My First Look At Picat as a Modeling Language for Constraint Solving and Planning

Håkan Kjellerstrand[1]

1. Independent Researcher, Malmoe, Sweden
E-mail: hakank@gmail.com

Abstract: Picat is a new and interesting programming language, combing many different programming paradigms: logic programming, imperative programming, functional programming, constraint programming, and tabling. This paper is a personal discussion of some of Picat's features with several code examples.

Key Words: Picat, Constraint Programming, Logic Programming, Planning, Prolog

INTRODUCTION

Picat [1] is a new programming language created by Neng-Fa Zhou and Jonathan Fruhman. The specification was published in December 2012. The first official (beta) release was in May 2013.

The main reason I started to try out Picat in May 2013 was that it has support for Constraint Programming (CP), which is a programming paradigm that I am very interested in. Right from the beginning I liked this language – and not just for the support for CP - and in this paper I hope to show why. I should also say that the examples shown here are not just for Constraint Programming, but also traditional backtracking (à la Prolog), tabling (memoization), imperative constructs, etc, i.e. what makes Picat Picat.

The name **Picat** is an acronym and is explained in the quote below, borrowed from the Picat site [1]:

Picat is a general-purpose language that incorporates features from logic programming, functional programming, and scripting languages. The letters in the name summarize Picat's features:

- ***P**attern-matching [...]*
- ***I**mperative [...]*
- ***C**onstraints [...]*
- ***A**ctors [...]*
- ***T**abling [...]*

Of these general features, I have mostly tested constraints, pattern-matching, imperative programming constructs, and tabling, but have not played much with actors directly and will not mention them here.

Let us start with some simple examples to show these features.

All the programs and models in this paper – and many more - are available at my My Picat Page [2] (http://www.hakank.org/picat/).

1 CONSTRAINTS

Here is an example for the inevitable SEND + MORE = MONEY problem (send_more_money.pi), which clearly shows the influences of Prolog:

```
import cp.

sendmore(Digits) =>
   Digits = [S,E,N,D,M,O,R,Y],
   Digits :: 0..9,

   all_different(Digits),
   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(Digits).
```

Except for the `=>` operator, it looks very much like Prolog using Constraint Logic Programming (CLP). I explain the `=>` and how it is different from Prolog's `:-` later on, and will also mention some other similarities/differences from/with Prolog.

Another mandatory example when describing Constraint Programming is Sudoku (sudoku.pi) where we now see more differences between traditional Prolog and Picat in the use of the `foreach` loop:

```
import cp.
import util.

go =>
   time2(sudoku(1)).

sudoku(ProblemName) =>
        problem(ProblemName, Board),
        print_board(Board),
        sudoku(3, Board),
        print_board(Board).

sudoku(N, Board) =>
   N2 = N*N,

   Vars = array_matrix_to_list(Board),
   Vars :: 1..N2,

   foreach(Row in Board) all_different(Row) end,
   foreach(Column in transpose(Board))
```

```
        all_different(Column) end.
   foreach(I in 1..N..N2, J in 1..N..N2)
      all_different([Board[I+K,J+L] : K in 0..N-1, L in 0..N-1])
   end,

   solve([ff,down], Vars).

print_board(Board) =>
   N = Board.length,
   foreach(I in 1..N)
      foreach(J in 1..N)
         X = Board[I,J],
         if var(X) then printf("  _") else printf("  %w", X) end
      end,
      nl
   end,
   nl.

problem(1, Data) =>
Data = [
    [_, _, 2, _, _, 5, _, 7, 9],
    [1, _, 5, _, _, 3, _, _, _],
    [_, _, _, _, _, _, 6, _, _],
    [_, 1, _, 4, _, _, 9, _, _],
    [_, 9, _, _, _, _, _, 8, _],
    [_, _, 4, _, _, 9, _, 1, _],
    [_, _, 9, _, _, _, _, _, _],
    [_, _, _, 1, _, _, 3, _, 6],
    [6, 8, _, 3, _, _, 4, _, _]].
```

What I have understood from [3], the loop construct was one of the reasons that Neng-Fa Zhou wanted to create a new language, since he was not satisfied with the `foreach` loop he implemented in B-Prolog [4]. (Many of my Picat models/programs are directly ported from the B-Prolog models I wrote in 2013. See my B-Prolog page [5], e.g. sudoku.pl.)

One thing that differs between Picat's **foreach** loop compared to B-Prolog's (as well as ECLiPSe's and SICStus Prolog's) is that the user does not have to explicitly declare local (or global) variables, which makes it simpler to use and read.

I will describe more CP in Picat below, but will first describe some other Picat features.

2 TABLING

"Tabling" here refers to memoization ("caching the results", see [6]). A simple example of tabling is to start with this the standard recursive definition of a Fibonacci number:

```
table
fibt(0)=1.
fibt(1)=1.
fibt(N)=F,N>1 => F=fibt(N-1)+fibt(N-2).
```

This is normally very slow for larger N because of the massive number of recursive steps needed. Using the table declaration before the predicate will automatically have all the calls and their answers cached. Subsequent calls are resolved by table lookups, which can speed up the program considerably.

Another example of its use is in euler2.pi (Project Euler problem #2), where the object is to find the sum of all even-valued terms in the sequence that do not exceed four million. Here is one way of implementing this, using **list comprehension**:

```
euler2 =>
   writeln(sum([F : N in 1..100, F = fibt(N), F < 4000000, F mod
2 =:= 0])).
```

Finding this sum takes 0.015s (with the overload of starting Picat etc). If we do not table the values then it is much (much!) slower: `fibt(100)` is 573147844013817084101 (~5.73e+20), which requires an exponential number of recursive steps to compute. Also, here we see that Picat supports **big integers** as well.

Please note that tabling in some cases might be slower because of the memory overhead involved, so one has to be careful and test both with and without tabling.

Another example of tabling is for **dynamic programming**. Here is an example for calculating the edit distance, which is taken from Picat's example exs.pi:

```
% computing the minimal editing distance
% of two given lists
table(+,+,min)
edit([],[],D) => D=0.
edit([X|Xs],[X|Ys],D) =>  % copy
    edit(Xs,Ys,D).
edit(Xs,[_Y|Ys],D)   ?=>  % insert
    edit(Xs,Ys,D1),
    D=D1+1.
edit([_X|Xs],Ys,D) =>     % delete
    edit(Xs,Ys,D1),
    D=D1+1.
```

(The symbols => and ?=> will be explained later.)

3 IMPERATIVE PROGRAMMING

Regarding loops (which can be seen as a trademark in imperative programming), we have already seen the `foreach` loop in the Sudoku example. Picat has some more of the traditional **imperative programming** constructs: `while` loops and assignments.

The Fibonacci example shown above (Project Euler #2, using the list comprehension) has the drawback of a hard coded limit that checks all `N` in the range 1..100 (it might be considered cheating to use a fixed range); this range was found by some experimentation and is actually not the smallest range. In Picat it is also possible to use a `while` loop for this, though not as nice looking as the list comprehension version. This is traditional imperative programming. It also shows an example of **(re)assignments** using the `:=` operator:

```
euler2b =>
  I = 1,
  Sum = I,
  F = fibt(I),
  while (F < 4000000)
     if F mod 2 == 0 then
        Sum := Sum + F
     end,
     I := I + 1,
     F := fibt(I)
  end,
  writeln(Sum).
```

This version also calculates the solution in 0.015s.

Sometimes the use of list comprehension and while loops might be slower than writing a predicate in traditional recursive ("Prolog") style, and it is sometimes worthwhile to try this approach as well.

4 MORE CONSTRAINT PROGRAMMING

Here are some more examples of using CP in Picat.

6.1 Element Constraint

In Picat, as in B-Prolog (and in many other CP systems), the `element` constraint (`List[Ith]=Value`) has to be written explicitly using the predicate

```
element(Ith, List, Value)
```

when `Ith` is a **decision variable**. However, if `Ith` is a pain integer, it can in Picat be stated more naturally as

```
List[Ith] = Value
```

or

```
List[Ith] #= Value
```

if `List` is a list of decision variables.

Readers of my My Constraint Programming Blog [7] might remember that I quite often complain how hard it is to write a **matrix element** constraint in different CP systems, and this applies to Picat as well. For example, in circuit.pi, in order to implement the circuit constraint I would like to use this matrix element construct to naturally state the constraint, but this is **not** allowed:

```
foreach(I in 2..N)
  % this is not allowed
  Z[I] = X[Z[I-1]]
end
```

Instead, I have to rewrite it. Here is one version:

```
import cp.

% ...

foreach(I in 2..N)
   Z1 #= Z[I-1],
   element(Z1,X,X2),
   Z[I] #= X2
end,
```

In stable_marriage.pi a more general version, `matrix_element`, is used:

```
matrix_element(X, I, J, Val) =>
  element(I, X, Row),
  element(J, Row, Val).
```

This version (using two `element`) can be very good sometimes, but depending on the type of the variables the above constraint does not work correctly; instead some of following variants have to be used. Sometimes all variants have to be tested. Note: all of them except one are commented out:

```
import cp.

% ...

matrix_element(X, I, J, Val) =>
  element(I, X, Row),
  element(J, Row, Val).

% matrix_element(X, I, J, Val) =>
%   nth(I, X, Row),
%   element(J, Row, Val).

% matrix_element(X, I, J, Val) =>
%   freeze(I, (nth(I, X, Row),freeze(J,nth(J,Row,Val)))).

% matrix_element(X, I, J, Val) =>
%   freeze(I, (element(I, X, Row),freeze(J,element(J,Row,Val)))).
```

The built-in `freeze(X, Call)` predicate delays the call to `Call` until `X` becomes a non-variable term.

6.2 Reification, alldifferent_except_0

Picat supports reifications with a nice syntax I expect from a high level CP system. Note that `#/\` is "and" for FD variables ("or" is denoted `#\/`). Here is the `alldifferent_except_0` constraint (see alldifferent_except_0.pi for the full model):

```
import cp.

alldifferent_except_0(Xs) =>
  foreach(I in 1..Xs.length, J in 1..I-1)
    (Xs[I] #!= 0 #/\ Xs[J] #!= 0) #=> (Xs[I] #!= Xs[J])
  end.
```

6.3 N-queens

Below are some different implementations of N-queens, taken from nqueens.pi.

```
import cp.

queens3(N, Q) =>
   Q=new_list(N),
   Q :: 1..N,
   all_different(Q),
   all_different([$Q[I]-I : I in 1..N]),
   all_different([$Q[I]+I : I in 1..N]),
   solve([ff],Q).

% Using all_distinct instead
queens3b(N, Q) =>
    Q=new_list(N),
    Q :: 1..N,
    all_distinct(Q),
    all_distinct([$Q[I]-I : I in 1..N]),
    all_distinct([$Q[I]+I : I in 1..N]),
    solve([ff],Q).

queens4(N, Q) =>
   Q = new_list(N),
   Q :: 1..N,
   foreach(A in [-1,0,1])
      all_different([$Q[I]+I*A : I in 1..N])
   end,
   solve([ff],Q).
```

Note that in `queens3` we have to use `$Q[I]-I` (i.e. `Q[I]` prepended with a `$`) since it is a term, not a function call, and must be evaluated ("escaped").

6.4 Magic Square

Below is a magic squares implementation in Picat, taken from magic_square.pi. One thing to notice is the use of `rows()`, `columns()`, `diagonal1()`, and `diagonal2()`, from Picat's `util` module.

```
import cp.
import utils.

magic2(N,Square) =>
   writef("\n\nN: %d\n", N),
   NN = N*N,
   Sum = N*(NN+1)//2,% magical sum
   writef("Sum = %d\n", Sum),

   Square = new_array(N,N),
   Square :: 1..NN,

  all_different(Square),

  foreach(Row in Square.rows()) Sum #= sum(Rows) end,
   foreach(Column in Square.columns()) Sum #= sum(Column) end,

   % diagonal sums
   Sum #= sum(Square.diagonal1()),
   Sum #= sum(Square.diagonal2()),

   % Symmetry breaking
   Square[1,1] #< Square[1,N],
   Square[1,1] #< Square[N,N],
   Square[1,1] #< Square[N,1],
   Square[1,N] #< Square[N,1],
```

```
   solve([ffd,updown],Square),

   print_square(Square).
```

6.5 Magic Sequence

Another standard CP problem is magic_sequence.pi:

```
import cp.

magic_sequence(N, Sequence) =>
   writef("\n%d: ",N),
   Sequence = new_list(N),
   Sequence :: 0..N-1,
   foreach(I in 0..N-1) count(I,Sequence,#=,Sequence[I+1]) end,
   N #= sum(Sequence),
   Integers = [ I : I in 0..N-1],
   N = scalar_product(Integers, Sequence),
   solve([ff], Sequence).

scalar_product(A, X) = Product =>
   Product #= sum([S : I in 1..A.length, S #= A[I]*X[I]]).
```

Note that I tend to use `count(Var, Vars, Relation, N)` instead of the built-in `global_cardinality(List, Pairs)` since it is more natural for me.

6.6 Least Diff problem, optimization

The Least Diff problem is a simple optimization problem in Picat: Minimize the difference between `ABCDE-FGHIJ` where the letters A..J are distinct integers (in the domain 0..9).

```
import cp.

least_diff(L,Diff) =>
  L = [A,B,C,D,E,  F,G,H,I,J],
  L :: 0..9,

  all_different(L),

  X #= 10000*A + 1000*B + 100*C + 10*D + E,
  Y #= 10000*F + 1000*G + 100*H + 10*I + J,

  Diff #= X - Y,
  Diff #> 0,
  solve([$min(Diff)], L).
```

The interesting part is the objective `$min(Diff)` (note the "`$`") which defines the objective value.

For more advanced models one can use the `$report(Goal)` to show the progress of the objective. Here is an example from einav_puzzle.pi:

```
   solve($[min(TotalSum), report(printf("Found %d\n",
TotalSum))], ffd],Vars),
```

Without `report(Goal)` the solution is only shown after the solver has finished the search, which may take a considerable amount time.

6.7 Table constraint

Picat has a `table` global constraint: the `in_table` built-in. Here is a simple example using table constraint: traffic_lights.pi (CSPLib #16) which use the list `Allowed` to state which combinations are allowed.

```
import cp.

traffic_lights_table(V, P) =>
  N  = 4,
  % allowed/1 as a table (translated)
  Allowed = [(1,1,3,3),
             (2,1,4,1),
             (3,3,1,1),
             (4,1,2,1)],
  V = new_list(N), V :: 1..N,
  P = new_list(N), P :: 1..N,
  foreach(I in 1..N, J in 1..N)
     JJ = (1+I) mod N,
     if J #= JJ then
        VI = V[I], PI = P[I],
        VJ = V[J], PJ = P[J],
        % Table constraint
        (VI, PI, VJ, PJ) in Allowed
     end
  end,
  Vars = V ++ P,
  solve(Vars).
```

The use of `in_table` requires that the values are integers so we have to translate to/from integers in this version.

Another example of table constraint is hidato.pi (though it is not very efficient).

6.8 Global Constraints

Picat supports the most common global constraints as shown in the list below. See Picat Guide [8], section 11.5 for details.

- `all_different(List)`
- `all_distinct(List)`
- `assignment(List)` (a.k.a. `inverse`)
- `circuit(List)`
- `count(Value,List,Rel,N)`
- `cumulative(Starts,Durations,Resources, Limit)`
- `diffn(RectangleList)`
- `disjunctive_tasks(Tasks)`
- `element(I,List, V)`
- `global_cardinality(List, Pairs)` (I tend to not use this so much since it is not as natural as using `count/4`)
- `neqs(NegList)`
- `serialized(Starts,Durations)`
- `subcircuit(List)`
- `in_table` used as the global constraint `table` (not to be confused with tabling).

In the module cp_utils.pi, I have also defined some other useful decompositions of global constraints:

- `matrix_element` (some different versions)
- `to_num`
- `latin_square`
- `increasing`
- `decreasing`
- `scalar_product` (with a relation)
- `alldifferent_except_0`
- `nvalue`
- `nvalues`
- `global_cardinality(Array, GccList)`
- `lex2`, `lex2eq`, `lex_less`, `lex_lesseq`, `lex_greater`, `lex_greatereq`, `lex_less2` (alternative variant)

6.9 Labeling

Most of the standard variable and value heuristics are supported, though I do miss options for using random assignments. Some of the examples above show how to label the variables in the `solve` predicate. Here the `ff` is

first-fail variable labeling, and down means that a variable is labeled from the largest value to the smallest.

```
solve([ff,down], Vars)
```

See the Picat Guide for descriptions of the options to solve and other solver options.

5 PATTERN MATCHING

Pattern matching is Picat is more influenced by pattern matching in functional programming than by Prolog's unification. This may confuse some programmers with a Prolog background.

Here is a definition of **quick sort** which shows some of the pattern matching constructs in Picat. It also shows how to define functions, i.e. predicates with one return value (the return value is placed after the predicate name, e.g. = L):

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

The two functions drop and take (standard in functional programming) can be defined as:

```
drop(Xs,0) = Xs.
drop([],_N) = [].
drop([_X|Xs],N) = drop(Xs,N-1).

take(_Xs,0) = [].
take([],_N) = [].
take([X|Xs],N) = [X] ++ take(Xs,N-1).
```

Reversing a list can be defined as follows using an accumulator:

```
reverse(L1,L2) => my_rev(L1,L2,[]).
my_rev([],L2,L3) => L2 = L3.
my_rev([X|Xs],L2,Acc) => my_rev(Xs,L2,[X|Acc]).
```

As will be mentioned again below, the matching between different variables - or part of variables - is often done in the body and not in the head.

There are some more comments about pattern matching below.

6 NONDETERMINISM AND OTHER PROLOG INFLUENCES/DIFFERENCES

Some of the examples above show clear influences from Prolog. Here we list some more influences (and differences).

Picat has - as Prolog has - support for nondeterminism using a **backtracking mechanism** to test different predicates. This is a really great feature in Picat and is one of the reasons I like Picat so much. Or rather: it is the **combination** of the backtracking possibility together with the imperative features, support for constraint programming etc that make me really like Picat.

6.1 Member

The nondeterministic member predicate in Picat works as in Prolog, i.e. it can be used both as a test for membership and also for generating all the elements:

```
Picat> member(2, [1,2,3])
yes
Picat> member(4, [1,2,3])
no
Picat> member(X, [1,2,3])
   X=1;
   X=2;
   X=3;
   no
```

As in Prolog we can get the next solution with a ; (semicolon) in the Picat shell.

A personal note: This “double nature” of predicates - bidirectionality, reversibility - is one of the features I really appreciate in Picat, Prolog, and in CP.

We can define member in Picat as:

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

In this definition we see two features of Picat:

- **pattern matching**: the pattern [Y|_] matches any list and "extracts" the first elements Y to be unified with X in the first clause.
- **backtracking**: The first rule is defined as a **backtrackable rule** using ?=> (the prepending "?" is the marker for backtrackable rules) and makes the call nondeterministic. The last rule in a definition should be stated as non-backtrackable, i.e. => (without the ?).

This predicate now works in the same way as the built-in member/2.

6.2 Predicate facts

Predicate facts are bodyless definitions akin to Prolog's "data definition" (but be careful to take this too literarily). Predicate facts are prepended by an **index declaration** to make them accessible as data (and this explicit declaration is one of the differences from Prolog).

For example, here is a stripped down version of the transitive closure example transitive_closure.pi:

```
top ?=>
    reach(X,Y),
    writeln([X,Y]),
    fail.
top => true.

% Transitive closure right
table
reach(X,Y) ?=> edge(X,Y).
reach(X,Y)  => edge(X,Z),reach(Z,Y).

% the graph
index(-,-)
edge(1,2).
edge(1,3).
edge(2,4).
edge(2,5).
edge(3,6).
% ....
```

The use of table will cache the reach predicate. In this simple example tabling is not needed, but for other applications it might be a great advantage. Another

advantage is that by using `table` we avoid the problem of possible infinite recursion in `reach`, and makes it independent on the order of `edge`/2 and `reach`/2.

6.3 Append

`append` in Picat is also nondeterministic as in Prolog. Here is an example where we generate the different sublists `X` and `Y` that make up the list `[1,2,3,4]`. Collecting all the results is done with `findall` which will be discussed little more below:

```
Picat> Z=findall([X,Y], append(X, Y, [1,2,3,4]))
Z = [[[],[1,2,3,4]],[[1],[2,3,4]],[[1,2],[3,4]],[[1,2,3],[4]],
[[1,2,3,4],[]]]
```

As with `member`, we can define it on our own:

```
append(Xs,Ys,Zs) ?=> Xs=[], Ys=Zs.
append(Xs,Ys,Zs) => Xs=[X|XsR], append(XsR,Ys,Zs).
```

6.4 Findall

As we saw in the example above , `findall` in Picat works much as in Prolog. One difference is that `findall` in Picat is a **function**, i.e. returns a value and that is why it was assigned to the `Z` variable above.

6.5 Other nondeterministic predicates

Here are the **nondeterministic** built-in predicates in Picat. Many of these are wrappers to the underlying B-Prolog predicates.

- `between(From, To, X)`
- `append(X,Y,Z)`
- `append(W,X,Y,Z)`
- `member(Term, List)`
- `select(X, List, ResList)`
- `find(String, Substring, From, To)`
- `find_ignore_case(String, Substring, From, To)`
- `repeat`
- `permutation(List, Perm)`
- `nth(I, List, Val)`
- `indomain(Var)` (CP)
- `indomain_down(Var)` (CP)
- `solve(Vars)` (CP)
- `solve(Options, Vars)` (CP)
- `statistics(Name,Value)`

One can also note that - in contrast to Prolog - `length` is a "plain" (deterministic) function and does not create a list, i.e. it is not bidirectional. It just returns the length of the list (array, structure). For defining a new list one has to use `new_list(Length)` instead.

The nondeterministic predicates are great when they are needed, but please don't overuse them. The Picat documentation generally recommends writing functions rather than predicates since functions are easier to debug and since they return exactly one value.

6.6 An advice to Prolog addicts

In general, Prolog addicts should be a little careful when trying to write Prolog code in Picat. Sometimes it works very well to write it as in Prolog (with some minor modifications) but sometimes it does not work at all. The biggest gotcha is probably pattern matching - as mentioned above - which looks much like Prolog's unification but is more like the pattern matching used in functional languages such as Haskell.

7 SAT SOLVER

Picat also supports solving problems using a SAT solver which makes it possible to use almost exactly the same code as using the CP solver. The only thing that is needed to change is

```
import cp.
```

to

```
import sat.
```

However, the SAT solver does not support all the global constraints from the CP solver, but it might be worth testing sometimes since it can be very fast compared to a CP approach. Still, I tend to rely on CP since then I can get **all** the solutions - or two - which is inefficient or not easy to do with the SAT solver used in Picat.

I have not used the SAT solver very much, but see Neng-Fa Zhou's (and others) programs at Picat's Projects page [9], e.g. mqueens.pi (it is one of the problems from the CP-2013 Model and Solve Competition [10]).

8 PLANNING

One of the things I really like in Picat is that it is now easy to implement certain planning problems. I first wrote a small planning module (bplan.pi, inspired by Hector J. Levesque's version "Thinking as Computation", a rather new Prolog AI book). Neng-Fa Zhou later built a much faster module (planner.pi) with a lot of different variants, both optimality versions (`best_plan*`) and non-optimality versions, bounded and unbounded, and with and without costs.

The `planning` module is quite simple to use:

- Define the **action** predicates (the legal moves): `action(FromState, ToState, Move, Cost)`
- Define the initialization state (the first `FromState`)
- Define the goal state(s)/definition: `final(State)` that checks if the goal state has been reached.
- call the `plan*` predicate with the appropriate arguments

A simple planning example is the M12 problem (test_planner_M12.pi) which has

- two actions: merge (perfect shuffle) and reverse
- initial state (the problem instance), e.g. `[8,11,6,1,10,9,4,3,12,7,2,5]`
- the goal state: `[1,2,3,4,5,6,7,8,9,10,11,12].`

(The goal state could be written as `1..12.`). Here is the code for the planner part. Using the `table` directive is sometimes a booster in these kind of planning problems, but not always:

```
go =>
  Init = [8,11,6,1,10,9,4,3,12,7,2,5],
  time(best_plan_downward(Init, Plan)),
  writeln(Plan),
  writeln(len=Plan.length),
  nl.

final(Goal) =>
  Goal=[1,2,3,4,5,6,7,8,9,10,11,12].

table
% merge move
action([M1,M12,M2,M11,M3,M10,M4,M9,M5,M8,M6,M7], To, M, Cost) ?
=>
   Cost=1, M=m,
   To=[M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12].
% reverse move
action([M12,M11,M10,M9,M8,M7,M6,M5,M4,M3,M2,M1], To,M, Cost) =>
   Cost=1,M=r, To=[M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12].
```

As mentioned above, it is important to declare the first `action` with `?=>` so that it is **backtrackable**, i.e. if that clause fails it will backtrack and test the next rule.

Picat finds the shortest solution for this problem instance (27 steps) in 0.97s:

```
[m,m,m,r,m,m,m,r,m,m,m,m,r,m,r,m,r,m,m,r,m,m,r,m,m,m,r]
```

where "`m`" refers to the merge action/move, and "`r`" to the reverse action/move.

For a maze problem (test_planner.pi), the action (move) is simply defined as

```
action(From,To,Move,Cost) =>
   maze(From,To),
   Move = [From,To],
    Cost = 1.
```

where `maze(From,To)` defines which nodes are connected in the maze (graph). And that is about as simple as it could be.

For these kind of "straight" planning problems it's - IMHO - much easier to use this approach in Picat than – for example - using Constraint Programming.

My other planning models are at http://www.hakank.org/picat/#planning_problems, mostly quite simple standard problems. Neng-Fa Zhou has written more advanced programs, shown at the Picat's Projects page. One neat example is his solution of the Doubleclock problem from the CP-2013 competition: doubleclock.pi [9] which solves each of the five problem instances in about 0.1s (except for #4 which takes 0.6s).

9 STRUCTURE AND MAP (HASH TABLE)

Picat has a construct for creating a structure - `new_struct` (a kind of poor man's OOP "object"). Here is a small example:

```
Picat> S = new_struct(point,3), Name = S.name, Len =
S.length
S = point(_3b0,_3b4,_3b8)
Name = point
Len = 3
```

A data structure that I use much more is the inevitable `map` (hash table):

```
Picat> M = new_map([one=1,two=2]), M.put(three,3), One =
M.get(one)
```

Below is a longer example (one_dimensional_cellular_automata.pi) which use a map (`Seen`) to detect fixpoints and cycles in the evolution of the patterns, i.e. if we have seen a pattern earlier. It uses `new_map()` for creating the map, `has_key(S)` for checking if pattern `S` already exists in the map, and `put(S,1)` to add the pattern S to the map. Also note the definition of the rules as function facts.

```
go =>
  %     _ # # # _ # # _ # _ # _ # _ # _ _ # _ _
  S = [0,1,1,1,0,1,1,0,1,0,1,0,1,0,1,0,0,1,0,0],
  All = ca(S),
  foreach(A in All) println(A.convert()) end,
  writeln(len=All.length),
  nl.

convert(L) = Res =>
  B = "_#",
  Res = [B[L[I]+1] : I in 1..L.length].

ca(S) = All =>
  Len = S.length,
  All := [S],
  % detect fixpoint and cycle
  Seen = new_map(),
  while (not Seen.has_key(S))
    Seen.put(S,1),
    T = [S[1]] ++ [rule(sublist(S,I-1,I+1)) : I in 2..Len-1] ++
[S[Len]],
    All := All ++ [T],
    S := T
  end.

% the rules
index(+)
rule([0,0,0]) = 0. %
rule([0,0,1]) = 0. %
rule([0,1,0]) = 0. % Dies without enough neighbours
rule([0,1,1]) = 1. % Needs one neighbour to survive
rule([1,0,0]) = 0. %
rule([1,0,1]) = 1. % Two neighbours giving birth
rule([1,1,0]) = 1. % Needs one neighbour to survive
rule([1,1,1]) = 0. % Starved to death.
```

10 STRINGS

There are some support for strings in Picat, such as

- `++` (for list/string concatenation)
- `atom_chars(Atom)`
- `find(String, Substring,From,To)` (nondet)
- `find_ignore_case(String, Substring,From,To)`
- `join(Tokens)`
- `join(Tokens,Separator)`

- `number_chars(Num)`
- `parse_term(String)`
- `split(String)`
- `split(String, Separators)`
- `to_atom(String)`
- `to_binary_string(Int)`
- `to_lowercase(String)`
- `to_uppercase(String)`
- `to_string(Term)`

Unfortunately there is - as of writing - not any support for regular expressions; I hope this will come soon enough in Picat. However, since a string is (just) an array of characters, much of the general list handling can be used for manipulating strings.

11 OTHER FEATURES IN PICAT

Picat has many other features that are not mentioned in this paper. For example:

- debugging and trace, akin to Prolog's debug facilities
- `os` module for handling files and directories. Reading files is often much easier to do than in standard Prolog
- `math` module for standard math functions, for example `primes(Int)`, `prime(Int)`, `random()`, `random2()`.

There are many other useful predicates that are not mentioned here. See the Picat User's Guide for a detailed description.

12 MY CONTRIBUTIONS TO PICAT

Picat was originally created by Neng-Fa Zhou and Jonathan Fruhman. After the release in May 2013, I have done some contributions to the code as well as bug reports, suggestions etc, and I am now sometimes mentioned as one of the Picat developers/collaborators. Please note that I only develop things in Picat, not in the C or B-Prolog level. Some of the modules I have contributed that are in the Picat distribution are:

- util.pi: One of the utilities modules
- picat_lib_aux.pi: Utilities used by other modules
- apl_util.pi: Utilities inspired by the languages APL and K. Most of this is as a proof-of-concept.
- set_util.pi: Set utilities (ported from Neng-Fa's set utilities in B-Prolog)

Also, many of the Project Euler programs above #20 at the Picat's Project page are mine. Also see my own collection of Euler programs (http://www.hakank.org/picat/#euler) where I have some different variants for the problems below problem #20.

As of writing there are almost 450 public Picat programs/models on my My Picat Page (http://www.hakank.org/picat/).

They are mostly in the following areas:

- Constraint Programming
- Planning problems
- Recreational mathematics and puzzles
- Project Euler [11]
- Rosetta Code [12] programs

REFERENCES

[1] http://picat-lang.org/

[2] http://www.hakank.org/picat/

[3] Neng-Fa Zhou: comp.lang.prolog, thread, "Picat: A New Logic-Based Language" https://groups.google.com/forum/#!msg/comp.lang.prolog/p6nUtaCAs8A/JQ7b0LY5vfsJ (Google Groups)

[4] http://www.probp.com/

[5] http://www.hakank.org/bprolog/

[6] http://en.wikipedia.org/wiki/Memoization

[7] http://www.hakank.org/constraint_programmming_blog/

[8] http://picat-lang.org/download/picat_guide.pdf

[9] http://picat-lang.org/projects.html

[10] http://cp2013.a4cp.org/program/competition

[11] http://projecteuler.net/

[12] http://rosettacode.org/wiki/Rosetta_Code