Goals

- $$(a, b) \quad (a; b) \quad \text{not } a \quad X=Y$$

- Constraints and Constraint Expressions

- $$X+Y \# = 100 \quad X \# \neq 0 \quad X \# \wedge Y$$

- Arrays

- $$\{2, 3, 5, 7, 11, 13, 17, 19\}$$



Built-ins

```
Picat> integer(2)
yes
Picat> integer(2.0)
no
Picat> real(3.0)
yes
Picat> not real(3.0)
no
Picat> var(X)
yes
Picat> X = 5, var(X)
no
Picat> true
yes
Picat> fail
no
```




Built-ins (Cont.)

```
Picat> X = to_binary_string(5), Y = to_binary_string(13)
X = ['1', '0', '1']
Y = ['1', '1', '0', '1']
```

```
% X is an attributed variable
```

```
Picat> put(X, age, 35), put(X, weight, 205), A = get(X, age)
A = 35
```

```
% X is a map
```

```
Picat> X = new_map([age=35, weight=205]), put(X, gender, male)
X = map([age=35, weight=205, gender=male])
```

```
Picat> S = $point(1.0, 2.0), Name = name(S), Arity = length(S)
Name = point
Arity = 2
```

```
Picat> I = read_int(stdin) % Read an integer from standard input
123
I = 123
```



Index Notation

$X[l_1, \dots, l_n]$: X references a compound value

```
Picat> L = [a,b,c,d], X = L[2]  
X = b
```

```
Picat> S = $student(marry,cs,3.8), GPA=S[3]  
GPA = 3.8
```

```
Picat> A = {{1, 2, 3}, {4, 5, 6}}, B = A[2, 3]  
B = 6
```



List Comprehension

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

```
Picat> L = [X : X in 1..5].  
L = [1,2,3,4,5]
```

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

```
Picat> L = [X : I in 1..5] % X is local  
L = [_bee8,_bef0,_bef8,_bf00,_bf08]
```

```
Picat> X=X, L = [X : I in 1..5] % X is non-local  
L = [X,X,X,X,X]
```

OOP Notation

```
Picat> Y = 13.to_binary_string()  
Y = ['1', '1', '0', '1']
```

```
Picat> Y = 13.to_binary_string().reverse()  
Y = ['1', '0', '1', '1']
```

% X becomes an attributed variable

```
Picat> X.put(age, 35), X.put(weight, 205), A = X.age  
A = 35
```

%X is a map

```
Picat> X = new_map([age=35, weight=205]), X.put(gender, male)  
X = (map)([age=35, weight=205, gender=male])
```

```
Picat> S = $point(1.0, 2.0), Name = S.name, Arity = S.length  
Name = point  
Arity = 2
```

```
Picat> I = math.pi      % module qualifier  
I = 3.14159
```

O.f(t1,...,tn)

-- means module qualified call if O is atom

-- means f(O,t1,...,tn) otherwise.

Explicit Unification $t_1=t_2$

```
Picat> X=1                                ← bind
X=1
Picat> $f(a,b) = $f(a,b)                  ← test
yes
Picat> [H|T] = [a,b,c]                    ← matching
H=a
T=[b,c]
Picat> $f(X,Y) = $f(a,b)                  ← matching
X=a
Y=b
Picat> $f(X,b) = $f(a,Y)                  ← full unification
X=a
Y=b
Picat> X = $f(X)                          ← without occur checking
X=f(f(.....
```

Predicates

- Relation with pattern-matching rules

```
fib(0,F) => F=1.  
fib(1,F) => F=1.  
fib(N,F),N>1 => fib(N-1,F1),fib(N-2,F2),F=F1+F2.  
fib(N,F) => throw $error(wrong_argument,fib,N).
```

- Backtracking (explicit non-determinism)

```
member(X,[Y|_]) ?=> X=Y.  
member(X,[_|L]) => member(X,L).
```

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

- Control backtracking

```
Picat> once(member(X,[1,2,3]))
```

Predicate Facts

| | | |
|-------------------------------|--------|------------------------------------|
| <code>index(+,-) (-,+)</code> | | <code>edge(a,Y) ?=> Y=b.</code> |
| <code>edge(a,b).</code> | | <code>edge(a,Y) => Y=c.</code> |
| <code>edge(a,c).</code> | | <code>edge(b,Y) => Y=c.</code> |
| <code>edge(b,c).</code> | —————> | <code>edge(c,Y) => Y=b.</code> |
| <code>edge(c,b).</code> | | <code>edge(X,b) ?=> X=a.</code> |
| | | <code>edge(X,c) ?=> X=a.</code> |
| | | <code>edge(X,c) => X=b.</code> |
| | | <code>edge(X,b) => X=c.</code> |

- Facts must be ground
- A call with insufficiently instantiated arguments fails
 - `Picat> edge(X,Y)`
`no`



Functions

- Always succeed with a return value

```
power_set([]) = [[]].  
power_set([H|T]) = P1++P2 =>  
    P1 = power_set(T),  
    P2 = [[H|S] : S in P1].
```

```
perm([]) = [[]].  
perm(Lst) = [[E|P] : E in Lst, P in perm(Lst.delete(E))].
```

```
matrix_multi(A,B) = C =>  
    C = new_array(A.length,B[1].length),  
    foreach(I in 1..A.length, J in 1..B[1].length)  
        C[I,J] = sum([A[I,K]*B[K,J] : K in 1..A[1].length])  
end.
```




More on Functions

- Ranges are always functions

`write($f(L..U))` is the same as `Lst=L..U, write($f(Lst))`

- Index notations are always functions

`X[1]+X[2] #= 100` is the same as `X1=X[1], X2=X[2], X1+X2 #= 100`

`write($f(X[I]))` is the same as `Xi=X[I], write($f(Xi))`

- List comprehensions are always functions

`sum([A[I,J] : I in 1..N, J in 1..N]) #= N*N`

is the same as

`L = [A[I,J] : I in 1..N, J in 1..N], sum(L) #= N*N`



Patterns in Heads

- Index notations, ranges, dot notations, and list comprehensions cannot occur in head patterns
- As-patterns

```
merge([], Ys) = Ys.  
merge(Xs, []) = Xs.  
merge([X|Xs], Ys@[Y|_])=[X|Zs], X<Y => Zs=merge(Xs, Ys).  
merge(Xs, [Y|Ys])=[Y|Zs] => Zs=merge(Xs, Ys).
```



Conditional Statements

■ If-then-else

```
fib(N)=F =>  
  if (N=0; N=1) then  
    F=1  
  elseif N>1 then  
    F=fib(N-1)+fib(N-2)  
  else  
    throw $error(wrong_argument,fib,N)$  
  end.
```

■ Prolog-style if-then-else

$(C \rightarrow A; B)$

■ Conditional Expressions

```
fib(N) = cond((N==0;N==1), 1, fib(N-1)+fib(N-2))
```

Assignments

- $X[I_1, \dots, I_n] := \text{Exp}$

Destructively update the component to Exp .
Undo the update upon backtracking.

- $\text{Var} := \text{Exp}$

The compiler changes it to $\text{Var}' = \text{Exp}$ and replace all subsequent occurrences of Var in the body of the rule by Var' .

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



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



Loops

■ Types

- `foreach(E1 in D1, ..., En in Dn) Goal end`
- `while (Cond) Goal end`
- `do Goal while (Cond)`

- Loops provide another way to write recurrences
- A loop forms a name scope: variables that do not occur before in the outer scope are local.
- Loops are compiled into tail-recursive predicates



Scopes of Variables


- Variables that occur within a loop but not before in its outer scope are local to each iteration

```
p(A) =>
  foreach(I in 1 .. A.length)
    A[I] = $node(X)
  end.
```

```
p(A) :-
  noop(X),
  foreach(I in 1 .. A.length)
    A[I] = $node(X)
  end.
```

```
q(L) =>
  L = [X : I in 1 .. 5].
```

```
q(L) =>
  noop(X),
  L = [X : I in 1 .. 5].
```



Loops (ex-1)

```
sum_list(L)=Sum =>  
    S=0,  
    foreach (X in L)  
        S:=S+X  
    end,  
    Sum=S.
```

■ Recurrences

```
S=0  
S1=L[1]+S  
S2=L[2]+S1  
...  
Sn=L[n]+Sn-1  
Sum = Sn
```

■ Query

```
Picat> S=sum_list([1,2,3])  
S=6
```

Loops (ex-2)

```
read_list=List =>
    L=[],
    E=read_int(),
    while (E != 0)
        L := [E|L],
        E := read_int()
    end,
    List=L.
```

■ Recurrences

```
L=[]
L1=[e1|L]
L2=[e2|L1]
...
Ln=[en|Ln-1]
List=Ln
```

■ Query

```
Picat> L=read_list()
1 2 3
L=[3,2,1]
```


Loops (ex-3)

```
read_list=List =>
  List=L,
  E=read_int(),
  while (E != 0)
    L = [E|T],
    L := T,
    E := read_int()
  end,
  L=[ ].
```

■ Recurrences

```
L=[e1|L1]
L1=[e2|L2]
...
Ln-1=[en|Ln]
Ln=[ ]
```

■ Query

```
Picat> L=read_list()
1 2 3
L=[1,2,3]
```



List Comprehensions to Loops

```
List = [(A,X) : A in [a,b], X in 1..2]
```




```
List = L,  
foreach(A in [a,b], X in 1..2)  
    L = [(A,X) | T],  
    L := T  
end,  
L = [].
```



Tabling

- Predicates define relations where a set of facts is implicitly generated by the rules
- The process of fact generation might never end, and can contain a lot of redundancy
- Tabling memorizes calls and their answers in order to prevent infinite loops and to limit redundancy



Tabling (example)

```
table
```

```
fib(0)=1.
```

```
fib(1)=1.
```

```
fib(N)=fib(N-1)+fib(N-2).
```

- Without tabling, $\text{fib}(N)$ takes exponential time in N
- With tabling, $\text{fib}(N)$ takes linear time



Selective Tabling

- A table mode declaration instructs the system on what answers to table
 - `table(M1, M2, ..., Mn)` where M_i is:
 - `+`: input
 - `-`: output
 - `min`: output, corresponding variable should be minimized
 - `max`: output, corresponding variable should be maximized
 - `nt`: not-tabled (only the last argument can be `nt`)
- Selective tabling is useful for dynamic programming problems

Dynamic Programming (examples)

■ Shortest Path

```
table(+, +, -, min)
shortest_path(X, Y, Path, W) =>
    Path = [(X, Y)],
    edge(X, Y, W),
shortest_path(X, Y, Path, W) =>
    Path = [(X, Z) | PathR],
    edge(X, Z, W1),
    shortest_path(Z, Y, PathR, W2),
    W = W1 + W2.
```

■ Knapsack Problem

```
table(+, +, -, max)
knapsack(_, 0, Bag, V) =>
    Bag = [],
    V = 0.
knapsack([_ | L], K, Bag, V), K > 0 ?=>
    knapsack(L, K, Bag, V).
knapsack([F | L], K, Bag, V), K >= F =>
    Bag = [F | Bag1],
    knapsack(L, K - F, Bag1, V1),
    V = V1 + 1.
```



Modules

```
module M.  
import M1,M2,...,Mn.
```

- The declared module name and the file name must be the same
- Files that do not begin with a module declaration are in the global module
- Atoms and structure names are global
- Picat has a global symbol table for atoms, a global symbol table for structure names, and a global symbol table for modules
- Each module has its own symbol table for the public predicates and functions



Modules (Cont.)

- Binding of normal calls to their definitions occurs at compile time
 - The compiler searches modules in the order that they were imported
- Binding of higher-order calls to their definitions occurs at runtime.
 - The runtime system searches modules in the order that they were loaded
- The environment variable `PICATPATH` tells where the compiler or runtime system searches for modules

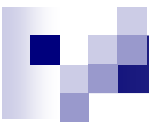


Modules (Cont.)

- No module variables are allowed

Recall that $M.f(\dots)$ stands for $f(M, \dots)$ if M is a variable

- No module-qualified higher-order calls



Modules (example)

```
% In file qsort.pi
module qsort.
sort([]) = [].
sort([H|T]) = sort([E : E in T, E <= H] ++ [H] ++ sort([E : E in T, E > H]).
```

```
% In file isort.pi
module isort.
sort([]) = [].
sort([H|T]) = insert(H, sort(T)).

private
insert(X,[]) = [X].
insert(X,Ys@[Y|_]) = Zs, X<Y => Zs=[X|Ys].
insert(X,[Y|Ys]) = [Y|insert(X,Ys)].
```

```
% another file test_sort.pi
import qsort,isort.
```

```
sort1(L)=S =>
    S=sort(L).
sort2(L)=S =>
    S=qsort.sort(L).
sort3(L)=S =>
    S=isort.sort(L).
```



The planner Module

- Useful for solving planning problems
 - `plan(State,Limit,Plan,PlanCost)`
 - `best_plan(State,Limit,Plan,PlanCost)`
 - ...
- Users only need to define `final/1` and `action/4`
 - `final(State)` is true if `State` is a final state
 - `action(State,NextState,Action,ActionCost)` encodes the state transition diagram
- Uses the early termination and resource-bounded search techniques to speedup search



Ex: The Farmer's Problem

```
import planner.

go =>
    S0=[s,s,s,s],
    best_plan(S0,Plan),
    writeln(Plan).

final([n,n,n,n]) => true.

action([F,F,G,C],S1,Action,ActionCost) ?=>
    Action=farmer_wolf,
    ActionCost = 1,
    opposite(F,F1),
    S1=[F1,F1,G,C],
    not unsafe(S1).

...
```



Constraints

- Picat can be used for constraint satisfaction and optimization problems
- Constraint Problems
 - Generate variables
 - Generate constraints over the variables
 - Solve the problem, finding an assignment of values to the variables that matches all the constraints
- Picat can be used as a modeling language for CP, SAT, LP/MIP
 - Loops are helpful for modeling



Constraints (example)

- SEND + MORE = MONEY

```
import cp.
```

```
go =>
```

```
Vars=[S,E,N,D,M,O,R,Y], % generate variables
Vars :: 0..9,           % define the domains
all_different(Vars),    % generate constraints
S #!= 0,
M #!= 0,
1000*S+100*E+10*N+D+1000*M+100*O+10*R+E
    #= 10000*M+1000*O+100*N+10*E+Y,
solve(Vars),            % search
writeln(Vars).
```



N-Queens Problem

```
import cp.  
  
queens(N) =>  
    Qs=new_array(N),  
    Qs :: 1..N,  
    foreach (I in 1..N-1, J in I+1..N)  
        Qs[I] #!= Qs[J],  
        abs(Qs[I]-Qs[J]) #!= J-I  
    end,  
    solve(Qs),  
    writeln(Qs).
```



Action Rules

- Syntax

Head, Condition, {EventSet} => Action

- **Agent**

$p(X_1, \dots, X_n)$

- **Condition**

- Inline tests (e.g., $\text{var}(X)$, $\text{nonvar}(X)$, $X==Y$, $X>Y$)

- **EventSet**

- $\text{event}(X, O)$ -- a general form event
 - $\text{ins}(X)$ -- X is instantiated
 - $\text{dom}(X, E)$ -- An inner element E of X 's domain is excluded
 - $\text{dom_any}(X, E)$ -- An arbitrary element E is excluded

- **Action**

- Same as a rule body



Applications of AR

- Co-routining and concurrency
 - `freeze(X,Call)` is compiled to AR
- Constraint propagation
 - Constraints in the `cp` module are compiled to AR
 - Users can program problem-specific propagators for global constraints
- Compiling CHR
- Interactive graphical user interfaces

Implementing freeze(X,Goal)

```
freeze(X,q(X,Y))
```



```
freeze_q(X,Y),var(X),{ins(X)} => true.  
freeze_q(X,Y) => q(X,Y).
```



Event-Handling

```
echo(X), {event(X, O)} => writeln(O).
```

```
Picat> echo(X), X.post_event(hello).  
hello
```

```
Picat> echo(X), repeat, X.post_event(hello), nl, fail.  
hello
```

```
hello
```

```
hello
```

```
...
```

Programming Constraint Propagators

■ Maintaining arc consistency for $aX=bY+c$

```
'aX in bY+c_arc'(A,X,B,Y,C),var(X),var(Y),
    {dom(Y,Ey)}
=>
    T = B*Ey+C,
    Ex = T//A,
    (A*Ex==T -> fd_set_false(X,Ex);true).
'aX in bY+c_arc'(A,X,B,Y,C) => true.
```

Whenever an element E_Y is excluded from Y 's domain,
exclude E_Y 's counterpart, E_X , from X 's domain.



Higher-Order Calls

- Functions and predicates that take calls as arguments
- `call(S, A1, ..., An)`
 - Calls the named predicate with the specified arguments
- `apply(S, A1, ..., An)`
 - Similar to `call`, except `apply` returns a value
- `findall(Template, Call)`
 - Returns a list of all possible solutions of `Call` in the form `Template`.
`findall` forms a name scope like a loop.

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

```
Picat> L = findall(X, member(X, [1, 2, 3]))
L = [1,2,3]
```



Higher-Order Functions

`map(_F, []) = [].`

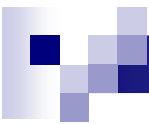
`map(F, [X | Xs]) = [apply(F, X) | map(F, Xs)].`

`map2(_F, [], []) = [].`

`map2(F, [X | Xs], [Y | Ys]) = [apply(F, X, Y) | map2(F, Xs, Ys)].`

`fold(_F, Acc, []) = Acc.`

`fold(F, Acc, [H | T]) = fold(F, apply(F, H, Acc), T).`



Using Higher-Order Calls is Discouraged

- List comprehensions are significantly faster than higher-order calls

- ☐ **X** `map(-, L)`

- ☐ **O** `[-X : X in L]`

- ☐ **X** `map2(+, L1, L2)`

- ☐ **O** `[X+Y : {X,Y} in zip(L1, L2)]`



Global Maps

- `get_heap_map()`
 - Created on the heap after the thread is created
 - Changes are undone when backtracking
- `get_global_map()`
 - Created in the global area when Picat is started
 - Changes are not undone when backtracking
- `get_table_map()`
 - Created in the table area when Picat is started
 - Keys and values are hash-consed
 - Changes are not undone when backtracking



Pros and Cons of Global Maps

■ Pros

- Allows data to be accessed everywhere without being passed as arguments
- Maps returned by `get_global_map()` and `get_table_map()` can be used to store global data that are shared by multiple branches of a search tree
 - Used in the implementation of `minof` and `maxof`.

■ Cons

- Affects locality of data and readability of programs



Global Heap Maps and Global Maps (example)

```
go ?=>
    get_heap_map().put(one,1),
    get_global_map().put(one,1),
    fail.
go =>
    if (get_heap_map().has_key(one)) then
        writef("heap map has key%n")
    else
        writef("heap map has no key%n")
    end,
    if (get_global_map().has_key(one)) then
        writef("global map has key%n")
    else
        writef("global map has no key%n")
    end.
```



Picat Vs. Prolog

- Picat is arguably more expressive

```
qsort([])=[].  
qsort([H|T])=qsort([E : E in T, E<H])++[H]++qsort([E : E in T, E>H]).
```

```
power_set([]) = [[]].  
power_set([H|T]) = P1++P2 =>  
    P1 = power_set(T),  
    P2 = [[H|S] : S in P1].
```

```
matrix_multi(A,B) = C =>  
    C = new_array(A.length,B[1].length),  
    foreach(I in 1..A.length, J in 1..B[1].length)  
        C[I,J] = sum([A[I,K]*B[K,J] : K in 1..A[1].length])  
    end.
```

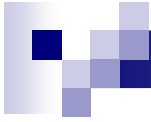


Picat Vs. Prolog

- Picat is more scalable because pattern-matching facilitates indexing rules

`L ::= (“abcd” | “abc” | “ab” | “a”)*`

```
p([a,b,c,d|T]) => p(T).  
p([a,b,c|T]) => p(T).  
p([a,b|T]) => p(T).  
p([a|T]) => p(T).  
p([]) => true.
```



Picat Vs. Prolog

- Picat is arguably more reliable than Prolog
 - Explicit unification and nondeterminism
 - Functions don't fail (at least built-in functions)
 - No cuts or dynamic predicates
 - No operator overloading
 - A simple static module system



Summary

- Picat is a hybrid of LP, FP and scripting
- Picat or Copycat?
 - Prolog (in particular B-Prolog), Haskell, Scala, Mercury, Erlang, Python, Ruby, C-family (C++, Java, C#), OCaml,...
- The first version is available at picat-lang.org
 - Reuses a lot of B-Prolog's code
- Supported modules
 - basic, io, sys, math, os, cp, sat, and util
- More modules will be added



Resources

- Users' Guide

- http://picat-lang.org/download/picat_guide.pdf

- Picat Book

- <http://www.springer.com/us/book/9783319258812>

- Hakan Kjellerstrand's Picat Page

- <http://www.hakank.org/picat/>

- Examples

- <http://picat-lang.org/download/exs.pi>

- Modules

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