Solving Puzzles with Constraint Programming

TEFcon 2022

Adrián Arroyo Calle

$ whoami

HaaC - Agente Único

https://adrianistan.eu

Contributor to Scryer Prolog

THE MYTH OF "CONSENSUAL" SEX
I CONSENT
I CONSENT
SWI Prolog
ISN'T THERE SOMEBODY YOU FORGOT TO ASK?

```
?- sex(X, Y).
   false.
?- 
```

We’re going to solve puzzles!

With the computer help

And without writing an algorithm

State problems in a declarative way

Then ask the solver for a solution

Constraint Satisfaction Problem

Mainly we’ll use how to solve them using Constraint Programming, but we’ll discuss other approaches too

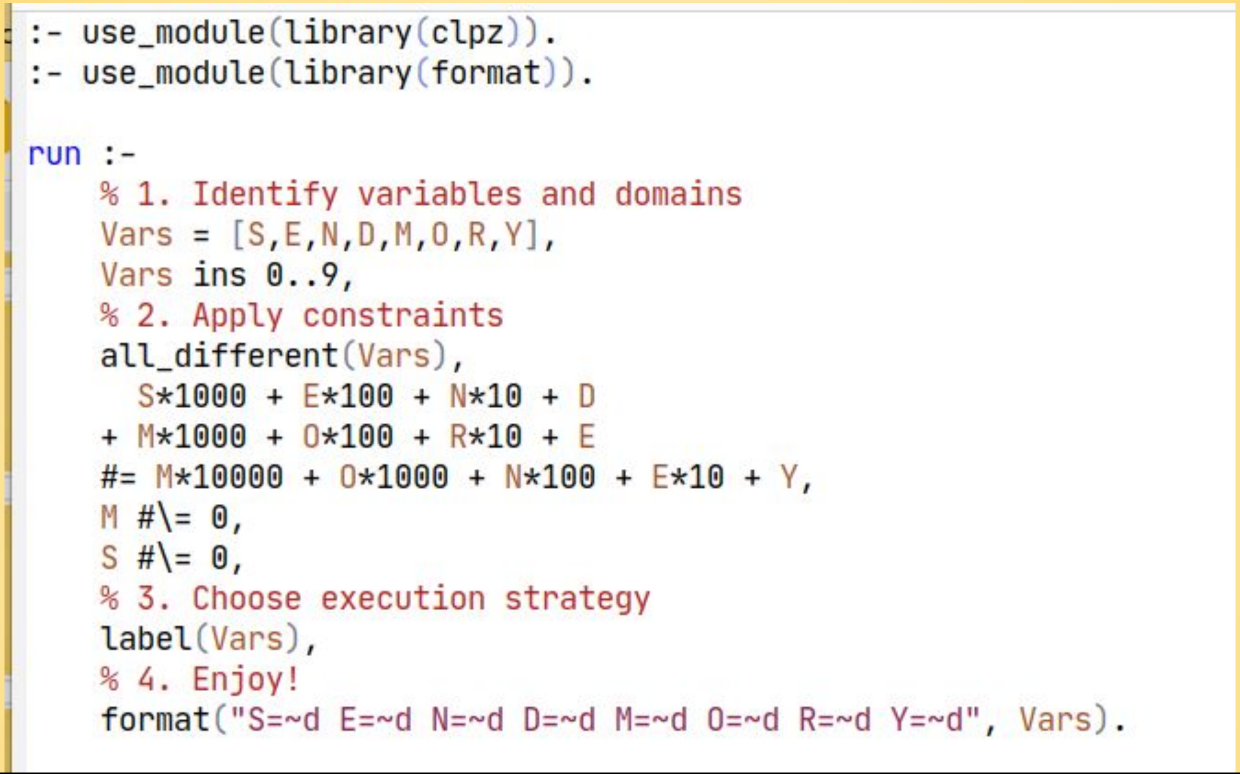
```
    SEND
+   MORE
-------
= MONEY
```

Each letter should be substituted by a digit between 0 and 9. The sum must be true

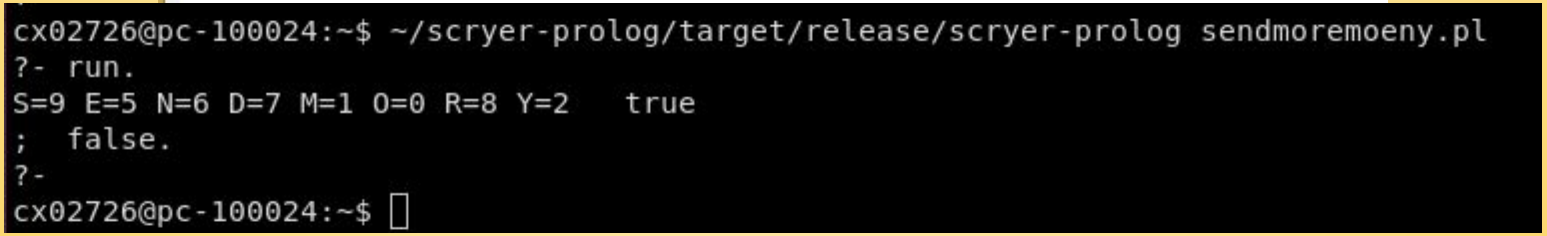
We can model it with constraints and ask the solver for the solution!

Let’s do it on Scryer Prolog

1. Identify the variables of the problem (and their domains)
2. Apply constraints on them
3. Choose an execution strategy
4. Enjoy!



```prolog
:- use_module(library(clpz)).
:- use_module(library(format)).

run :-
    % 1. Identify variables and domains
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    % 2. Apply constraints
    all_different(Vars),
      S*1000 + E*100 + N*10 + D
    + M*1000 + O*100 + R*10 + E
    #= M*10000 + O*1000 + N*100 + E*10 + Y,
    M #\= 0,
    S #\= 0,
    % 3. Choose execution strategy
    label(Vars),
    % 4. Enjoy!
    format("S=~d E=~d N=~d D=~d M=~d O=~d R=~d Y=~d", Vars).
```

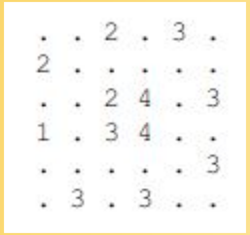


```
cx02726@pc-100024:~$ ~/scryer-prolog/target/release/scryer-prolog sendmoremoeny.pl
?- run.
S=9 E=5 N=6 D=7 M=1 O=0 R=8 Y=2   true
;  false.
?-
cx02726@pc-100024:~$ 
```

Overview of techniques in the CSP field

- CP (Constraint Programming). Apply constraint propagation over the domains to reduce the ranges of valid values. If it still doesn’t get a solution, starts a search strategy.
- SAT. Boolean satisfiability problem. The problem gets mapped to the 3SAT and solved using algorithms like DPLL or CDCL.
- MIP (Mixed Integer Programming). Optimize an objective function. The evolution of George Dantzig algorithms (Simplex) including integer variables.

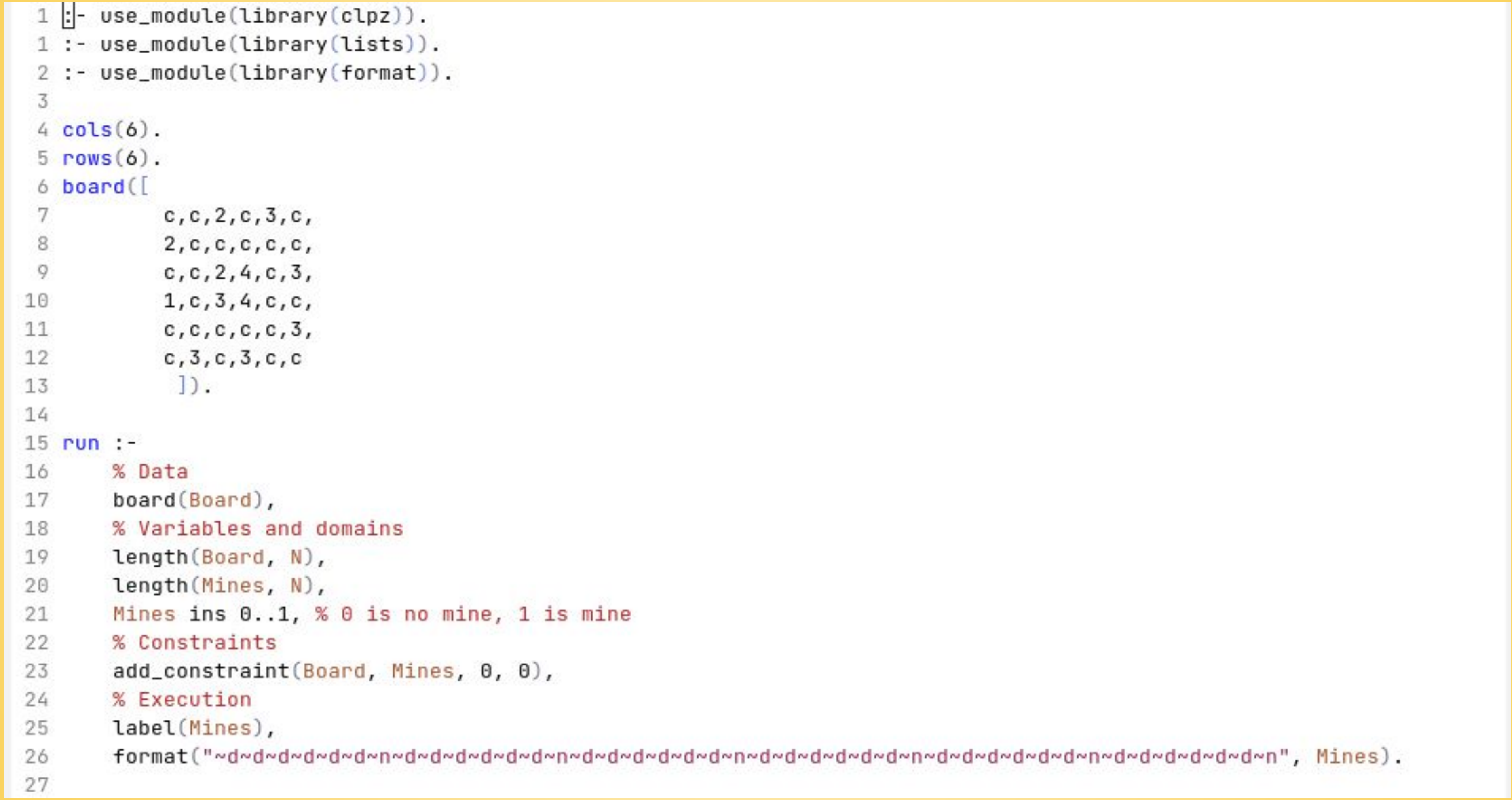
Minesweeper



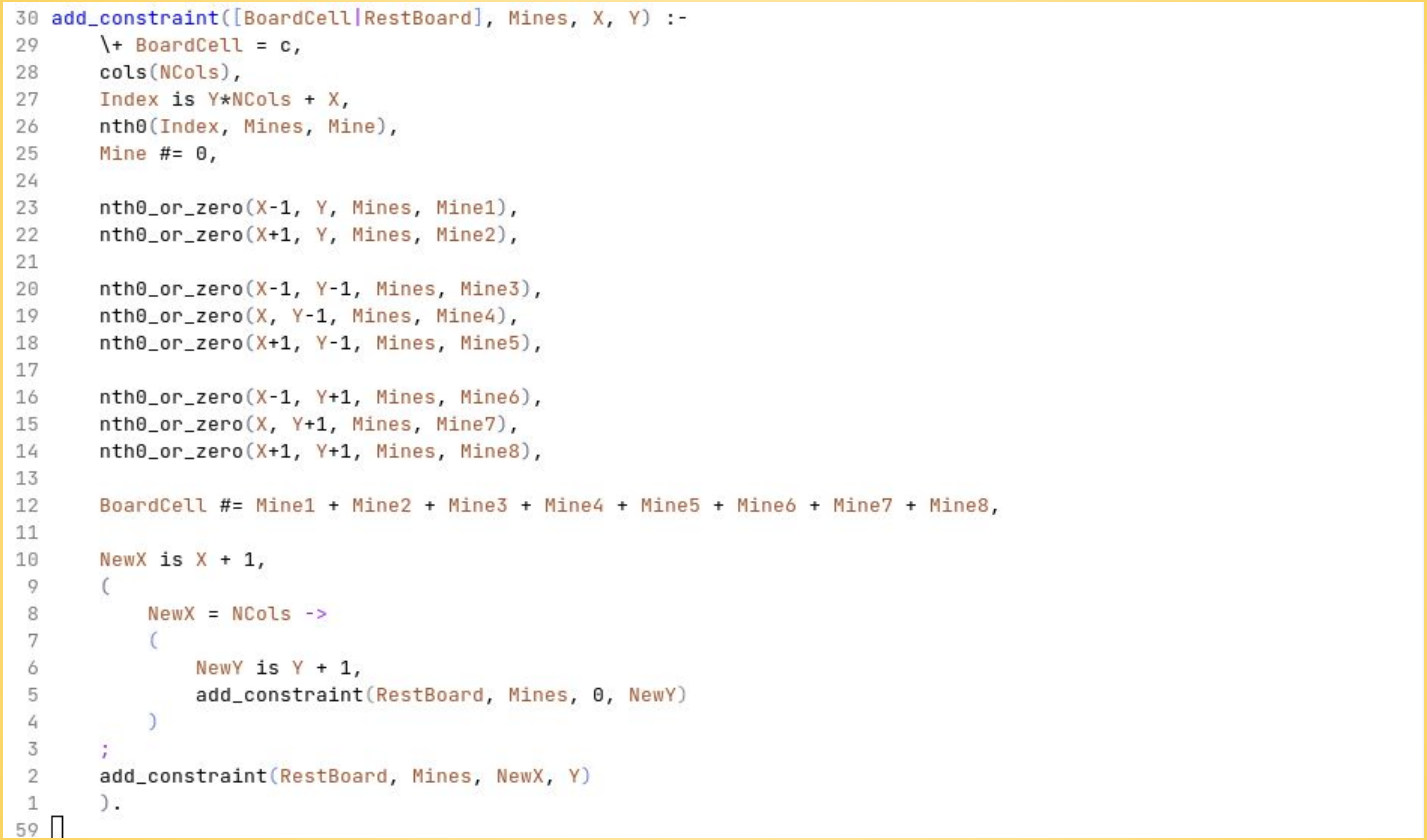
```
. . 2 . 3 .
2 . . . . .
. . 2 4 . 3
1 . 3 4 . .
. . . . . 3
. 3 . 3 . .
```

Unlike the GUI program, in this problem we already have unlocked some cells. They tell us how many mines are around that cell. Determine which locked cells are mines and which are not.

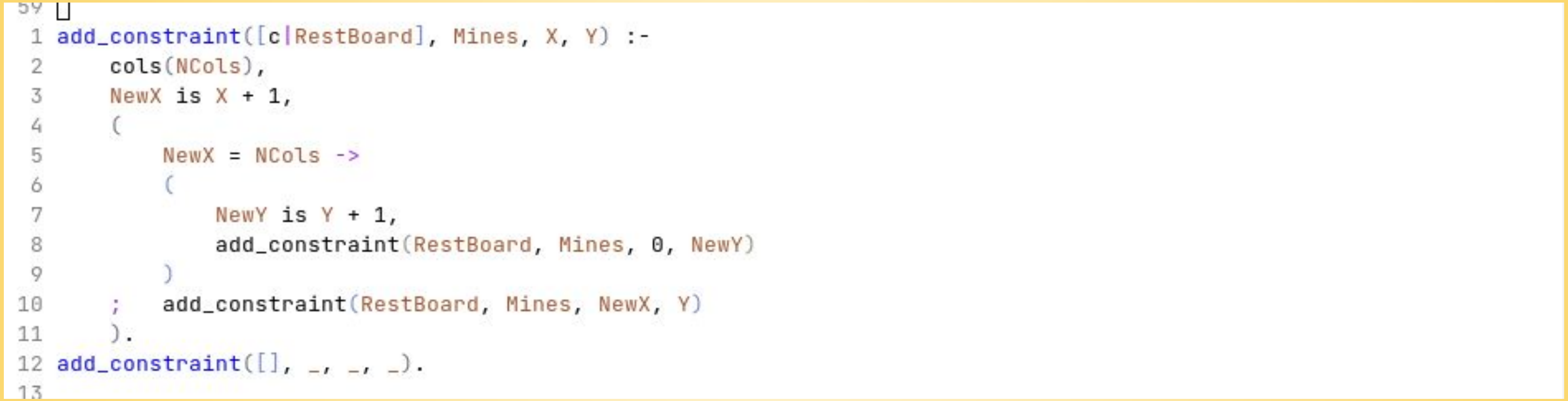
The idea is simple: have two matrices. One with the board information and the other full of variables. Each cell can contain a mine (1) or not (0)



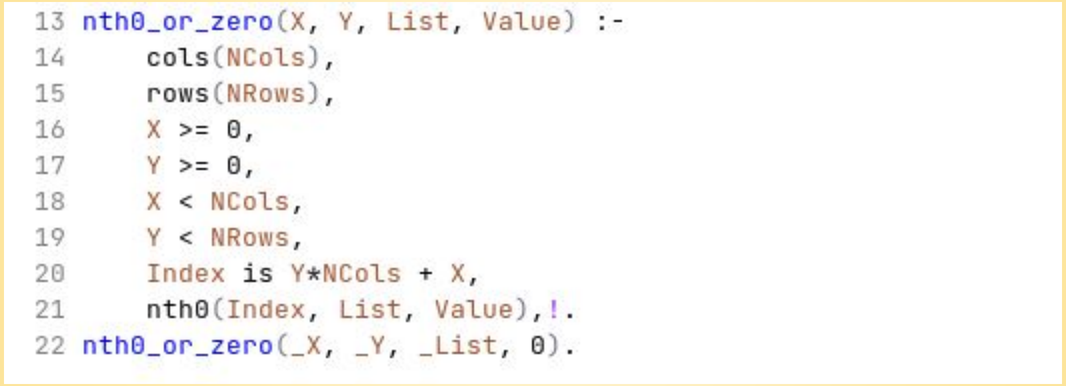
```prolog
:- use_module(library(clpz)).
:- use_module(library(lists)).
:- use_module(library(format)).

cols(6).
rows(6).
board([
        c,c,2,c,3,c,
        2,c,c,c,c,c,
        c,c,2,4,c,3,
        1,c,3,4,c,c,
        c,c,c,c,c,3,
        c,3,c,3,c,c
         ]).

run :-
    % Data
    board(Board),
    % Variables and domains
    length(Board, N),
    length(Mines, N),
    Mines ins 0..1, % 0 is no mine, 1 is mine
    % Constraints
    add_constraint(Board, Mines, 0, 0),
    % Execution
    label(Mines),
    format("~d~d~d~d~d~d~n~d~d~d~d~d~d~n~d~d~d~d~d~d~n~d~d~d~d~d~d~n~d~d~d~d~d~d~n~d~d~d~d~d~d~n", Mines).
```



```prolog
add_constraint([BoardCell|RestBoard], Mines, X, Y) :-
    \+ BoardCell = c,
    cols(NCols),
    Index is Y*NCols + X,
    nth0(Index, Mines, Mine),
    Mine #= 0,

    nth0_or_zero(X-1, Y, Mines, Mine1),
    nth0_or_zero(X+1, Y, Mines, Mine2),

    nth0_or_zero(X-1, Y-1, Mines, Mine3),
    nth0_or_zero(X, Y-1, Mines, Mine4),
    nth0_or_zero(X+1, Y-1, Mines, Mine5),

    nth0_or_zero(X-1, Y+1, Mines, Mine6),
    nth0_or_zero(X, Y+1, Mines, Mine7),
    nth0_or_zero(X+1, Y+1, Mines, Mine8),

    BoardCell #= Mine1 + Mine2 + Mine3 + Mine4 + Mine5 + Mine6 + Mine7 + Mine8,

    NewX is X + 1,
    (
        NewX = NCols ->
        (
            NewY is Y + 1,
            add_constraint(RestBoard, Mines, 0, NewY)
        )
    ;
    add_constraint(RestBoard, Mines, NewX, Y)
    ).
```



```prolog
add_constraint([c|RestBoard], Mines, X, Y) :-
    cols(NCols),
    NewX is X + 1,
    (
        NewX = NCols ->
        (
            NewY is Y + 1,
            add_constraint(RestBoard, Mines, 0, NewY)
        )
    ;   add_constraint(RestBoard, Mines, NewX, Y)
    ).
add_constraint([], _, _, _).
```



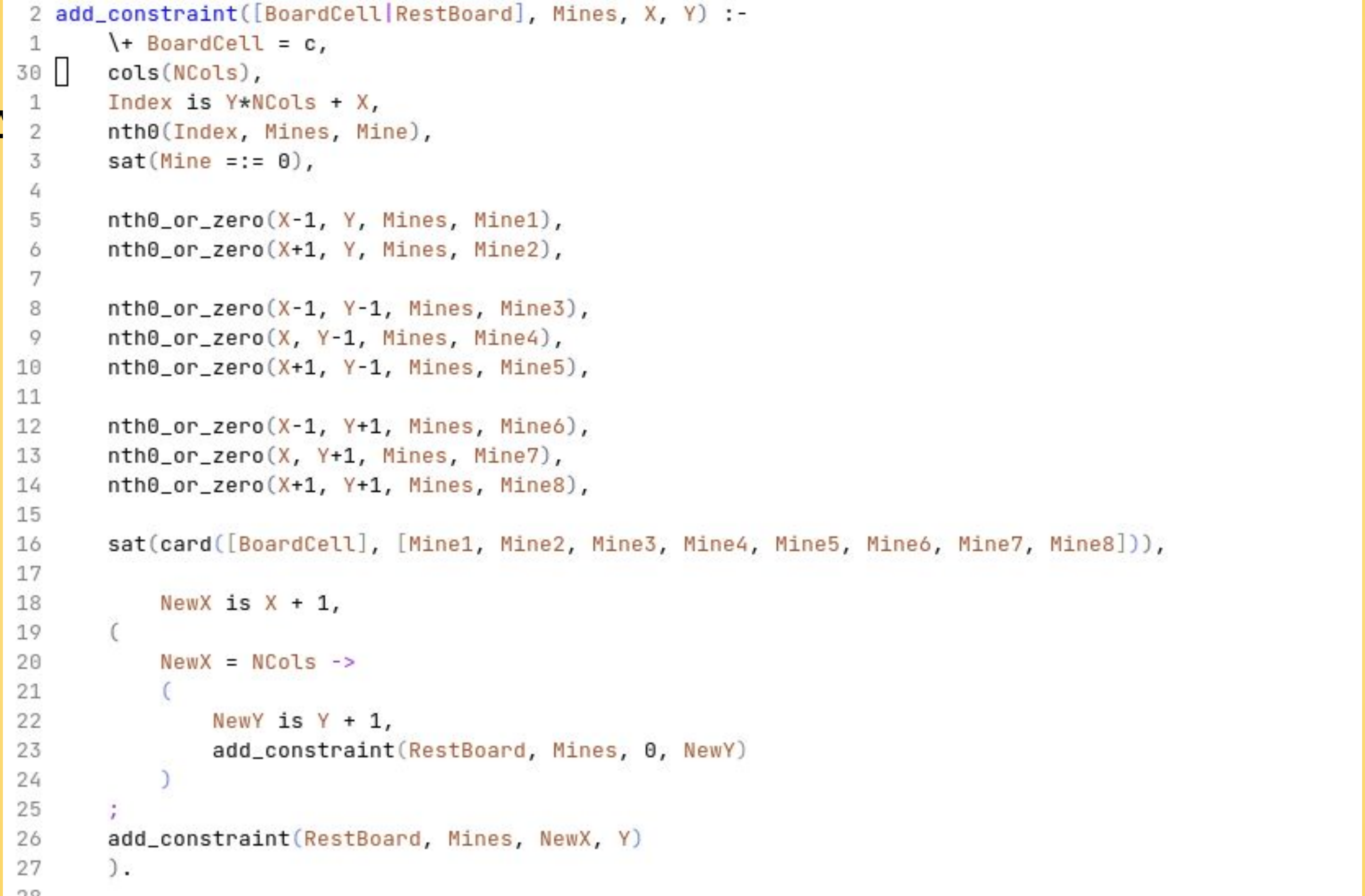
```prolog
nth0_or_zero(X, Y, List, Value) :-
    cols(NCols),
    rows(NRows),
    X >= 0,
    Y >= 0,
    X < NCols,
    Y < NRows,
    Index is Y*NCols + X,
    nth0(Index, List, Value),!.
nth0_or_zero(_X, _Y, _List, 0).
```

But the domain of the variables is always 0 or 1. That’s boolean!

We can model them using specific SAT techniques

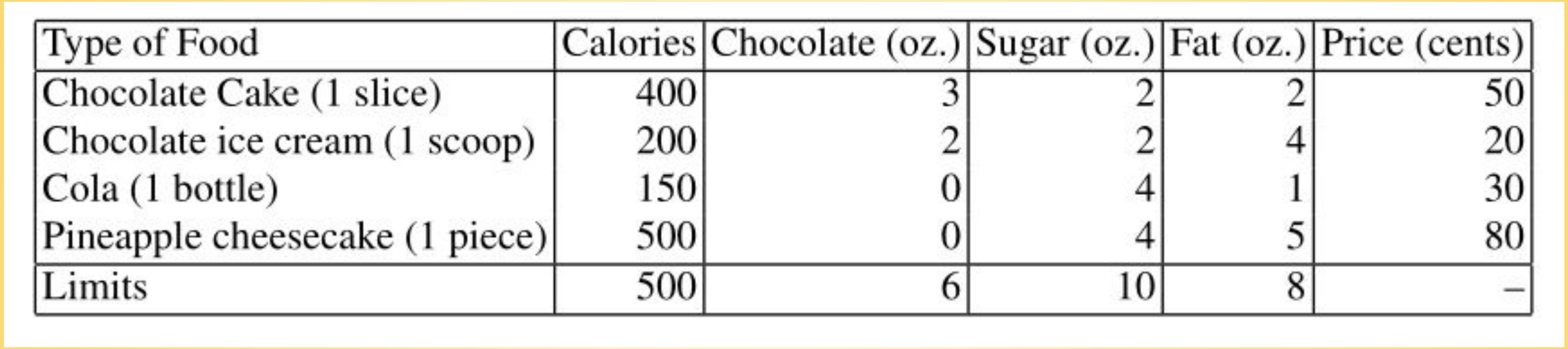
- Some systems do automatic translation from CP to SAT if requested (Picat, B-Prolog,...)
- Lots of very good SAT solvers: Google OR-Tools, Microsoft Z3, PicatSAT...
- Scryer Prolog includes clp(B)
- SAT or CP is better depending on the specific problem. Do benchmarks!
- In this case, smaller problem instances are faster with CP, larger instances perform better with SAT

SA



```
add_constraint([BoardCell|RestBoard], Mines, X, Y) :-
    \+ BoardCell = c,
    cols(NCols),
    Index is Y*NCols + X,
    nth0(Index, Mines, Mine),
    sat(Mine =:= 0),

    nth0_or_zero(X-1, Y, Mines, Mine1),
    nth0_or_zero(X+1, Y, Mines, Mine2),

    nth0_or_zero(X-1, Y-1, Mines, Mine3),
    nth0_or_zero(X, Y-1, Mines, Mine4),
    nth0_or_zero(X+1, Y-1, Mines, Mine5),

    nth0_or_zero(X-1, Y+1, Mines, Mine6),
    nth0_or_zero(X, Y+1, Mines, Mine7),
    nth0_or_zero(X+1, Y+1, Mines, Mine8),

    sat(card([BoardCell], [Mine1, Mine2, Mine3, Mine4, Mine5, Mine6, Mine7, Mine8])),

        NewX is X + 1,
    (
        NewX = NCols ->
        (
            NewY is Y + 1,
            add_constraint(RestBoard, Mines, 0, NewY)
        )
    ;
    add_constraint(RestBoard, Mines, NewX, Y)
    ).
```

Diet

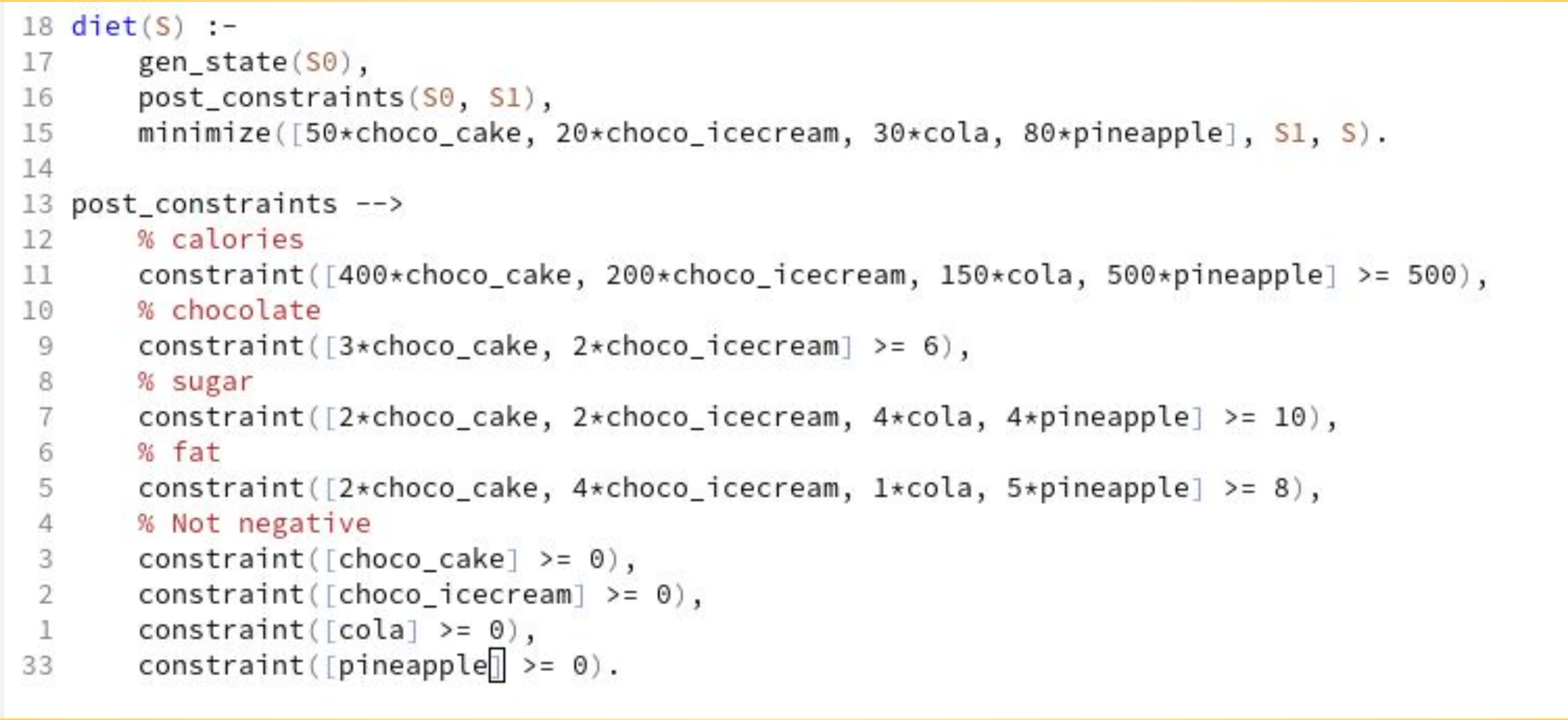


Type of Food	Calories	Chocolate (oz.)	Sugar (oz.)	Fat (oz.)	Price (cents)
Chocolate Cake (1 slice)	400	3	2	2	50
Chocolate ice cream (1 scoop)	200	2	2	4	20
Cola (1 bottle)	150	0	4	1	30
Pineapple cheesecake (1 piece)	500	0	4	5	80
Limits	500	6	10	8	–

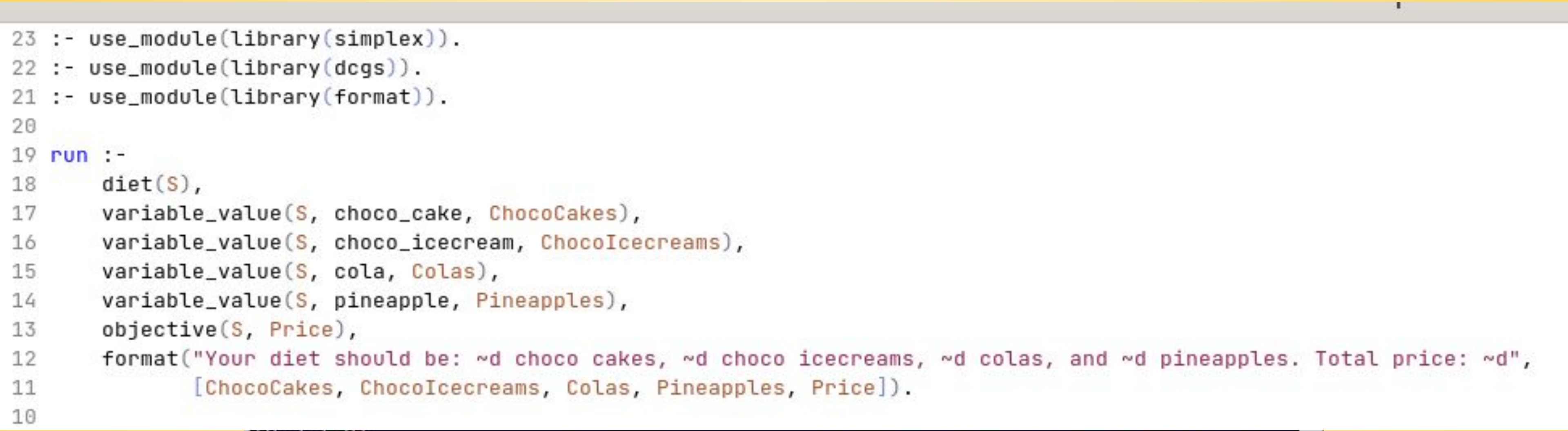
Meet the nutritional requirements while optimizing the cost of the food

When we have an optimization problem like this, it’s a good idea to use MIP to solve the puzzle.

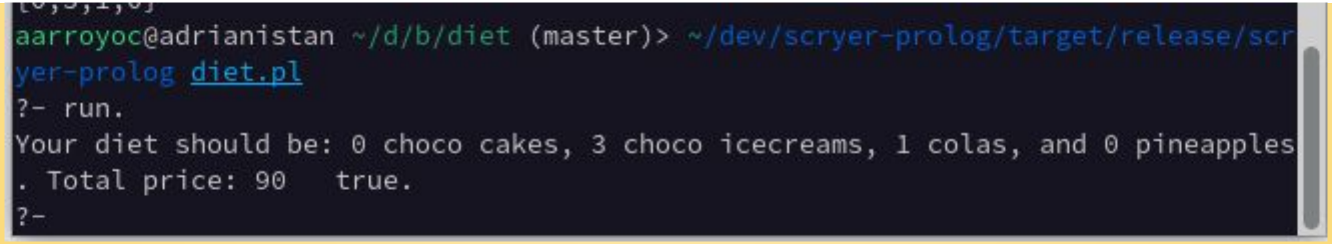
In this case, the problem is just Linear Programming



```
diet(S) :-
    gen_state(S0),
    post_constraints(S0, S1),
    minimize([50*choco_cake, 20*choco_icecream, 30*cola, 80*pineapple], S1, S).

post_constraints -->
    % calories
    constraint([400*choco_cake, 200*choco_icecream, 150*cola, 500*pineapple] >= 500),
    % chocolate
    constraint([3*choco_cake, 2*choco_icecream] >= 6),
    % sugar
    constraint([2*choco_cake, 2*choco_icecream, 4*cola, 4*pineapple] >= 10),
    % fat
    constraint([2*choco_cake, 4*choco_icecream, 1*cola, 5*pineapple] >= 8),
    % Not negative
    constraint([choco_cake] >= 0),
    constraint([choco_icecream] >= 0),
    constraint([cola] >= 0),
    constraint([pineapple] >= 0).
```



```prolog
:- use_module(library(simplex)).
:- use_module(library(dcgs)).
:- use_module(library(format)).

run :-
    diet(S),
    variable_value(S, choco_cake, ChocoCakes),
    variable_value(S, choco_icecream, ChocoIcecreams),
    variable_value(S, cola, Colas),
    variable_value(S, pineapple, Pineapples),
    objective(S, Price),
    format("Your diet should be: ~d choco cakes, ~d choco icecreams, ~d colas, and ~d pineapples. Total price: ~d",
           [ChocoCakes, ChocoIcecreams, Colas, Pineapples, Price]).

```



```
[0,3,1,0]
aarroyoc@adrianistan ~/d/b/diet (master)> ~/dev/scryer-prolog/target/release/scr
yer-prolog diet.pl
?- run.
Your diet should be: 0 choco cakes, 3 choco icecreams, 1 colas, and 0 pineapples
. Total price: 90   true.
?-
```

MIP overview

- MIP problems can be solved very efficiently
- If it can be solved with the Linear Programming subset, even more (integers are problematic)
- Lots of high quality solvers: IBM ILOG CPLEX, FICO Xpress, Gurobi, GLPK, JuMP,...
- Here we’re using Triska’s simplex library, included in Scryer Prolog
- Usually the best option if we need to do an optimization to get to a solution and feasible solutions are easy to find.

The Zebra Puzzle

Five men with different nationalities live in the first five houses of a street. They practice

five distinct professions, and each of them has a favorite animal and a favorite drink, all

of them different. The five houses are painted in different colors.

The Englishman lives in a red house.

The Spaniard owns a dog.

The Japanese is a painter.

The Italian drinks tea.

The Norwegian lives in the first house on the left.

The owner of the green house drinks coffee.

The Zebra Puzzle II

The green house is on the right of the white one.

The sculptor breeds snails.

The diplomat lives in the yellow house.

Milk is drunk in the middle house.

The Norwegian’s house is next to the blue one.

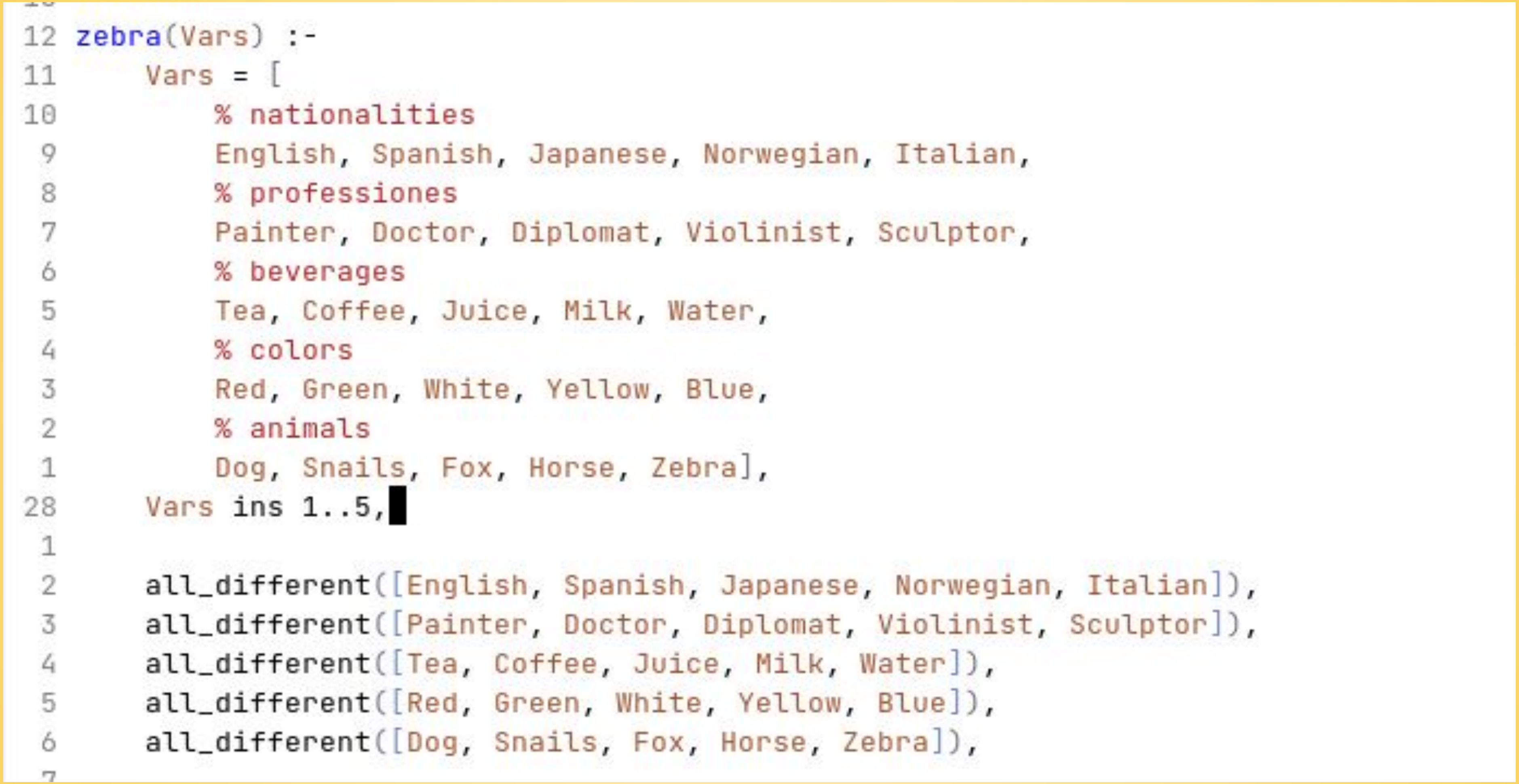
The violinist drinks fruit juice.

The fox is in a house next to that of the doctor.

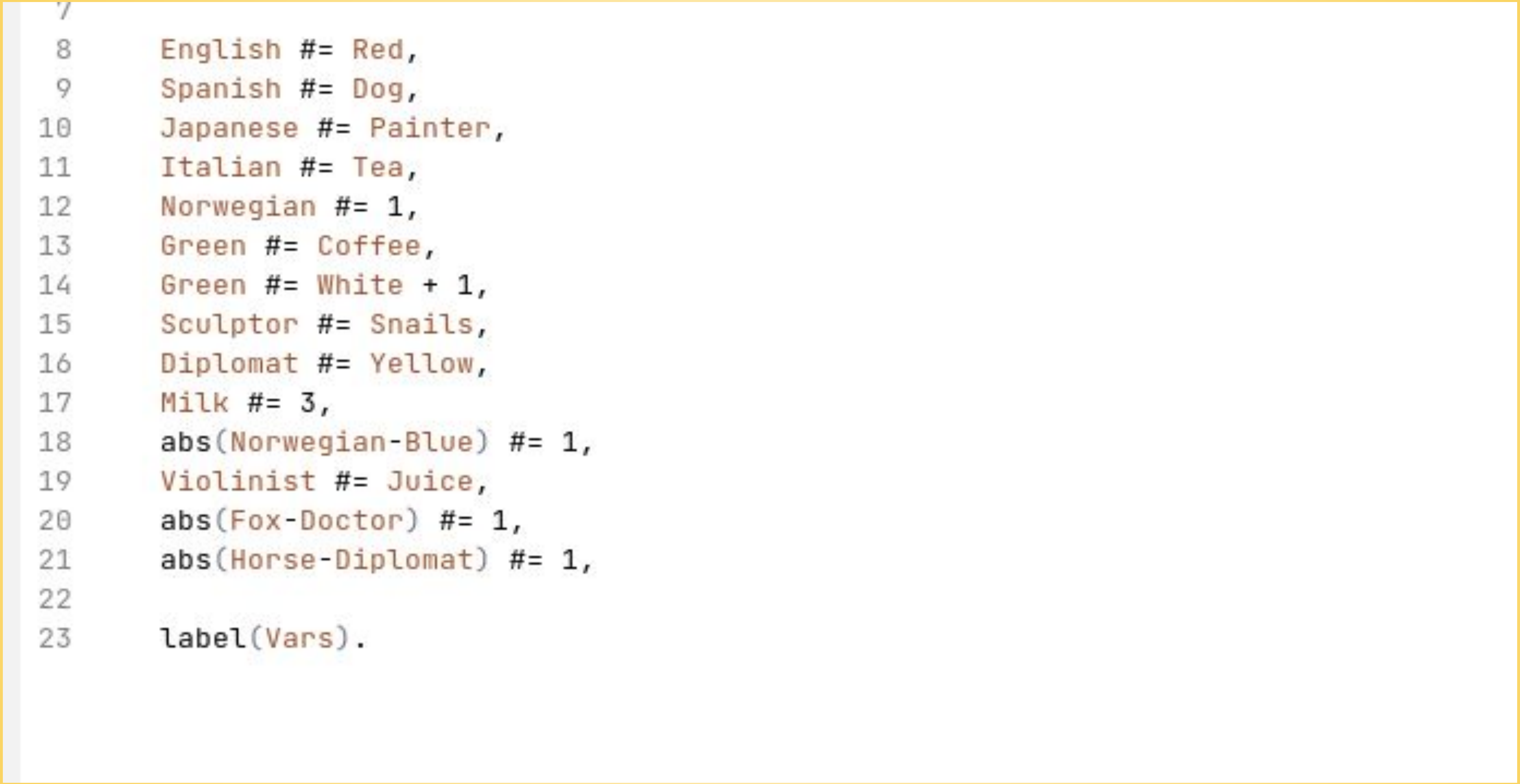
The horse is in a house next to that of the diplomat.

Who owns a **zebra**, and who drinks **water**?

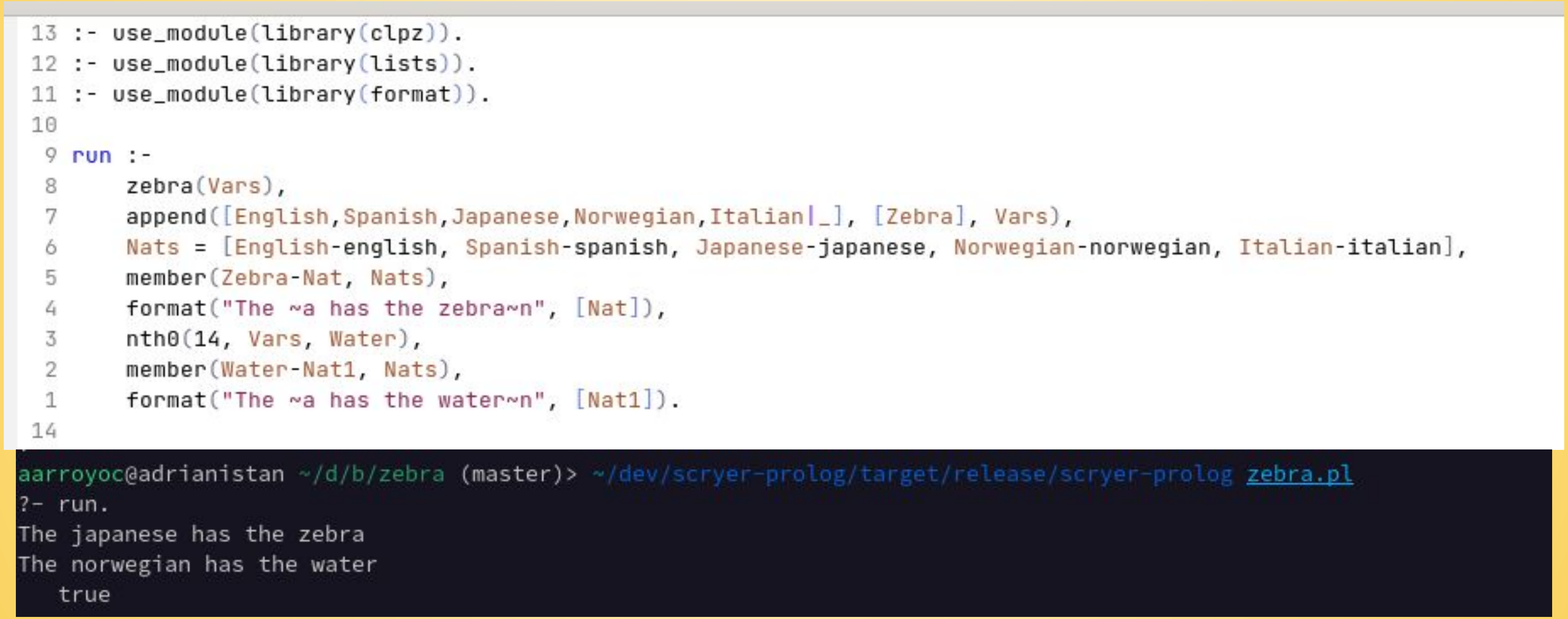
Hint: assign a number to each house



```
zebra(Vars) :-
    Vars = [
        % nationalities
        English, Spanish, Japanese, Norwegian, Italian,
        % professiones
        Painter, Doctor, Diplomat, Violinist, Sculptor,
        % beverages
        Tea, Coffee, Juice, Milk, Water,
        % colors
        Red, Green, White, Yellow, Blue,
        % animals
        Dog, Snails, Fox, Horse, Zebra],
    Vars ins 1..5,

    all_different([English, Spanish, Japanese, Norwegian, Italian]),
    all_different([Painter, Doctor, Diplomat, Violinist, Sculptor]),
    all_different([Tea, Coffee, Juice, Milk, Water]),
    all_different([Red, Green, White, Yellow, Blue]),
    all_different([Dog, Snails, Fox, Horse, Zebra]),
```



```
    English #= Red,
    Spanish #= Dog,
    Japanese #= Painter,
    Italian #= Tea,
    Norwegian #= 1,
    Green #= Coffee,
    Green #= White + 1,
    Sculptor #= Snails,
    Diplomat #= Yellow,
    Milk #= 3,
    abs(Norwegian-Blue) #= 1,
    Violinist #= Juice,
    abs(Fox-Doctor) #= 1,
    abs(Horse-Diplomat) #= 1,

    label(Vars).
```



```
:- use_module(library(clpz)).
:- use_module(library(lists)).
:- use_module(library(format)).

run :-
    zebra(Vars),
    append([English,Spanish,Japanese,Norwegian,Italian|_], [Zebra], Vars),
    Nats = [English-english, Spanish-spanish, Japanese-japanese, Norwegian-norwegian, Italian-italian],
    member(Zebra-Nat, Nats),
    format("The ~a has the zebra~n", [Nat]),
    nth0(14, Vars, Water),
    member(Water-Nat1, Nats),
    format("The ~a has the water~n", [Nat1]).

```

```
aarroyoc@adrianistan ~/d/b/zebra (master)> ~/dev/scryer-prolog/target/release/scryer-prolog zebra.pl
?- run.
The japanese has the zebra
The norwegian has the water
   true
```

Moving furniture with 4 friends

Minimize the time needed to clean a room full of furniture. Some furniture needs more than 1 person working on it.

Moving furniture with 4 friends

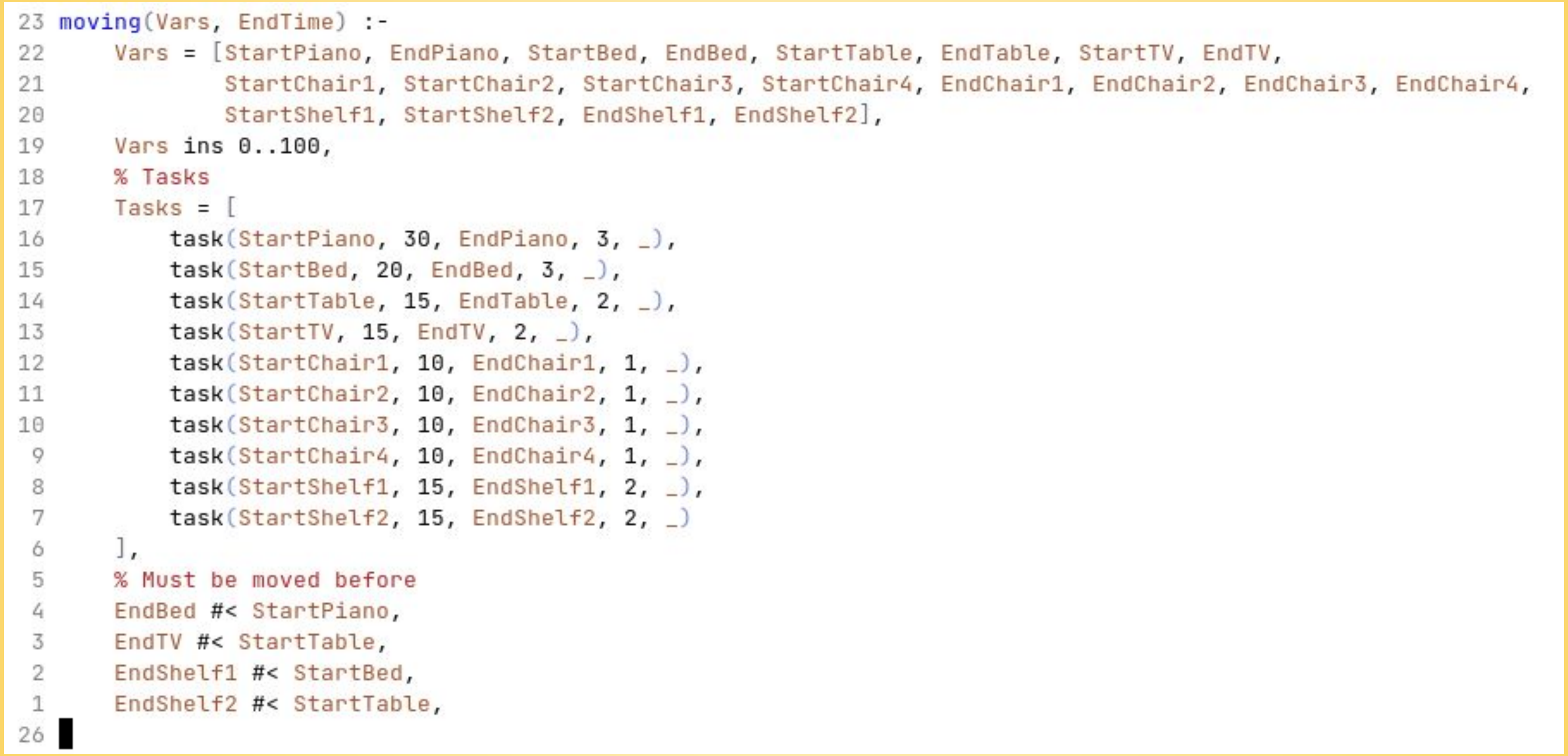
Item	People needed	Time needed	Notes
Piano	3	30	
Bed	3	20	Must be moved before piano
Table	2	15	
TV	2	15	Must be moved before the table
Chairs (from 1 to 4)	1	10	
Shelf 1	2	15	Must be moved before bed
Shelf 2	2	15	Must be moved before table

cumulative constraint to the rescue!

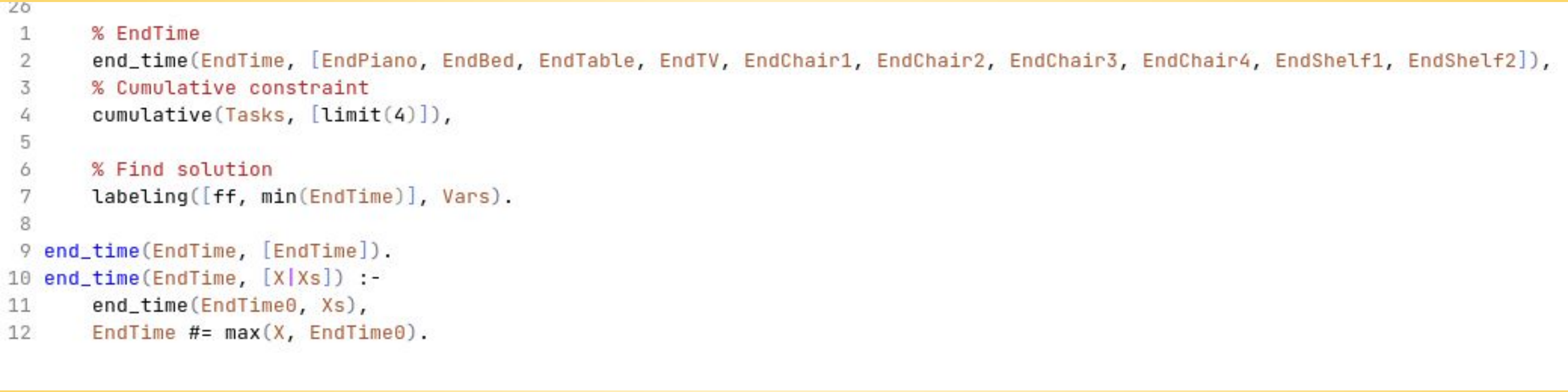
cumulative constraint allows us to pass a set of tasks that use a limited resource, with an Start Time, a Duration, an Ending Time and a Resource Usage

We can apply moving restrictions via traditional constraints over Start/End times.

We can find a solution that is optimal, by minimizing the max End time!



```
moving(Vars, EndTime) :-
    Vars = [StartPiano, EndPiano, StartBed, EndBed, StartTable, EndTable, StartTV, EndTV,
            StartChair1, StartChair2, StartChair3, StartChair4, EndChair1, EndChair2, EndChair3, EndChair4,
            StartShelf1, StartShelf2, EndShelf1, EndShelf2],
    Vars ins 0..100,
    % Tasks
    Tasks = [
        task(StartPiano, 30, EndPiano, 3, _),
        task(StartBed, 20, EndBed, 3, _),
        task(StartTable, 15, EndTable, 2, _),
        task(StartTV, 15, EndTV, 2, _),
        task(StartChair1, 10, EndChair1, 1, _),
        task(StartChair2, 10, EndChair2, 1, _),
        task(StartChair3, 10, EndChair3, 1, _),
        task(StartChair4, 10, EndChair4, 1, _),
        task(StartShelf1, 15, EndShelf1, 2, _),
        task(StartShelf2, 15, EndShelf2, 2, _)
    ],
    % Must be moved before
    EndBed #< StartPiano,
    EndTV #< StartTable,
    EndShelf1 #< StartBed,
    EndShelf2 #< StartTable,

```



```
    % EndTime
    end_time(EndTime, [EndPiano, EndBed, EndTable, EndTV, EndChair1, EndChair2, EndChair3, EndChair4, EndShelf1, EndShelf2]),
    % Cumulative constraint
    cumulative(Tasks, [limit(4)]),

    % Find solution
    labeling([ff, min(EndTime)], Vars).

end_time(EndTime, [EndTime]).
end_time(EndTime, [X|Xs]) :-
    end_time(EndTime0, Xs),
    EndTime #= max(X, EndTime0).
```

Optimal time: 82 “slots”

Printing a nice result left as a task for the viewer :)

The three jugs problem

We have three jugs, of 3L, 5L and 8L. The one of 8L is full, the rest is empty. Pouring water from one to another, get a jug with exactly 4L.

Multiple ways to solve this

However we’re in a CP talk

Some problems can be encoded as an Automaton

Then solved with CP

Automaton?

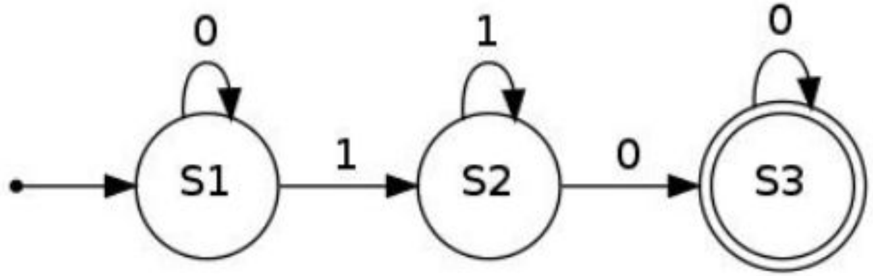


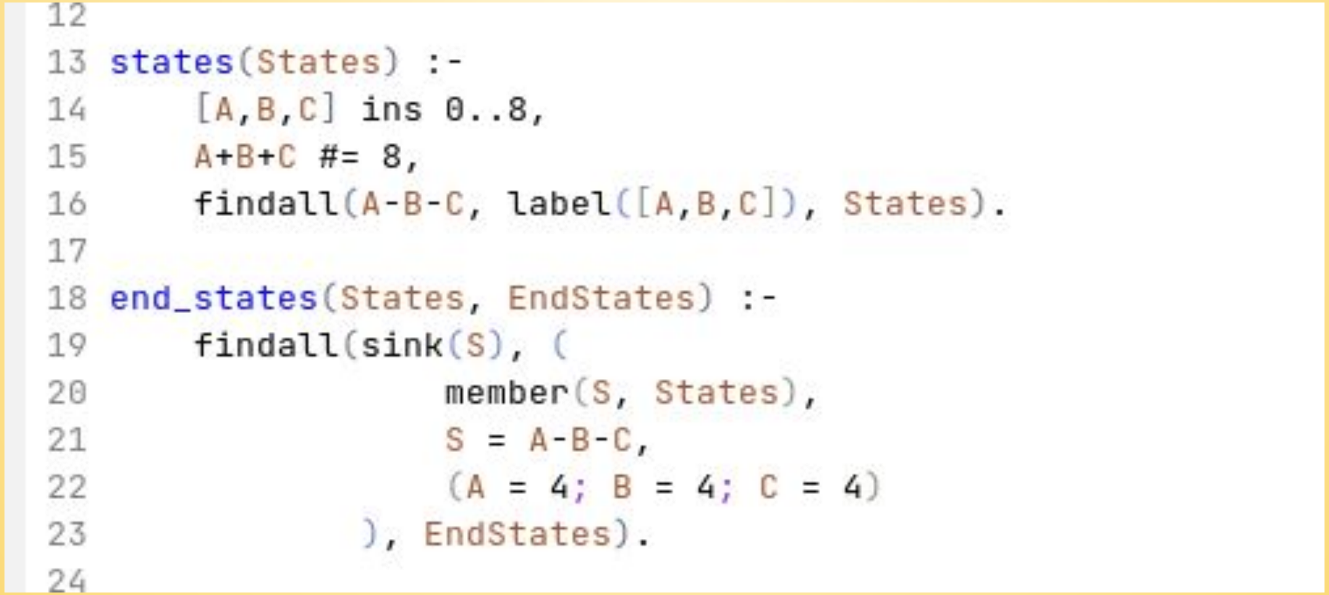
Fig. 3.10 DFA for "0*1*0*"

DFA: Deterministic Finite Automaton

Automaton!

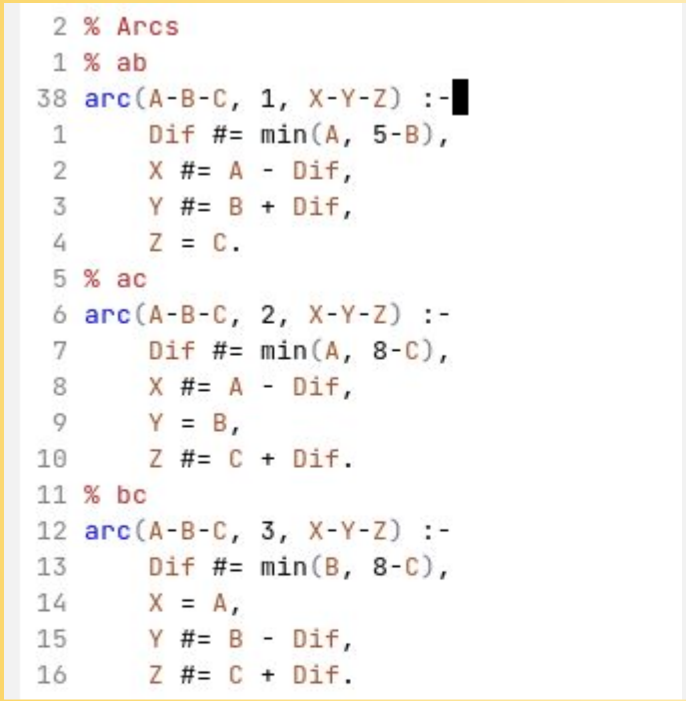
Source states: just one, 0-0-8.

Sink states: enumerate all states and filter those who have 4L on a single jug.

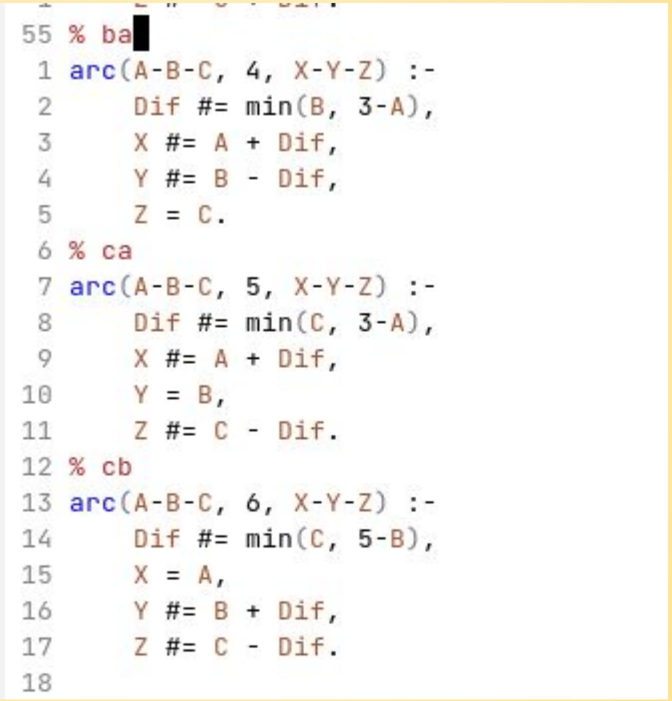


```prolog

states(States) :-
    [A,B,C] ins 0..8,
    A+B+C #= 8,
    findall(A-B-C, label([A,B,C]), States).

end_states(States, EndStates) :-
    findall(sink(S), (
                  member(S, States),
                  S = A-B-C,
                  (A = 4; B = 4; C = 4)
              ), EndStates).

```

Automaton! II

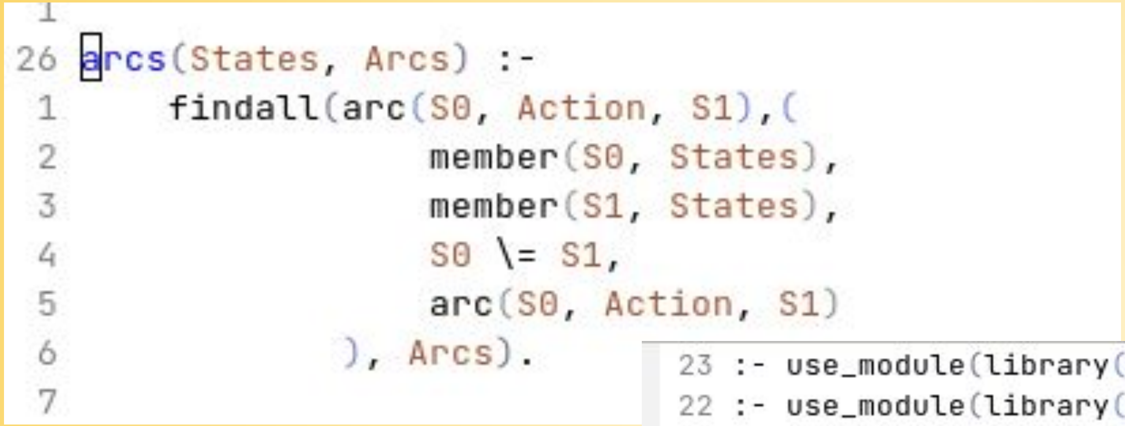


```prolog
% Arcs
% ab
arc(A-B-C, 1, X-Y-Z) :-
    Dif #= min(A, 5-B),
    X #= A - Dif,
    Y #= B + Dif,
    Z = C.
% ac
arc(A-B-C, 2, X-Y-Z) :-
    Dif #= min(A, 8-C),
    X #= A - Dif,
    Y = B,
    Z #= C + Dif.
% bc
arc(A-B-C, 3, X-Y-Z) :-
    Dif #= min(B, 8-C),
    X = A,
    Y #= B - Dif,
    Z #= C + Dif.
```

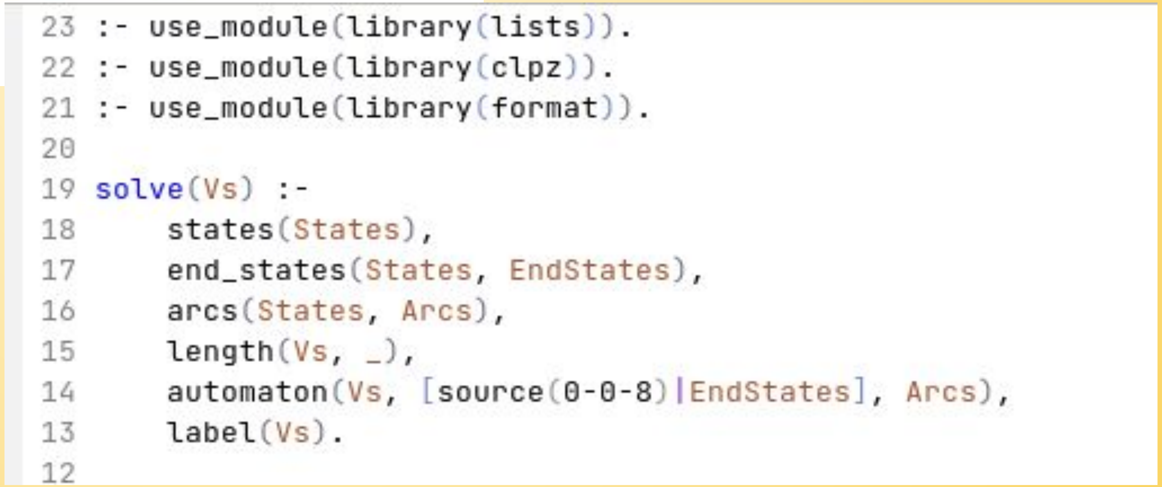


```prolog
% ba
arc(A-B-C, 4, X-Y-Z) :-
    Dif #= min(B, 3-A),
    X #= A + Dif,
    Y #= B - Dif,
    Z = C.
% ca
arc(A-B-C, 5, X-Y-Z) :-
    Dif #= min(C, 3-A),
    X #= A + Dif,
    Y = B,
    Z #= C - Dif.
% cb
arc(A-B-C, 6, X-Y-Z) :-
    Dif #= min(C, 5-B),
    X = A,
    Y #= B + Dif,
    Z #= C - Dif.
```

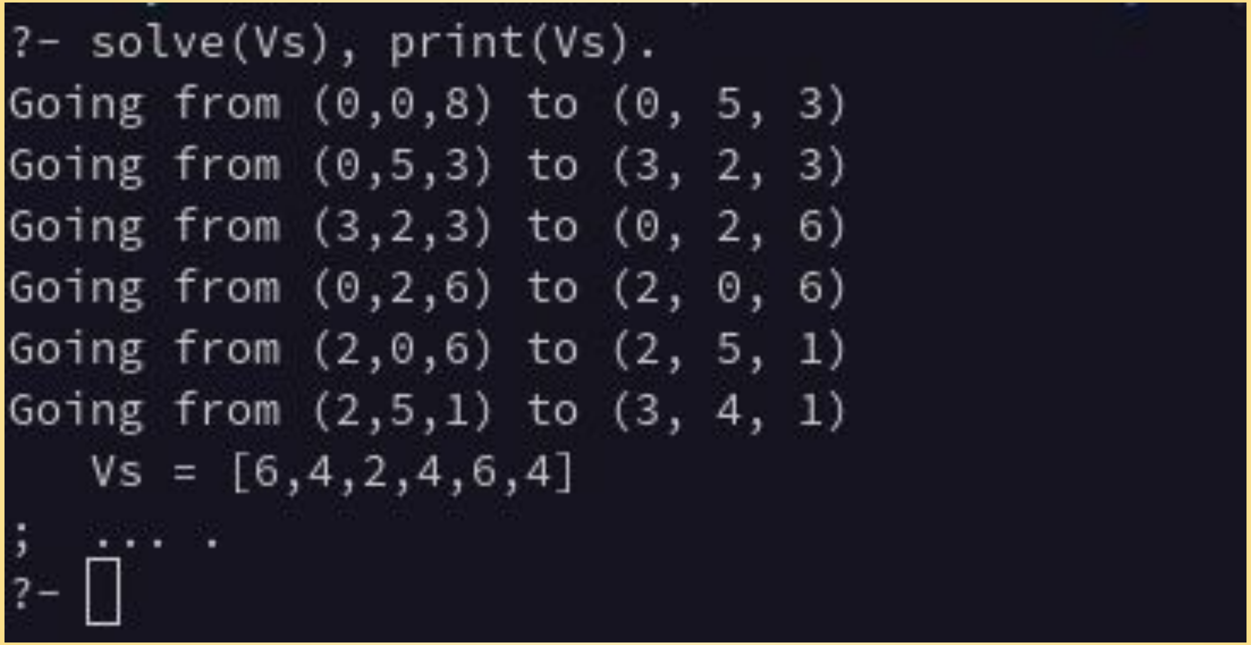
Automaton! III



```
arcs(States, Arcs) :-
    findall(arc(S0, Action, S1),(
                    member(S0, States),
                    member(S1, States),
                    S0 \= S1,
                    arc(S0, Action, S1)
                ), Arcs).
```

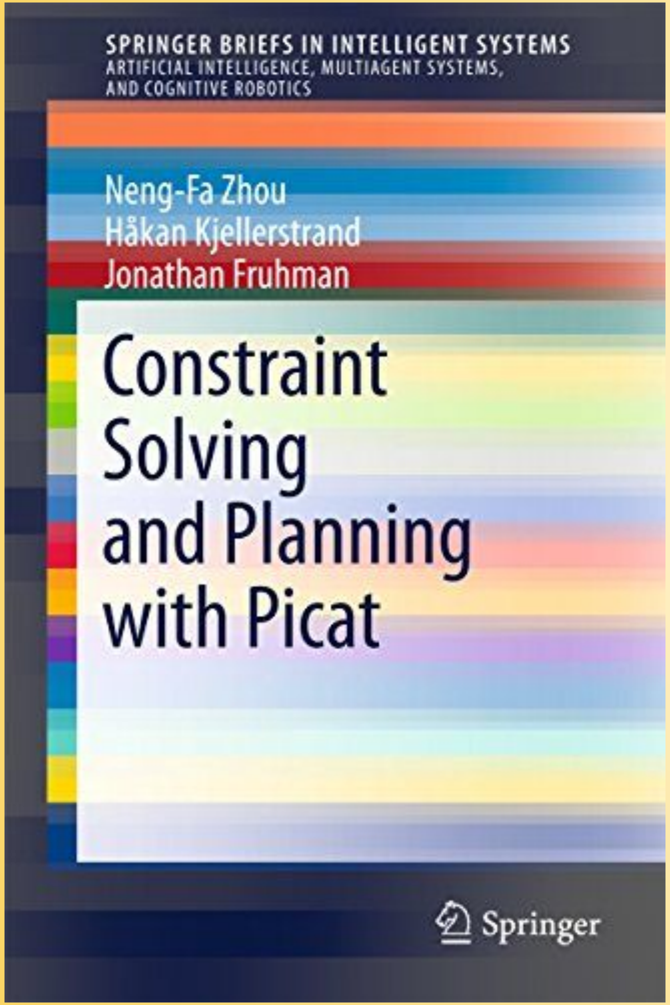


```
:- use_module(library(lists)).
:- use_module(library(clpz)).
:- use_module(library(format)).

solve(Vs) :-
    states(States),
    end_states(States, EndStates),
    arcs(States, Arcs),
    length(Vs, _),
    automaton(Vs, [source(0-0-8)|EndStates], Arcs),
    label(Vs).

```



```
?- solve(Vs), print(Vs).
Going from (0,0,8) to (0, 5, 3)
Going from (0,5,3) to (3, 2, 3)
Going from (3,2,3) to (0, 2, 6)
Going from (0,2,6) to (2, 0, 6)
Going from (2,0,6) to (2, 5, 1)
Going from (2,5,1) to (3, 4, 1)
   Vs = [6,4,2,4,6,4]
;  ... .
?- 
```

Put a bit of CP in your life!


SPRINGER BRIEFS IN INTELLIGENT SYSTEMS
ARTIFICIAL INTELLIGENCE, MULTIAGENT SYSTEMS, AND COGNITIVE ROBOTICS

Neng-Fa Zhou
Håkan Kjellerstrand
Jonathan Fruhman

Constraint Solving and Planning with Picat

Springer

Credits

- Constraint Solving and Planning with Picat
- Markus Triska clp(Z), clp(B), simplex libraries
- Mark Thom and other developers of Scryer Prolog

UNIVERSITE DE PARIS
FACULTE DE MEDECINE
PAR M VINCENT AURIOL
PRESIDENT DE LA REPUBLIQUE
M ANDRE MARIE
M JEAN SARRAILH
M LEON BINET

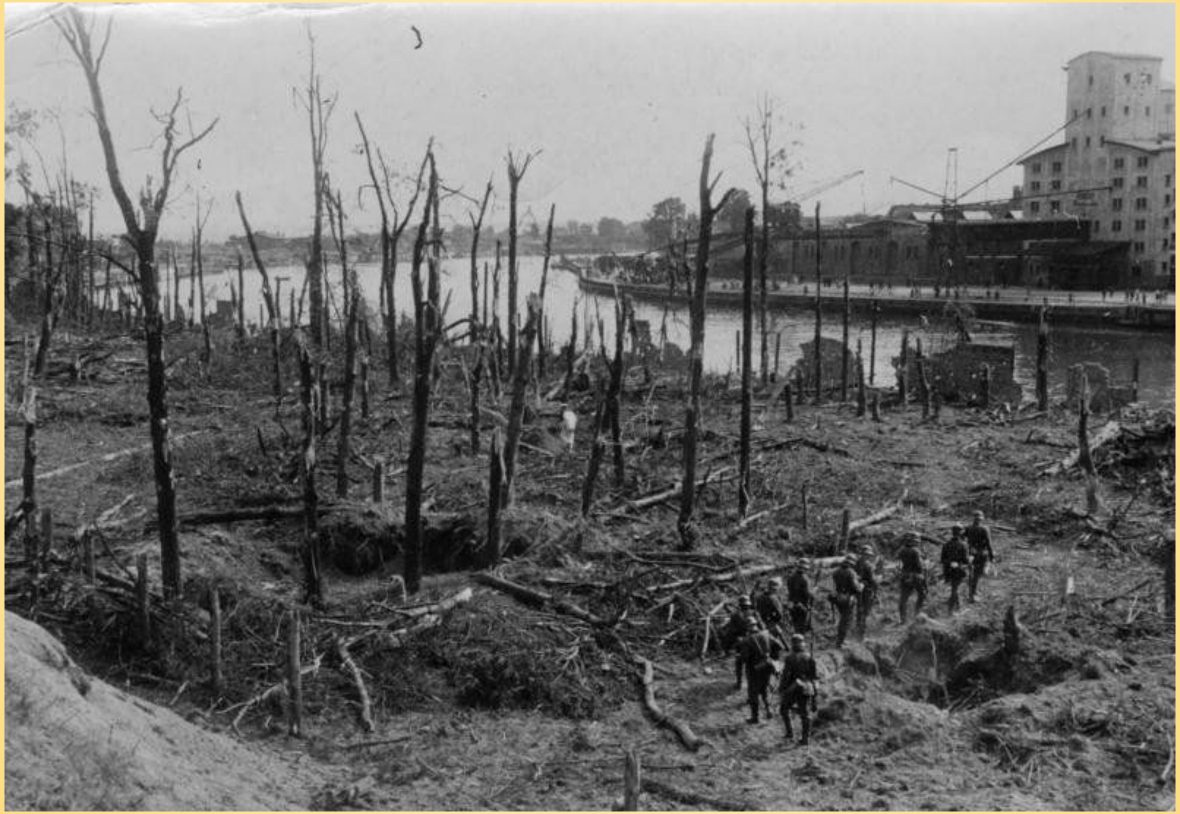
¿ ?

Questions, and, hopefully, answers

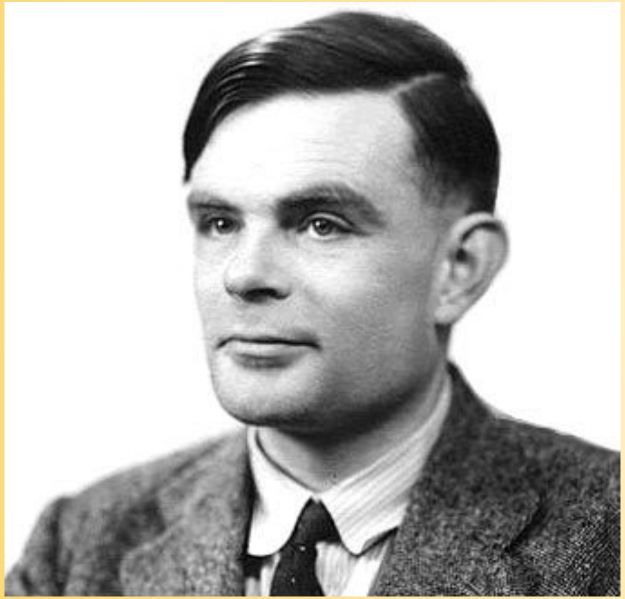
LE ROU
SLAVI
bien dormir, ouvrez les yeux
KIPLI

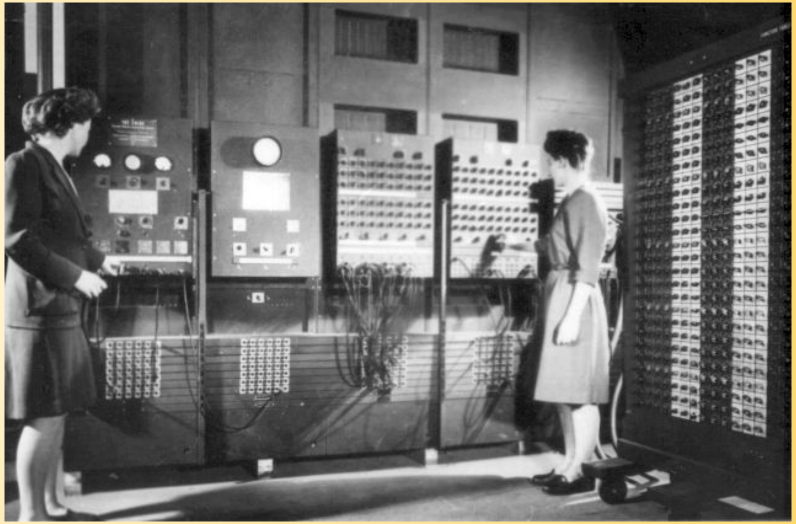
More? circuit, label strategies

1939 - Gdansk (Danzig)



First practical use of computers


Cryptography


Physical calculations

While important, cryptography and physics calcs are not the only thing you need to win a war!

Operations Research or:

How many supplies I need? What's the optimal way to deliver them?

George Dantzig