Conceptual Modeling with Alloy

THE SOCIETY FOR MODELING & SIMULATION INTERNATIONAL™

Dr. Ross Gore - Virginia Modeling, Analysis and Simulation Center (VMASC) - Suffolk, Virginia

Conceptual Modeling

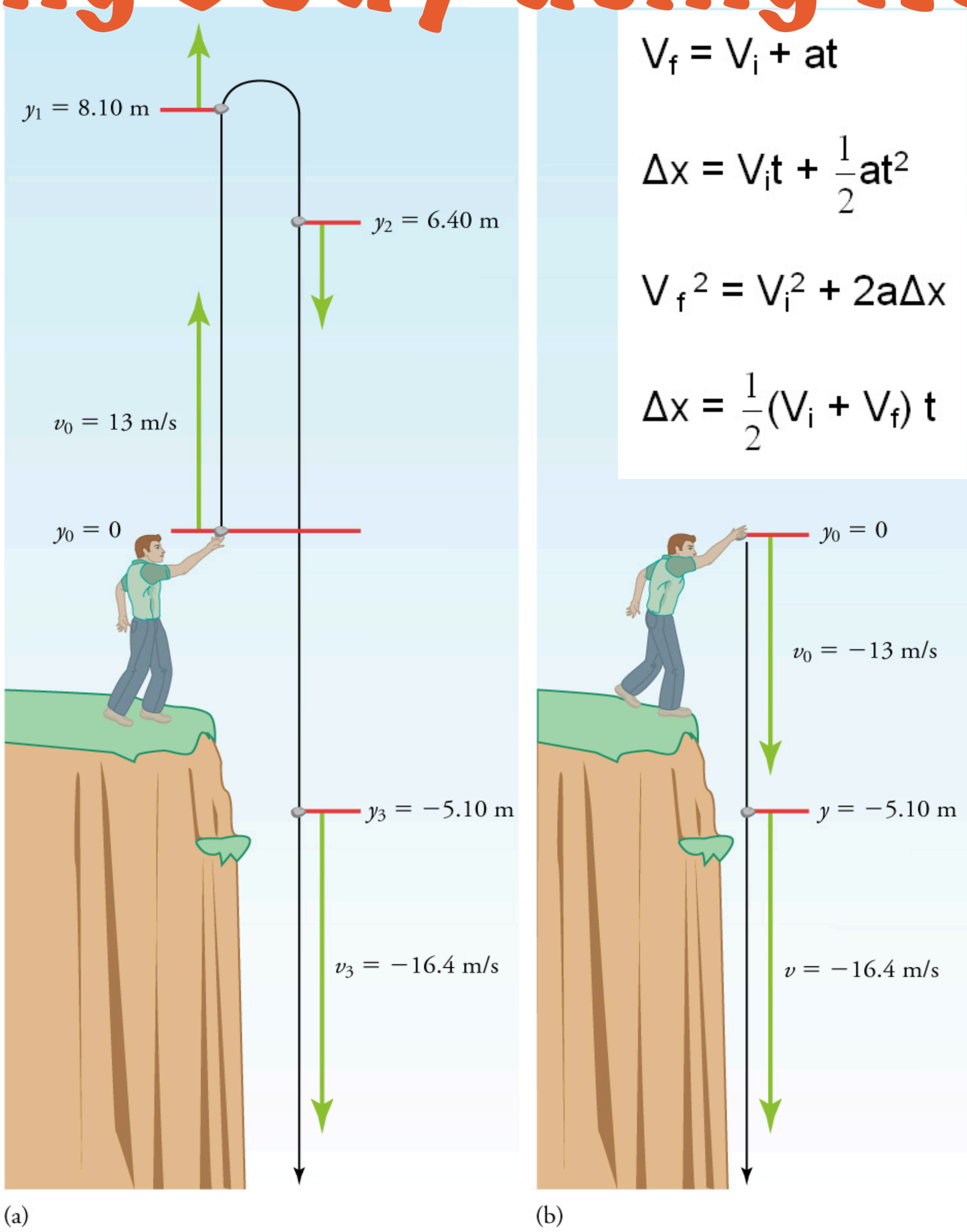
Falling Body using Newton

$$V_f = V_i + at$$

$$\Delta x = V_i t + \frac{1}{2}at^2$$

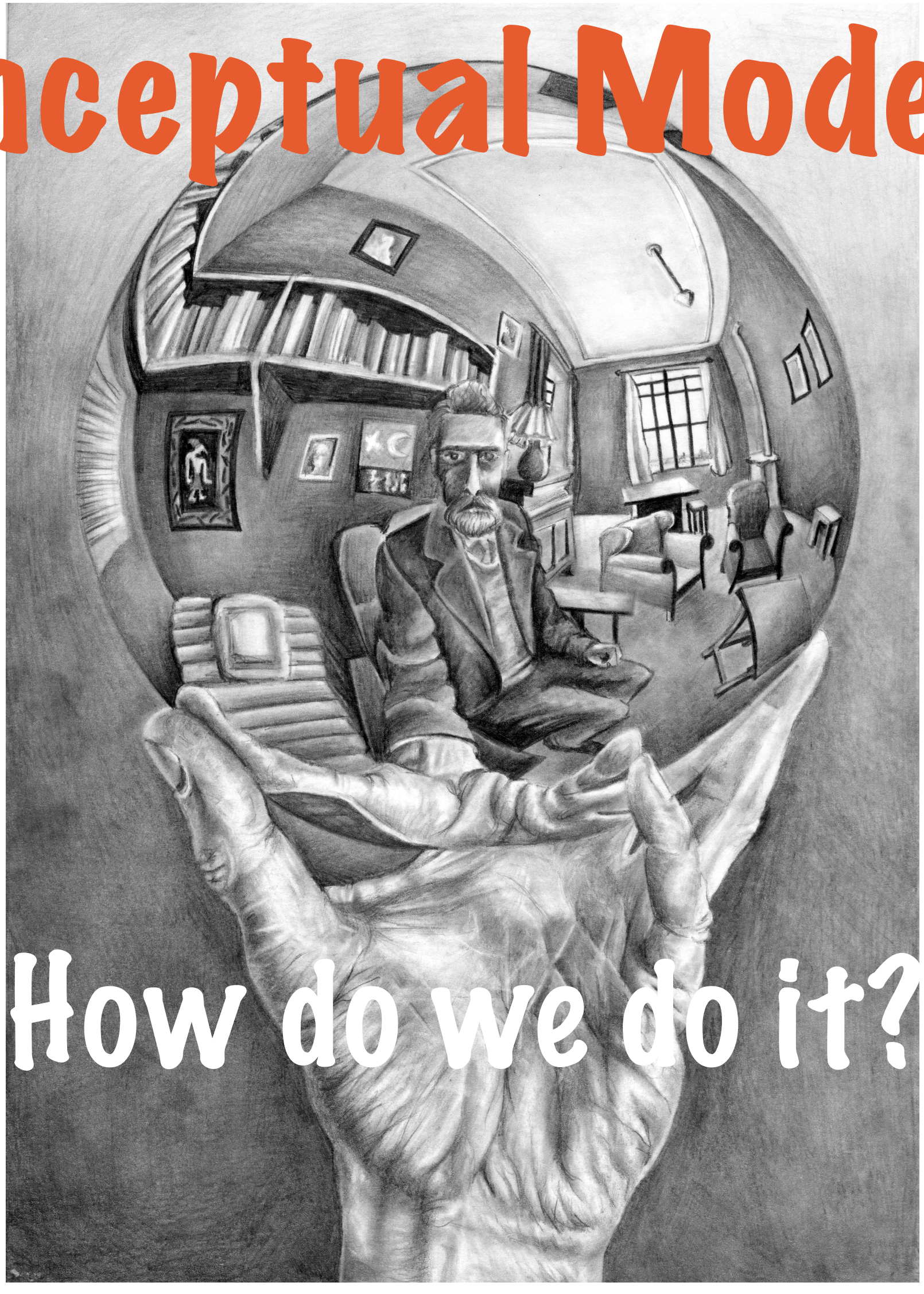
$$V_f^2 = V_i^2 + 2a\Delta x$$

$$\Delta x = \frac{1}{2}(V_i + V_f)\, t$$



Complex Physical Sims

Conceptual Modeling



How do we do it?

what we're already doing

thinking & sketching
simulating features

normal design practice
copying good ideas

evaluating products
user feedback

discarding failed designs
"refactoring"

To design something really well, you have to get it. You have to really grok what it's all about. It takes a passionate commitment to really thoroughly understand something, chew it up, not just quickly swallow it. Most people don't take the time to do that. --Steve Jobs

what we're not doing

being explicit

focusing: what are the concepts?
relating: how are they related to each other?
analyzing: what properties do they have?

hypothesis

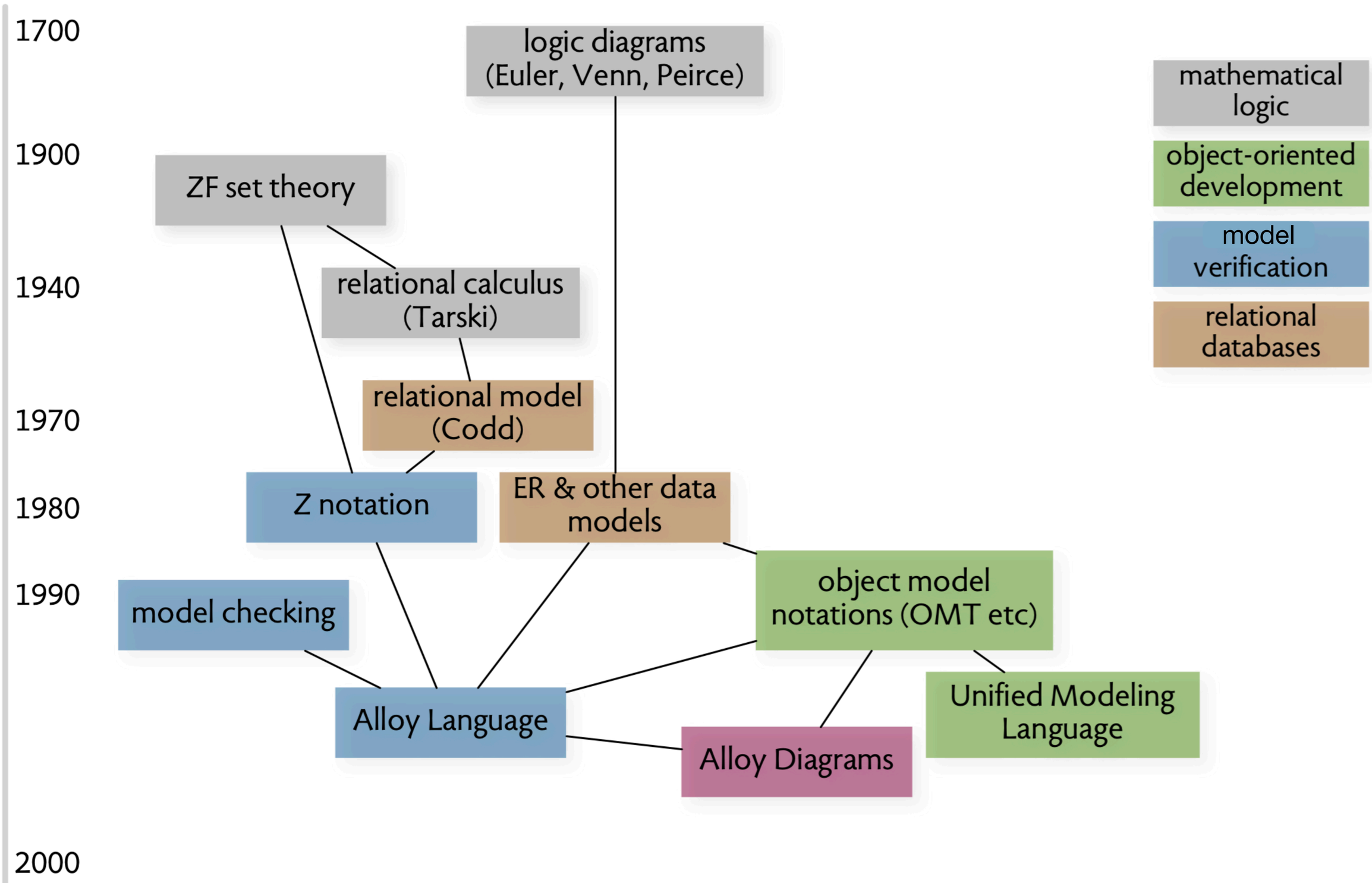
Incomplete Model	Complete Model
hard to use	easy to use
a mess to maintain	easy to maintain
unreliable & buggy	trusted & reliable

Today we'll:

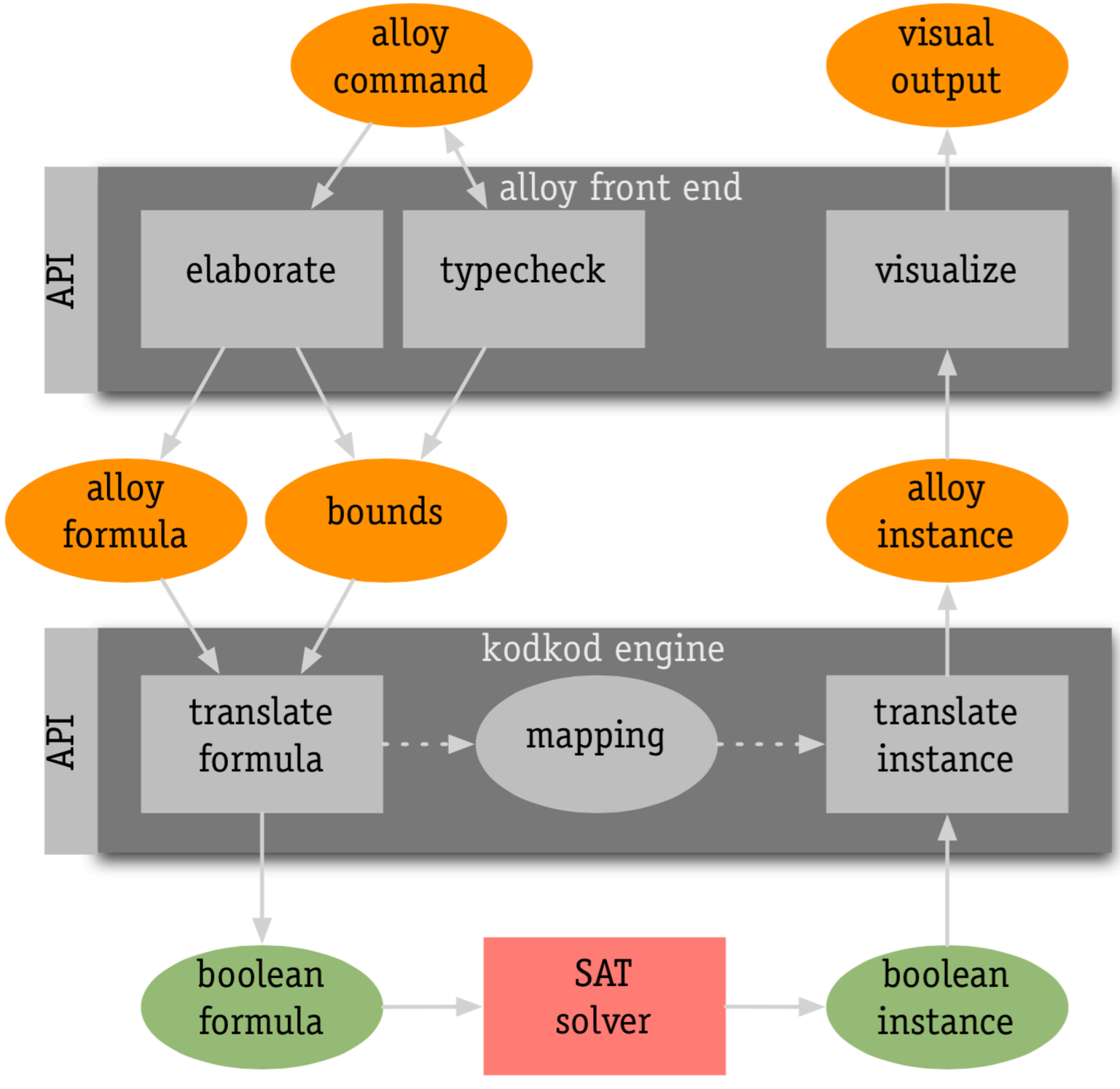
Build a conceptual model of Pacman

that specifies an explicit design

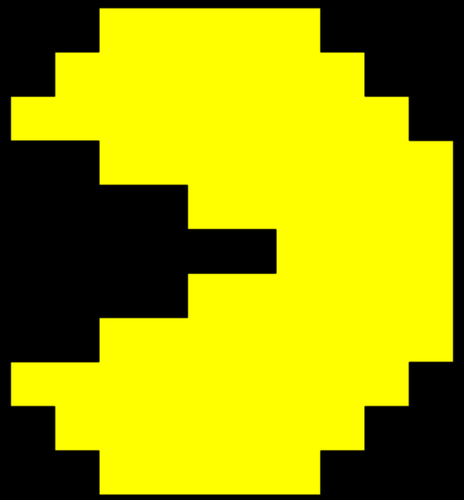
alloy: a notation



the alloy analyzer: a model finder

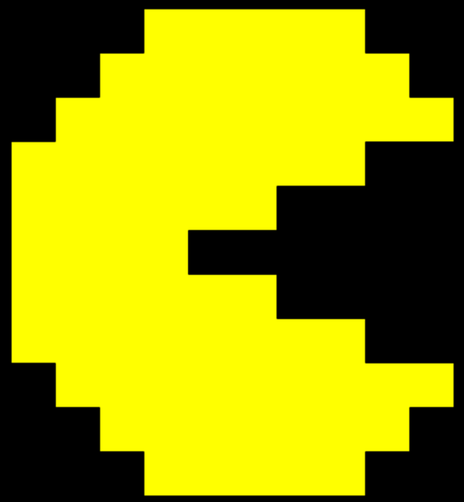


What's In PacMan?



What's In PacMan?

* Board
  - Position
  - Legal Moves
* PacMan
* Blinky(s)
* Walls
* Food



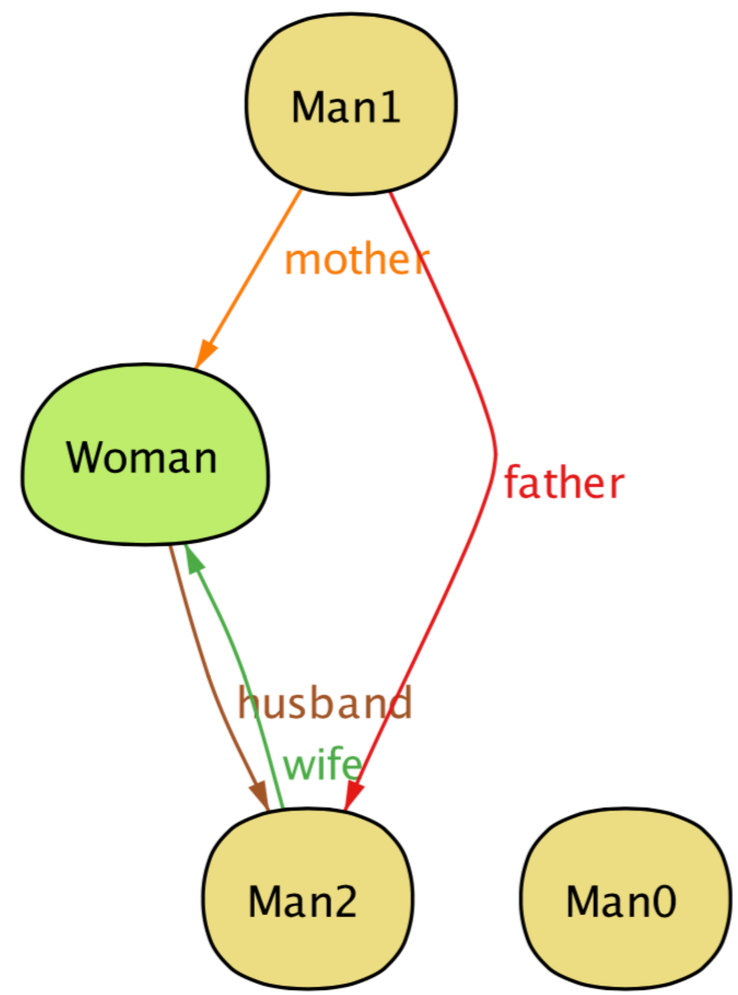
* Winning the Game
* Losing the Game

semantic concepts

atom
indivisible
immutable
uninterpreted

relation
collection of atom tuples

set
collection of atoms
(ie, a unary relation)



language: signatures

```
sig A {}
```
set of atoms A

```
sig A {}
sig B {}
```
disjoint sets A and B (no A & B)

```
sig Position {}

sig State {}
```

```
abstract sig A {}
sig B extends A {}
sig C extends A {}
```
A partitioned by disjoint subsets B and C

```
one sig A {}
```
A is a singleton set

```
abstract sig status {}
one sig Play extends status {}
one sig Win  extends status {}
one sig Loss extends status {}
```

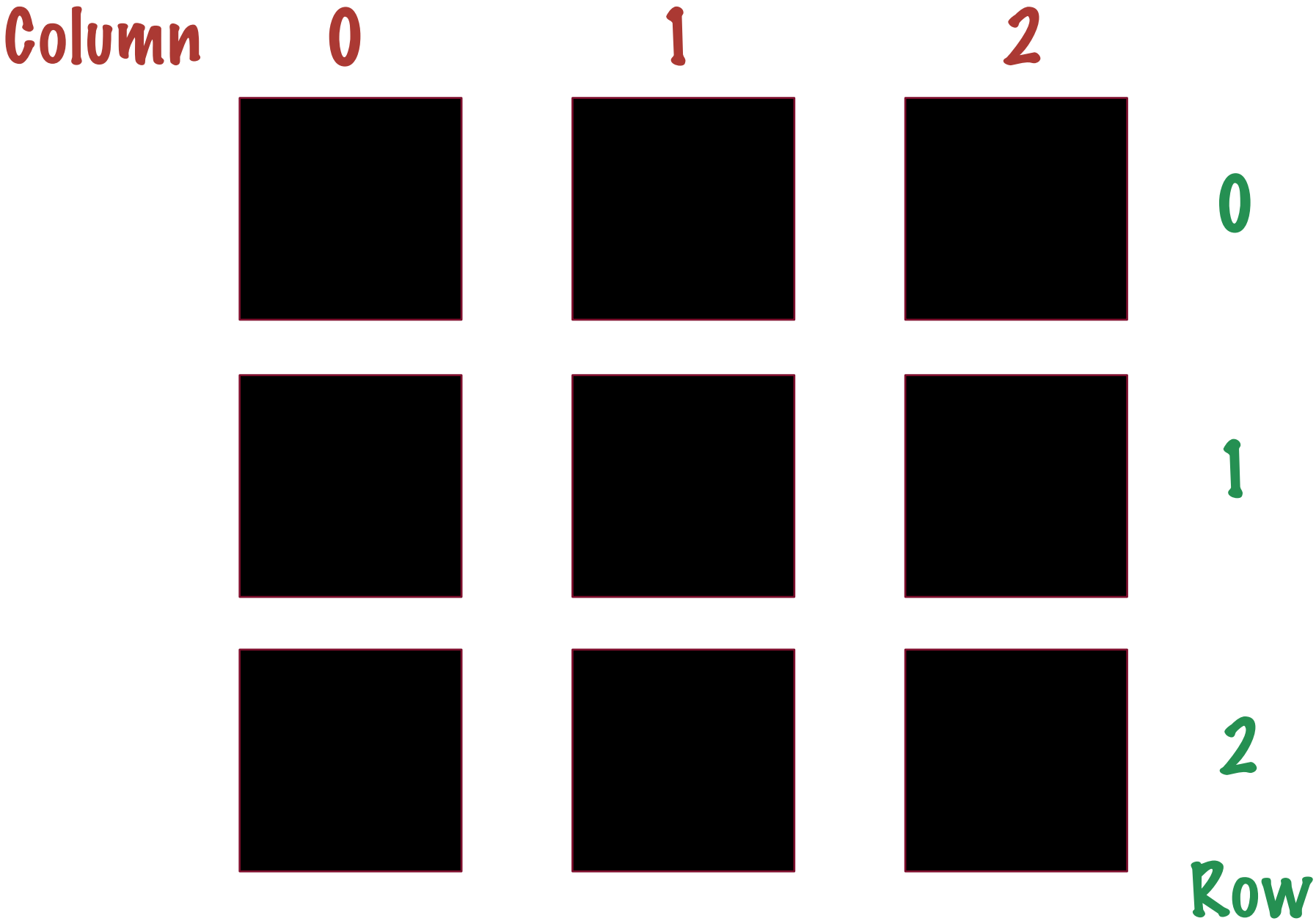
language: fields

```
sig A {f: e}
```

f is a binary relation with domain A and range given by expression e
f is constrained to be a function
(f: A -> one e) or (all a: A | a.f: e)

```
sig A {
  f1: one  e1,
  f2: lone e2,
  f3: some e3,
  f4: set  e4
}
```

```
sig State {
    walls  : set Position,
    food   : set Position,
    pacman : one Position,
    blinky : one Position,
    status : one Status
}
```


Column
0
1
2
0
1
2
Row

language: fields

```
sig A {f: e}
```

f is a binary relation with domain A and range given by expression e
f is constrained to be a function (f: A -> one e) or (all a: A | a.f: e)

```
sig A {
  f1: one  e1,
  f2: lone e2,
  f3: some e3,
  f4: set  e4
}
```

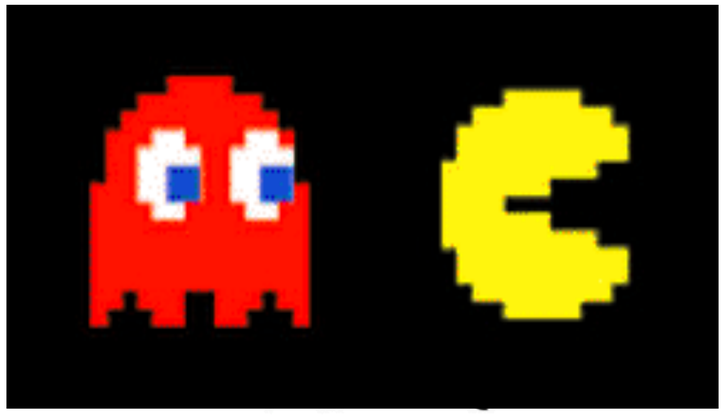
```
sig Position {
    adjacent : set Position,
    coordinates : Int->Int,
    row : Int,
    column : Int
}
// constraints on the fields
{
    int[column]>=0
    int[column]<int[Width.i]
    int[row]>=0
    int[row]<int[Height.i]
    coordinates = column->row
}
```

language: run command

```
pred p[x: X, y: Y, ...] { F }
run p scope
```

instructs analyzer to search for instance of predicate within scope

if model has facts M, finds solution to M && (some x: X, y: Y, ... | F)



```
pred GamePlay() {
   last.status = Play
}

run GamePlay for 1 State, exactly 9 Position
```

Model Requirement

Duplicate Positions
Are Not Cool!!!!

Each Position Must
Be You-nique

language: assertions & check command

instructs analyzer to search for counterexample to assertion within scope

```
assert a { F }
check a scope
```

if model has facts M finds solution to M && !F

```
assert NoDuplicatePositions{
    all p1, p2: Position |
    (p1 != p2) => (p1.column!=p2.column) ||
    (p1.row!=p2.row)
}

check NoDuplicatePositions for 9
```

Show any row or column equal to 0

Position2 ($NoDuplicatePositions_p1, row)	Position3 (column)	Position4	Position5 (column)	Position6	Position7 ($NoDuplicatePositions_p2, row)

Show any row or column equal to 2

Position2 ($NoDuplicatePositions_p1, column)	Position3 (row)	Position4 (column)	Position5 (row)	Position6 (column)	Position7 ($NoDuplicatePositions_p2, column)

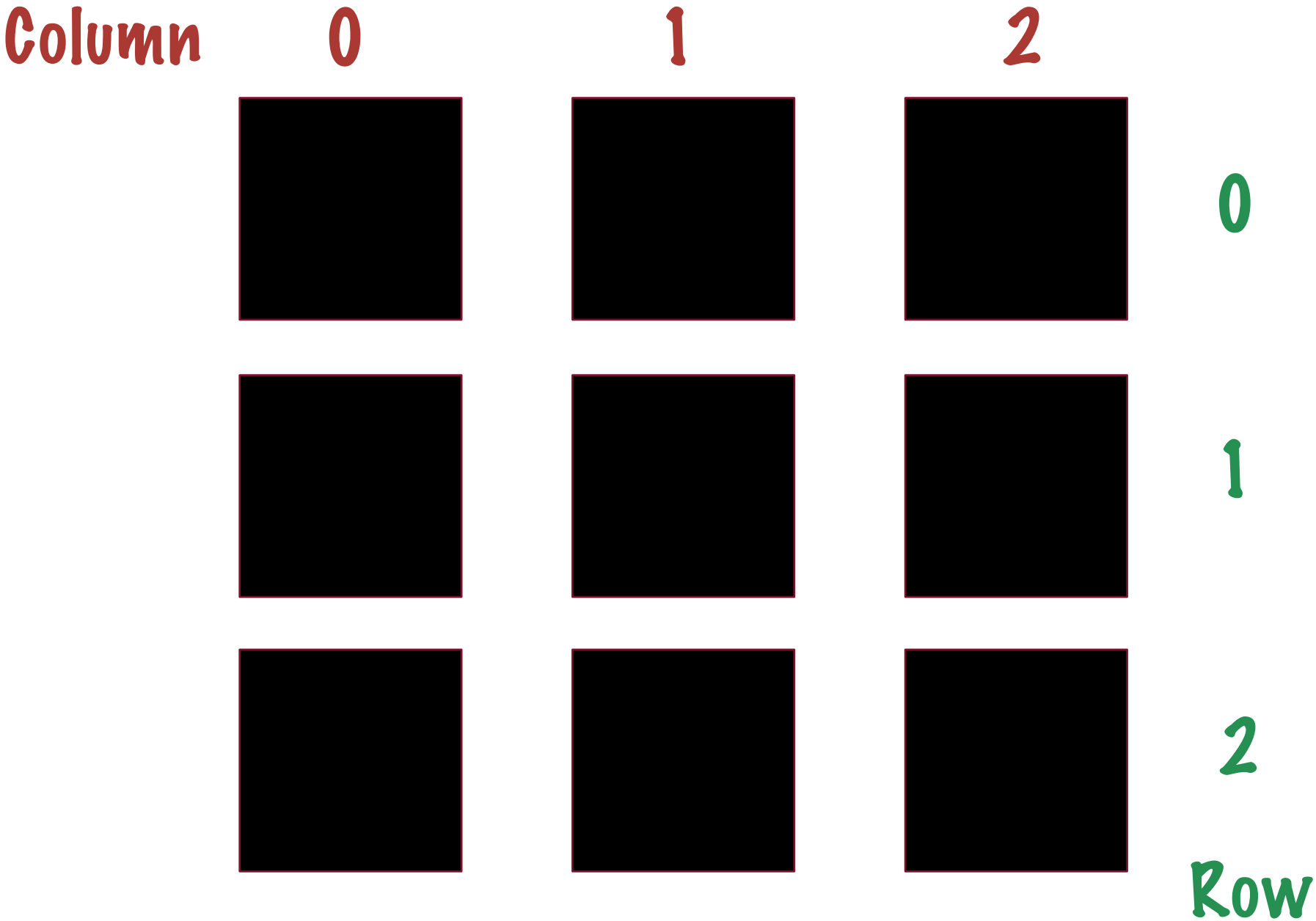
Duplicate Position at Col 2, Row 0 !!

language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
fact NoDuplicatePositions{
    all p1, p2: Position |
    (p1 != p2) => (p1.column!=p2.column) ||
    (p1.row!=p2.row)
}
```

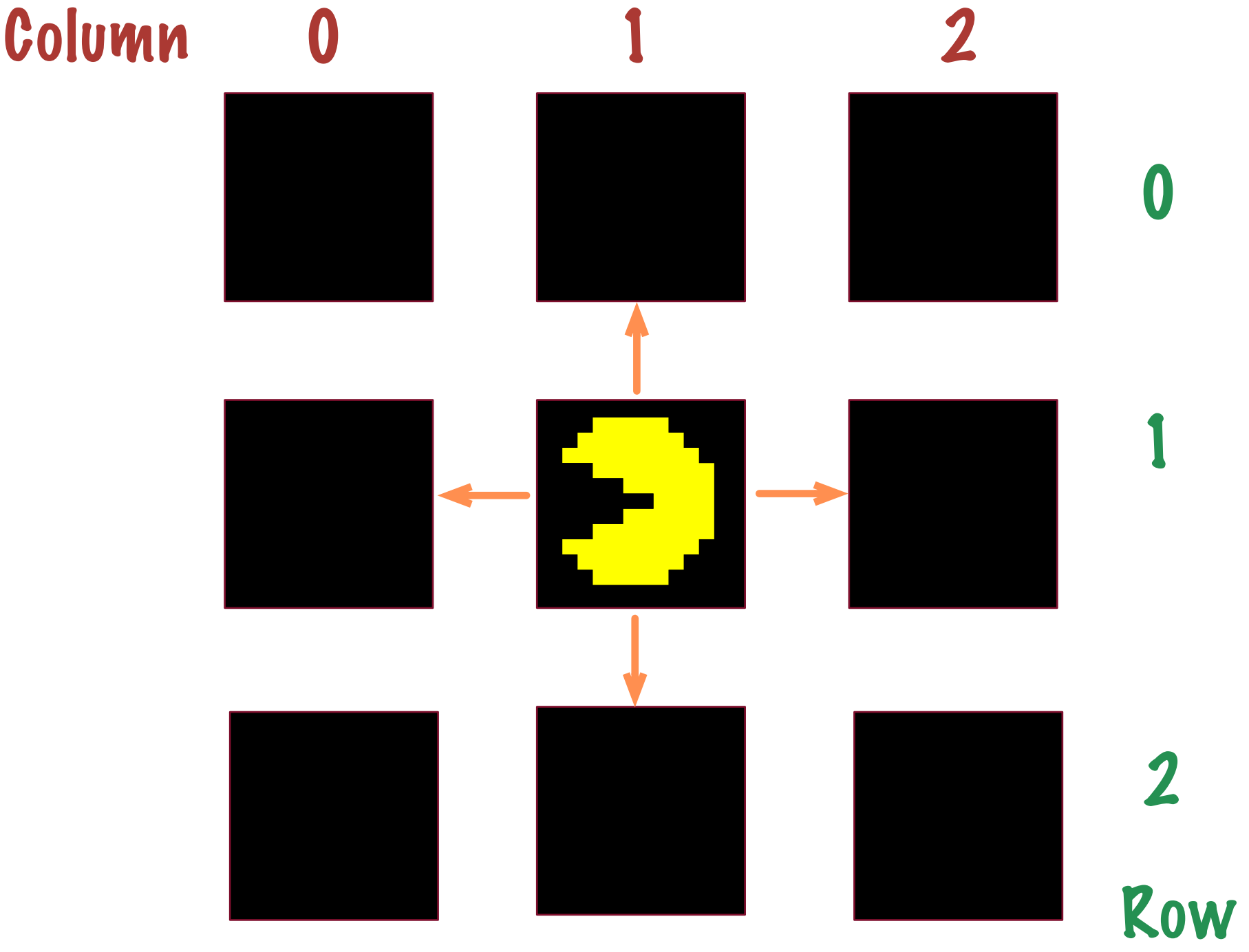

Column
0
1
2
0
1
2
Row

language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
// Adjacent columns differ by one
// Adjacent rows differ by Width

fact PositionOrder{
      all p: Position, p': p.next{
      let a = int[p.row].mul[int[Width.i]],
          a'= int[p'.row].mul[int[Width.i]] {
          (a + int[p.column]) <
          (a' + int[p'.column])
      }
    }
}
```

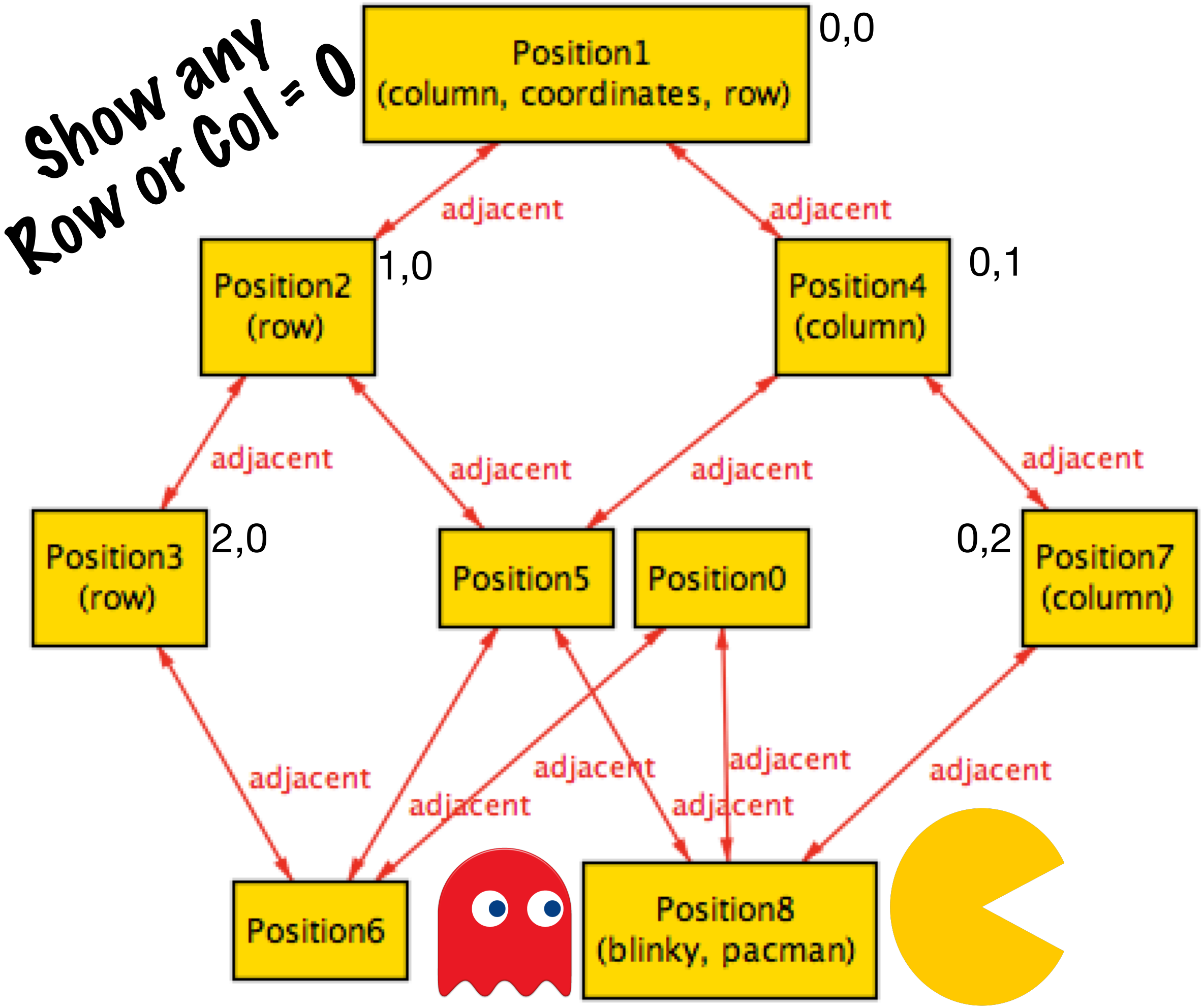

Column
0
1
2
0
1
2
Row

language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
fact AdjacentPositions{
      all p, a: Position{
         (a in p.adjacent) iff{
             int[a.row] = int[p.row] and{
                 int[a.column] = int[p.column].add[1] or
                 int[a.column] = int[p.column].sub[1]
             }

             or

             int[a.column] = int[p.column] and{
                 int[a.row] = int[p.row].add[1] or
                 int[a.row] = int[p.row].sub[1]
             }
         }
     }
}
```

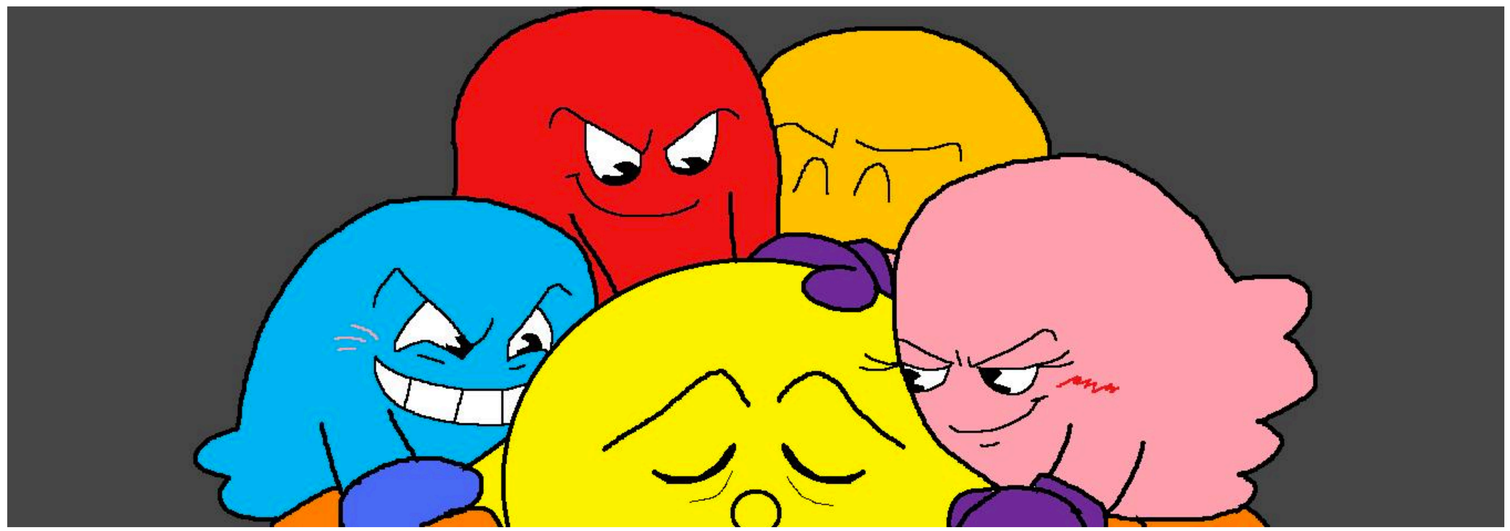

Show any
Row or Col = 0
Position1
(column, coordinates, row)
0,0
Position2
(row)
1,0
Position4
(column)
0,1
Position3
(row)
2,0
Position5
Position0
0,2
Position7
(column)
Position6
Position8
(blinky, pacman)
adjacent

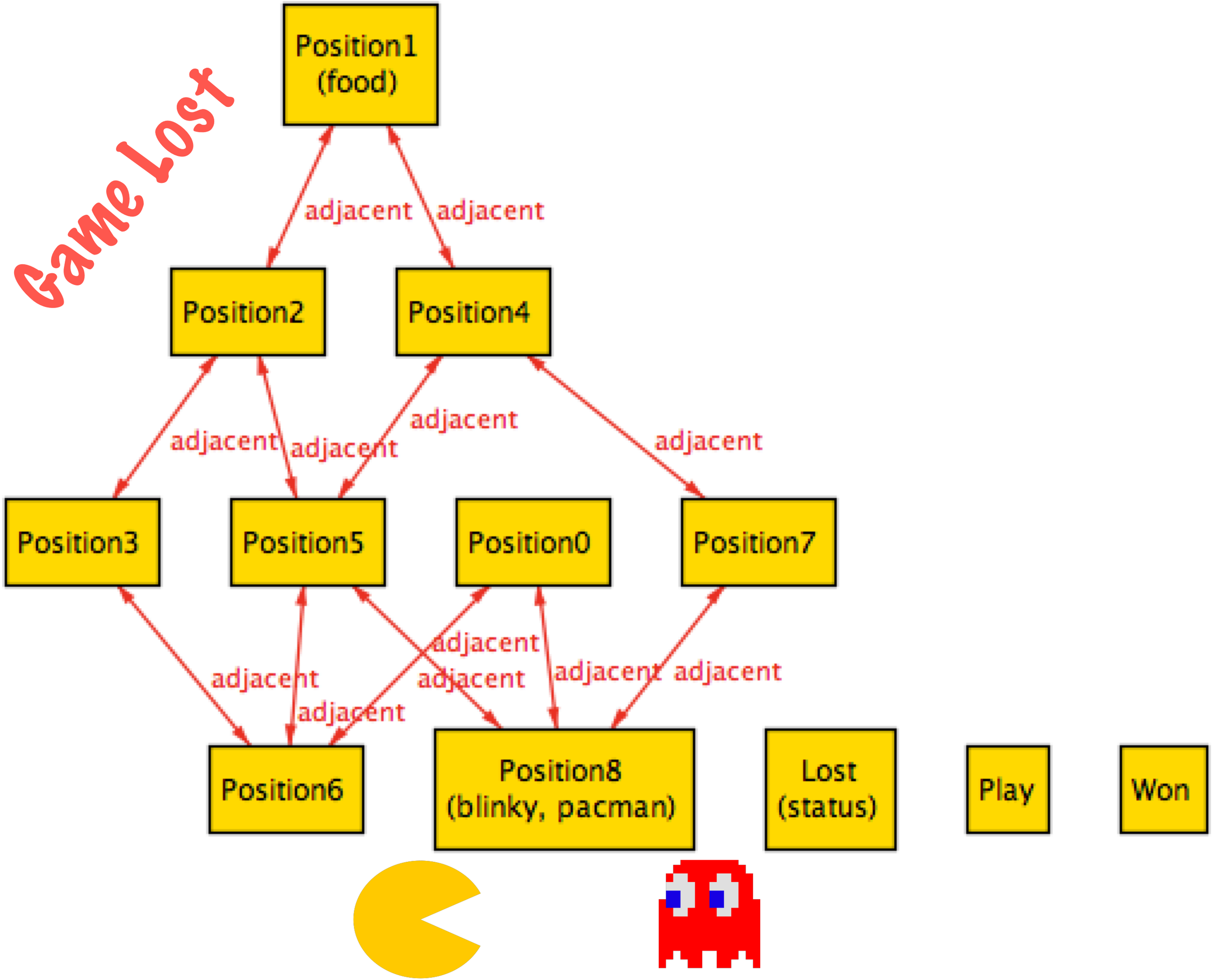
language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
fact GameIsLostWhenBlinkyGetsPacman {
    all s : State |
        s.status = Lost iff s.blinky = s.pacman
}
```



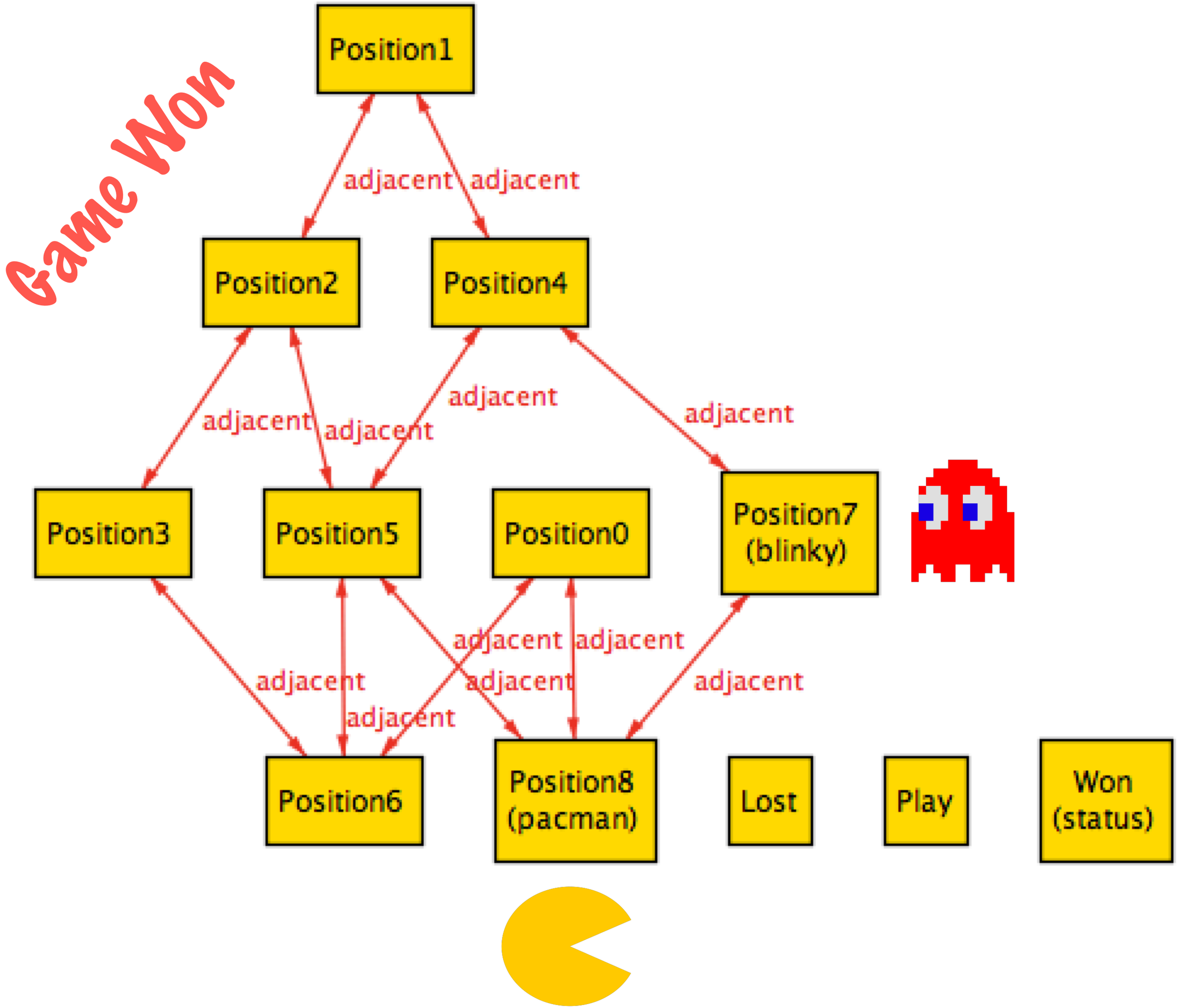

Game Lost
Position1
(food)
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
Position7
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(blinky, pacman)
Lost
(status)
Play
Won

language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
fact GameIsWonWhenAllTheFoodIsGone {
    all s : State |
        s.status= Won iff s.food = none
}
```

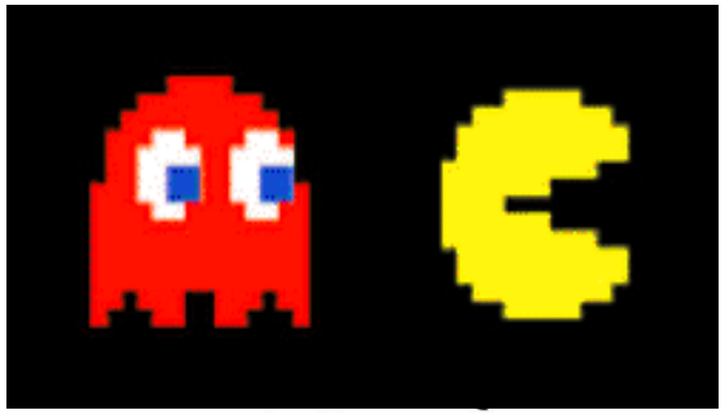

Game Won
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
Position7
(blinky)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(pacman)
Lost
Play
Won
(status)

language: run command

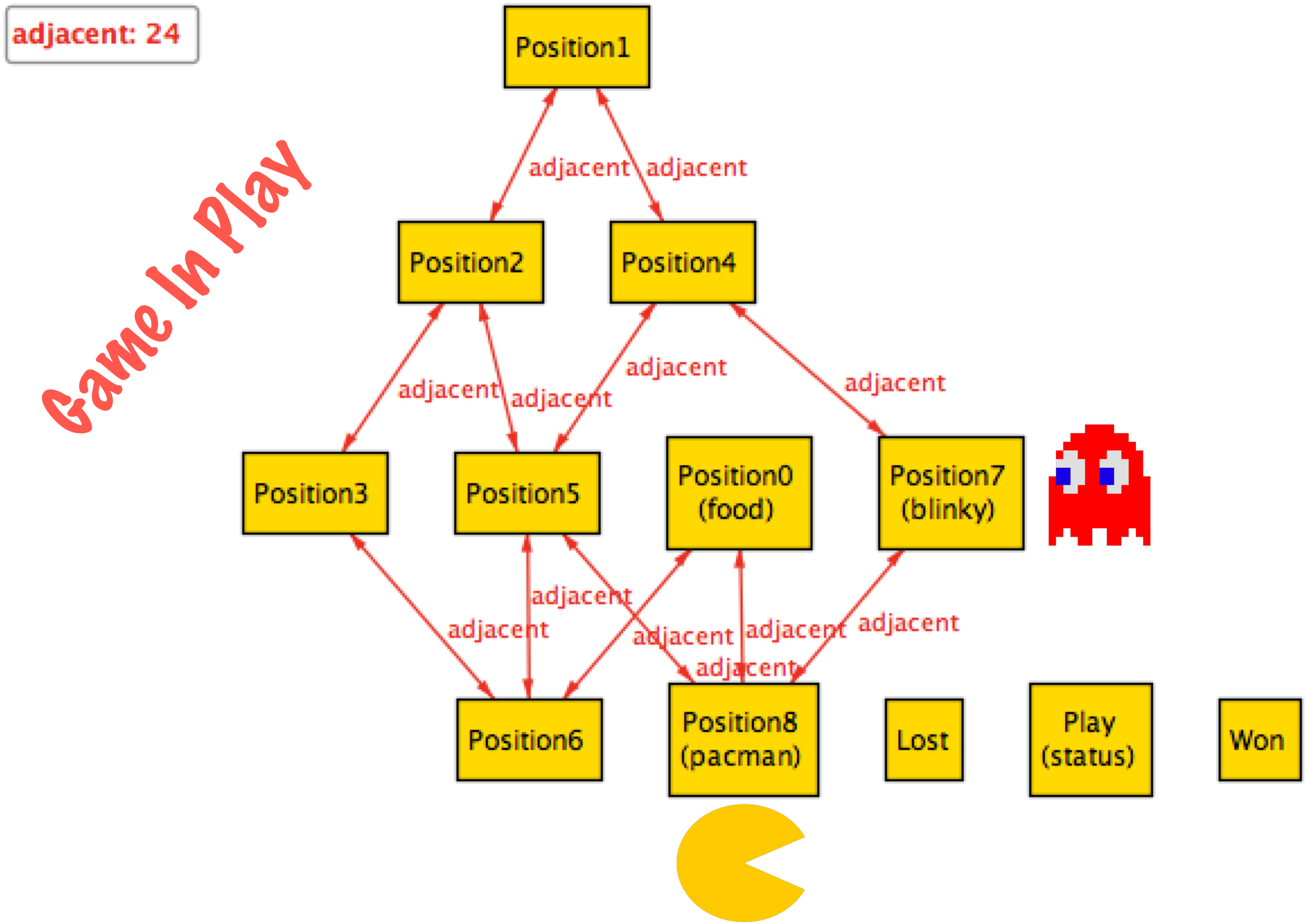
```
pred p[x: X, y: Y, ...] { F }
run p scope
```

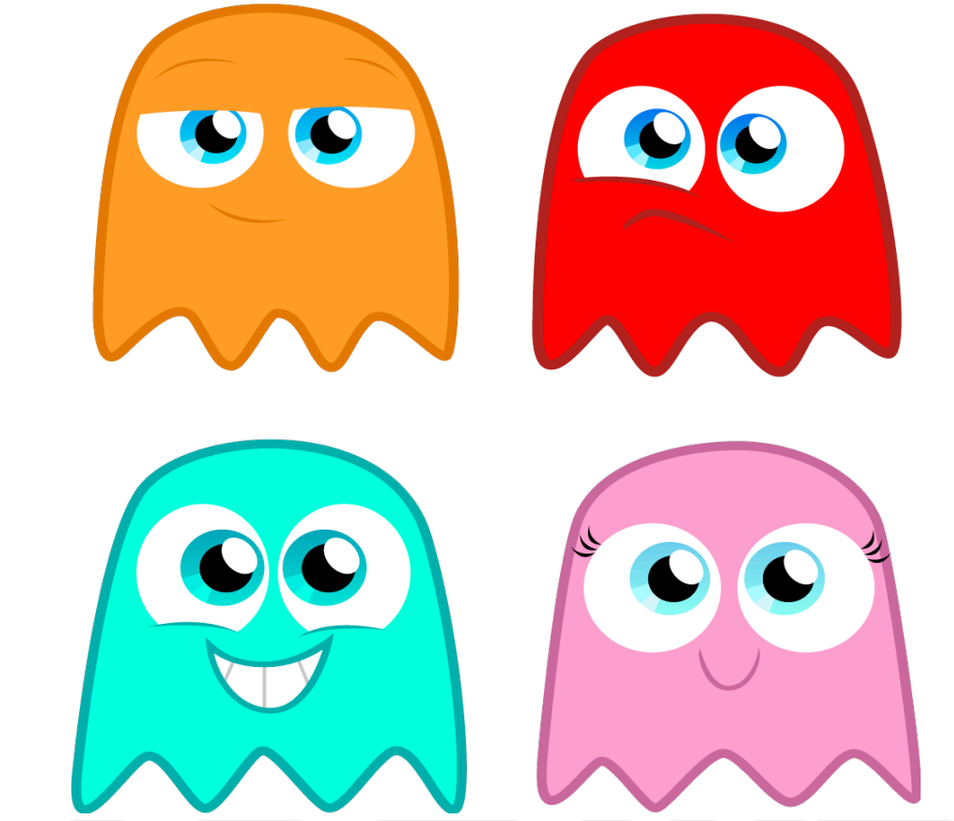
instructs analyzer to search for instance of predicate within scope

if model has facts M, finds solution to M && (some x: X, y: Y, ... | F)

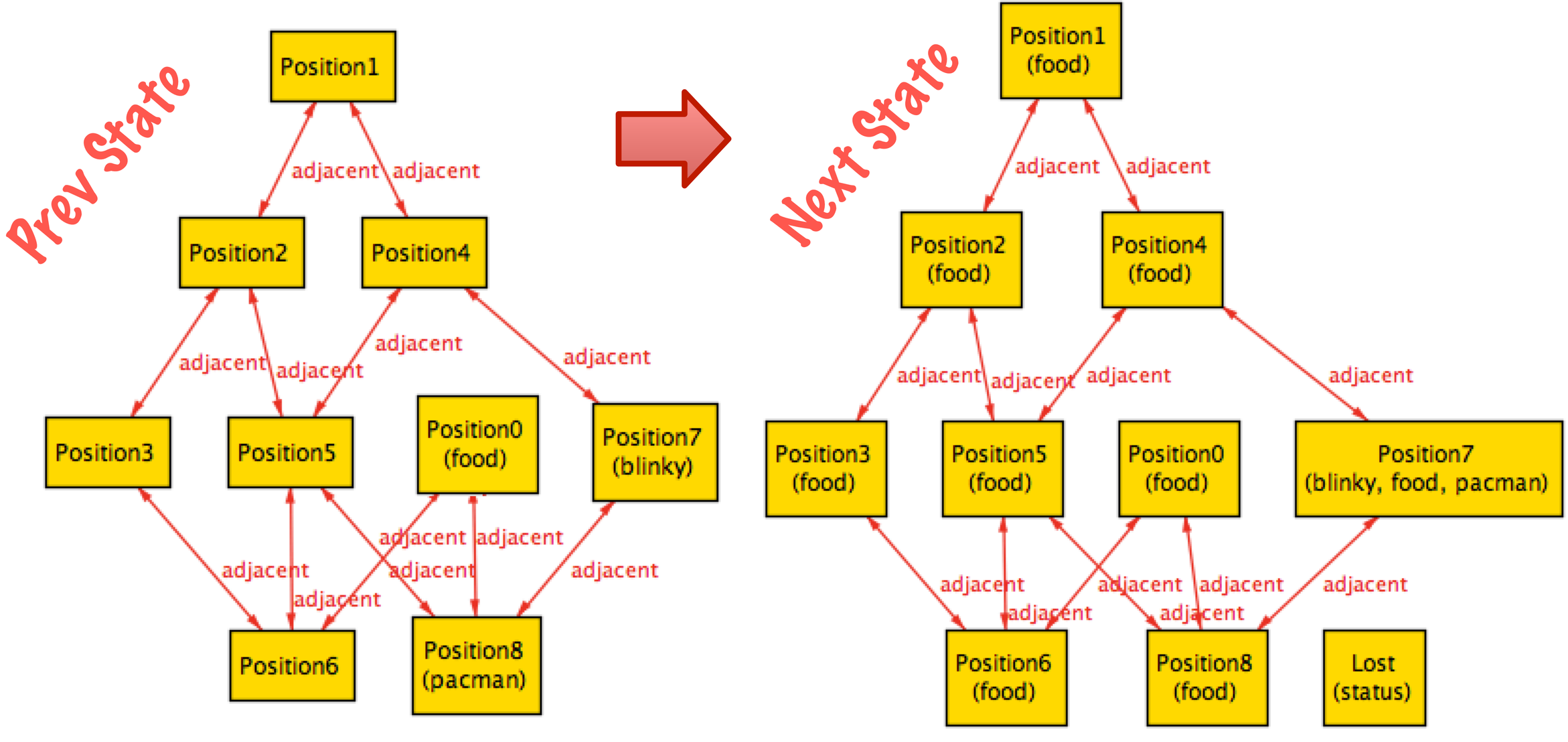


```
pred GamePlay() {
   last.status = Play
}

run GamePlay for 1 State, exactly 9 Position
```


adjacent: 24
Game In Play
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
(food)
Position7
(blinky)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(pacman)
Lost
Play
(status)
Won



ARE WE DONE YET?

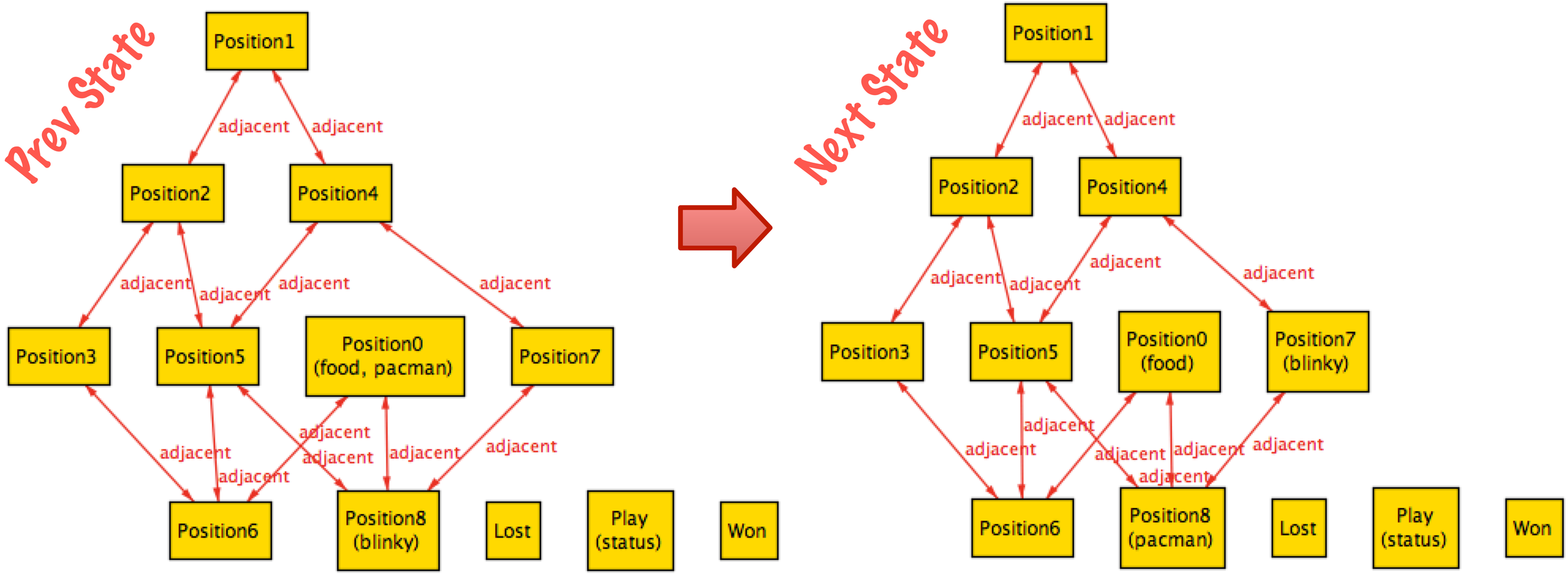

Prev State
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
(food)
Position7
(blinky)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(pacman)
Next State
Position1
(food)
adjacent
adjacent
Position2
(food)
Position4
(food)
adjacent
adjacent
adjacent
adjacent
Position3
(food)
Position5
(food)
Position0
(food)
Position7
(blinky, food, pacman)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
(food)
Position8
(food)
Lost
(status)

language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

```
pred FoodSupplyNeverIncreases[s, s' : State]
{
    s'.food in s.food
}
```

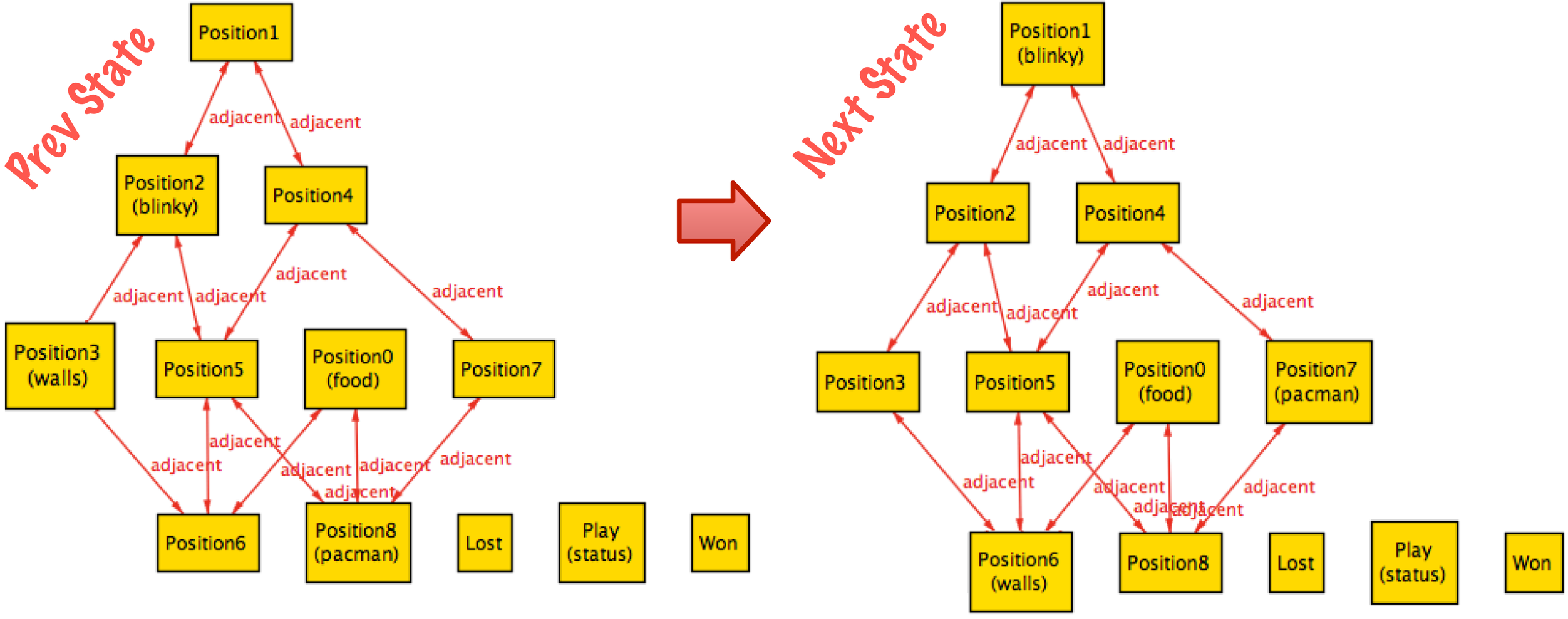

Prev State
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
(food, pacman)
Position7
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(blinky)
Lost
Play
(status)
Won
Next State
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
(food)
Position7
(blinky)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(pacman)
Lost
Play
(status)
Won

language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

```
pred FoodIsEatenByPacman[s, s': State]
{
    s'.food = s.food - s.pacman
}
```

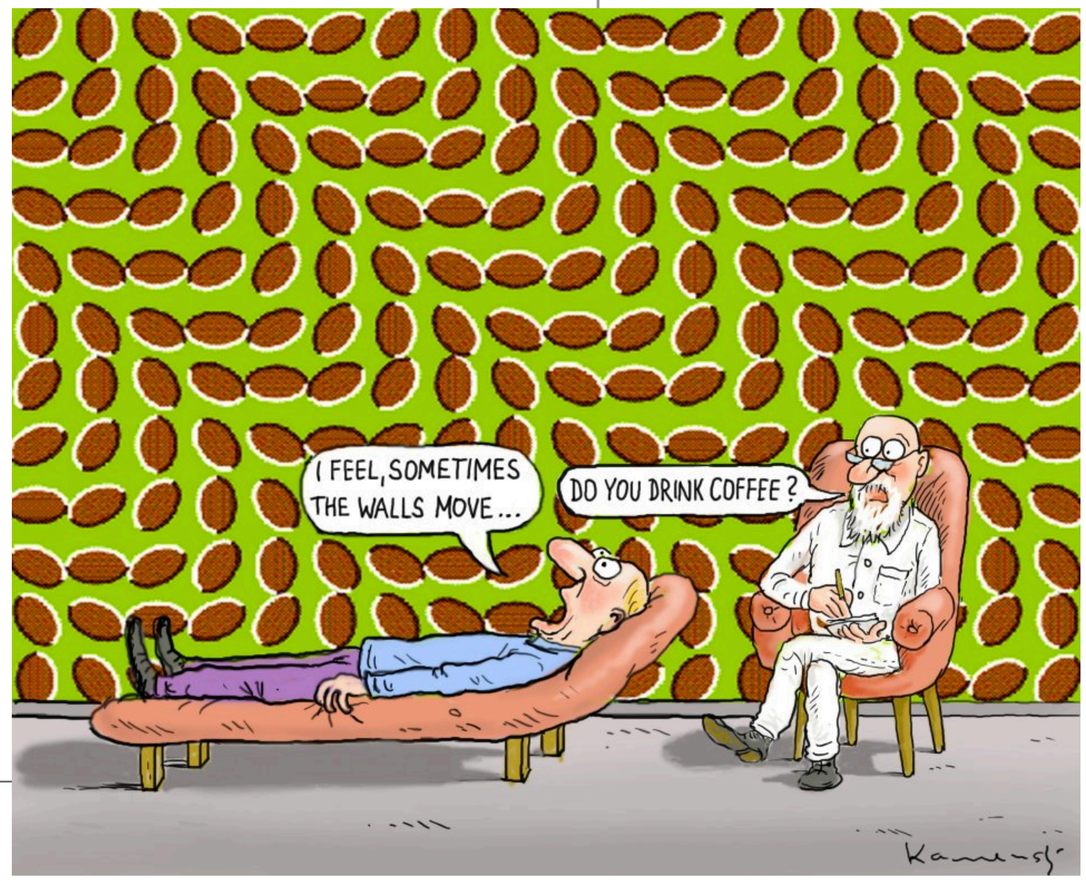

Prev State
Position1
adjacent
adjacent
Position2
(blinky)
Position4
adjacent
adjacent
adjacent
adjacent
Position3
(walls)
Position5
Position0
(food)
Position7
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(pacman)
Lost
Play
(status)
Won
Next State
Position1
(blinky)
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
(food)
Position7
(pacman)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
(walls)
Position8
Lost
Play
(status)
Won

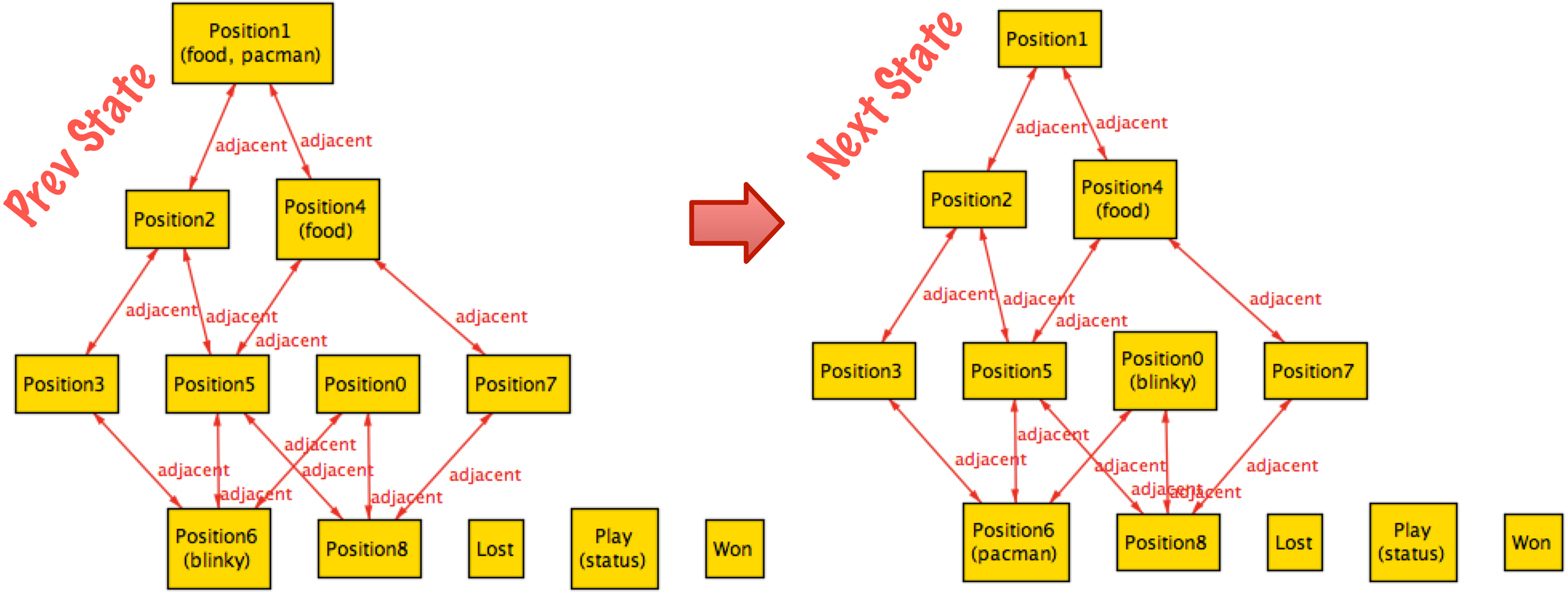
language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

```
pred WallsDontMove[ s, s' : State]
{
    s.walls = s'.walls
}
```



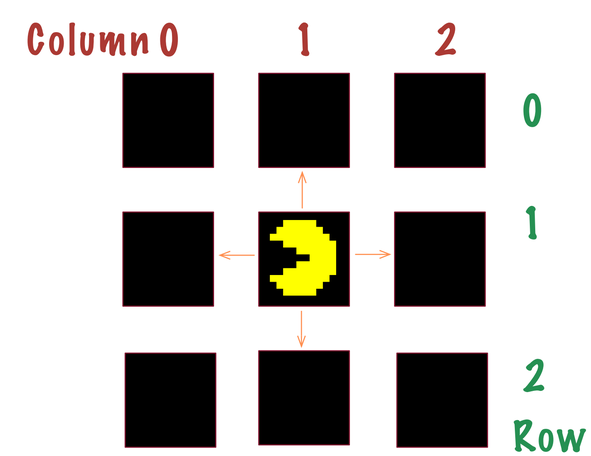

Prev State
Position1
(food, pacman)
adjacent
adjacent
Position2
Position4
(food)
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
Position7
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
(blinky)
Position8
Lost
Play
(status)
Won
Next State
Position1
adjacent
adjacent
Position2
Position4
(food)
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
(blinky)
Position7
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
(pacman)
Position8
Lost
Play
(status)
Won

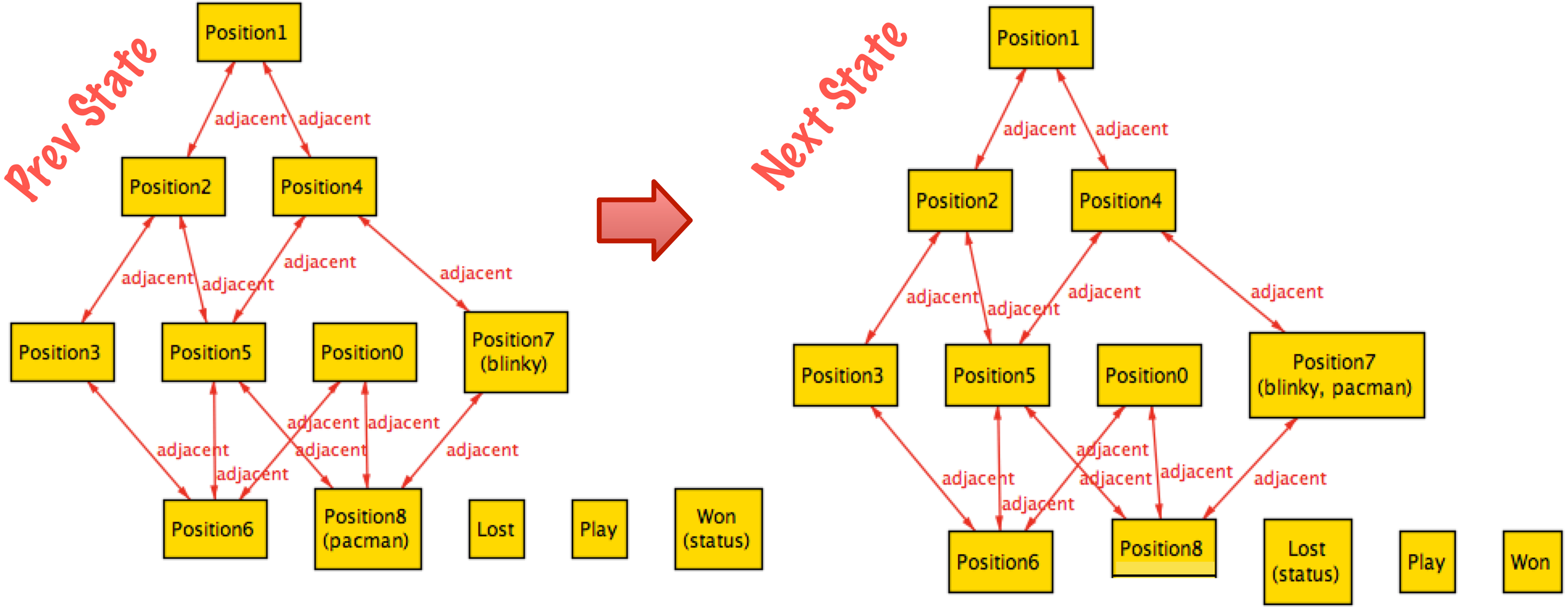
language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

```
pred MovementIsOnlyToAdjacentPosition[s, s':State]
{
    let p = (s'.pacman), q = (s.pacman) | p->q in adjacent
    let p = (s'.blinky), q = (s.blinky) | p->q in adjacent
}
```




Prev State
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
Position7
(blinky)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
(pacman)
Lost
Play
Won
(status)
Next State
Position1
adjacent
adjacent
Position2
Position4
adjacent
adjacent
adjacent
adjacent
Position3
Position5
Position0
Position7
(blinky, pacman)
adjacent
adjacent
adjacent
adjacent
adjacent
adjacent
Position6
Position8
Lost
(status)
Play
Won

language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

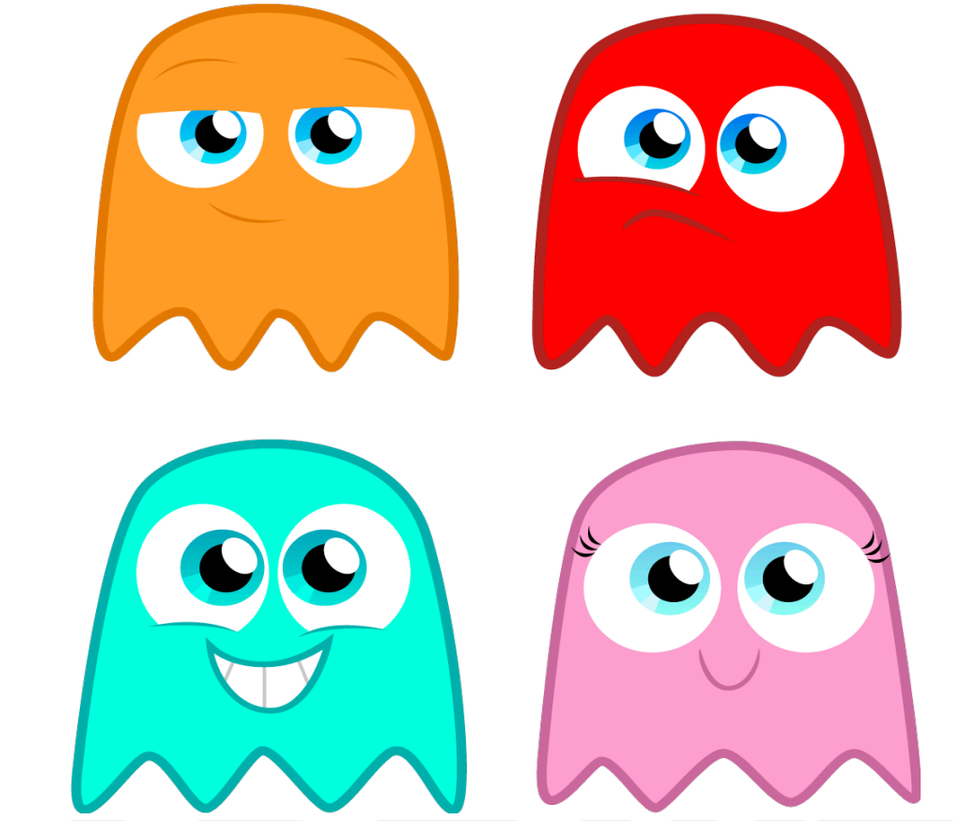
```
pred MovementIsPossibleOnlyWhilePlaying[s,s':State]
{
    s.status != Play implies s = s'
}
```

language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

facts introduce constraints that are assumed to always hold

```
fact ValidBehavior {
    all s: State, s': s.next {
        FoodSupplyNeverIncreases[s,s'] and
        FoodIsEatenByPacman[s,s'] and
        MovementIsOnlyToAdjacentPosition[s,s'] and
        WallsDontMove[s,s'] and
        MovementIsPossibleOnlyWhilePlaying[s,s']
    }
}
```



ARE WE DONE YET?

Where
do I START?

language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

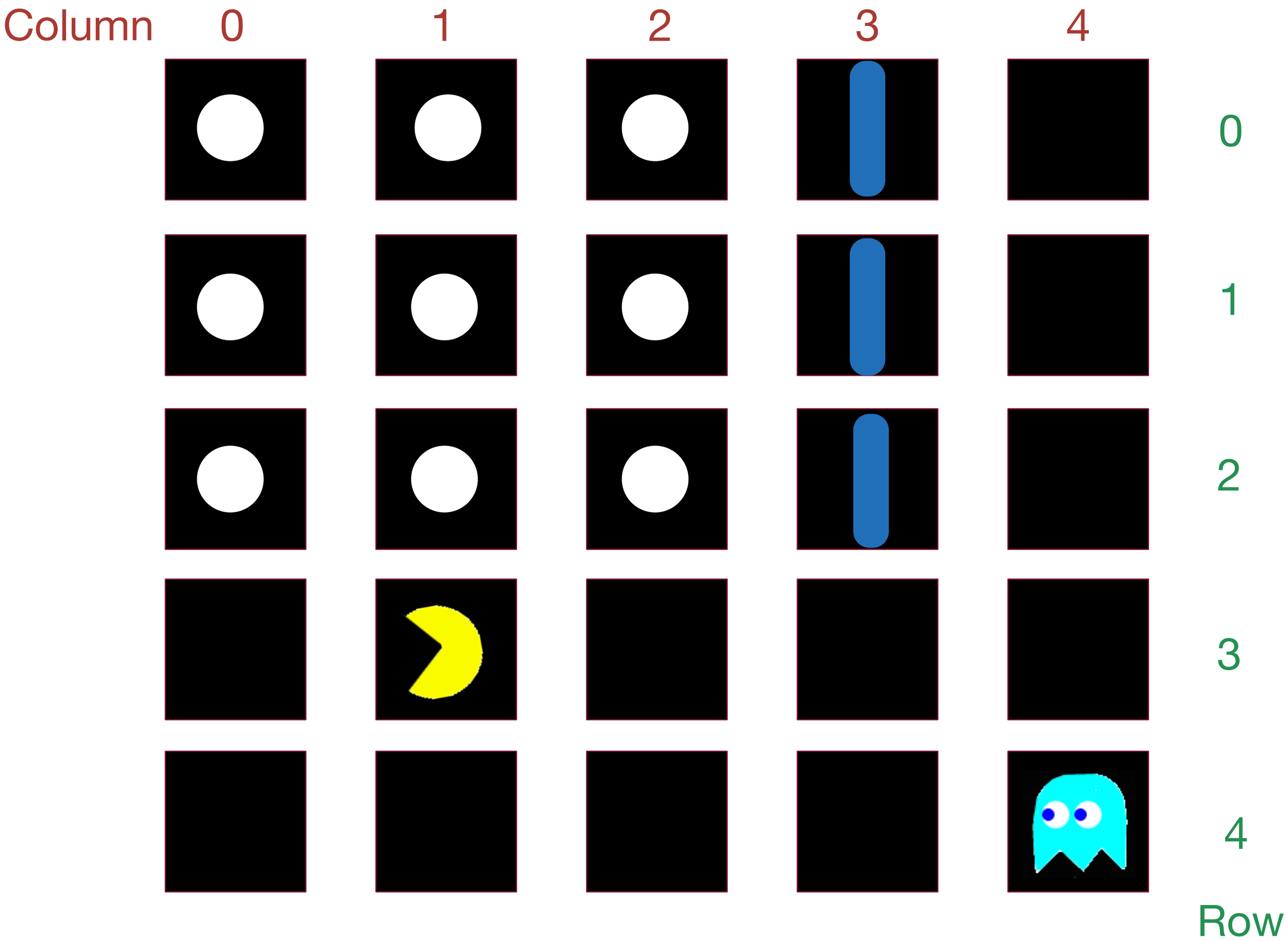
```
pred InitGame[s : State]
{
   s.walls.coordinates = Int[3]->Int[0] + Int[3]->Int[1] +
                         Int[3]->Int[2] and

   s.food.coordinates =  Int[0]->Int[0] + Int[1]->Int[0] +
                         Int[2]->Int[0] + Int[0]->Int[1] +
                         Int[1]->Int[1] + Int[2]->Int[1] +
                         Int[0]->Int[2] + Int[1]->Int[2] +
                         Int[2]->Int[2] and
   s.status = Play
}
```

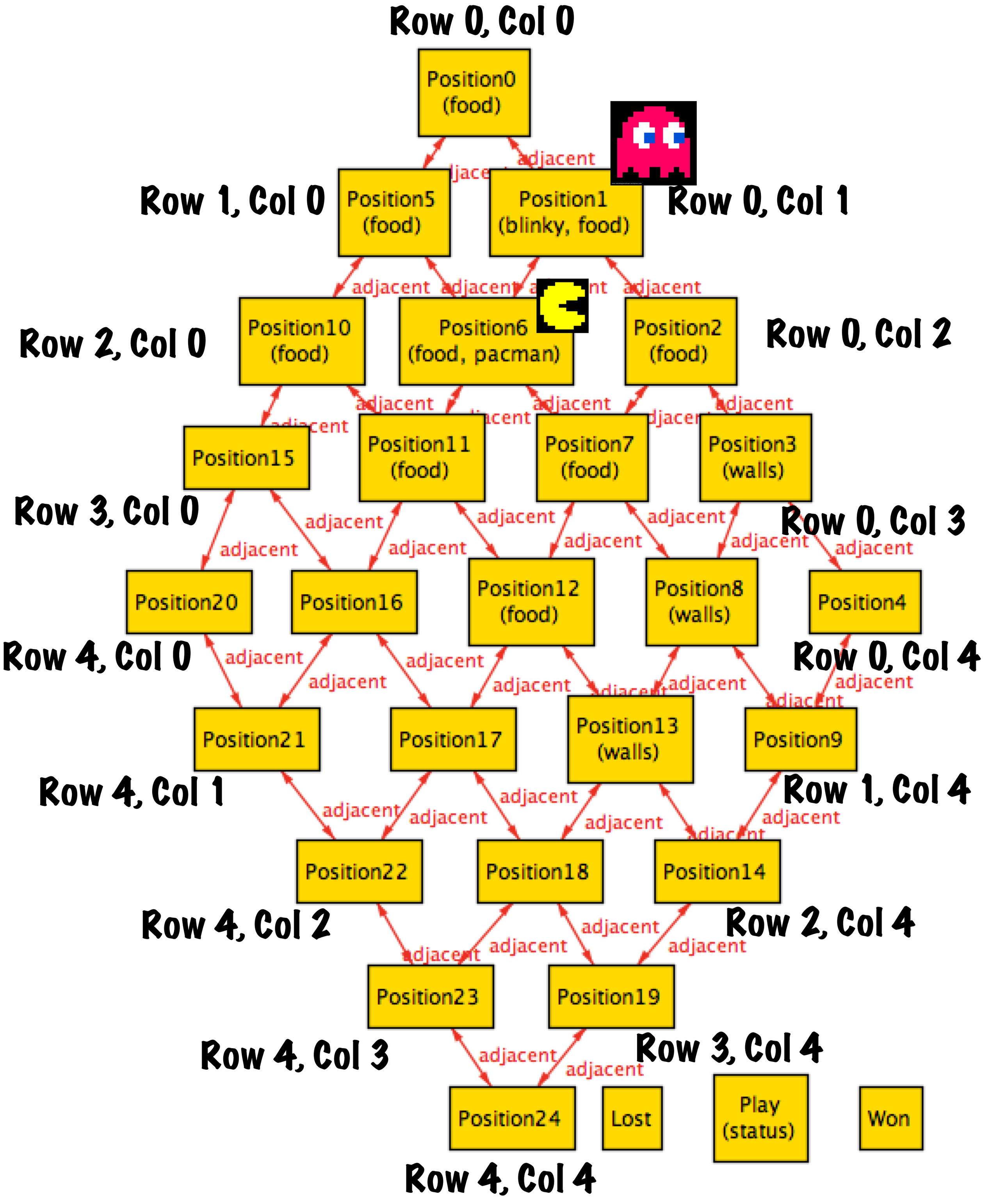
language: facts

```
fact { F }
fact f { F }
sig S { ... }{ F }
```

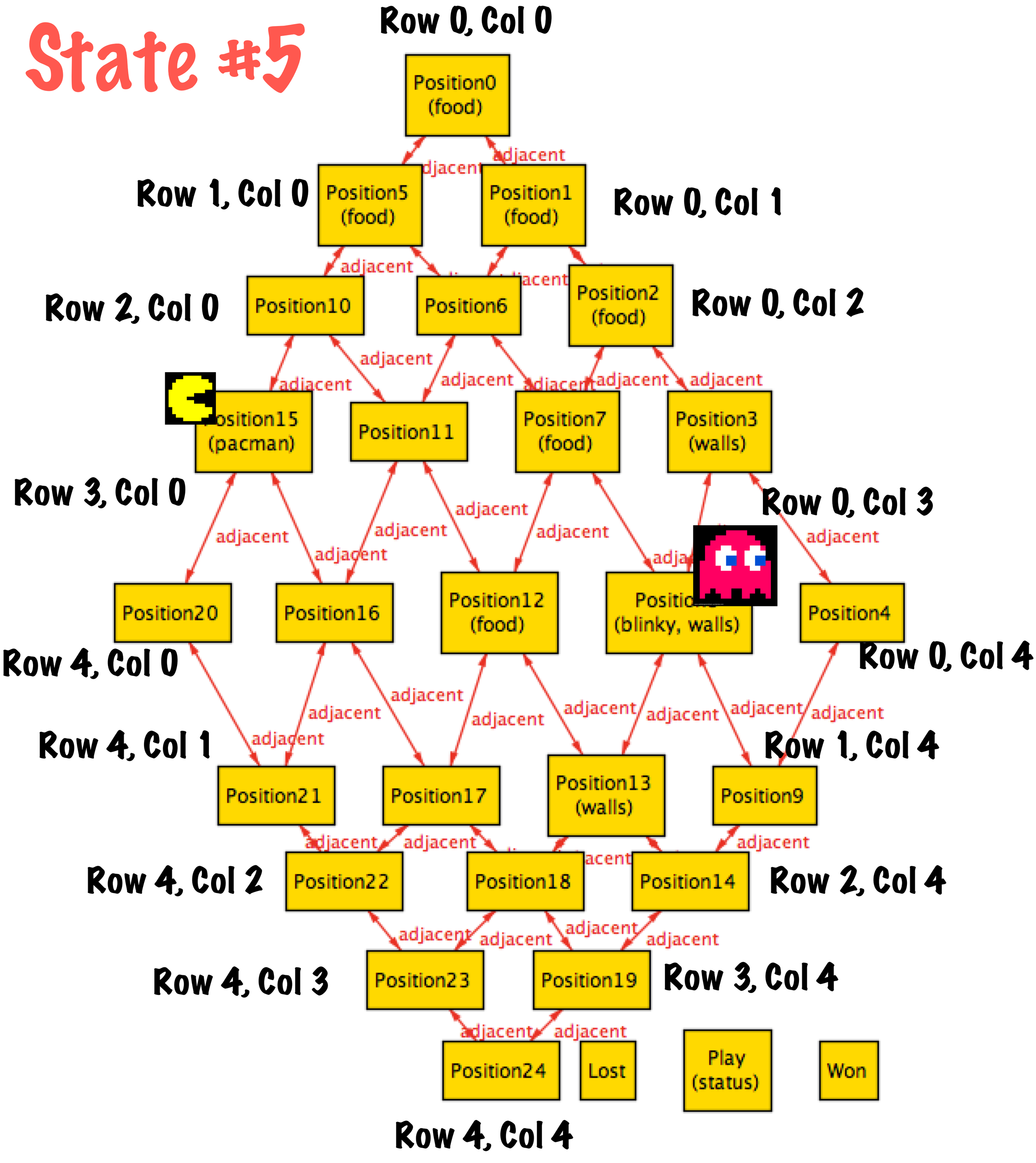
facts introduce constraints that are assumed to always hold

```
fact initialState {
  InitGame[first]
}
```


Column
0
1
2
3
4
0
1
2
3
4
Row

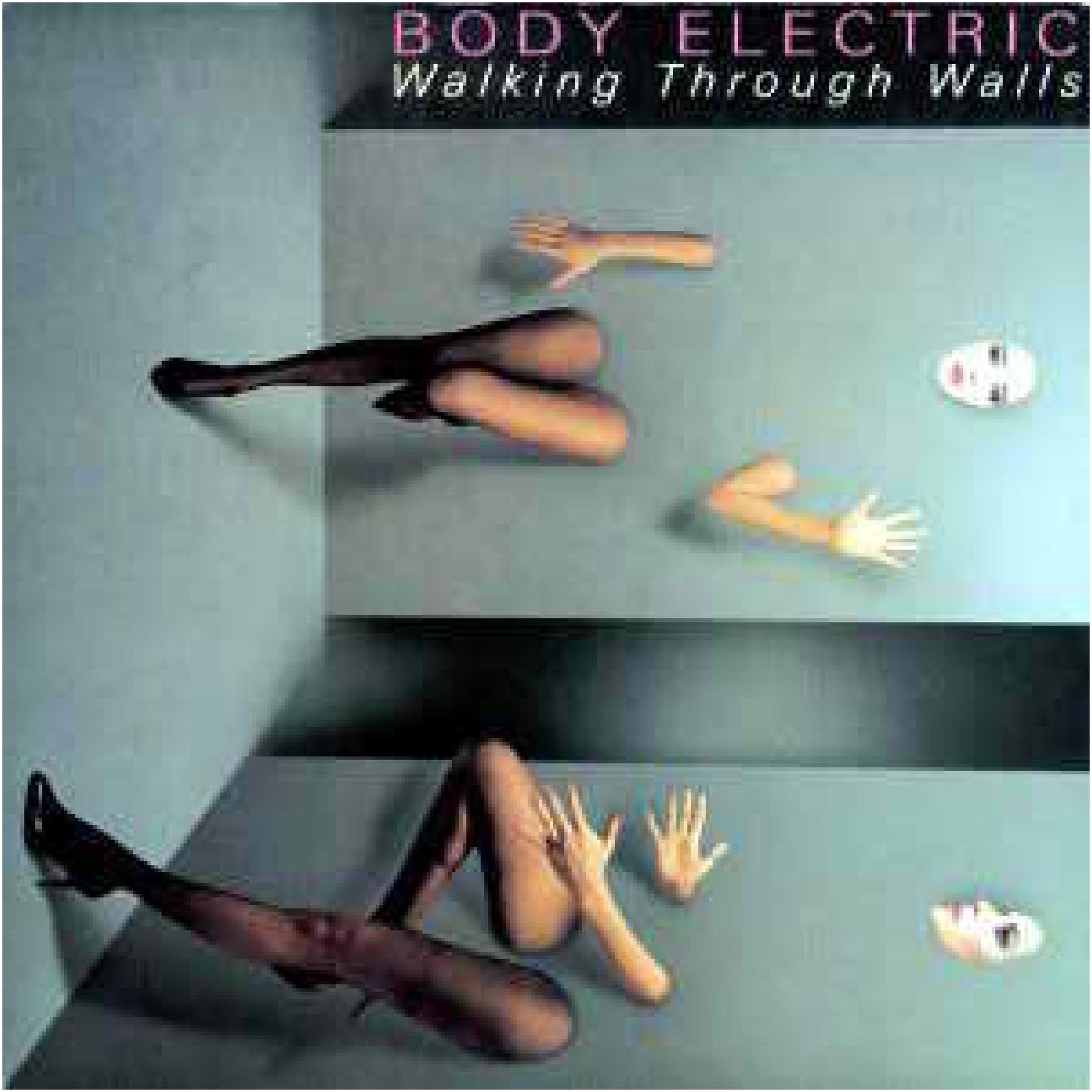

Row 0, Col 0
Position0 (food)
Row 1, Col 0
Position5 (food)
Position1 (blinky, food)
Row 0, Col 1
Row 2, Col 0
Position10 (food)
Position6 (food, pacman)
Position2 (food)
Row 0, Col 2
Position15
Position11 (food)
Position7 (food)
Position3 (walls)
Row 3, Col 0
Row 0, Col 3
Position20
Position16
Position12 (food)
Position8 (walls)
Position4
Row 4, Col 0
Row 0, Col 4
Position21
Position17
Position13 (walls)
Position9
Row 4, Col 1
Row 1, Col 4
Position22
Position18
Position14
Row 4, Col 2
Row 2, Col 4
Position23
Position19
Row 4, Col 3
Row 3, Col 4
Position24
Lost
Play (status)
Won
Row 4, Col 4
adjacent

State #5


Row 0, Col 0
Position0 (food)
Row 1, Col 0
Position5 (food)
Position1 (food)
Row 0, Col 1
Row 2, Col 0
Position10
Position6
Position2 (food)
Row 0, Col 2
Position15 (pacman)
Position11
Position7 (food)
Position3 (walls)
Row 3, Col 0
Row 0, Col 3
Position20
Position16
Position12 (food)
(blinky, walls)
Position4
Row 4, Col 0
Row 0, Col 4
Row 4, Col 1
Row 1, Col 4
Position21
Position17
Position13 (walls)
Position9
Row 4, Col 2
Position22
Position18
Position14
Row 2, Col 4
Row 4, Col 3
Position23
Position19
Row 3, Col 4
Position24
Lost
Play (status)
Won
Row 4, Col 4
adjacent

BODY ELECTRIC

Walking Through Walls



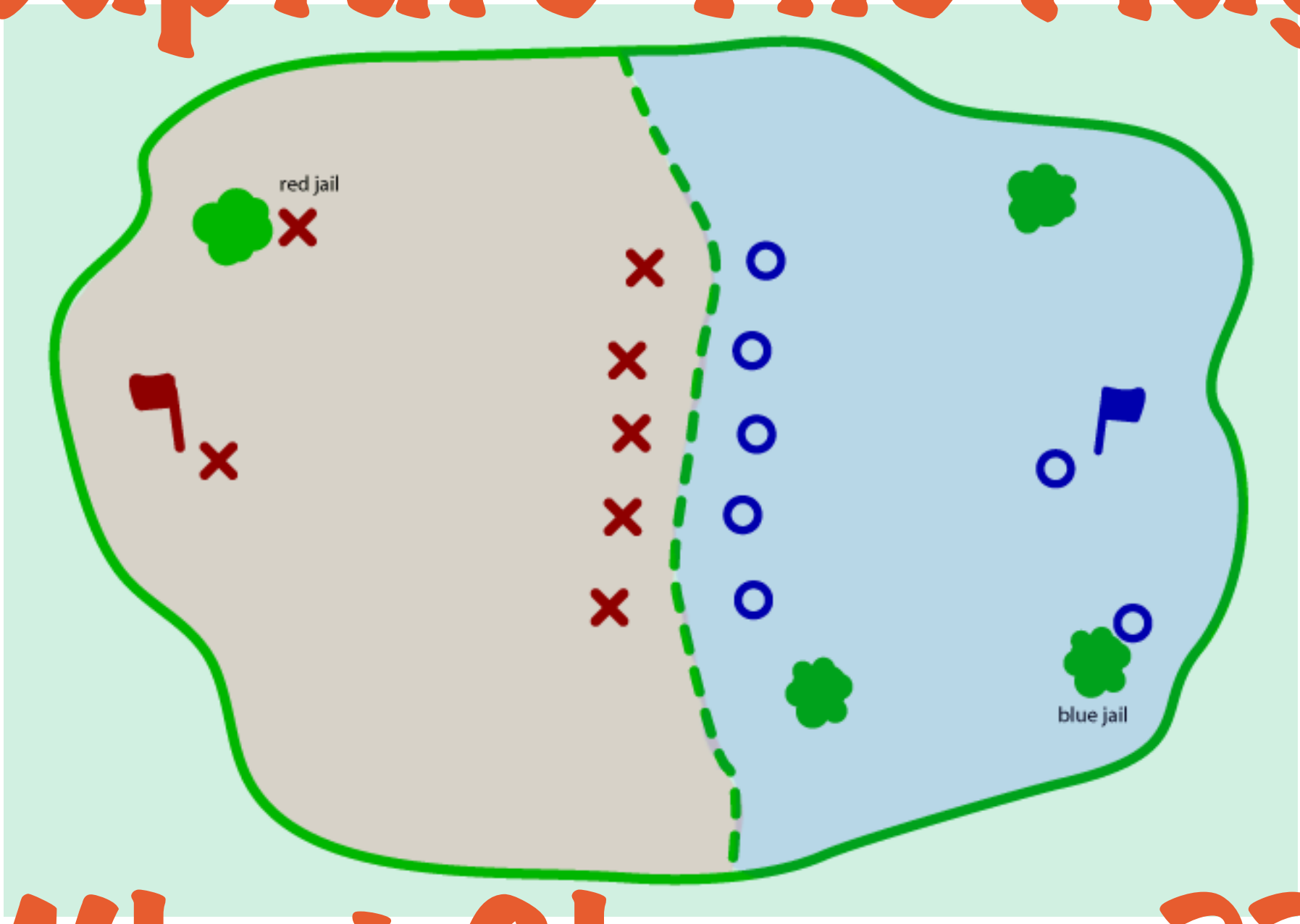
language: predicates

```
pred p[x1: e1, ..., xn: en] { F }
```

named formula describing changes in model state over time

```
pred MovementIsOnlyToAdjacentPosition[s, s':State]
{
    let p = (s'.pacman), q = (s.pacman) | (p->q in adjacent)
            and not( q in s.walls || p in s'.walls )
    let p = (s'.blinky), q = (s.blinky) |  (p->q in adjacent)
            and not( q in s.walls || p in s'.walls )
}
```

Capture The Flag



What Changes??

more examples: alloy.mit.edu

about | community | download | documentation | book | applications | people | thanks

alloy: a language & tool for relational models

about alloy

Alloy is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.

An Alloy model is a collection of constraints that describes (implicitly) a set of structures, for example: all the possible security configurations of a web application, or all the possible topologies of a switching network. Alloy's tool, the Alloy Analyzer, is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples. Structures are displayed graphically, and their appearance can be customized for the domain at hand.

At its core, the Alloy language is a simple but expressive logic based on the notion of relations, and was inspired by the Z specification language and Tarski's relational calculus. Alloy's syntax is designed to make it easy to build models incrementally, and was influenced by modeling languages (such as the object models of OMT and UML). Novel features of Alloy include a rich subtype facility for factoring out common features and a uniform and powerful syntax for navigation expressions.

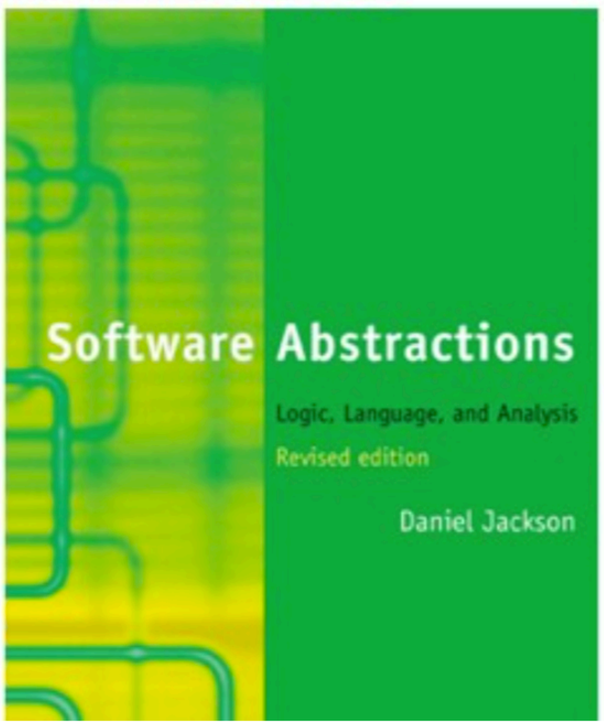
The Alloy Analyzer works by reduction to SAT. Version 4 was a complete rewrite that included Kodkod, a new model finding engine that optimizes the reduction, and a new front end.

news

ASM, Alloy, B and Z Conference: papers now due January 22!

Research programmer position available on Alloy project!

Revised edition of book now out! Available from MIT Press.



unity of purpose?

Conceptual integrity is the most important consideration in system design. It is better to have a system **omit certain anomalous features** and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.

-- Fred Brooks, 1975