Computational Law
Dynamics

Michael Genesereth
Computer Science Department
Stanford University

Operations

Queries and View Definitions

```
goal(X,Y) :- p(X,Y) & ~q(Y)

r(X,Y) :- p(X,Y) & ~q(Y)
s(X,Y) :- r(X,Y) & r(Y,Z)
```

Updates and Operation Definitions

```
p(X) & ~q(X) ==> ~p(X) & q(X)

flip(X) :: p(X) & ~q(X) ==> ~p(X) & q(X)
flop(X) :: r(X,Y) ==> flip(X) & flop(Y)
```

Syntax

Operation Constants

Operation constants represent operations.

`tick` - tick of the clock

`click` - click a button on a web page

`stack` - place one block on another

`mark` - place a specific mark in a row and a column

Same spelling conventions as other constants.

Like constructors, and predicates, each has a specific arity.

```
tick/0
click/1
stack/2
mark/3
```

Actions

An **action** is an application of an operation to objects.

In what follows, we denote actions using a syntax similar to that of compound terms, viz. an *n*-ary operation constant followed by *n* terms enclosed in parentheses (as appropriate) and separated by commas.

Examples:

```
tick
click(a)
stack(a,b)
mark(x,2,3)
```

Syntactically, actions are treated as terms.

Operation Definition

```
c(a) :: p(a,b) & q(a) ==> ~q(a) & c(b)
```

head *(action)*

conditions *(ordinary literals)*

effects *(base literals or actions)*

Variables

```
c(X) :: p(X,Y) & q(X) ==> ~q(X) & c(Y)
```

Degenerate Rules

Degenerate Rule

```
c(X) :: true ==> ~p(X) & q(X)
```

Shorthand

```
c(X) :: ~p(X) & q(X)
```

Dynamic Logic Programs

An *operation definition* is a finite collection of operation rules with the same operation in the head.

Example

```
c(X) :: p(X) & q(X)
c(X) :: ~r(X) ==> ~p(X) & r(X)
```

A *dynamic logic program* is a collection of view definitions and operation definitions.

Safety

A operation rule is **safe** if and only if every variable in every literal on the right hand side appears in a positive literal on the left hand side. Also, every variable in a negative literal on the left hand side appears in a prior positive literal.

Safe Operation Rule

```
c(X) ::
  p(X,Y) & ~q(X) ==>
  ~p(X,Y) & q(X) & c(Y)
```

Unsafe Operation Rule

```
c(X) ::
  p(X,Y) & ~q(Z) ==>
  ~p(X,Y) & q(W) & c(Y)
```

Semantics

Intuition

Given a rule set, the result of applying an action to a dataset is the dataset that results from *deleting all of the negative effects* of the action from the dataset and *adding in all of the positive effects*.

Example

Dataset: `{p(a),p(b),p(c),q(a),q(b),q(c),r(b)}`

Ruleset:

```
s(X) :- p(X) & q(X)
u(X) :: s(X) & ~r(X) ==> ~p(X) & r(X)
```

Actionset: `{u(a),u(b)}`

Extension: `{p(a),p(b),p(c),q(a),q(b),q(c),r(b),`
`s(a),s(b),s(c)}`

Active Instances of operation rule:

```
u(a) :: s(a) & ~r(a) ==> ~p(a) & r(a)
```

Inactive Instance:

```
u(b) :: s(b) & ~r(b) ==> ~p(b) & r(b)
u(c) :: s(a) & ~r(c) ==> ~p(c) & r(c)
```

Example

Dataset: `{p(a),p(b),p(c),q(a),q(b),q(c),r(b)}`

Ruleset:

```
s(X) :- p(X) & q(X)
u(X) :: s(X) & ~r(X) ==> ~p(X) & r(X)
```

Actionset: `{u(a),u(b)}`

Extension: `{p(a),p(b),p(c),q(a),q(b),q(c),r(b),`
`s(a),s(b),s(c)}`

Active Instances of operation rule:

```
u(a) :: s(a) & ~r(a) ==> ~p(a) & r(a)
```

Inactive Instance:

```
u(b) :: s(b) & ~r(b) ==> ~p(b) & r(b)
u(c) :: s(a) & ~r(c) ==> ~p(c) & r(c)
```

Example

Dataset: `{p(a),p(b),p(c),q(a),q(b),q(c),r(b)}`

Ruleset:

```
s(X) :- p(X) & q(X)
u(X) :: s(X) & ~r(X) ==> ~p(X) & r(X)
```

Actionset: `{u(a),u(b)}`

Active Instances of operation rules:

```
u(a) :: s(a) & ~r(a) ==> ~p(a) & r(a)
```

Expansion: `{~p(a), r(a)}`
Negative Updates: `{p(a)}`
Positive Updates: `{r(a)}`

Example

Dataset: {`p(a),p(b),p(c),q(a),q(b),q(c),r(b)`}

Ruleset:

```
s(X) :- p(X) & q(X)
u(X) :: s(X) & ~r(X) ==> ~p(X) & r(X)
```

Actionset: {`u(a),u(b)`}

Negative Updates: {`p(a)`}
Positive Updates: {`r(a)`}

Result: {`p(b),p(c),q(a),q(b),q(c),r(a),r(b)`}

Production Systems

A **production system** is a set of condition-action rules. On each step in the execution of a production system, an active rule is chosen and the actions are performed. The cycle then repeats on the new state.

```
if p(X), then del p(X) and add q(X)
if q(X), then del q(X) and add p(X)
```

Before: {`p(a),q(b)`}
Step 1: {`q(a),q(b)`}
Step 2: {`p(a),q(b)`} *or* {`p(b),q(a)`}

When do we stop?

Dynamic Logic Programs

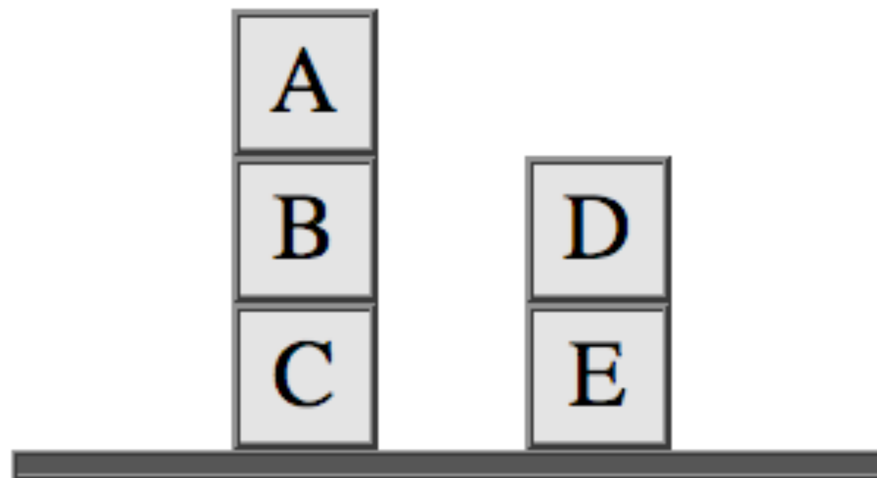
Dynamic logic programs differ from production systems in that all active transition rules “fire” at the same time. (1) All updates are computed *before* any changes are made, and (2) all changes are made simultaneously.

```
tick :: p(X) ==> ~p(X) & q(X)
tick :: q(X) ==> ~q(X) & p(X)
```

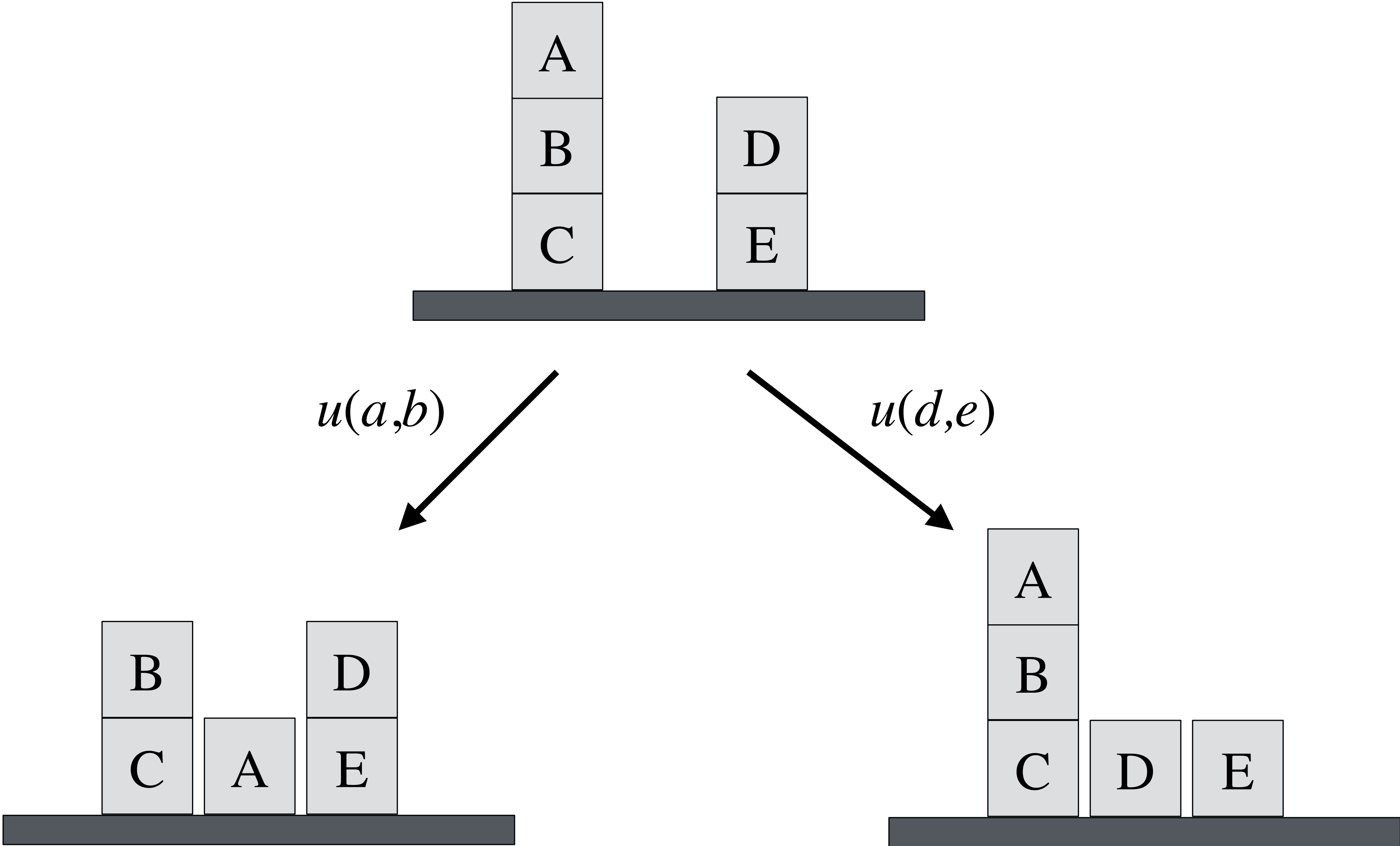
Before: {`p(a), q(b)`}
After: {`p(b), q(a)`}

Blocks World

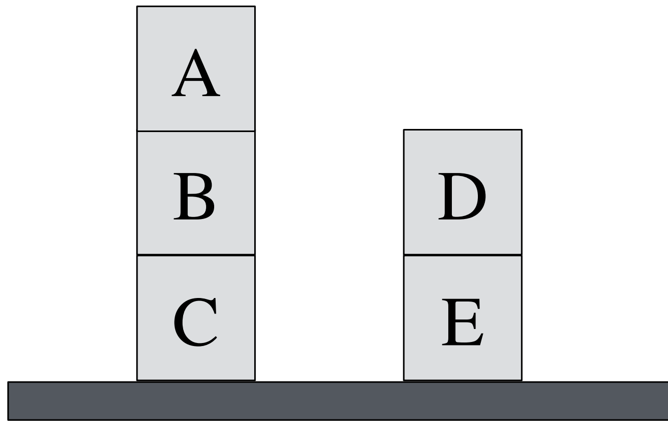
Blocks World



External Actions

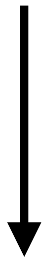


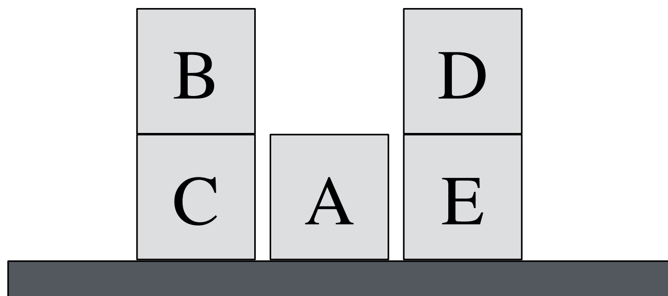
Describing States




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

u(a,b)






```
clear(a)
table(a)
clear(b)
on(b,c)
on(d,e)
  ...
```

Operation Definitions

Operations:

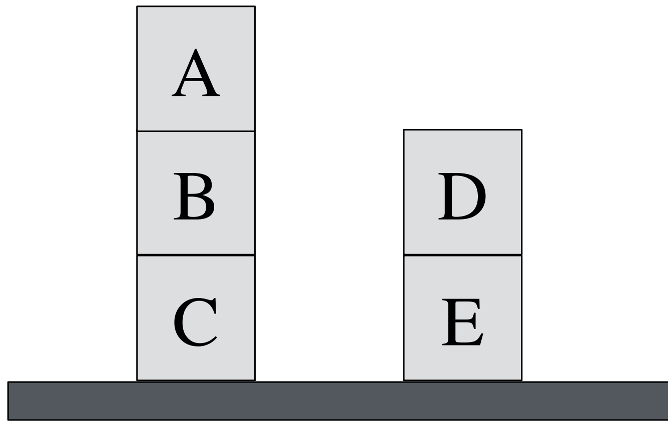
`u(x,y)` means that x is moved from y to the table.

`s(x,y)` means that x is moved from the table to y.

Operation Definitions:

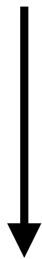
```
u(X,Y) ::
  clear(X) & on(X,Y)
  ==> ~on(X,Y) & table(X) & clear(Y)

s(X,Y) ::
  table(X) & clear(X) & clear(Y)
  ==> ~table(X) & ~clear(Y) & on(X,Y)
```

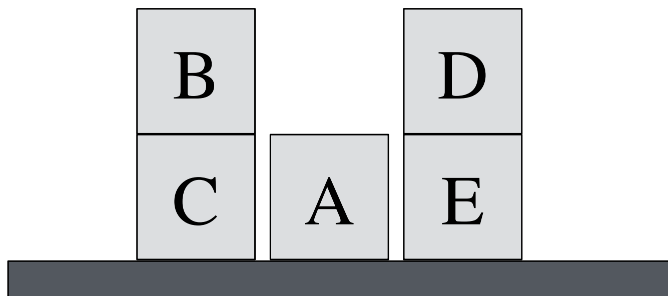
Describing States




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

u(a,b)






```
clear(a)
table(a)
clear(b)
on(b,c)
on(d,e)
  ...
```

The Game of Life

World

Rules of the Game

(1) Any *live* cell with *two or three* live neighbors lives on to the next generation.

(2) Any *live* cell with *fewer than two* live neighbors dies (as if caused by underpopulation).

(3) Any *live* cell with *more than three* live neighbors dies (as if by overpopulation).

(4) Any *dead* cell with *exactly three* live neighbors becomes a live cell (as if by reproduction).

Vocabulary

Symbols: `c11, c12, …`

Unary Predicates:

`on` - cell is live

`cell` - true of cells

Binary Predicates:

`neighbor` - cells are neighbors

Starvation

Any *live* cell with *fewer than two* live neighbors dies.

```
tick ::
 on(Y) & evaluate(countofall(X,neighbor(X,Y)&on(X)),0)
 ==> ~on(Y)

tick ::
 on(Y) & evaluate(countofall(X,neighbor(X,Y)&on(X)),1)
 ==> ~on(Y)
```

Overcrowding

Any *live* cell with *more than three* live neighbors dies.

```
tick ::
  on(Y) &
  evaluate(min(countofall(X,neighbor(X,Y)&on(X)),4),4)
  ==> ~on(Y)
```

Transition Rules

Any *dead* cell with *exactly three* live neighbors becomes live.

```
tick ::
  cell(Y) & ~on(Y) &
  evaluate(countofall(X,neighbor(X,Y)&on(X)),3)
  ==> on(Y)
```

Example

http://logicprogramming.stanford.edu/examples/gameoflife.html

Tic Tac Toe

States

X		
	O	
		X

```
cell(1,1,x)
cell(1,2,b)
cell(1,3,b)
cell(2,1,b)
cell(2,2,o)
cell(2,3,b)
cell(3,1,b)
cell(3,2,b)
cell(3,3,x)
control(o)
```

Legal Moves

```
legal(M,N) :- cell(M,N,b)
```

State:

```
cell(1,1,x)
cell(1,2,b)
cell(1,3,b)
cell(2,1,b)
cell(2,2,o)
cell(2,3,b)
cell(3,1,b)
cell(3,2,b)
cell(3,3,x)
control(o)
```

X		
	O	
		X

Legal Moves:

```
mark(1,2)
mark(1,3)
mark(2,1)
mark(2,3)
mark(3,1)
mark(3,2)
```

Actions

```
mark(M,N) ::
   control(Z) ==> ~cell(M,N,b) & cell(M,N,Z)
mark(M,N) ::
   control(x) ==> ~control(x) & control(o)
mark(M,N) ::
   control(o) ==> ~control(o) & control(x)
```

Before	Action	After
cell(1,1,x) cell(1,2,b) cell(1,3,b) cell(2,1,b) cell(2,2,o) cell(2,3,b) cell(3,1,b) cell(3,2,b) cell(3,3,x) control(o)	mark(1,3) →	cell(1,1,x) cell(1,2,b) cell(1,3,o) cell(2,1,b) cell(2,2,o) cell(2,3,b) cell(3,1,b) cell(3,2,b) cell(3,3,x) control(x)

Supporting Concepts

```
row(M,Z) :- cell(M,1,Z) & cell(M,2,Z) & cell(M,3,Z)
col(M,Z) :- cell(1,N,Z) & cell(2,N,Z) & cell(3,N,Z)
diag(Z) :- cell(1,1,Z) & cell(2,2,Z) & cell(3,3,Z)
diag(Z) :- cell(1,3,Z) & cell(2,2,Z) & cell(3,1,Z)
```

```
line(Z) :- row(M,Z)
line(Z) :- col(M,Z)
line(Z) :- diag(Z)
```

Goals and Termination

```
win(x) :- line(x)
win(o) :- line(o)


terminal :- win(Z)
terminal :-
  evaluate(countofall([M,N],cell(M,N,b)),0)
```

Sierra

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

```
edge(a,b)
edge(b,d)
edge(b,e)
```

Execute

Action

Expand Expand on update

Execute Run on clock tick

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

```
edge(a,b)
edge(b,d)
edge(b,e)
```

Execute

Action copy(b,c)

Expand Expand on update

Execute Run on clock tick

```
edge(c,d)
edge(c,e)
```

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Display a menu

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

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

Execute

Action copy(b,c)

Expand ☐ Expand on update

Execute ☐ Run on clock tick

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Display a menu

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

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

Execute

Action invert(c)

Expand Expand on update

Execute Run on clock tick

```
~edge(c,d)
edge(d,c)
~edge(c,e)
edge(e,c)
```

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Display a menu

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

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

Execute

Action invert(c)

Expand ☐ Expand on update

Execute ☐ Run on clock tick

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Display a menu

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

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

Execute

Action insert(w,b)

Expand ☐ Expand on update

Execute ☐ Run on clock tick

```
edge(w,b)
edge(w,d)
edge(w,c)
edge(w,e)
```

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Display a menu

Not Secure — logicprogramming.stanford.edu

File Dataset Channel Ruleset Operation Settings

Lambda

Save Revert Sort

```
edge(a,b)
edge(b,d)
edge(b,e)
edge(d,c)
edge(e,c)
edge(w,b)
edge(w,d)
edge(w,c)
edge(w,e)
```

Execute

Action insert(w,b)

Expand ☐ Expand on update

Execute ☐ Run on clock tick

Library

Save Revert

```
copy(X,Y) :: edge(X,Z) ==> edge(Y,Z)

invert(X) :: edge(X,Y) ==> ~edge(X,Y) & edge(Y,X)

insert(X,Y) :: edge(X,Y)
insert(X,Y) :: edge(Y,Z) ==> insert(X,Z)
```

Display a menu

Assignments

Readings

Reading 4.1 - Dynamics

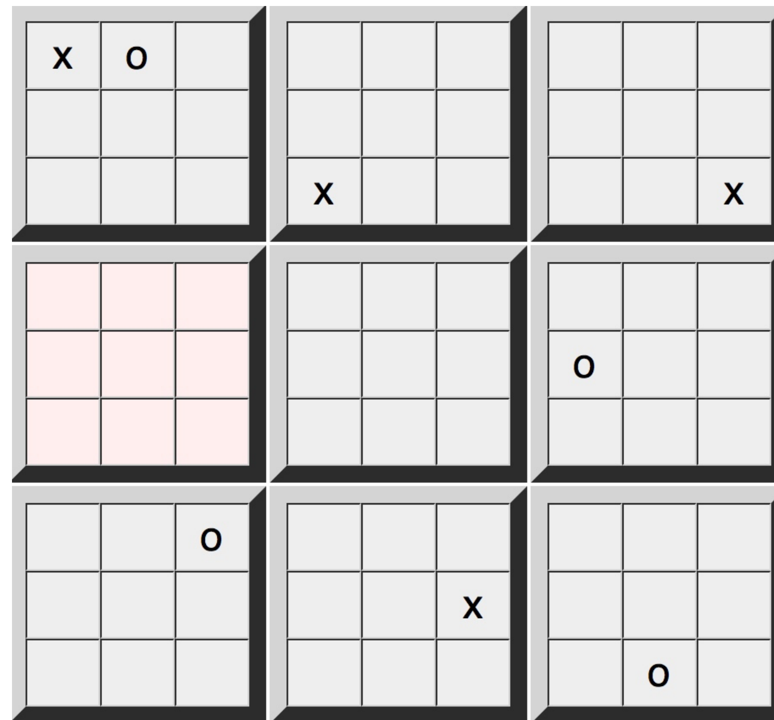
Assignment - Sierra

The goal of this exercise is for you to familiarize yourself with the Sierra capabilities for editing and using action definitions. Go to http://epilog.stanford.edu and click on the Sierra link.

In a separate window, open the documentation for Sierra. To access the documentation, go to http://epilog.stanford.edu, click on Documentation, and then click on the Sierra item on the resulting drop-down menu.

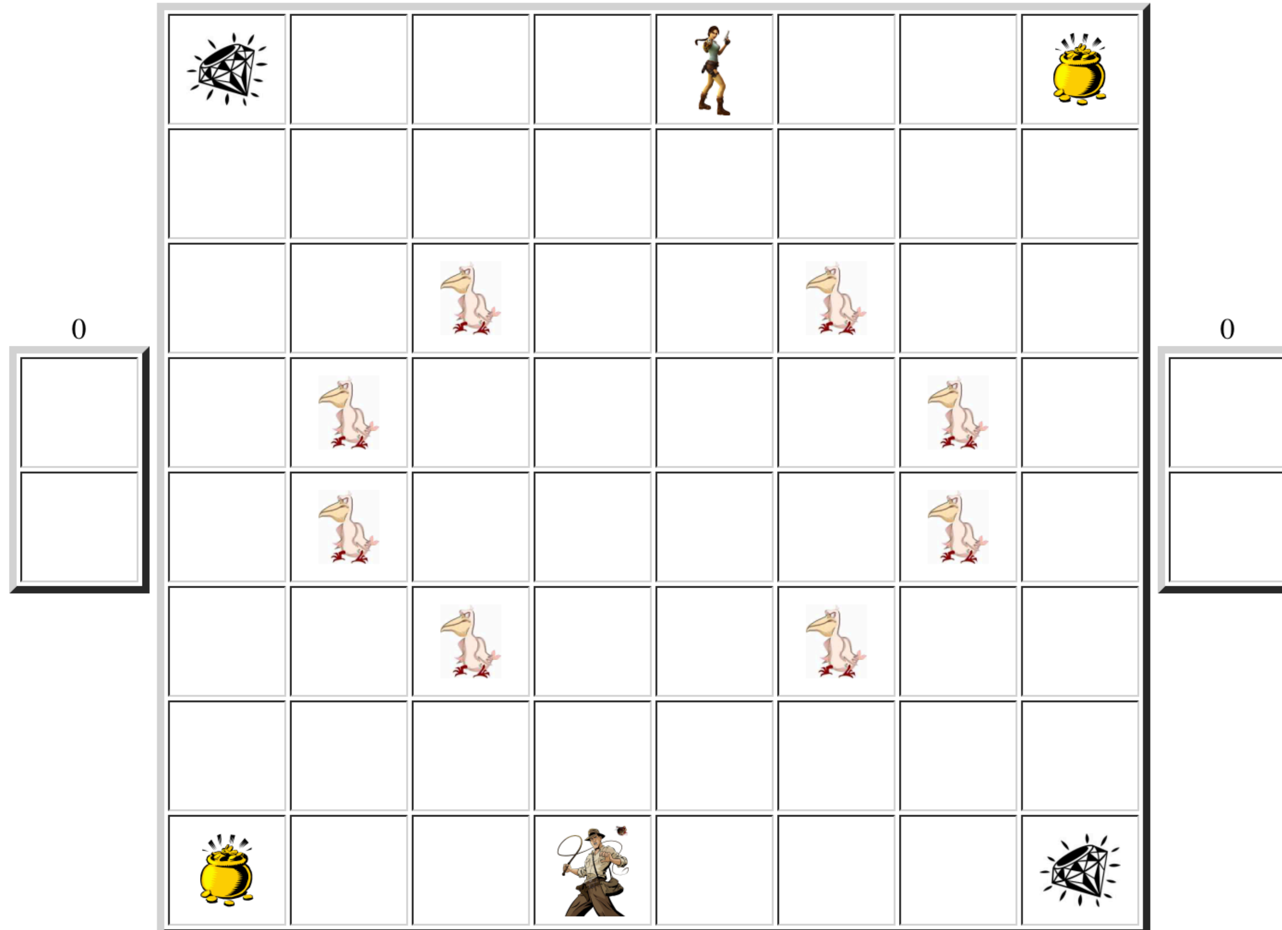
Read though section 7 of the documentation and reproduce the examples in the Sierra window you opened earlier. Once you have done this, experiment on your own. Try different data and different actions.

Assignment - Nineboard Tic Tac Toe



http://complaw.stanford.edu/chapters/nineboard.html

Pelican Hunters



http://complaw.stanford.edu/assignments/pelican_basic.html

CODEX

The Stanford Center for Legal Informatics

Legal Empowerment through Information Technology

CODEX

The Stanford Center for Legal Informatics

Legal Empowerment through Information Technology