Logic Programming
Simple Examples

Michael Genesereth
Computer Science Department
Stanford University

Programme

Examples, Examples, Examples

- Kinship
- Blocks World
- Boolean Logic
- Tournament

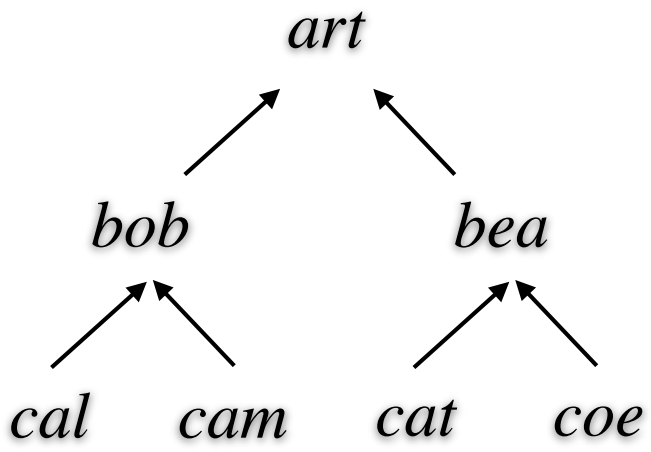
Next Time

- Lists
- Sets
- Trees

Kinship

Datasets

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```



Example

Grandparents:

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
grandparent(art,cal)
grandparent(art,cam)
grandparent(art,cat)
grandparent(art,coe)
```

Example

Grandparents:

```
grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)
```

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
grandparent(art,cal)
grandparent(art,cam)
grandparent(art,cat)
grandparent(art,coe)
```

Example

Personhood:

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
person(art)
person(bob)
person(cal)
person(cam)
person(bea)
person(cat)
person(coe)
```

Example

Personhood:

```
person(X) :- parent(X,Y)
person(Y) :- parent(X,Y)
```

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
person(art)
person(bob)
person(cal)
person(cam)
person(bea)
person(cat)
person(coe)
```

Example

Childlessness:

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
childless(cal)
childless(cam)
childless(cat)
childless(coe)
```

Example

Childlessness using aggregate:

```
childless(X) :-
  evaluate(countofall(Y,parent(X,Y)),0)
```

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
childless(cal)
childless(cam)
childless(cat)
childless(coe)
```

Childlessness using negation:

```
childless(X) :- person(X) & ~isparent(X)
isparent(X) :- parent(X,Y)
```

Example

Ancestors:

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
ancestor(art,bob)
ancestor(art,bea)
ancestor(bob,cal)
ancestor(bob,cam)
ancestor(bea,cat)
ancestor(bea,coe)
ancestor(art,cal)
ancestor(art,cam)
ancestor(art,cat)
ancestor(art,coe)
```

Example

Ancestors:

```
ancestor(X,Z) :- parent(X,Z)
ancestor(X,Z) :- parent(X,Y) & ancestor(Y,Z)
```

Data:

```
parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
```

View:

```
ancestor(art,bob)
ancestor(art,bea)
ancestor(bob,cal)
ancestor(bob,cam)
ancestor(bea,cat)
ancestor(bea,coe)
ancestor(art,cal)
ancestor(art,cam)
ancestor(art,cat)
ancestor(art,coe)
```

Computing `ancestor` Top-Down

Ancestors:

```
ancestor(X,Z) :- parent(X,Z)
ancestor(X,Z) :- parent(X,Y) & ancestor(Y,Z)
```

Trace:

```
  Call: ancestor(X,Z)
    Call: parent(X,Z)
    Exit: parent(art,bob)
* Exit: ancestor(art,bob)
  Redo: ancestor(X,Z)
    Redo: parent(X,Z)
    Exit: parent(bob,cal)
* Exit: ancestor(bob,cal)
  Redo: ancestor(X,Z)
    Redo: parent(X,Z)
    Fail: parent(X,Z)
```

```
    Call: parent(X,V8)
    Exit: parent(art,bob)
    Call: ancestor(bob,Z)
      Call: parent(bob,Z)
      Exit: parent(bob,cal)
    Exit: ancestor(Y,cal)
* Exit: ancestor(art,cal)
  Redo: ancestor(X,Z)
    Redo: ancestor(Y,Z)
      Redo: parent(bob,Z)
      Fail: parent(bob,cal)
            ...
```

Computing `ancestor` Top-Down

Ancestors:

```
ancestor(X,Z) :- parent(X,Z)
ancestor(X,Z) :- ancestor(X,Y) & ancestor(Y,Z)
```

Trace:

```
Call: ancestor(X,Z)
  Call: parent(X,Z)
  Fail: parent(X,Z)
  Call: ancestor(X,V144)
    Call: parent(X,V144)
    Fail: parent(X,V144)
    Call: ancestor(X,V145)
      Call: parent(X,V145)
      Fail: parent(X,V145)
      Call: ancestor(X,V146)
      ...
```

Constraints

Up to now - unconstrained datasets

any dataset from its Herbrand base is acceptable

Not always the case

A person cannot be both dead and alive.

A person cannot be own parent.

Every parent in parent relation must be in adult relation.

`illegal`

Boolean / 0-ary predicate

true if and only if dataset violates at least one constraint

We encode constraints by defining `illegal`.

Examples

A person cannot be both dead and alive.

```
illegal :- dead(X) & alive(X)
```

Examples

A person cannot be both dead and alive.

```
illegal :- dead(X) & alive(X)
```

A person cannot be his own parent. - using `parent/2`

Examples

A person cannot be both dead and alive.

```
illegal :- dead(X) & alive(X)
```

A person cannot be his own parent. - using `parent/2`

```
illegal :- parent(X,X)
```

Examples

A person cannot be both dead and alive.

```
illegal :- dead(X) & alive(X)
```

A person cannot be his own parent. - using `parent/2`

```
illegal :- parent(X,X)
```

Every parent is an adult. - using `parent/2` and `adult/1`

Examples

A person cannot be both dead and alive.

```
illegal :- dead(X) & alive(X)
```

A person cannot be his own parent. - using `parent/2`

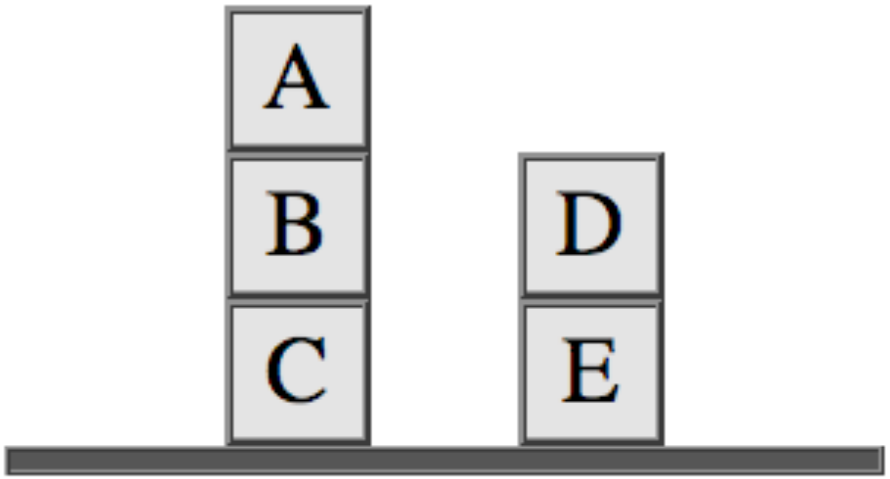
```
illegal :- parent(X,X)
```

Every parent is an adult. - using `parent/2` and `adult/1`

```
illegal :- parent(X,Y) & ~adult(X)
```

Blocks World

Blocks World



Vocabulary

Symbols: `a`, `b`, `c`, `d`, `e`

Unary Predicates:

- `clear` - blocks with no blocks on top
- `cluttered` - blocks with something on top
- `supported` - blocks resting on other blocks
- `table` - blocks on the table

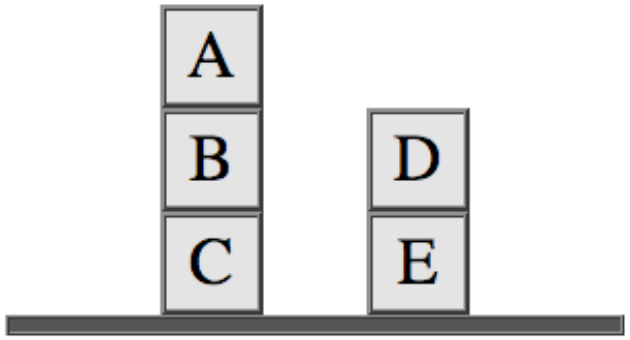
Binary Predicates:

- `on` - pairs of blocks in which first is on the second
- `above` - pairs in which first block is above the second

Ternary Predicates:

- `stack` - triples of blocks arranged in a stack

Data




```
block(a)
block(b)
block(c)
block(d)
block(e)
```

```
on(a,b)
on(b,c)
on(d,e)
```

```
clear(a)
clear(d)

cluttered(b)
cluttered(c)
cluttered(e)
```

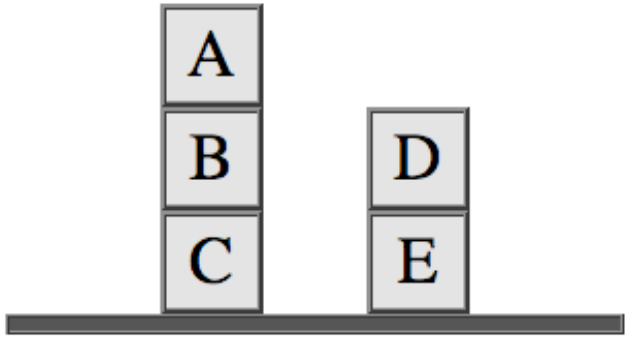
```
table(c)
table(e)

supported(a)
supported(b)
supported(d)
```

```
above(a,b)
above(b,c)
above(a,c)
above(d,e)

stack(a,b,c)
```

Blocks World



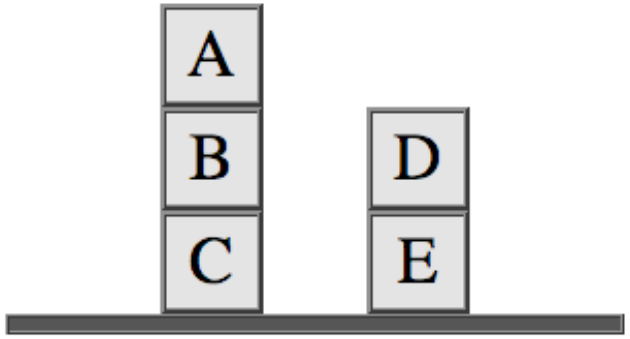

```
block(a)
block(b)
block(c)
block(d)
block(e)
```

```
on(a,b)
on(b,c)
on(d,e)
```

```
cluttered(b)
cluttered(c)
cluttered(e)
```

```
clear(a)
clear(d)
```

Blocks World



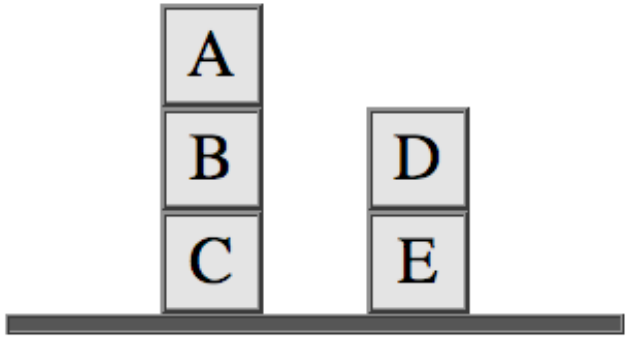

```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
cluttered(Y) :- on(X,Y)
clear(Y) :- block(Y) & ~cluttered(Y)
```

```
cluttered(b)   clear(a)
cluttered(c)   clear(d)
cluttered(e)
```

```
clear(Y) :-
  block(Y) & evaluate(countofall(X,on(X,Y)),0)
```

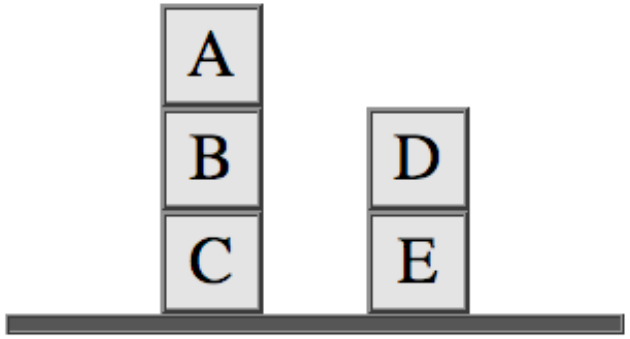
Blocks World




```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
supported(a)   table(c)
supported(b)   table(e)
supported(d)
```

Blocks World



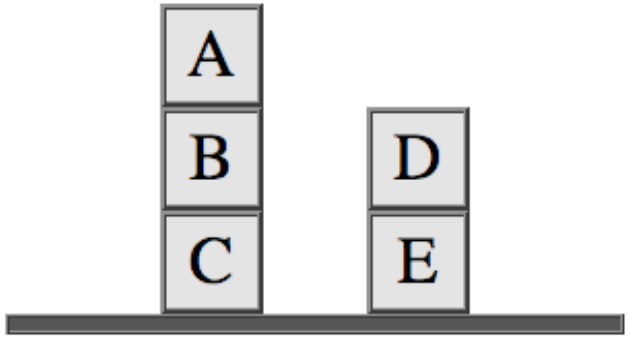

```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
supported(X) :- on(X,Y)
table(X) :- block(X) & ~supported(X)
```

```
supported(a)   table(c)
supported(b)   table(e)
supported(d)
```

```
table(X) :-
  block(X) & evaluate(countofall(Y,on(X,Y)),0)
```

Blocks World

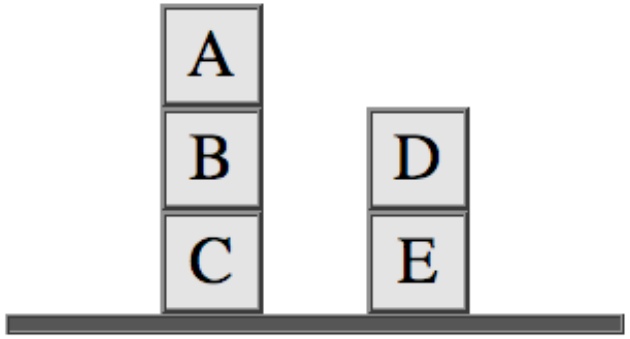



```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
stack(X,Y,Z) :- on(X,Y) & on(Y,Z)
```

```
stack(a,b,c)
```

Blocks World




```
block(a)
block(b)      on(a,b)
block(c)      on(b,c)
block(d)      on(d,e)
block(e)
```

```
above(X,Z) :- on(X,Z)
above(X,Z) :- on(X,Y) & above(Y,Z)
```

```
above(a,b)
above(b,c)
above(d,e)
above(a,c)
```

Boolean Logic

Metalevel Logic

Boolean Logic sentences are expressions written using propositional constants and logical operators like ¬ (not), ∧ (and), and ∨ (not).

$$(p \wedge \neg q) \vee (\neg p \wedge q)$$

Basic idea: represent sentences in Boolean Logic as Logic Programming terms and write rules to define basic properties and relationships in Boolean Logic.

NB: We can extend to defining Logic Programming within Logic Programming as well. The formalization is messier, and some nasty problems need to be handled (notably paradoxes).

Syntactic Metavocabulary

Object Constants (propositions)

`p, q, r`

Function constants / constructors

`not/1`
`and/2`
`or/2`

$(p \wedge \neg q) \vee (\neg p \wedge q)$

is represented as

`or(and(p,not(q)),and(not(p),q))`

Sentences in Boolean logic are represented by ***terms*** in Epilog. Using these terms, we can write sentences in Epilog *about* sentences in Boolean Logic.

Syntactic Metavocabulary

Object Constants (propositions)

`p, q, r`

Function constants

`not/1`
`and/2`
`or/2`

Unary Relation Constants

`proposition/1`
`negation/1`
`conjunction/1`
`disjunction/1`
`sentence/1`

Example: `negation(not(p))`

Syntactic Metadefinitions

```
proposition(p)
proposition(q)
proposition(r)

negation(not(X)) :- sentence(X)

conjunction(and(X,Y)) :-
  sentence(X) & sentence(Y)

disjunction(or(X,Y)) :-
  sentence(X) & sentence(Y)
```

Syntactic Metadefinitions

```
proposition(p)
proposition(q)
proposition(r)

negation(not(X)) :- sentence(X)

conjunction(and(X,Y)) :-
  sentence(X) & sentence(Y)

disjunction(or(X,Y)) :-
  sentence(X) & sentence(Y)

sentence(X) :- proposition(X)
sentence(X) :- negation(X)
sentence(X) :- conjunction(X)
sentence(X) :- disjunction(X)
```

Examples

```
sentence(p)
sentence(q)
sentence(r)

negation(not(p))
conjunction(and(q,r))
disjunction(or(q,r))

sentence(not(p))
sentence(and(q,r))
sentence(or(q,r))
```

Examples

```
sentence(p)
sentence(q)
sentence(r)

sentence(not(p))
sentence(and(q,r))
sentence(or(q,r))

conjunction(and(p,and(q,r)))
conjunction(and(p,not(p)))
disjunction(or(q,and(q,r)))
disjunction(or(p,not(p)))

sentence(and(p,and(q,r)))
sentence(and(p,not(p)))
sentence(or(q,and(q,r)))
sentence(or(p,not(p)))
```

Semantic Metavocabulary

Binary Relation Constant: `value`

e.g. `value(p,true)`

e.g. `value(q,false)`

Unary Relation Constant: `istrue`

e.g. `istrue(p)`

e.g. `istrue(or(p,not(p)))`

Semantic Metadefinitions

Definitions:

```
istrue(P) :- value(P,true)
istrue(not(P)) :- sentence(P) & ~istrue(P)
istrue(and(P,Q)) :- istrue(P) & istrue(Q)
istrue(or(P,Q)) :- istrue(P)
istrue(or(P,Q)) :- istrue(Q)
```

Example

Definitions:

```
istrue(P) :- value(P,true)
istrue(not(P)) :- sentence(P) & ~istrue(P)
istrue(and(P,Q)) :- istrue(P) & istrue(Q)
istrue(or(P,Q)) :- istrue(P)
istrue(or(P,Q)) :- istrue(Q)
```

Dataset:

```
value(p,true)
value(q,false)
value(r,true)
```

Example

Definitions:

```
istrue(P) :- value(P,true)
istrue(not(P)) :- sentence(P) & ~istrue(P)
istrue(and(P,Q)) :- istrue(P) & istrue(Q)
istrue(or(P,Q)) :- istrue(P)
istrue(or(P,Q)) :- istrue(Q)
```

Dataset:

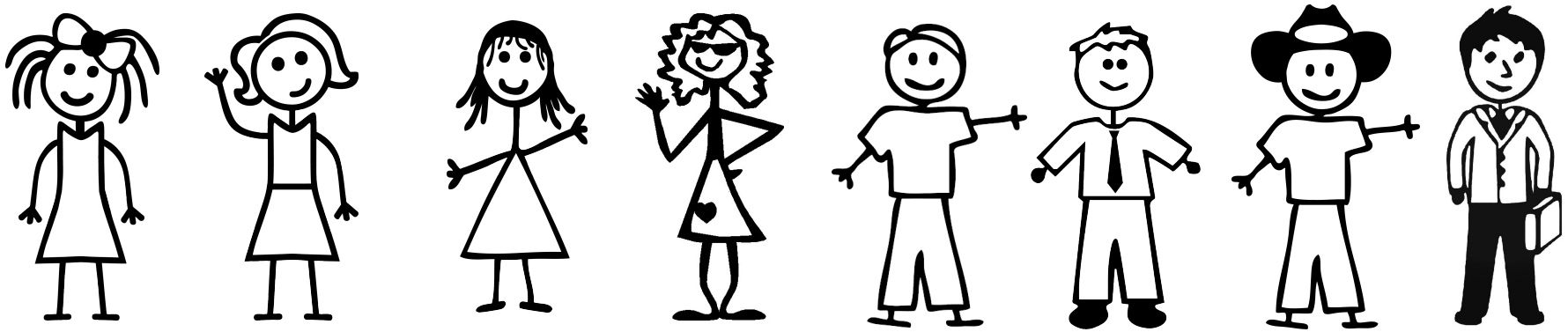
```
value(p,true)
value(q,false)
value(r,true)
```

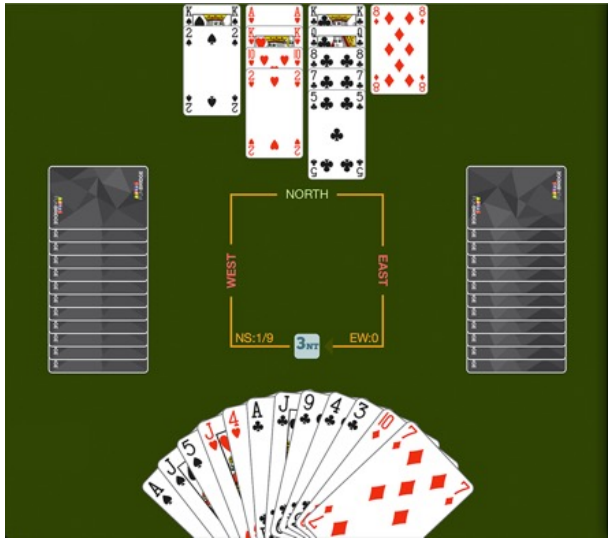
Conclusions:

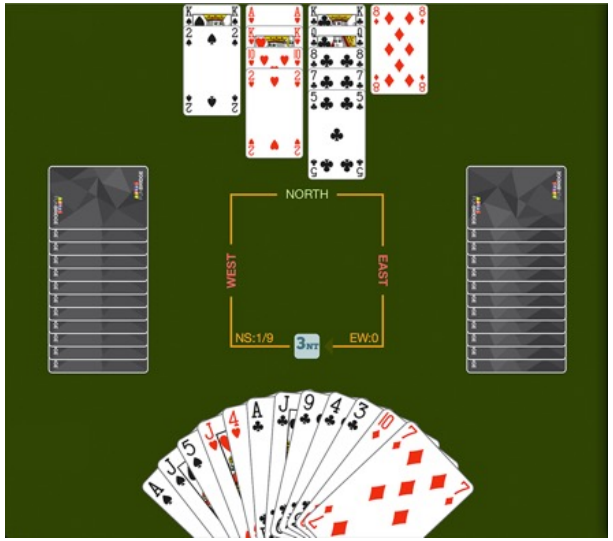
```
istrue(p)
istrue(not(q))
istrue(and(p,not(q)))
istrue(or(q,r))
istrue(or(q,not(q)))
```

Tournament

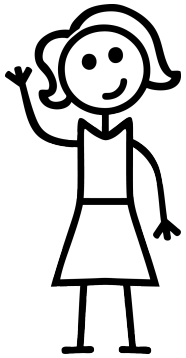
Bridge

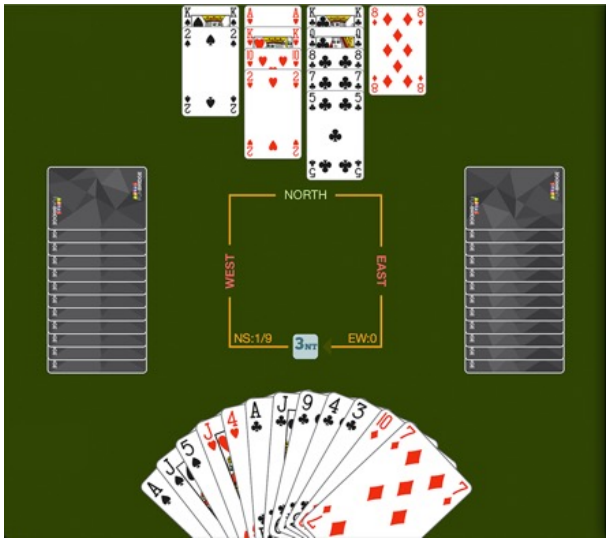


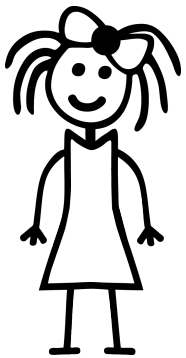




Sample Pair







Pairs

```
ispair(pair(X,Y)) :-
  player(X) & player(Y)
```

```
player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ispair(pair(1,1))
ispair(pair(1,2))
ispair(pair(1,3))
ispair(pair(1,4))
ispair(pair(2,1))
ispair(pair(2,2))
ispair(pair(2,3))
ispair(pair(2,4))
   ...
ispair(pair(8,5))
ispair(pair(8,6))
ispair(pair(8,7))
ispair(pair(8,8))
```

Pairs

```
ispair(pair(X,Y)) :-
  player(X) & player(Y)
```

```
player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ispair(pair(1,1)) ?
ispair(pair(1,2))
ispair(pair(1,3))
ispair(pair(1,4))
ispair(pair(2,1))
ispair(pair(2,2)) ?
ispair(pair(2,3))
ispair(pair(2,4))
   ...
ispair(pair(8,5))
ispair(pair(8,6))
ispair(pair(8,7))
ispair(pair(8,8)) ?
```

Pairs

```
ispair(pair(X,Y)) :-
  player(X) & player(Y) & distinct(X,Y)
```

```
player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ispair(pair(1,2))
ispair(pair(1,3))
ispair(pair(1,4))
ispair(pair(2,1))
ispair(pair(2,3))
ispair(pair(2,4))
   ...
ispair(pair(8,5))
ispair(pair(8,6))
ispair(pair(8,7))
```

Pairs

```
ispair(pair(X,Y)) :-
  player(X) & player(Y) & distinct(X,Y)
```

```
player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ispair(pair(1,2)) ?
ispair(pair(1,3))
ispair(pair(1,4))
ispair(pair(2,1)) ?
ispair(pair(2,3))
ispair(pair(2,4))
   ...
ispair(pair(8,5))
ispair(pair(8,6))
ispair(pair(8,7))
```

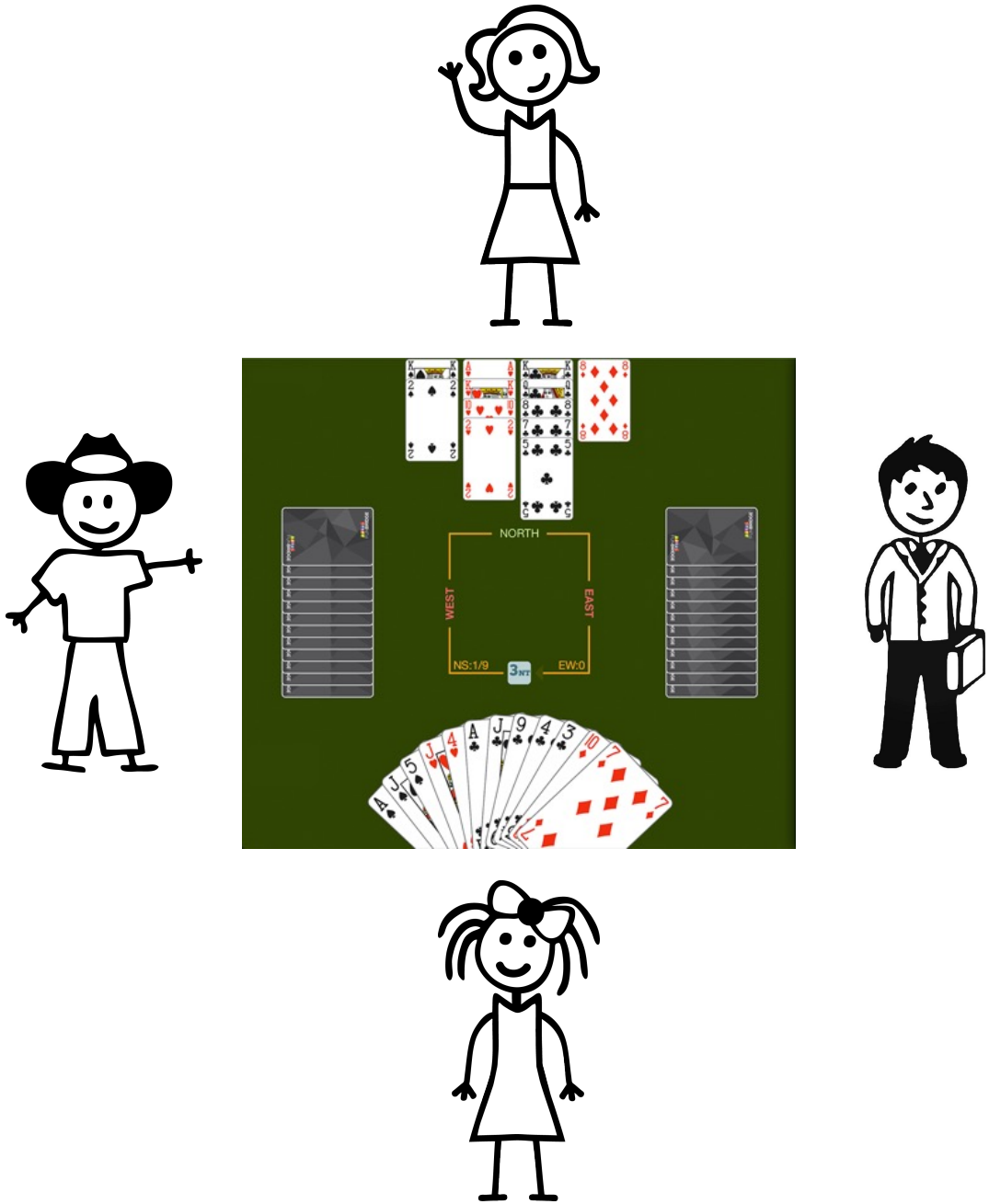
Pairs

```
ispair(pair(X,Y)) :-
  player(X) & player(Y) & distinct(X,Y) &
  leq(X,Y)
```

```
player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ispair(pair(1,2))
ispair(pair(1,3))
ispair(pair(1,4))
ispair(pair(2,3))
ispair(pair(2,4))
   ...
ispair(pair(6,7))
ispair(pair(6,8))
ispair(pair(7,8))
```

Sample Match



Matches

```
ismatch(match(P,Q)) :- ispair(P) & ispair(Q)

player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ismatch(match(pair(1,2), pair(1,2))) ?
ismatch(match(pair(1,2), pair(1,3)))
...
```

Matches

```
ismatch(match(P,Q)) :-
  ispair(P) & ispair(Q) & distinct(P,Q)

player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

ismatch(match(pair(1,2), pair(1,3))) ?

...

Matches

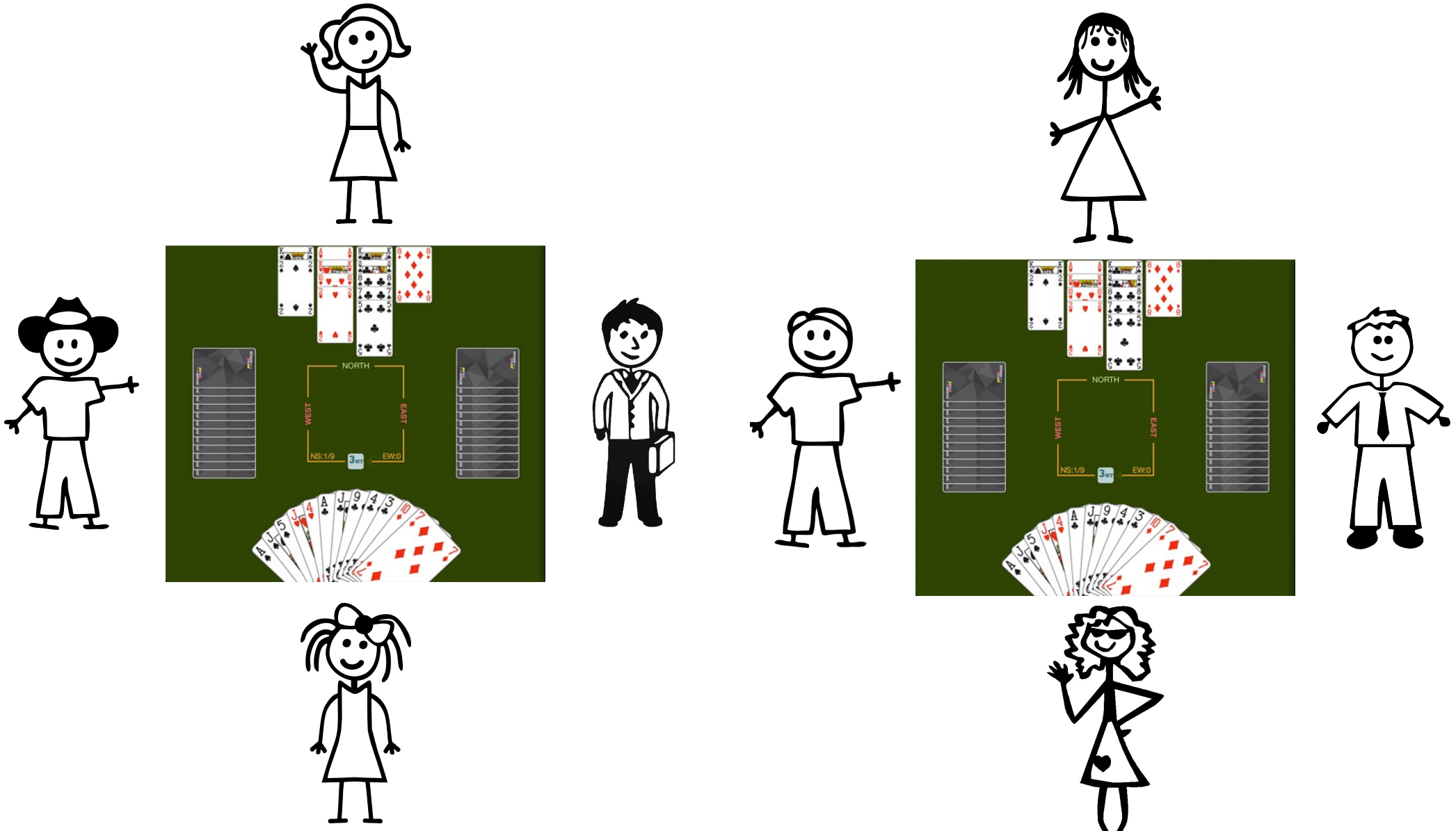
```
ismatch(match(P,Q)) :-
  ispair(P) & ispair(Q) & dispair(P,Q)

dispair(pair(U,V),pair(X,Y)) :- mutex(U,V,X,Y)

player(1)
player(2)
player(3)
player(4)
player(5)
player(6)
player(7)
player(8)
```

```
ismatch(match(pair(1,2), pair(7,8)))
...
```

Sample Round



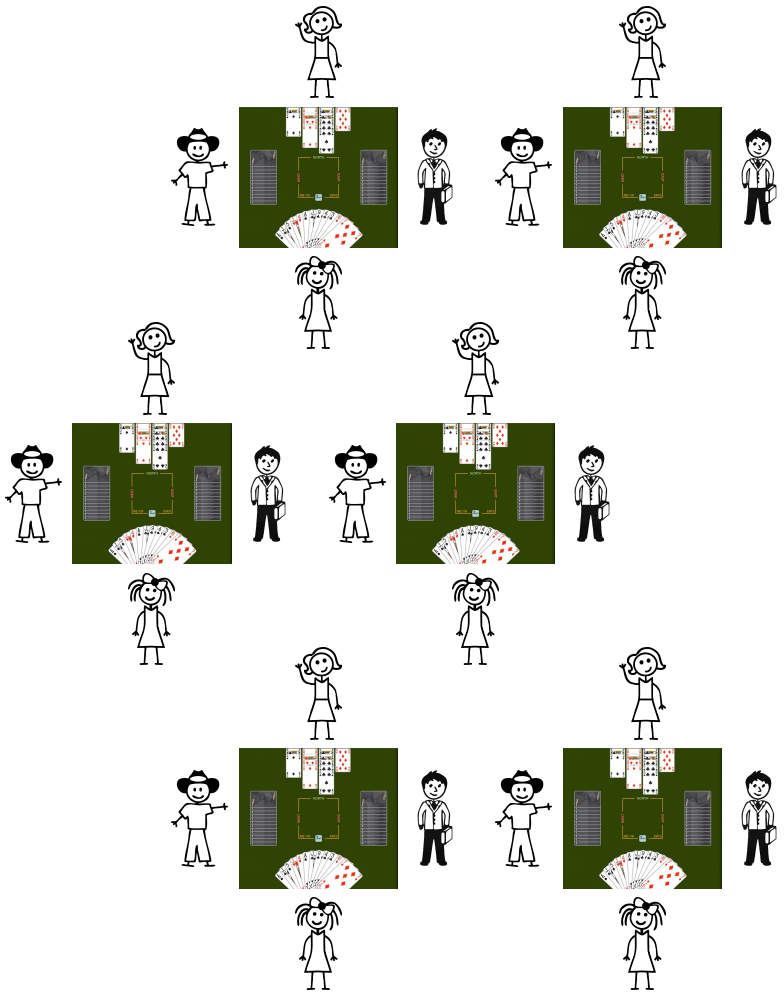
Rounds

```
isround(round(M1,M2)) :-
  ismatch(M1) & ismatch(M2) & dismatch(M1,M2)

dismatch(match(pair(X1,X2),pair(X3,X4)),
         match(pair(X5,X6),pair(X7,X8))) :-
  mutex(X1,X2,X3,X4,X5,X6,X7,X8)
```

```
isround(round(match(pair(1,2), pair(7,8)),
              match(pair(3,4), pair(5,6))))
```

Sample Schedule



Seven rounds in schedule.

Schedule

```
schedule(R1,R2,R3,R4,R5,R6,R7) :-
  isround(R1) &
  isround(R2) & diff(R2,R1) &
  isround(R3) & diff(R3,R1) & diff(R3,R2) &
  isround(R4) & diff(R4,R1) & diff(R4,R2) & diff(R4,R3) &
  isround(R5) & diff(R5,R1) & diff(R5,R2) & diff(R5,R3) &
                diff(R5,R3) & diff(R5,R4) &
  isround(R6) & diff(R6,R1) & diff(R6,R2) & diff(R6,R3) &
                diff(R6,R4) & diff(R6,R4) & diff(R6,R5) &
  isround(R7) & diff(R7,R1) & diff(R7,R2) & diff(R7,R3) &
                diff(R7,R4) & diff(R7,R4) & diff(R7,R5) &
                diff(R7,R6)

diff(round(match(P1,P2),match(P3,P4)),
     round(match(P5,P6),match(P7,P8))) :-
  mutex(P1,P2,P3,P4,P5,P6,P7,P8)
```

File Dataset Channel Ruleset Operation Settings

Library

Save Revert

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% tournament
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

schedule(R1,R2,R3,R4,R5,R6,R7) :-
  isround(R1) &
  isround(R2) & diff(R2,R1) &
  isround(R3) & diff(R3,R1) & diff(R3,R2) &
  isround(R4) & diff(R4,R1) & diff(R4,R2) & diff(R4,R3) &
  isround(R5) & diff(R5,R1) & diff(R5,R2) & diff(R5,R3) & diff(R5,R3) &
                diff(R5,R4) &
  isround(R6) & diff(R6,R1) & diff(R6,R2) & diff(R6,R3) & diff(R6,R4) &
                diff(R6,R4) & diff(R6,R5) &
  isround(R7) & diff(R7,R1) & diff(R7,R2) & diff(R7,R3) & diff(R7,R4) &
                diff(R7,R4) & diff(R7,R5) & diff(R7,R6)

isround(round(M1,M2)) :-
  ismatch(M1) &
  ismatch(M2) &
  dismatch(M1,M2)

ismatch(match(P,Q)) :-
  ispair(P) &
  ispair(Q) &
  dispair(P,Q)

diff(round(match(P1,P2),match(P3,P4)),round(match(P5,P6),match(P7,P8))) :-
  mutex(P1,P2,P3,P4,P5,P6,P7,P8)

ispair(pair(X,Y)) :-
  player(X) &
  player(Y) &
  distinct(X,Y) &
  evaluate(min(X,Y),X)

dismatch(match(pair(X1,X2),pair(X3,X4)),match(pair(X5,X6),pair(X7,X8))) :-
  mutex(X1,X2,X3,X4,X5,X6,X7,X8)

dispair(pair(U,V),pair(X,Y)) :-
  distinct(U,X) &
  distinct(U,Y) &
  distinct(V,X) &
  distinct(V,Y)

player(1)
player(2)
player(3)
player(4)
player(5)
```

Not Secure — epilog.stanford.edu

File Dataset Channel Ruleset Operation Settings

Library

Save Revert

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% tournament
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

schedule(R1,R2,R3,R4,R5,R6,R7) :-
  isround(R1) &
  isround(R2) & diff(R2,R1) &
  isround(R3) & diff(R3,R1) & diff(R3,R2) &
  isround(R4) & diff(R4,R1) & diff(R4,R2) & diff(R4,R3) &
  isround(R5) & diff(R5,R1) & diff(R5,R2) & diff(R5,R3) & diff(R5,R3) &
                diff(R5,R4) &
  isround(R6) & diff(R6,R1) & diff(R6,R2) & diff(R6,R3) & diff(R6,R4) &
                diff(R6,R4) & diff(R6,R5) &
  isround(R7) & diff(R7,R1) & diff(R7,R2) & diff(R7,R3) & diff(R7,R4) &
                diff(R7,R4) & diff(R7,R5) & diff(R7,R6)

isround(round(M1,M2)) :-
  ismatch(M1) &
  ismatch(M2) &
  dismatch(M1,M2)

ismatch(match(P,Q)) :-
  ispair(P) &
  ispair(Q) &
  dispair(P,Q)

diff(round(match(P1,P2),match(P3,P4)),round(match(P5,P6),match(P7,P8))) :-
  mutex(P1,P2,P3,P4,P5,P6,P7,P8)

ispair(pair(X,Y)) :-
  player(X) &
  player(Y) &
  distinct(X,Y) &
  evaluate(min(X,Y),X)

dismatch(match(pair(X1,X2),pair(X3,X4)),match(pair(X5,X6),pair(X7,X8))) :-
  mutex(X1,X2,X3,X4,X5,X6,X7,X8)

dispair(pair(U,V),pair(X,Y)) :-
  distinct(U,X) &
  distinct(U,Y) &
  distinct(V,X) &
  distinct(V,Y)

player(1)
player(2)
player(3)
player(4)
player(5)
```

Compute

Query `schedule(R1,R2,R3,R4,R5,R6,R7)`

Show Next 1 result(s) Autorefresh

1108436 unification(s)

```
schedule(round(match(pair(1,2),pair(3,4)),match(pair(5,6),pair(7,8))),round(match(pai
r(1,3),pair(2,4)),match(pair(5,7),pair(6,8))),round(match(pair(1,4),pair(2,3)),match(
pair(5,8),pair(6,7))),round(match(pair(1,5),pair(2,6)),match(pair(3,7),pair(4,8))),ro
und(match(pair(1,6),pair(2,5)),match(pair(3,8),pair(4,7))),round(match(pair(1,7),pair
(2,8)),match(pair(3,5),pair(4,6))),round(match(pair(1,8),pair(2,7)),match(pair(3,6),p
air(4,5))))
```