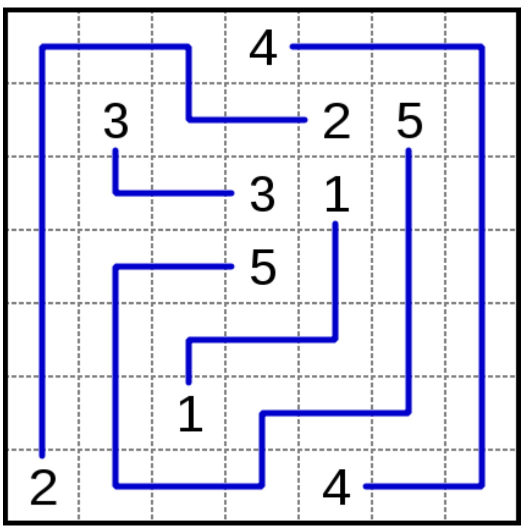
Modeling and Solving AI Problems in Picat

Roman Barták, Neng-Fa Zhou


Numberlink

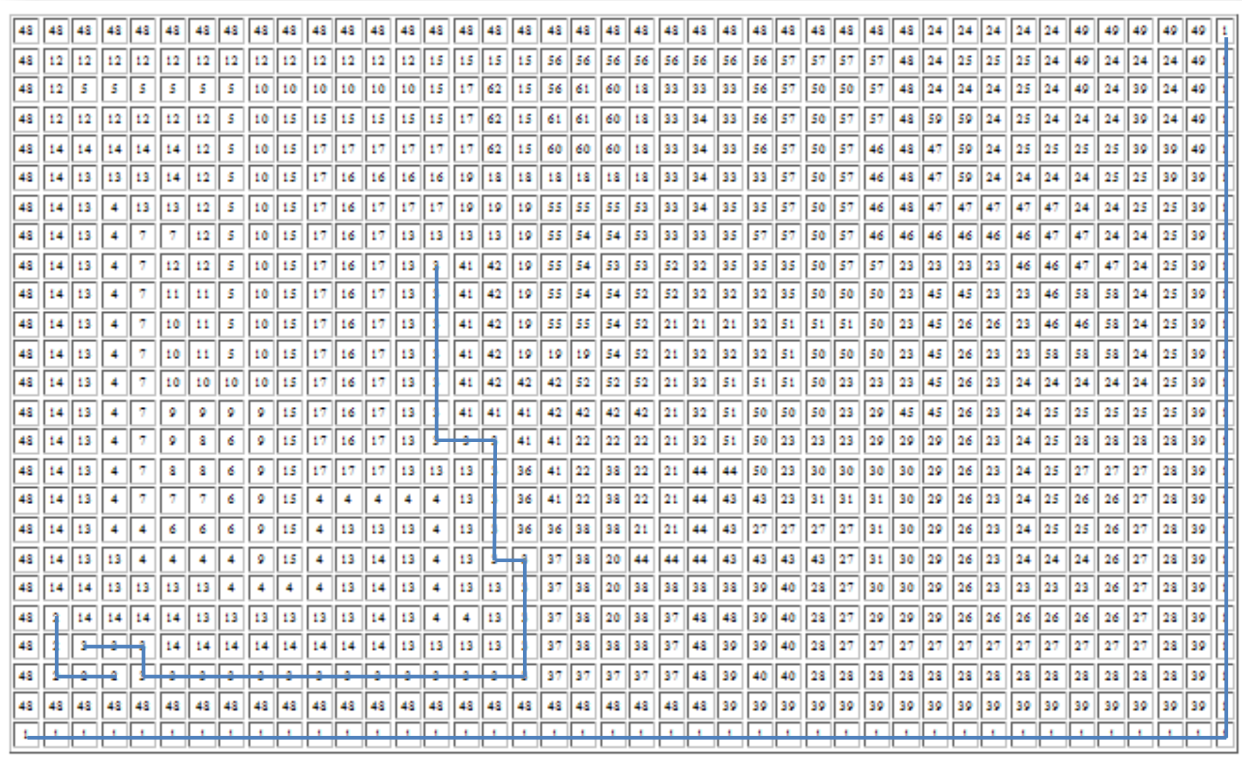


Pair up all the matching numbers on the grid with single continuous lines (or paths).

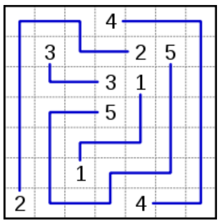
- The **lines cannot branch off** or **cross** over each other, and
- the numbers have to fall at the end of each line (i.e., not in the middle).

It is considered all the cells in the grid are filled.

Source: wiki



Solved with the sat module of Picat and the Lingeling solver in 40s.




```
{{0,0,0,4,0,0,0},
 {0,3,0,0,2,5,0},
 {0,0,0,3,1,0,0},
 {0,0,0,5,0,0,0},
 {0,0,0,0,0,0,0},
 {0,0,1,0,0,0,0},
 {2,0,0,0,4,0,0}}
```

```
import sat.

numberlink(NP,NR,NC,InputM) =>
     M = new_array(NP,NR,NC),
     M :: 0..1,
     % no two numbers occupy the same square
    foreach(J in 1..NR, K in 1..NC)
        sum([M[I,J,K] : I in 1..NP]) #=1
    end,
     % connectivity constraints
    foreach(I in 1..NP, J in 1..NR, K in 1..NC)
        Neibs = [M[I,J1,K1] : (J1,K1) in [(J-1,K),(J+1,K),(J,K-1),(J,K+1)],
                              J1>=1, K1>=1, J1=<NR, K1=<NC],
        (InputM[J,K]==I ->
            M[I,J,K] #=1, sum(Neibs) #= 1
        ;
            M[I,J,K] #=> sum(Neibs) #= 2
        )
    end,
    solve(M).
```

Tutorial outline

Part I: From Prolog to Picat

- *Introduction to Picat's programming constructs*
- *Behind the scene*

Part II. Combinatorial (optimization) problems in Picat

- *A very short introduction to SAT, CP, MIP modules*
- *Examples of combinatorial (optimization) problems and their encodings in Picat*
- *Behind the scene*

Part III. Classical action planning in Picat

- *A very short introduction to formal models of classical planning problems*
- *Examples of planning problems and their encodings in Picat*
- *Behind the scene*

Wrap up

Part I:

FROM PROLOG TO PICAT

Why the name "PICAT"?

- **P**attern-matching, **I**ntuitive, **C**onstraints, **A**ctors, **T**abling

Core logic programming concepts:

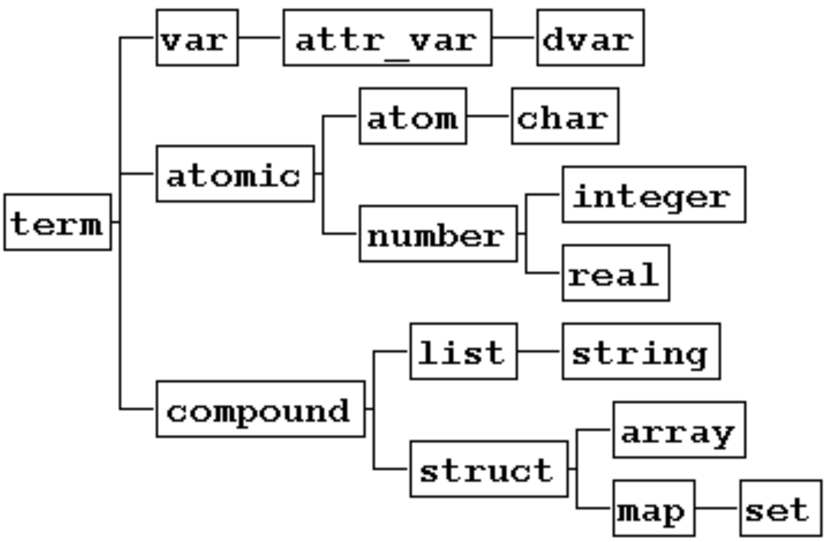
- logic variables (arrays and maps are terms)
- implicit pattern-matching and explicit unification
- explicit non-determinism

Language constructs for scripting and modeling:

- functions, loops, list and array comprehensions, and assignments

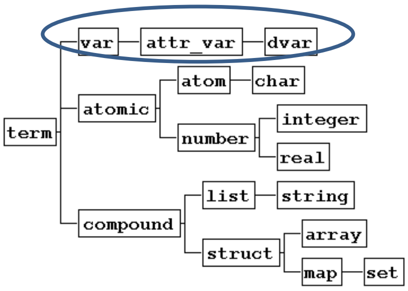
Facilities for combinatorial search:

- tabling for dynamic programming
- the `cp`, `sat`, and `mip` modules for CSPs
- the `planner` module for planning



Variables

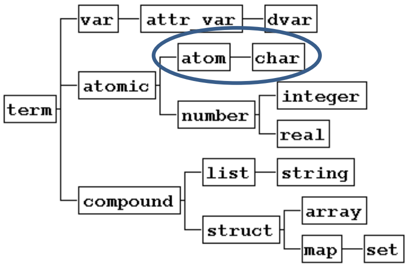
A variable name begins with a capital letter or the underscore.




```
Picat> var(X)
yes

Picat> X = a, var(X)
no

Picat> X.put_attr(a,1), attr_var(X)
yes

Picat> X.put_attr(a,1), Val = X.get_attr(a)
Val = 1
yes

Picat> import cp
Picat> X :: 1..10, dvar(X)
X = DV_010b48_1..10
yes
```

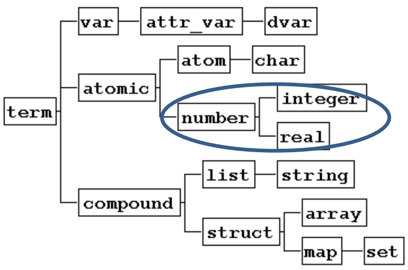
Atoms and Characters

An unquoted atom name begins with a lower-case letter.
A character is a single-letter atom.




```
Picat> atom(abc)
yes

Picat> atom('_abc')
yes

Picat> char(a)
yes

Picat> Code = ord(a)
Code = 97

Picat> A = chr(97)
A = a
```

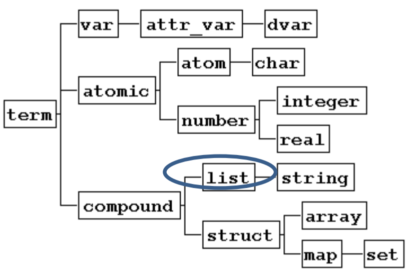
Numbers




```
Picat> int(123)
yes

Picat> Big = 9999999999999999999999
Big = 9999999999999999999999

Picat> X = 0b111101
X = 61

Picat> X = 0xff0
X = 4080

Picat> real(1.23)
yes

Picat> X = 1.23e10
X = 12300000000.0
```

Lists

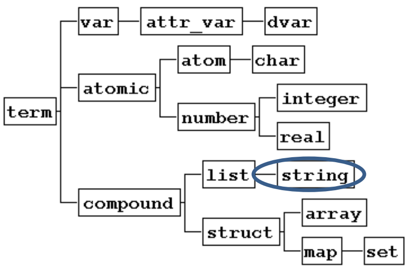
Lists are singly-linked lists.




```
Picat> L = [a,b,c], list(L)
L = [a,b,c]
yes

Picat> L = new_list(3)
L = [_101c8,_101d8,_101e8]

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

Picat> L = [X : X in 1..10, even(X)]
L = [2,4,6,8,10]

Picat> L = [a,b,c], Len = len(L)
L = [a,b,c]
Len = 3

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

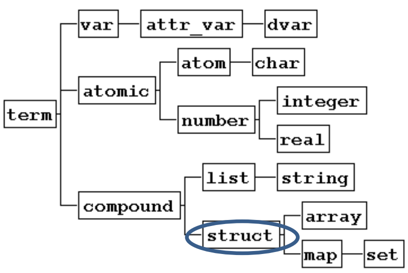
Strings

Strings are lists of characters.



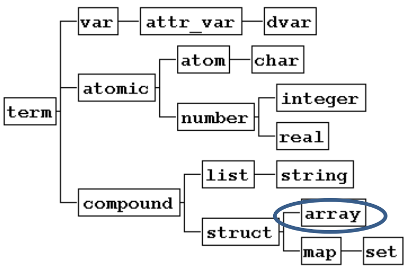

```
Picat> S = "hello"
S = [h,e,l,l,o]

Picat> S = "hello" ++ "Picat"
S = [h,e,l,l,o,'P',i,c,a,t]

Picat> S = to_string(abc)
S = [a,b,c]

Picat> S = to_radix_string(123,16)
S = ['7','B']

Picat> X = to_int("123")
X = 123

Picat> X = parse_term("[1,2,3]")
X = [1,2,3]
```

Structures




```
Picat> S = $student(mary,cs,3.8)
S = student(mary,cs,3.8)

Picat> S = new_struct(mary,3)
S = mary(_12ad0,_12ad8,_12ae0)

Picat> S = $f(a), A = arity(S), N = name(S)
A = 1
N = f

Picat> And = (a,b)
And = (a,b)

Picat> Or = (a;b)
Or = (a;b)

Picat> Constr = (X #= Y)
Constr = (_10f18 #= _10f20)
```

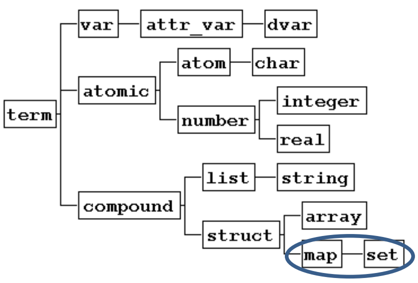
Arrays




```
Picat> A = {a,b,c}, array(A)
A = {a,b,c}
yes

Picat> A = new_array(3)
A = {_10528,_10530,_10538}

Picat> A = new_array(3,3)
A = {{_fdb0,_fdb8,_fdc0},…}

Picat> A = {X : X in 1..10, even(X)}
A = {2,4,6,8,10}

Picat> L = [a,b,c], A = to_array(L)
L = [a,b,c]
A = {a,b,c}

Picat> A = {a,b} ++ {c,d}
A = {a,b,c,d}
```

Maps and Sets

Maps and sets are hash tables.




```
Picat> M = new_map([ichi=1, ni=2]), map(M)
M = (map)[ni = 2,ichi = 1]
yes

Picat> M = new_map([ni=2]), Ni = M.get(ni)
Ni = 2

Picat> M = new_map(), M.put(ni,2)
M = (map)[ni = 2]

Picat> M = new_map(), Ni = M.get(ni,unknown)
M = (map)[]
Ni = unknown

Picat> S = new_set([a,b,c])
S = (map)[c,b,a]

Picat> S = new_set([a,b,c]), S.has_key(b)
yes
```

Index Notation

X[I1,…,In] : X references a compound value

Linear-time access of **list** elements.

```
Picat> L = [a,b,c,d], X = L[4]
X = d
```

Constant-time access of **structure** and **array** elements.

```
Picat> S = $student(mary,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

$$[T : E_1 \ \mathtt{in}\ D_1, Cond_n , . . ., E_n \ \mathtt{in}\ D_n, Cond_n]$$

```
Picat> L = [X : X in 1..10, even(X)]
L = [2,4,6,8,10]

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

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

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

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

OOP Notation

O.f(t1,…,tn)
-- means module qualified call if O is atom
-- means f(O,t_1,…,t_n) otherwise.

```
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_attr(age, 35), X.put_attr(weight, 205), A =
   X.get_attr(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.len
Name = point
Arity = 2

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

Explicit Unification

```
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
```

Nondeterministic Predicates

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

Picat> between(1,3,X)

Picat> select(X,[1,2,3],R)

Picat> nth(I,[1,2,3],E)

Picat> append(L1,L2,[1,2,3])
```


Control backtracking

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

Higher-Order

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

Picat> Sin = apply(sin,0.5)
Sin = 0.479425538604203

Picat> R = map(to_real,[1,2,3])
R = [1.0,2.0,3.0]

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

Picat> time(_ = 1..1000000)
CPU time 0.033 seconds.

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

The `io` Module


```
Picat> X = read_int()
123
X = 123

Picat> X = read_file_lines()
hello
Picat
X = [[h,e,l,l,o],['P',i,c,a,t]]

Picat> S = open("t"), Line = S.read_line(),
S.close()
S = (stream)[10002]
Line = [h,e,l,l,o,' ','P',i,c,a,t]
```

The `math` Module

$$\int x\mathbf{e}^{6x}\,dx = \frac{x}{6}\mathbf{e}^{6x} - \int \frac{1}{6}\mathbf{e}^{6x}\,dx = \frac{x}{6}\mathbf{e}^{6x} - \frac{1}{36}\mathbf{e}^{6x} + c$$

```
Picat> X = sign(-2)
X = -1

Picat> X = sin(pi()/3)
X = 0.866025403784439

Picat> X = sqrt(5)
X = 2.23606797749979

Picat> X = factorial(30)
X = 265252859812191058636308480000000

Picat> X = gcd(100000,388)
X = 4

Picat> X = primes(17)
X = [2,3,5,7,11,13,17]
```

The `util` Module

```
Picat> import util

Picat> Ts = split("ab cd ef"), S = Ts.join()
Ts = [[a,b],[c,d],[e,f]]
S = [a,b,' ',c,d,' ',e,f]

Picat> permutation([1,2,3],P)
P = [1,2,3] ?;
P = [1,3,2] ?
…

Picat> Ps = permutations([1,2,3])
Ps = [[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]
```

Statements

```
Picat> (2 > 1, 2 < 3)          % conjunction
yes

Picat> (X = a; X = b)          % disjunction
X = a ?;
X = b

Picat> not X = a               % negation

Picat> if var(X) then writeln(var) else writeln(no) end
var

Picat> (var(X) -> writeln(var); writeln(no))
var

Picat> X = cond(2>1, a, b)      % conditional exp
X = a
```

The foreach Loop

`foreach`(E_1 `in` D_1, $Cond_1$,…, E_n `in` D_n, $Cond_n$)
Goal
`end`

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

```
Picat> A = new_array(5), foreach(I in 1..5) A[I] = X end
A = {_15bd0,_15bd8,_15be0,_15be8,_15bf0}

Picat> X = _, A = new_array(5), foreach(I in 1..5) A[I] = X end
A = {X,X,X,X,X}
```

Assignments

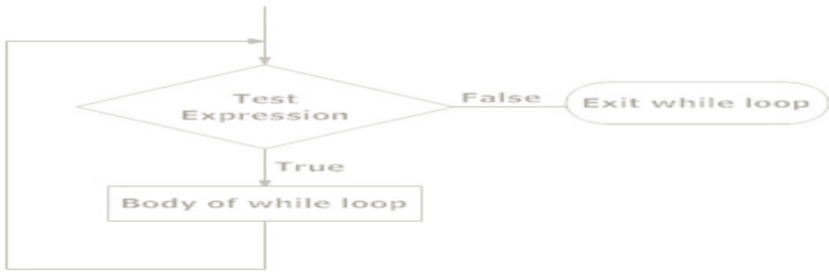
`X[I1,…,In] := Exp`
Destructively update the component to `Exp`. Undo the update upon backtracking.

`Var := Exp`
The compiler changes it to `Var' = Exp` and replaces all subsequent occurrences of `Var` in the scope by `Var'`.

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

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

```
while (Cond)
    Goal
end
```

```
Picat> X = read_int(), while (X !== 0) X := read_int() end
```



Non-backtrackable

Head, Cond => Body.

Backtrackable

Head, Cond ?=> Body.

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

membchk(X,[X|_] => true.
membchk(X,[_|L]) => membchk(X,L).
```

- Pattern-matching rules
  - No laziness or freeze
    The call `membchk(X,_)` fails
  - Facilitates indexing
- Explicit unification
- Explicit non-determinism

Predicate Facts

```
index(+,-)  (-,+)
edge(a,b).
edge(a,c).
edge(b,c).
edge(c,b).
```

```
edge(a,Y)  ?=> Y=b.
edge(a,Y)  =>  Y=c.
edge(b,Y)  =>  Y=c.
edge(c,Y)  =>  Y=b.
edge(X,b)  ?=> X=a.
edge(X,c)  ?=> X=a.
edge(X,c)  =>  X=b.
edge(X,b)  =>  X=c.
```

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

Functional Programming in Picat

`Head = Exp, Cond => Body.`

```
fib(0) = 1.
fib(1) = 1.
fib(N) = fib(N-1)+fib(N-2).

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

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

Dynamically typed

List and array comprehensions

Strict (not lazy)

Higher-order functions

Patterns in Heads

Function calls cannot occur in head patterns.

Index notations, ranges, dot notations, and 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).
```

Dynamic Programming in Picat

```
table
fib(0) = 0.
fib(1) = 1.
fib(N) = fib(N-1)+fib(N-2).
```

- Linear tabling
- Mode-directed tabling
- Term sharing

Dynamic Programming: Binomial Coefficient

$$\binom{n}{0} = \binom{n}{n} = 1 \quad \text{for all integers } n \geq 0,$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{for all integers } n, k : 1 \leq k \leq n-1,$$

```
table
c(_, 0) = 1.
c(N, N) = 1.
c(N, K) = c(N-1, K-1) + c(N-1, K).
```

Scripting in Picat

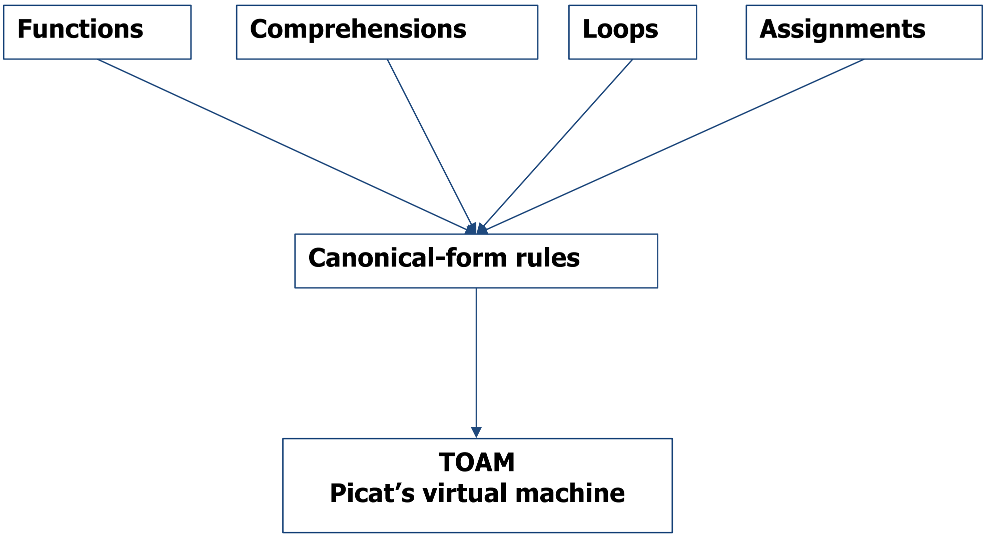
```
main =>
    print("enter an integer:"),
    N = read_int(),
    foreach(I in 0..N)
        Num := 1,
        printf("%*s", N-I, ""),     % print N-I spaces
        foreach(K in 0..I)
            printf("%d ", Num),
            Num := Num*(I-K) div (K+1)
        end,
        nl
    end.
```

SSA (Static Single Assignment)

Loops

```
$ picat pascal
enter an integer:5
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

The Picat Compiler



Transformation of Functions

f(A1,A2,…,An) = Exp, Cond => Body.

↓

p(A1,A2,…,An,V), Cond => Body, V=Exp.

```
conc([], Ys) = Ys.
conc([X|Xs], Ys) = [X | conc(Xs, Ys)].
```

↓

```
conc_p([], Ys, Zs) => Zs = Ys.
conc_p([X|Xs], Ys, Zs) =>
    Zs = [X|Zs1],
    conc_p(Xs, Ys, Zs1).
```

Tail-recursive

Transformation of Comprehensions

```
L = [Exp : E1 in D1, Condn , . . ., En in Dn, Condn]
```

↓

```
L = Tail,
foreach (E1 in D1, Condn , . . ., En in Dn, Condn)
    Tail = [Exp|NewVar],
    Tail := NewVar,
end,
Tail = []
```

Transformation of Aggregates of Comprehensions

```
Sum = sum([f(I) : I in 1..100])
```

↓

```
S = 0,
foreach (I in 1..100)
    S := S + f(I)
end,
Sum = S
```

Deforestation

Transformation of `foreach`

```
foreach (E in D)
    Goal
end
```

`V1,…,Vn` are global vars in `Goal`
`D` is a list

```
p(V1,…,Vn,[]) => true.
p(V1,…,Vn,[E|T]) => Goal, p(V1,V1,…,Vn,T).
```

Transformation of LHS := RHS, No if-then-else, no loops

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

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

Static Single Assignment form

Transformation of LHS := RHS: in if-then-else

```
go(Z) =>
    X = 1, Y = 2,
    if Z > 0 then
        X := X * Z
    else
        Y := Y + Z
    end,
    print([X,Y]).
```

```
go(Z) =>
       X = 1, Y = 2,
       p(X, Xout, Y, Yout, Z),
       println([Xout,Yout]).

   p(Xin, Xout, Yin, Yout, Z), Z > 0 =>
       Xout = Xin * Z,
       Yout = Yin.
   p(Xin, Xout, Yin, Yout, Z) =>
       Xout = Xin,
       Yout = Yin + Z.
```

Transformation of LHS := RHS, in loops

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

```
sum_list(L, Sum) =>
    S = 0,
    p(L, S, Sout),
    Sum = Sout.

p([], Sin, Sout) =>
    Sout = Sin.
p([E|T], Sin, Sout) =>
    St = Sin + E,
    p(T, St, Sout).
```

Programming Exercise: Count Zeros

Write a function that returns the number of zeros in a given simple list of numbers.

```
count_zeros(L) = sum([1 : 0 in L]).
```

```
count_zeros(L) = Count =>
    count_zeros(L, 0, Count).

count_zeros([], Count0, Count) => Count = Count0.
count_zeros([0|L], Count0, Count) =>
    count_zeros(L, Count0+1, Count).
count_zeros([_|L], Count0, Count) =>
    count_zeros(L, Count0, Count).
```

Programming Exercise: Replicate Elements

Replicate the elements of a list a given number of times.

Example:

`repli([a,b],3)` returns `[a,a,a,b,b,b]`.

```
repli(L, N) = [X : X in L, _ in 1..N].
```

Given a list of space-separated words, reverse the order of the words [from GCJ].

Input

```
3
this is a test
foobar
all your base
```

Output

```
Case #1: test a is this
Case #2: foobar
Case #3: base your all
```

Programming Exercise: Reverse Words

Given a list of space-separated words, reverse the order of the words [from GCJ].

```
3
this is a test
foobar
all your base
```

```
Case #1: test a is this
Case #2: foobar
Case #3: base your all
```


```
import util.

main =>
    T = read_line().to_int(),
    foreach (TC in 1..T)
        Words = read_line().split(),
        printf("Case #%w: %s\n", TC, Words.reverse().join())
    end.
```

Programming Exercise: Store Credit

Given an integer C, and a sequence of integers, find the indices of the two items that sum up to C (from GCJ).

Input

```
2
100
3
5 75 25
200
7
150 24 79 50 88 345 3
```

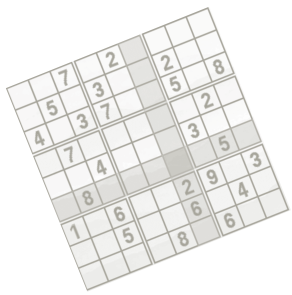
Output

```
Case #1: 2 3
Case #2: 1 4
```

Programming Exercise: Store Credit, Brute-force, $O(n^2)$

```
main =>
    T = read_int(),
    foreach (TC in 1..T)
        C = read_int(),
        N = read_int(),
        Items = {read_int() : _ in 1..N},
        do_case(TC, C, Items)
    end.

do_case(TC, C, Items),
   between(1, len(Items)-1, I),
   between(I+1, len(Items), J),
   C == Items[I]+Items[J]
=>
   printf("Case #%w: %w %w\n", TC, I, J).
```

```
main =>
    T = read_int(),
    foreach (TC in 1..T)
        C = read_int(),
        N = read_int(),
        Items = {read_int() : _ in 1..N},
        Map = new_map(),
        foreach (I in N..-1..1)
            Is = Map.get(Items[I], []),
            Map.put(Items[I],[I|Is])
        end,
        do_case(TC, C, Items, Map)
    end.

do_case(TC, C, Items, Map),
   between(1, len(Items)-1, I),
   Js = Map.get(C-Items[I], []),
   member(J, Js),
   I < J
=>
   printf("Case #%w: %w %w\n", TC, I, J).
```



Part II.

COMBINATORIAL (OPTIMIZATION) PROBLEMS IN PICAT

Combinatorial puzzle, whose goal is to enter digits 1-9 in cells of 9×9 table in such a way, that no digit appears twice or more in every row, column, and 3×3 sub-grid.

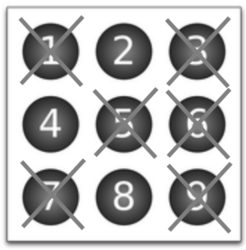
9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Solving Sudoku

×	×	6		(1)	3			
3	9	×					(1)	
2	1	8				4		

Use information that each digit appears exactly once in each row, column and sub-grid.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



We can see every cell as a **variable** with possible values from **domain** {1,…,9}.

There is a binary inequality **constraint** between all pairs of variables in every row, column, and sub-grid.

Such formulation of the problem is called a **constraint satisfaction problem.**

Constraint satisfaction problem consists of:

- a finite set of **variables**
  - describe some features of the world state that we are looking for, for example positions of queens at a chessboard
- **domains** – finite sets of values for each variable
  - describe “options” that are available, for example the rows for queens
  - sometimes, there is a single common “superdomain” and domains for particular variables are defined via unary constraints
- a finite set of **constraints**
  - a constraint is a *relation* over a subset of variables
    for example rowA ≠ rowB
  - a constraint can be defined *in extension* (a set of tuples satisfying the constraint) or using a *formula* (see above)

A feasible solution of a constraint satisfaction problem is a complete consistent assignment of values to variables.

- **complete** = each variable has assigned a value
- **consistent** = all constraints are satisfied

Sometimes we may look for all the feasible solutions or for the number of feasible solutions.

An optimal solution of a constraint satisfaction problem is a feasible solution that minimizes/maximizes a value of some objective function.

- **objective function** = a function mapping feasible solutions to integers

- For each variable we define its **domain.**
  - we will be using discrete finite domains only
  - such domains can be mapped to integers
- We define **constraints/relations** between the variables.

```
[X,Y] :: 0..100, 3#=X+Y, Y#>=2, X#>=1.
```

- Recall a **constraint satisfaction problem.**
- We want the system to find the values for the variables in such a way that all the constraints are satisfied.

```
X=1, Y=2
```


How is constraint satisfaction realized?

- For each variable the system keeps its actual domain.
- When a constraint is added, the inconsistent values are removed from the domain.

Example:

	X	Y
	inf..sup	inf..sup
[X,Y] :: 0..100	0..100	0..100
3#=X+Y	0..3	0..3
Y#>=2	0..1	2..3
X#>=1	1	2

SEND+MORE=MONEY

Assign different digits to letters such that SEND+MORE=MONEY holds and S≠0 and M≠0.

Idea:

generate assignments with different digits and check the constraint

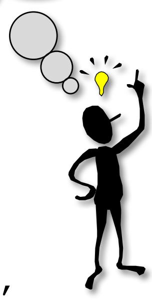
```
crypto_naive(Sol) =>
   Sol = [S,E,N,D,M,O,R,Y],
   Digits1_9 = 1..9,
   Digits0_9 = 0..9,
   member(S, Digits1_9),
   member(E, Digits0_9), E!=S,
   member(N, Digits0_9), N!=S, N!=E,
   member(D, Digits0_9), D!=S, D!=E, D!=N,
   member(M, Digits1_9), M!=S, M!=E, M!=N, M!=D,
   member(O, Digits0_9), O!=S, O!=E, O!=N, O!=D, O!=M,
   member(R, Digits0_9), R!=S, R!=E, R!=N, R!=D, R!=M, R!=O,
   member(Y, Digits0_9), Y!=S, Y!=E, Y!=N, Y!=D, Y!=M, Y!=O, Y!=R,
              1000*S + 100*E + 10*N + D +
              1000*M + 100*O + 10*R + E =
   10000*M + 1000*O + 100*N + 10*E + Y.
```

1.7 s

SEND+MORE=MONEY (better)

```
crypto_better(Sol) =>
   Sol = [S,E,N,D,M,O,R,Y],
   Digits1_9 = 1..9,
   Digits0_9 = 0..9,
   % D+E = 10*P1+Y
   member(D, Digits0_9),
   member(E, Digits0_9), E!=D,
   Y is (D+E) mod 10, Y!=D, Y!=E,
   P1 is (D+E) // 10, % carry bit

   % N+R+P1 = 10*P2+E
   member(N, Digits0_9), N!=D, N!=E, N!=Y,
   R is (10+E-N-P1) mod 10, R!=D, R!=E, R!=Y, R!=N,
   P2 is (N+R+P1) // 10,

   % E+O+P2 = 10*P3+N
   O is (10+N-E-P2) mod 10, O!=D, O!=E, O!=Y, O!=N, O!=R,
   P3 is (E+O+P2) // 10,

   % S+M+P3 = 10*M+O
   member(M, Digits1_9), M!=D, M!=E, M!=Y, M!=N, M!=R, M!=O,
   S is 9*M+O-P3,
   S>0,S<10, S!=D, S!=E, S!=Y, S!=N, S!=R, S!=O, S!=M.
```

Some letters can be computed from other letters and invalidity of the constraint can be checked before all letters are know



0.001 s

SEND+MORE=MONEY (CLP)

Domain filtering can take care about computing values for letters that depend on other letters.


```
import cp.
crypto(Sol) =>
  Sol=[S,E,N,D,M,O,R,Y],
  Sol :: 0..9,
  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,
  all_different(Sol),
  solve(Sol).
```

assign values (from domains) to variables – depth first search

Note: It is also possible to use a model with carry bits.

CLP(FD)

A typical structure of CLP programs in Picat:

```
import cp.

problem(Variables) =>

  declare_variables(Variables),

  post_constraints(Variables),

  solve(Variables).
```

Definition of CLP operators, constraints and solvers

Definition of variables and their domains

Definition of constraints

Declarative model

Control part
- **exploration of space of assignments**
- **assigning values to variables**
- **looking for one, all, or optimal solution**

Domain in Picat is a set of integers

- other values must be mapped to integers
- integers are naturally ordered

Frequently, domain is an interval

- `ListOfVariables :: MinVal..MaxVal`
- defines variables with the initial domain {MinVal,…,MaxVal}

For each variable we can define a separate domain (it is possible to use any expression providing a list of integers)

- `X :: Expr`
- `X :: [1,2,3,8,9,15]++[27,28]`

Classical arithmetic constraints with operations +,-, *, /, abs, min, max,… operations are built-in

It is possible to use comparison to define a constraint #=, #<, #>, #=<, #>=, #!=

```
Picat> A+B #=< C-2.
```

What if we define a constraint before defining the domains?

- For such variables, the system assumes initially the infinite domain -MinInt..+MaxInt

Arithmetic (reified) constraints can be connected using logical operations:

- **`#~ :Q`** negation
- **`:P #/\ :Q`** conjunction
- **`:P #\/ :Q`** disjunction
- **`:P #=> :Q`** implication
- **`:P #<=> :Q`** equivalence

P and Q could be Boolean variables (constants) or arithmetic, domain or Boolean constraints

Constraints alone frequently do not set the values to variables. We need to instantiate the variables via search.

- **`indomain(X)`**
  - assign a value to variable X (values are tried in the increasing order upon backtracking)
- **`solve(Vars)`**
  - instantiate variables in the list Vars
  - algorithm MAC – maintaining arc consistency during backtracking

`solve(:Options, +Variables)`

- variable ordering
  - **`forward, backward, degree, constr, min, max, min, ff, ffc, ffd,`** …
- value ordering
  - **`split, reverse_split`**
  - **`down, rand`**
- optimization
  - **`$min(X), $max(X)`**

Which **decision variables** are needed?

– variables denoting the problem solution

– they also define the search space

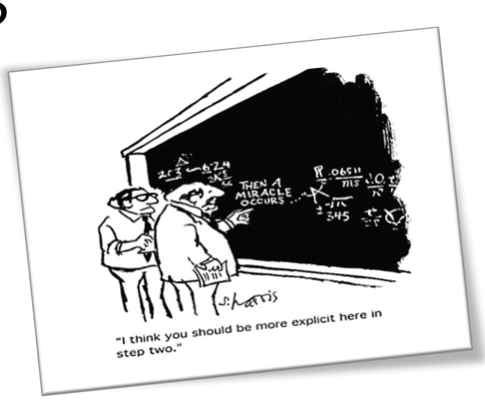
Which **values** can be assigned to variables?

– the definition of domains influences the constraints used

How to formalise **constraints**?

– available constraints

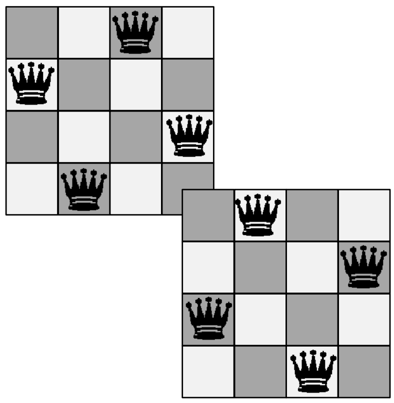
– auxiliary variables may be necessary



Propose a constraint model for solving the N-queens problem (place four queens to a chessboard of size N×N such that there is no conflict).

```
import cp.

queens(N,Queens) =>
   QR = new_list(N), QR :: 1..N,        % position in rows
   QC = new_list(N), QC :: 1..N,        % position in columns
   Queens = zip(QR,QC),                 % coordinates of queens
   foreach(I in 1..N, J in (I+1)..N)
       QR[I] #!= QR[J],                 % different rows
       QC[I] #!= QC[J],                 % different columns
       QC[I]-QR[I] #!= QC[J]-QR[J],     % different diagonals
       QC[I]+QR[I] #!= QC[J]+QR[J]
   end,
   solve(QR++QC).
```


```
Picat> queens(4,Q).
Q = [{1,2},{2,4},{3,1},{4,3}] ? ;
Q = [{1,3},{2,1},{3,4},{4,2}] ? ;
Q = [{1,2},{2,4},{4,3},{3,1}] ? ;
Q = [{1,3},{2,1},{4,2},{3,4}] ? ;
Q = [{1,2},{3,1},{2,4},{4,3}] ? ;
Q = [{1,3},{3,4},{2,1},{4,2}] ? ;
Q = [{1,2},{3,1},{4,3},{2,4}] ? ;
Q = [{1,3},{3,4},{4,2},{2,1}] ? ;
…
```



Where is the problem?

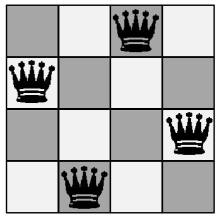
- Different assignments describe the same solution!
- There are only two different solutions (very „similar“ solutions).
- The search space is non-necessarily large.

Solution

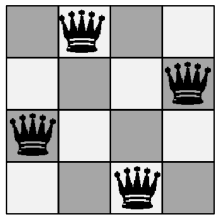
- pre-assign queens to rows (or to columns)

N-queens: a better model

```
import cp.

queens2(N,Queens) =>
   QR = 1..N,
   QC = new_list(N), QC :: 1..N,
   Queens = zip(QR,QC),
   all_different(QC),
   all_different([$QC[I]-I : I in 1..N]),
   all_different([$QC[I]+I : I in 1..N]),
   solve(QC).
```



```
Picat> queens2(4,Q).
Q = [{1,2},{2,4},{3,1},{4,3}] ?;
Q = [{1,3},{2,1},{3,4},{4,2}] ?;
no
```



Model properties:

- less variables (= smaller state space)
- less constraints (= faster propagation)

Homework:

- think about further improvements (symmetry breaking)

N-queens: a dual model

A dual model swaps the roles of values and variables.

Instead of looking for positions of queens we will be deciding whether or not a given cell contains a queen.

import ~~cp~~ **sat**.

```
queens_dual(N,Board) =>
   Board = new_array(N,N),
   Board :: 0..1,
   foreach(R in 1..N)    % exactly one queen per row
        sum([Board[R,C] : C in 1..N]) #= 1
   end,
   foreach(C in 1..N)    % exactly one queen per column
        sum([Board[R,C] : R in 1..N]) #= 1
   end,
   foreach(D in 0..(N-1)) % at most one queen per diagonal
        sum([Board[I,I+D] : I in 1..(N-D)]) #=< 1,
        sum([Board[I+D,I] : I in 1..(N-D)]) #=< 1,
        sum([Board[N-I+1,I+D] : I in 1..(N-D)]) #=< 1,
        sum([Board[N-I+1-D,I] : I in 1..(N-D)]) #=< 1
   end,
   sum([Board[R,C] : R in 1..N, C in 1..N]) #= N,
   solve(Board).

Picat> queens2(4,B).
B = {{0,0,1,0},{1,0,0,0},{0,0,0,1},{0,1,0,0}} ?;
B = {{0,1,0,0},{0,0,0,1},{1,0,0,0},{0,0,1,0}} ?;
no
```

Comment:

- The above model is much better suited for SAT.

model	#backtracks (8 queens)
naive	24
classical	24
dual	21

Notes on SAT

The constraints need to be translated to CNF (conjunctive normal form) to be solved by SAT solvers.

The Picat does the translation automatically.

Example of encoding:

$$
\begin{aligned}
& max(\{X_1, X_2, \ldots, X_n\} = Y : \\
& \quad Y = 1 \Rightarrow X_1 \vee X_2 \vee \cdots \vee X_n \\
& \quad Y = 0 \Rightarrow \neg X_1 \wedge \neg X_2 \wedge \cdots \wedge \neg X_n \\
& sum(\{X_1, X_2, \ldots, X_n\} = Y : \\
& \quad Y = 1 \Rightarrow exactly\_one(\{X_1, X_2, \ldots, X_n\}) \\
& \quad Y = 0 \Rightarrow \neg X_1 \wedge \neg X_2 \wedge \cdots \wedge \neg X_n \\
& exactly\_one(\{X_1, X_2, \ldots, X_n\}) \Leftrightarrow \\
& \quad at\_most\_one(\{X_1, X_2, \ldots, X_n\}) \wedge \\
& \quad at\_least\_one(\{X_1, X_2, \ldots, X_n\})
\end{aligned}
$$

Back to Sudoku

```
import cp.

sudoku(Board) =>
   N = Board.length,
   N1 = ceiling(sqrt(N)),
   Board :: 1..N,
   foreach(R in 1..N)
       all_different([Board[R,C] :
                          C in 1..N])
   end,
   foreach(C in 1
       all_differ
   end,
   foreach(R in 1
       all_differ

   end,
   solve(Board).
```

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

```
board(Board) =>
  Board = {{_, 6, _, 1, _, 4, _, 5, _},
           {_, _, 8, 3, _, 5, 6, _, _},
           {2, _, _, _, _, _, _, _, 1},
           {8, _, _, 4, _, 7, _, _, 6},
           {_, _, 6, _, _, _, 3, _, _},
           {7, _, _, 9, _, 1, _, _, 4},
           {5, _, _, _, _, _, _, _, 2},
           {_, _, 7, 2, _, 6, 9, _, _},
           {_, 4, _, 5, _, 8, _, 7, _}}.
```

Back to Sudoku

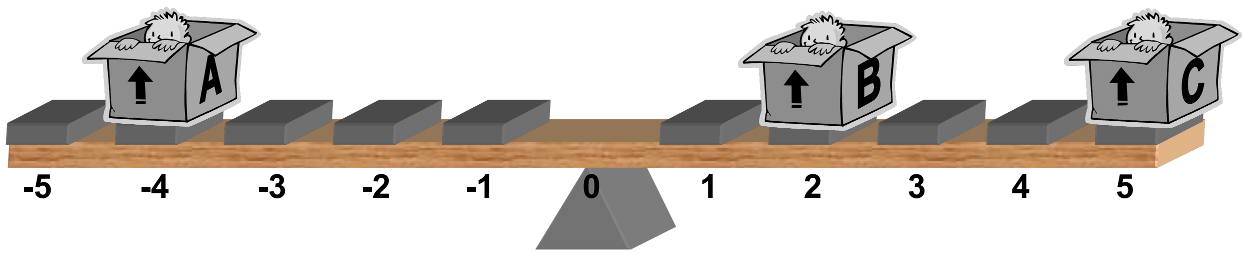
```
import cp.

sudoku(Board) =>
  N = Board.length,
  N1 = ceiling(sqrt(N)),
  Board :: 1..N,
  foreach(R in 1..N)
      all_different([Board[R,C] :
                     C in 1..N])
  end,
  foreach(C in 1..N)
      all_different([Board[R,C] : R in 1..N])
  end,
  foreach(R in 1..N1..N, C in 1..N1..N)
      all_different([Board[R+I,C+J] :
                     I in 0..N1-1, J in 0..N1-1])
  end,
  solve(Board).
```

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Seesaw problem

The problem:

Adam (36 kg), Boris (32 kg) and Cecil (16 kg)
want to sit on a seesaw with the length 10 foots
such that the minimal distances between them are more than 2 foots and the seesaw is balanced.



A CSP model:

- A,B,C in -5..5 — position
- 36*A+32*B+16*C = 0 — equilibrium state
- |A-B|>2, |A-C|>2, |B-C|>2 — minimal distances

Seesaw problem - implementation

```
import cp.

seesaw(Sol) =>
    Sol = [A,B,C],
    Sol :: -5..5,

    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

    solve(Sol).
```

```
Picat> seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;
X = [4,-5,1] ? ;
X = [4,-4,-1] ? ;
X = [4,-2,-5] ? ;

no
```

Symmetry breaking

– important to reduce search space

```
import cp.

seesaw(Sol) =>
    Sol = [A,B,C],
    Sol :: -5..5,

    A #=< 0,
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2, abs(A-C)#>2, abs(B-C)#>2,

    solve(Sol).
```

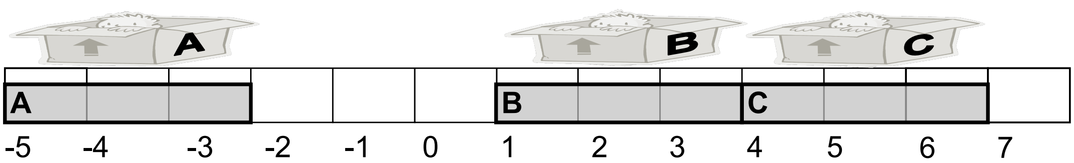
```
Picat> seesaw(X).

X = [-4,2,5] ? ;
X = [-4,4,1] ? ;
X = [-4,5,-1] ? ;

no
```

Seesaw problem - a different perspective

```
    [A,B,C] :: -5..5,
    A #=< 0,
    36*A+32*B+16*C #= 0,
    abs(A-B)#>2,
    abs(A-C)#>2,
    abs(B-C)#>2
```

```
A in -5..0
B in -2..5
C in -5..5
```

A set of similar constraints typically indicates a structured sub-problem that can be represented using a **global constraint.**



We can use a global constraint describing **allocation of activities to an exclusive resource**.

```
    [A,B,C] :: -5..5,
    A #=< 0,
    36*A+32*B+16*C #= 0,
    cumulative([A,B,C],[3,3,3],[1,1,1],1)
```

cumulative(starts,durations,resources,limit)

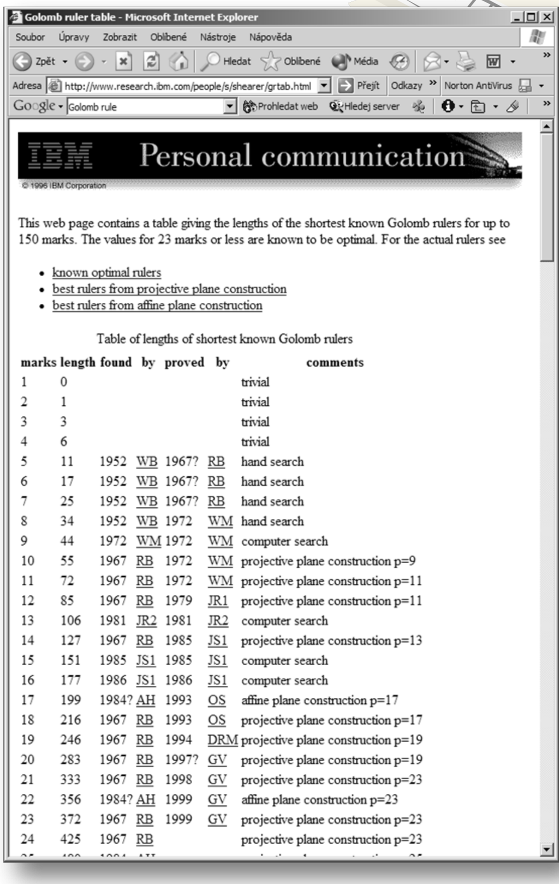
```
A in -5..0
B in -2..5
C in -5..5
```

A **ruler with M marks** such that **distances** between any two marks are **different**.

The **shortest ruler** is the optimal ruler.

Hard for $M \geq 16$, no exact algorithm for $M \geq 24$!

Applied in **radioastronomy**.



Golomb ruler table - Microsoft Internet Explorer

IBM Personal communication

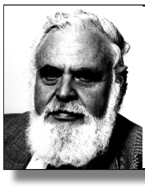
© 1996 IBM Corporation

This web page contains a table giving the lengths of the shortest known Golomb rulers for up to 150 marks. The values for 23 marks or less are known to be optimal. For the actual rulers see

- known optimal rulers
- best rulers from projective plane construction
- best rulers from affine plane construction

Table of lengths of shortest known Golomb rulers

marks	length	found	by	proved	by	comments
1	0					trivial
2	1					trivial
3	3					trivial
4	6					trivial
5	11	1952	WB	1967?	RB	hand search
6	17	1952	WB	1967?	RB	hand search
7	25	1952	WB	1967?	RB	hand search
8	34	1952	WB	1972	WM	hand search
9	44	1972	WM	1972	WM	computer search
10	55	1967	RB	1972	WM	projective plane construction p=9
11	72	1967	RB	1972	WM	projective plane construction p=11
12	85	1967	RB	1979	JR1	projective plane construction p=11
13	106	1981	JR2	1981	JR2	computer search
14	127	1967	RB	1985	JS1	projective plane construction p=13
15	151	1985	JS1	1985	JS1	computer search
16	177	1986	JS1	1986	JS1	computer search
17	199	1984?	AH	1993	OS	affine plane construction p=17
18	216	1967	RB	1993	OS	projective plane construction p=17
19	246	1967	RB	1994	DRM	projective plane construction p=19
20	283	1967	RB	1997?	GV	projective plane construction p=19
21	333	1967	RB	1998	GV	projective plane construction p=23
22	356	1984?	AH	1999	GV	affine plane construction p=23
23	372	1967	RB	1999	GV	projective plane construction p=23
24	425	1967	RB			projective plane construction p=23



Solomon W. Golomb
Professor
University of Southern California
http://csi.usc.edu/faculty/golomb.html

A base model:

Variables $X_1, \ldots, X_M$ with the domain 0..M*M

$\mathbf{X_1 = 0}$ *ruler start*

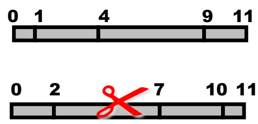
$\mathbf{X_1 < X_2 < \ldots < X_M}$ ***no permutations of variables***

$\mathbf{\forall i<j \; D_{i,j} = X_j - X_i}$ ***difference variables***

all_different$(\{D_{1,2}, D_{1,3}, \ldots D_{1,M}, D_{2,3}, \ldots D_{M-1,M}\})$

Model extensions:

$\mathbf{D_{1,2} < D_{M-1,M}}$ ***symmetry breaking***



better bounds (**implied constraints**) for $D_{i,j}$

$D_{i,j} = D_{i,i+1} + D_{i+1,i+2} + \ldots + D_{j-1,j}$

so $\mathbf{D_{i,j} \geq \Sigma_{j-i} = (j-i)*(j-i+1)/2}$ ***lower bound***

$X_M = X_M - X_1 = D_{1,M} = D_{1,2} + D_{2,3} + \ldots D_{i-1,i} + D_{i,j} + D_{j,j+1} + \ldots + D_{M-1,M}$

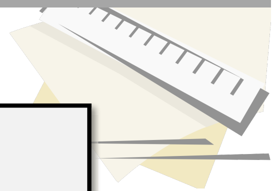
$D_{i,j} = X_M - (D_{1,2} + \ldots D_{i-1,i} + D_{j,j+1} + \ldots + D_{M-1,M})$

so $\mathbf{D_{i,j} \leq X_M - (M-1-j+i)*(M-j+i)/2}$ ***upper bound***

Golomb ruler in Picat

```
import cp.
golomb(M,X) =>
   X = new_list(M),
   X :: 0..(M*M),                          % domains for marks
   X[1] = 0,

   foreach(I in 1..(M-1))
      X[I] #< X[I+1]                       % no permutaions
   end,

   D = new_array(M,M),                     % distances
   foreach(I in 1..(M-1),J in (I+1)..M)
      D[I,J] #= X[J] - X[I],
      D[I,J] #>= (J-I)*(J-I+1)/2,          % bounds
      D[I,J] #=< X[M] - (M-1-J+I)*(M-J+I)/2
   end,

   D[1,2] #< D[M-1,M],                     % symmetry breaking
   all_different([$D[I,J] : I in 1..(M-1),
                            J in (I+1)..M]),
   solve($[min(X[M])],X).
```

Golomb ruler - some results



What is the effect of different constraint models?

size	base model	base model + symmetry	base model + symmetry + implied constraints
7	12	7	4
8	94	44	21
9	860	353	143
10	7 494	3 212	1 091
11	147 748	57 573	23 851

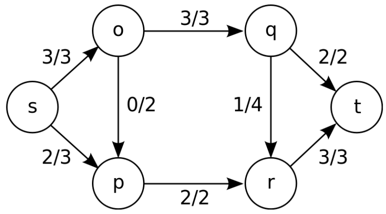
time in milliseconds on 1,7 GHz Intel Core i7, Picat 1.9#6

What is the effect of different search strategies?

size	fail first		leftmost first	
	enum	*split*	*enum*	*split*
7	9	9	5	4
8	67	68	23	21
9	537	537	170	143
10	4 834	4 721	1 217	1 091
11	134 071	132 046	26 981	23 851

time in milliseconds on 1,7 GHz Intel Core i7, Picat 1.9#6

Source: wiki




```
import mip.

maxflow(CapM,Source,Sink) =>
    N = CapM.length,
    M = new_array(N,N),
    foreach(I in 1..N, J in 1..N)    % capacity
        M[I,J] :: 0..CapM[I,J]
    end,
    foreach(I in 1..N, I != Source, I != Sink)    % conservation
        sum([M[J,I] : J in 1..N]) #= sum([M[I,J] : J in 1..N])
    end,
    Total #= sum([M[Source,I] : I in 1..N]),
    Total #= sum([M[I,Sink] : I in 1..N]),
    solve([$max(Total)],M),
    writeln(M).
```


Part III.

CLASSICAL ACTION PLANNING IN PICAT

Example: The farmer's problem

> Locations of
> Farmer, Wolf, Goat, and Cabbage

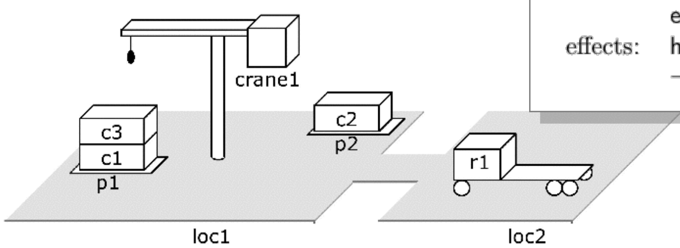
```
action([F,W,G,C],S1,Action,Cost), F=W ?=>
    Action=farmer_wolf,
    opposite(F,F1),
    S1=[F1,F1,G,C], safe(S1), Cost=1.
action([F,W,G,C],S1,Action,Cost), F=G ?=>
    Action=farmer_goat,
    opposite(F,F1),
    S1=[F1,W,F1,C], safe(S1), Cost=1.
action([F,W,G,C],S1,Action,Cost), F=C ?=>
    Action=farmer_cabbage,
    opposite(F,F1),
    S1=[F1,W,G,F1], safe(S1) , Cost=1.
action([F,W,G,C],S1,Action,Cost) =>
    Action=farmer_alone,
    opposite(F,F1),
    S1=[F1,W,G,C], safe(S1), Cost=1.
```

Modeling planning problems

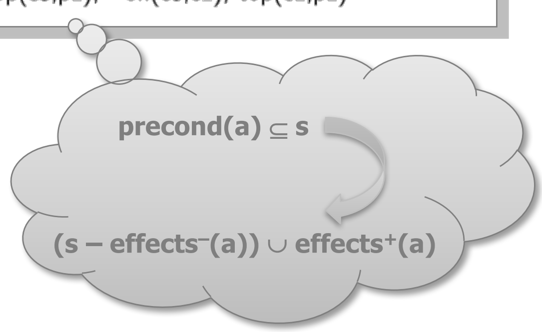
Representing **world states** as sets of atoms (factored representation).

Representing **actions** as entities changing validity of certain atoms.

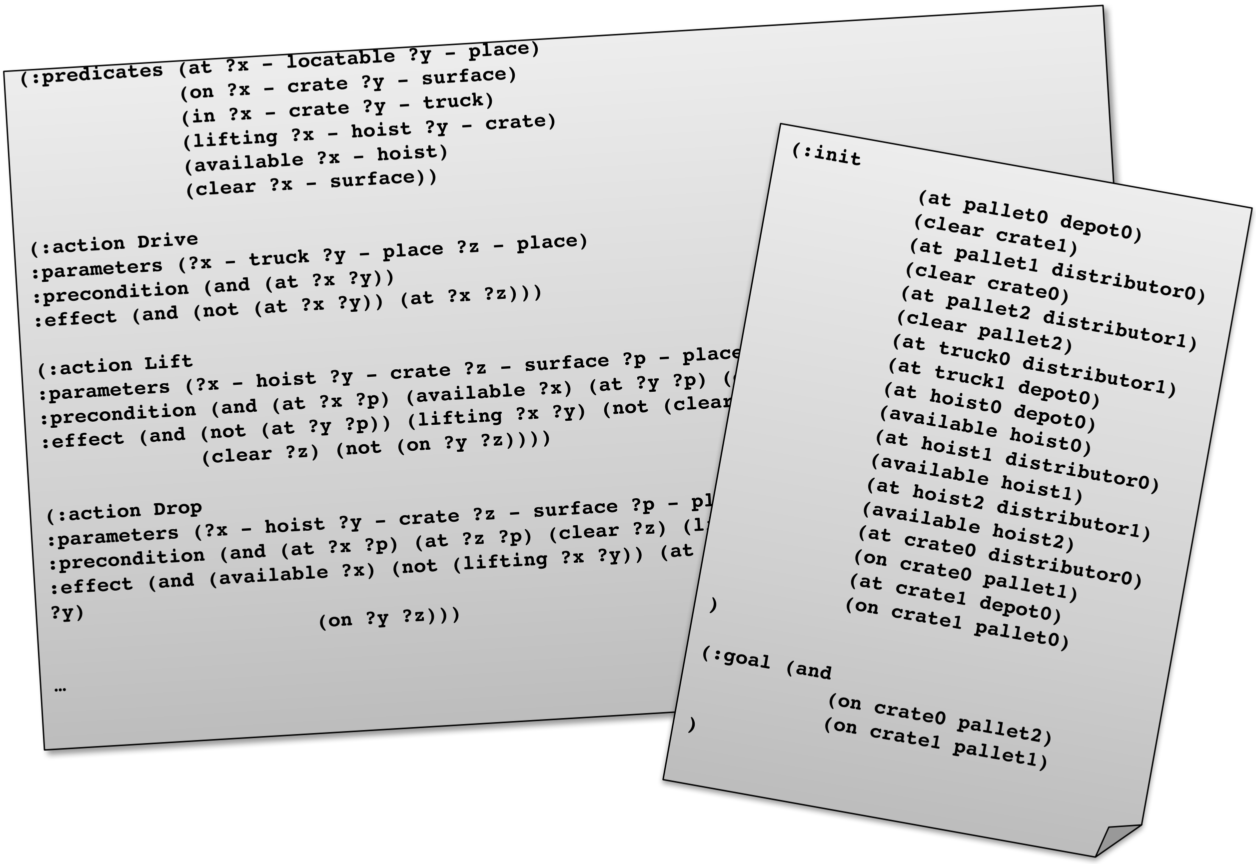
```
take(crane1,loc1,c3,c1,p1)
   ;; crane crane1 at location loc1 takes c3 off c1 in pile p1
   precond: belong(crane1,loc1), attached(p1,loc1),
            empty(crane1), top(c3,p1), on(c3,c1)
   effects: holding(crane1,c3), ¬empty(crane1), ¬in(c3,p1),
            ¬top(c3,p1), ¬on(c3,c1), top(c1,p1)
```



{attached(p1,loc1), in(c1,p1), in(c3,p1), top(c3,p1), on(c3,c1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.



Planning Domain Description Language (PDDL)


```
(:predicates (at ?x - locatable ?y - place)
             (on ?x - crate ?y - surface)
             (in ?x - crate ?y - truck)
             (lifting ?x - hoist ?y - crate)
             (available ?x - hoist)
             (clear ?x - surface))

(:action Drive
:parameters (?x - truck ?y - place ?z - place)
:precondition (and (at ?x ?y))
:effect (and (not (at ?x ?y)) (at ?x ?z)))

(:action Lift
:parameters (?x - hoist ?y - crate ?z - surface ?p - place
:precondition (and (at ?x ?p) (available ?x) (at ?y ?p) (
:effect (and (not (at ?y ?p)) (lifting ?x ?y) (not (clear
             (clear ?z) (not (on ?y ?z))))

(:action Drop
:parameters (?x - hoist ?y - crate ?z - surface ?p - pl
:precondition (and (at ?x ?p) (at ?z ?p) (clear ?z) (l
:effect (and (available ?x) (not (lifting ?x ?y)) (at
?y)                   (on ?y ?z)))

…
```

```
(:init
            (at pallet0 depot0)
            (clear crate1)
            (at pallet1 distributor0)
            (clear crate0)
            (at pallet2 distributor1)
            (clear pallet2)
            (at truck0 distributor1)
            (at truck1 depot0)
            (at hoist0 depot0)
            (available hoist0)
            (at hoist1 distributor0)
            (available hoist1)
            (at hoist2 distributor1)
            (available hoist2)
            (at crate0 distributor0)
            (on crate0 pallet1)
            (at crate1 depot0)
            (on crate1 pallet0)
)

(:goal (and
            (on crate0 pallet2)
)           (on crate1 pallet1)
```

State-space planning

The search space corresponds to the state space of the planning problem.

- search nodes correspond to world states
- arcs correspond to state transitions by means of actions
- the task is to find a path from the initial state to some goal state

Basic approaches

- forward search (progression)
  - start in the initial state and apply actions until reaching a goal state
- backward search (regression)
  - start with the goal and apply actions in the reverse order until a subgoal satisfying the initial state is reached
  - lifting (actions are only partially instantiated)

Domain knowledge

Heuristics guide the planner towards a goal state by ordering alternative plans. They do not solve the problem with the **large number of alternatives**.

Example (blockworld)

- If a block is placed correctly (consistent with the goal) then any action that moves that block just enlarges the plan.
- If a block is on a wrong place and there is an action that moves it to the correct place then any action that moves the block elsewhere just enlarges the plan.

It is possible to describe desirable/forbidden sequences of states using linear temporal logic.

- **control rules**

It is possible to describe expected plans via task decompositions.

- **hierarchical task networks**

Control rules in practice

Planners with control rules (TLPlan, TALPlanner) — HTN planning (SHOP2) — Forward planning (FF)

Domain	# insts	TLPlan	TALPlanner	SHOP2	FF
Depots	22	**22**	**22**	**22**	**22**
DriverLog	20	**20**	**20**	**20**	15
Zenotravel	20	**20**	**20**	**20**	**20**
Rovers	20	**20**	**20**	**20**	**20**
Satellite	20	**20**	**20**	**20**	**20**
Total	-	**894 (100%)**	**610 (100%)**	**899 (99%)**	237 (83%)

problems solved

Control rules in practice

```
…
(forall (?x ?y) (on ?x ?y)
          (and
          (print ?stream "(on ~A ~A) --" ?x ?y)
          (implies (good-tower ?x)
                   (print ?stream "  (good-tower ~A) " ?x))
          (implies (bad-tower ?x)
                   (print ?stream "  (bad-tower ~A) " ?x))
          (implies (good-tower ?y)
                   (print ?stream "  (good-tower ~A)~%" ?y))
          (implies (bad-tower ?y)
                   (print ?stream "  (bad-tower ~A)~%" ?y))))

   (forall (?x ?y) (in ?x ?y)
          (and
          (print ?stream "(in ~A ~A) " ?x ?y)
          (exists (?l) (at ?y ?l)
                   (print ?stream "(at ~A ~A) " ?y ?l))
          (implies (has-goal-loc ?x)
                   (print ?stream "(crate-goal-location ~A) = ~A (crate-goal-surface ~A)= ~A"
                          ?x (crate-goal-location ?x) ?x (crate-goal-surface ?x)))

          (print ?stream "~%")))
…
```

933 lines of code!

Picat planning module

Forward planning in Picat language (using tabling):

```
table (+,-,min)
plan(S,Plan,Cost),final(S) =>
    Plan=[],Cost=0.
plan(S,Plan,Cost) =>
    action(S,S1,Action,ActionCost),
    plan(S1,Plan1,Cost1),
    Plan = [Action|Plan1],
    Cost = Cost1+ActionCost.
```

Cost optimization done via:

- iterative deepening (**best_plan**)
- branch-and-bound (**best_plan_bb**)

Picat Planning Domain Model

Goal condition

```
final(+State) => goal_condition.
```

Action description

```
action(+State,-NextState,-Action,-Cost),
   precondition,
   [control_knowledge]
?=>
   description_of_next_state,
   action_cost_calculation,
   [heuristic_and_deadend_verification].
```

15-Puzzle

4		3	6
12	1	11	7
9	5	10	15
13	8	14	2

Initial state

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Goal state

State representation

Position of gap | Position of 1 | Position of 2

```
main =>
    Init = [(1,2),(2,2),(4,4),(1,3),(1,1),(3,2),(1,4),(2,4),
            (4,2),(3,1),(3,3),(2,3),(2,1),(4,1),(4,3),(3,4)],
    best_plan(Init,Plan).

final(S) => S = [(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),
                 (3,1),(3,2),(3,3),(3,4),(4,1),(4,2),(4,3),(4,4)].
```

15-Puzzle: Actions

```
action([P0@(R0,C0)|Tiles],NextS,Action,Cost) =>
    Cost = 1,
    (R1 = R0-1, R1 >= 1, C1 = C0, Action = up;
     R1 = R0+1, R1 =< 4, C1 = C0, Action = down;
     R1 = R0, C1 = C0-1, C1 >= 1, Action = left;
     R1 = R0, C1 = C0+1, C1 =< 4, Action = right),
    P1 = (R1,C1),
    slide(P0,P1,Tiles,NTiles),
    NextS = [P1|NTiles].

% slide the tile at P1 to the empty square at P0
slide(P0,P1,[P1|Tiles],NTiles) =>
    NTiles = [P0|Tiles].
slide(P0,P1,[Tile|Tiles],NTiles) =>
    NTiles=[Tile|NTilesR],
    slide(P0,P1,Tiles,NTilesR).
```

15-Puzzle: Heuristics and Performance

Heuristic function

```
heuristic(Tiles) = Dist =>
    final([_|FTiles]),
    Dist = sum([abs(R-FR)+abs(C-FC) :
               {(R,C),(FR,FC)} in zip(Tiles,FTiles)]).
```

Performance

- Picat planner easily solves 15-puzzle instances
- It can even solve some hard 24-puzzle instances if a better heuristic is used

NoMystery problem

A truck moves between locations to pickup and deliver packages while consuming fuel during moves.

- setting:
  - initial locations of packages and truck
  - goal locations of packages
  - initial fuel level, fuel cost for moving between locations
- possible actions: **load**, **unload**, **drive**
- assumption: track can carry any number of packages

Nomystery: State representation

Factored representation

- state = a set of atoms that hold in that state (a vector of values of state variables)

```
{at(p0,l2),at(p1,l2),at(p2,l1),at(t0,l2),
 in(p3,t0),in(p4,t0),in(p5,t0),
 fuel(t0,level84)}
```

Structured representation

- state = a term describing objects and their relations

  objects represented by properties rather than by names to break object symmetries

```
s(l2, level84, [l2,l2,l4], [[l1|l3], [l2|l3], [l2|l4]])
```

truck location

fuel level

destinations of loaded packages

current and desired locations of waiting packages

Factored representation

```
action(S,NextS,Act,Cost),
   truck(T), member(at(T,L),S),
   select(at(P,L),S,RestS), P != T
?=>
   Act = load(L,P,T), Cost = 1,
   NewS = insert_ordered(RestS,in(P,T)).
```

Structured representation

```
action(s(Loc,Fuel,LPs,WPs),NextS,Act,Cost),
   select([Loc|PkGoal],WPs,WPs1)
?=>
   Act = load(Loc,PkGoal), Cost = 1,
   LPs1 = insert_ordered(LPs,PkGoal),
   NextS = s(Loc,Fuel,LPs1,WPs1).
```


Estimate distance to goal

Precise heuristic for Nomystery domain:

- each package must be loaded and unloaded
- each place with packages to load or unload must be visited

```
action(S,NextS,Act,Cost),
   truck(T), member(at(T,L),S),
   select(at(P,L),S,RestS), P != T
?=>
   Act = load(L,P,T), Cost = 1,
   NewS = insert_ordered(RestS,in(P,T)),
   heuristics(NewS) < current_resource().
```

Tell the planner what to do at a given state based on the goal

- unload all packages destined for current location (and only those packages)

```
action(s(Loc,Fuel,LoadedPks,WaitPks), NextState, Action, Cost),
       select(Loc,LoadedPks,LoadedPks1)
=>
       Action = unload(Loc,Loc),
       NextState = s(Loc,Fuel,LoadedPks1, WaitPks),
       Cost = 1.
```

- load all undelivered packages at current location
- move somewhere
  - move to a location with waiting package or to a destination of some loaded package

NoMystery Model

```
action(s(Loc,Fuel,LoadedCGs,Cargoes), NextState, Action, Cost),
    select(Loc,LoadedCGs,LoadedCGs1)
=>
    Action = unload(Loc,Loc),
    NextState = s(Loc,Fuel,LoadedCGs1,Cargoes), Cost = 1.

Action(s(Loc,Fuel,LoadedCGs,Cargoes), NextState, Action, Cost),
    select([Loc|CargoGoal],Cargoes,Cargoes1)
=>
    insert_ordered(CargoGoal,LoadedCGs,LoadedCGs1),
    Action = load(Loc,CargoGoal),
    NextState = s(Loc,Fuel,LoadedCGs1,Cargoes1) , Cost = 1.

Action(s(Loc,Fuel,LoadedCGs,Cargoes), NextState, Action, Cost)
?=>
    Action = drive(Loc,Loc1),
    NextState = s(Loc1,Fuel1,LoadedCGs,Cargoes),
    fuelcost(FuelCost,Loc,Loc1),
    Fuel1 is Fuel-FuelCost,
    Fuel1 >= 0, Cost = 1.
```

Experimental setting

Four domains from International Planning Competitions:

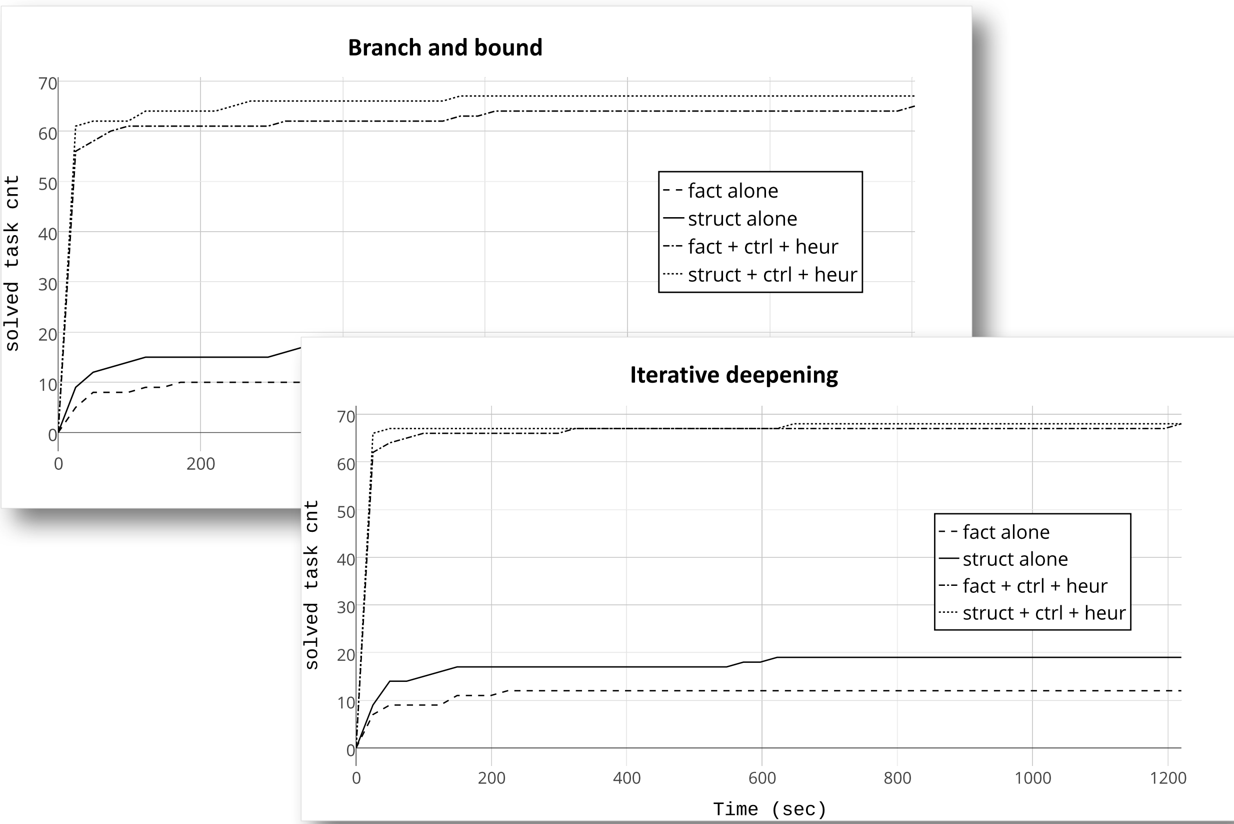
domain	#instances	#optimal
Depots	20	13
Nomystery	30	30
Visitall	20	5
Childsnack	20	20

For each domain the following models (each for structured and factored representation of states):

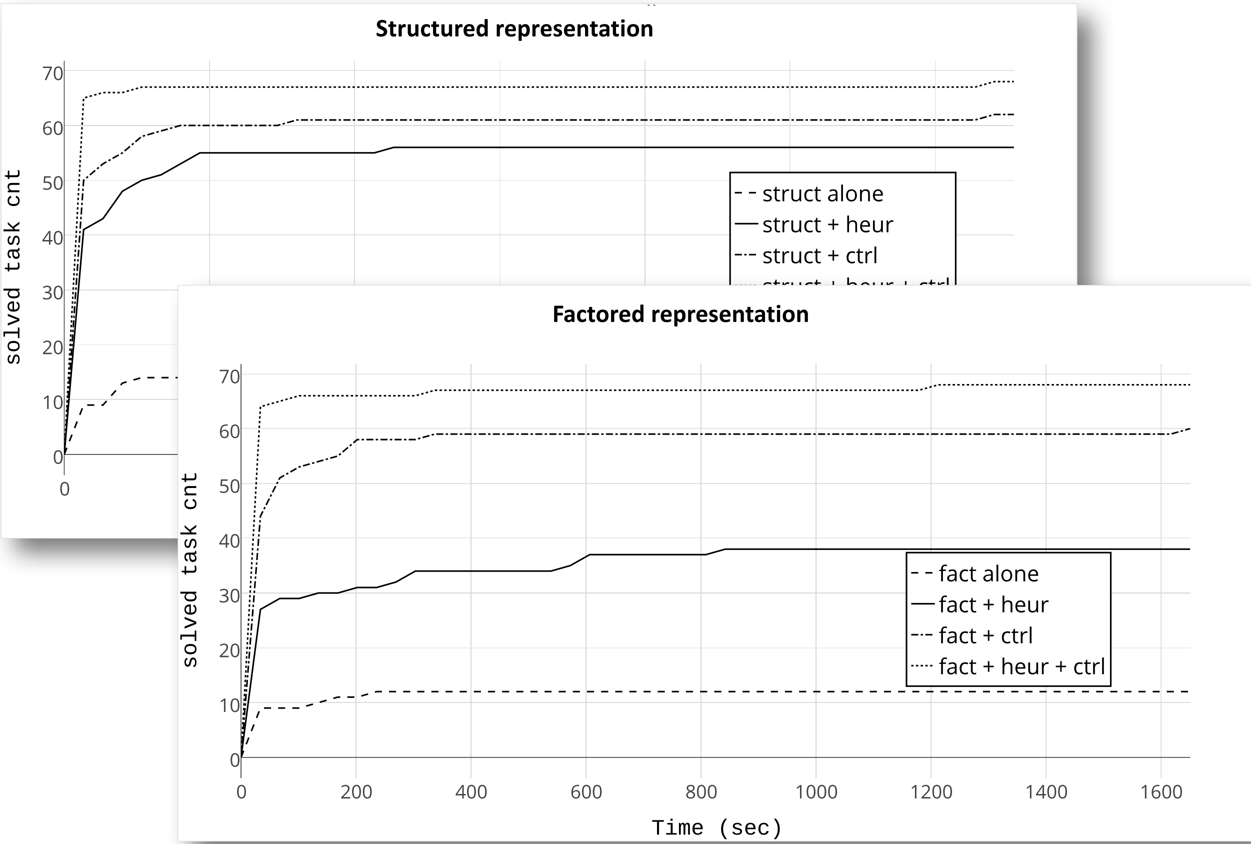
- pure model ("physics only")
- model with heuristics
- model with control knowledge
- model with heuristics + control knowledge

Compare #solved problems (30 minutes per problem)

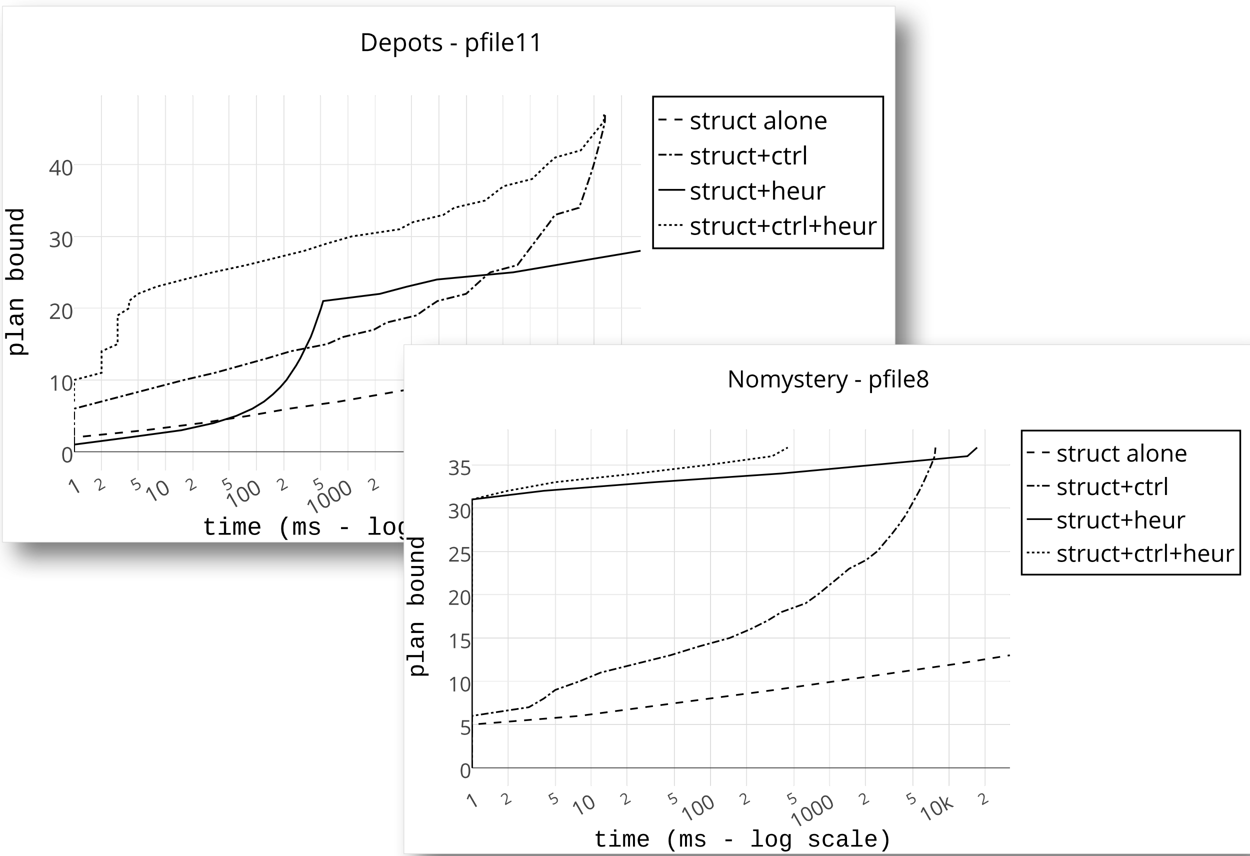
Factored vs. structured representations



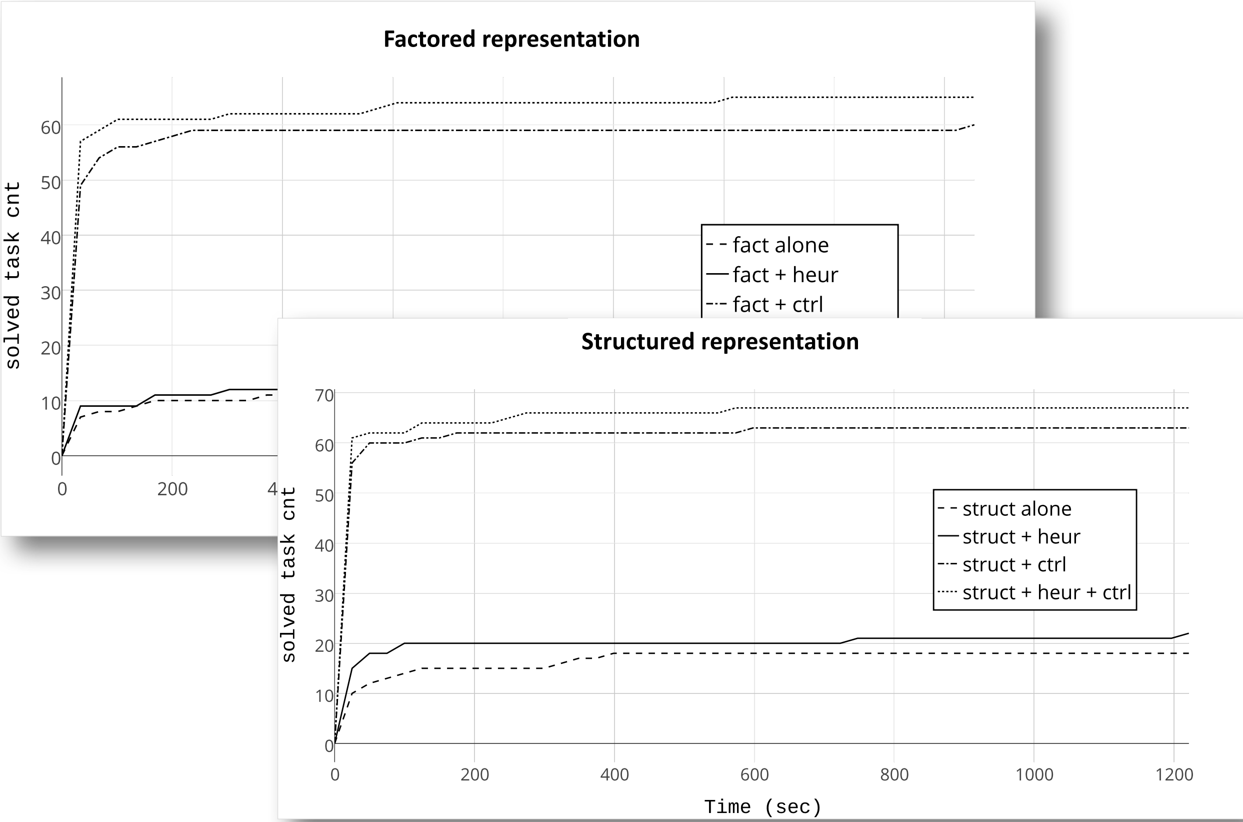
Heuristics vs. control knowledge (ID)



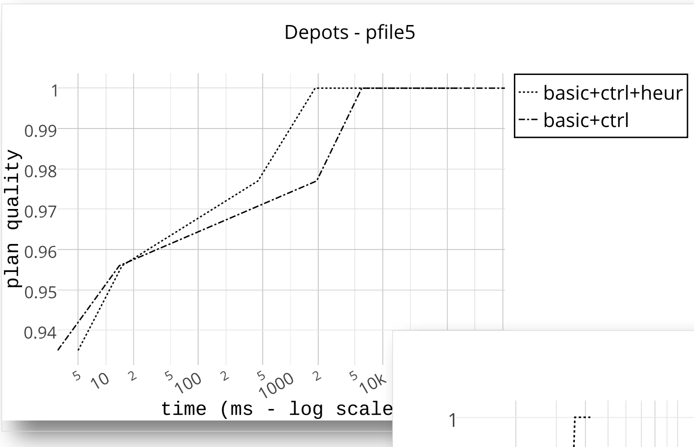
ID behavior



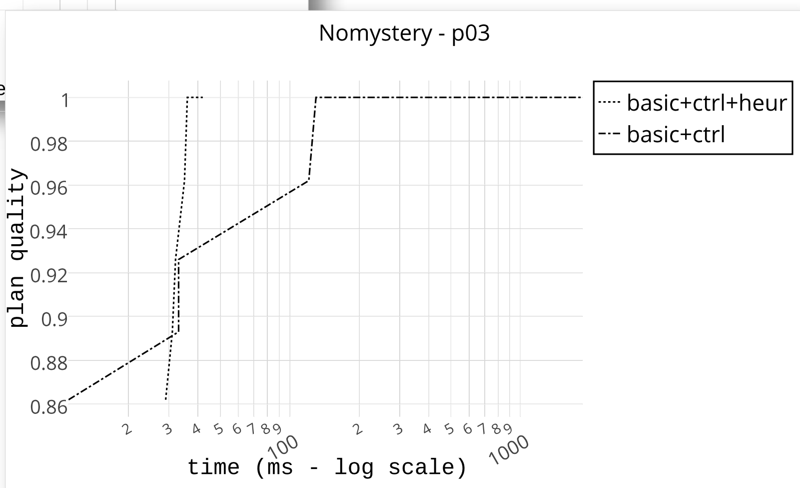
Heuristics vs. control knowledge (B-and-B)



B-and-B behavior



$$\text{quality\_score} = \frac{Q^*}{Q}$$



Comparison to PDDL planners

no tabling used

IPC 2014 winner

Domain	# insts	Picat	Picat-nt	SymbA
Barman	14	**14**	0	6
Cave	20	**20**	0	3
Childsnack	20	**20**	**20**	3
Citycar	20	**20**	17	17
Floortile	20	**20**	0	**20**
GED	20	**20**	19	19
Parking	20	**11**	4	1
Tetris	17	**13**	**13**	10
Transport	20	**10**	0	8

number of optimally solved problems

Comparison to domain-dependent planners

Planners with control rules

Task hierarchies

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**22**	**22**	**22**	**22**
Zenotravel	20	**20**	**20**	**20**	**20**
Driverlog	20	**20**	**20**	**20**	**20**
Satellite	20	**20**	**20**	**20**	**20**
Rovers	20	**20**	**20**	**20**	**20**
Total	102	**102**	**102**	**102**	**102**

problems solved

Comparison to domain-dependent planners

Planners with control rules (TLPlan, TALPlanner)

Task hierarchies (SHOP2)

Domain	# insts	Picat	TLPlan	TALPlanner	SHOP2
Depots	22	**21.94**	19.93	20.52	18.63
Zenotravel	20	**19.86**	18.40	18.79	17.14
Driverlog	20	17.21	17.68	**17.87**	14.16
Satellite	20	**20.00**	18.33	16.58	17.16
Rovers	20	**20.00**	17.67	14.61	17.57
Total	102	**99.01**	92.00	88.37	84.65

quality score (after 5 mins)

Comparison to domain-dependent planners

Domain	PDDL	Picat	TLPlan
Depots	42	156	933
Zenotravel	61	109	308
Driverlog	79	190	1395
Satellite	75	132	186
Rovers	119	223	914
Total	376	810	3736

encoding size

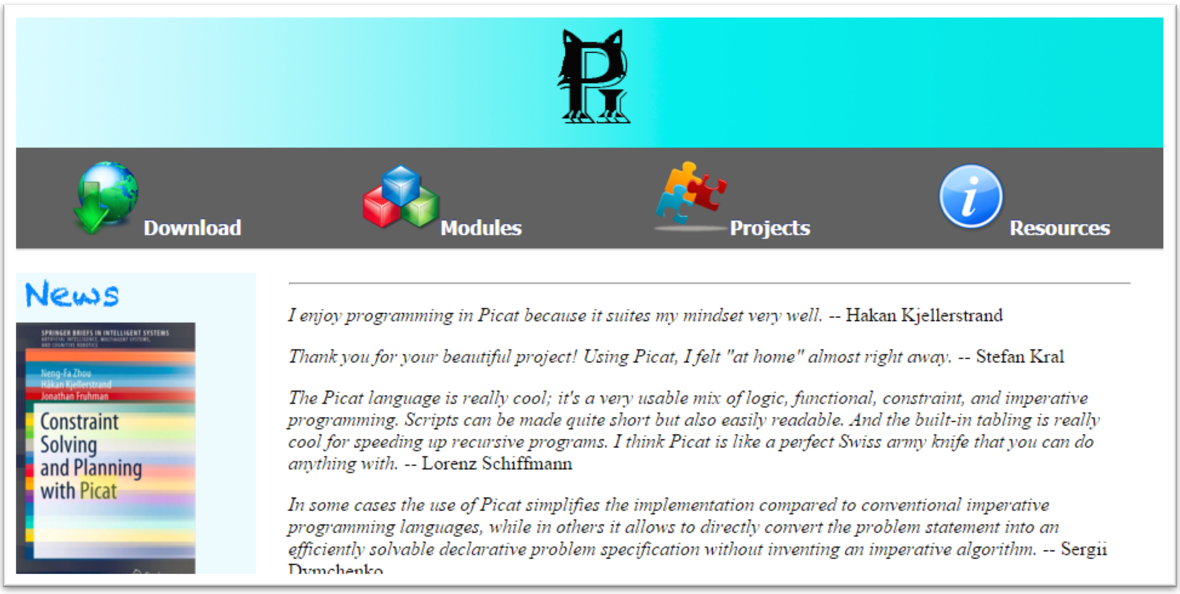
WRAP UP

Summary

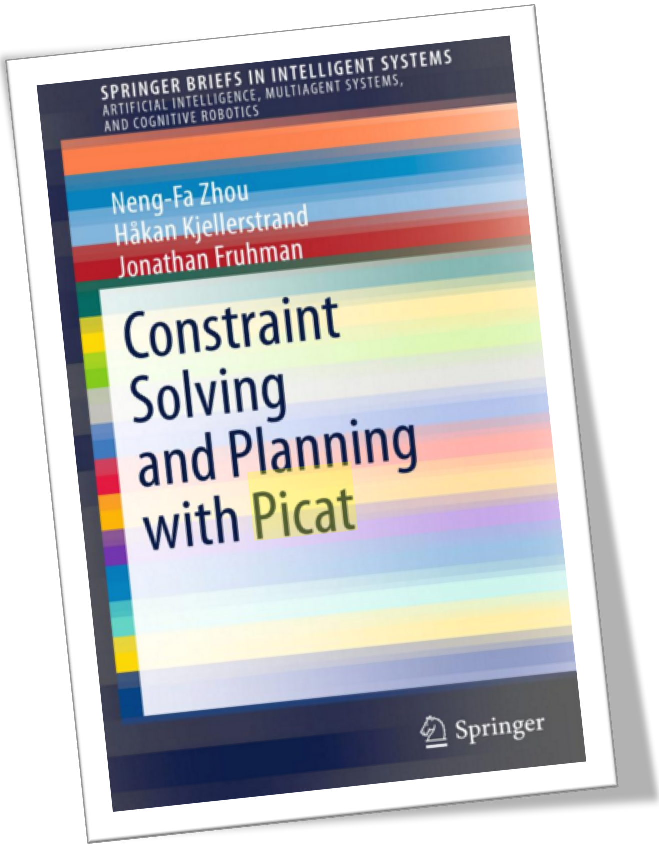
Picat is a logic-based multi-paradigm language that integrates logic programming, functional programming, constraint programming, and scripting.

- logic variables, unification, backtracking, pattern-matching rules, functions, list/array comprehensions, loops, assignments
- tabling for dynamic programming and **planning**
- **constraint solving** with CP (constraint programming), SAT (satisfiability), and MIP (mixed integer programming).

picat-lang.org



Picat book



1. H. Kjellerstrand:
   Picat: A Logic-based Multi-paradigm Language, ALP Newsletter, 2014.
2. R. Barták and N.-F. Zhou:
   Using Tabled Logic Programming to Solve the Petrobras Planning Problem, TPLP 2014.
3. R. Barták, A. Dovier, and N.-F. Zhou:
   On Modeling Planning Problems in Tabled Logic Programming, PPDP 2015.
4. S. Dymchenko and M. Mykhailova:
   Declaratively Solving Google Code Jam Problems with Picat, PADL 2015.
5. S. Dymchenko:
   An Introduction to Tabled Logic Programming with Picat, Linux Journal, August, 2015.
6. N.-F. Zhou:
   Combinatorial Search With Picat, ICLP invited talk, 2014.
7. N.-F. Zhou, R. Barták, and A. Dovier:
   Planning as Tabled Logic Programming, TPLP 2015.
8. N.-F. Zhou, H. Kjellerstrand, and J. Fruhman:
   Constraint Solving and Planning with Picat, Springer, 2015.
9. N.-F. Zhou, H. Kjellerstrand:
   The Picat-SAT Compiler, PADL 2016.