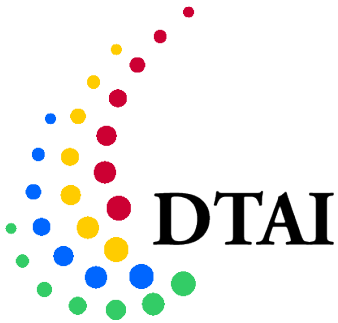
Constraint Handling Rules

A Tutorial for (Prolog) Programmers

Tom Schrijvers

Katholieke Universiteit Leuven, Belgium
Tom.Schrijvers@cs.kuleuven.be

ICLP 2008 – December 9-13, 2008



About This Tutorial

Expected background:

- know & like programming

About This Tutorial

Expected background:

- know & like programming
- know Prolog, ...

About This Tutorial

Expected background:

- know & like programming
- know Prolog, ...
- but have an itch that Prolog doesn't scratch

About This Tutorial

Expected background:

- know & like programming
- know Prolog, ...
- but have an itch that Prolog doesn't scratch
- ...or you will have at the end of this tutorial

About This Tutorial

Expected mindset:

- forget about Prolog for now

About This Tutorial

Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog

About This Tutorial

Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
- really, they won't be the same!!!

About This Tutorial

Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
- really, they won't be the same!!!

You've been warned!

About This Tutorial

Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
- really, they won't be the same!!!

You've been warned! Also:

- do not expect anything from the word "constraint"

About This Tutorial

Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
- really, they won't be the same!!!

You've been warned! Also:

- do not expect anything from the word "constraint"
- CHR constraints aren't

About this tutorial

What you'll learn:

About this tutorial

What you'll learn:

1. how CHR works (operationally)

About this tutorial

What you'll learn:

1. how CHR works (operationally)
2. how to program in CHR

About this tutorial

What you'll learn:

1. how CHR works (operationally)
2. how to program in CHR
3. just maybe, what CHR is good for

Art 101: Mixing Paint

As novice, you have to start at the bottom of the ladder:

Art 101: Mixing Paint

As novice, you have to start at the bottom of the ladder:

- Learning how to obtain different colors

Art 101: Mixing Paint

As novice, you have to start at the bottom of the ladder:

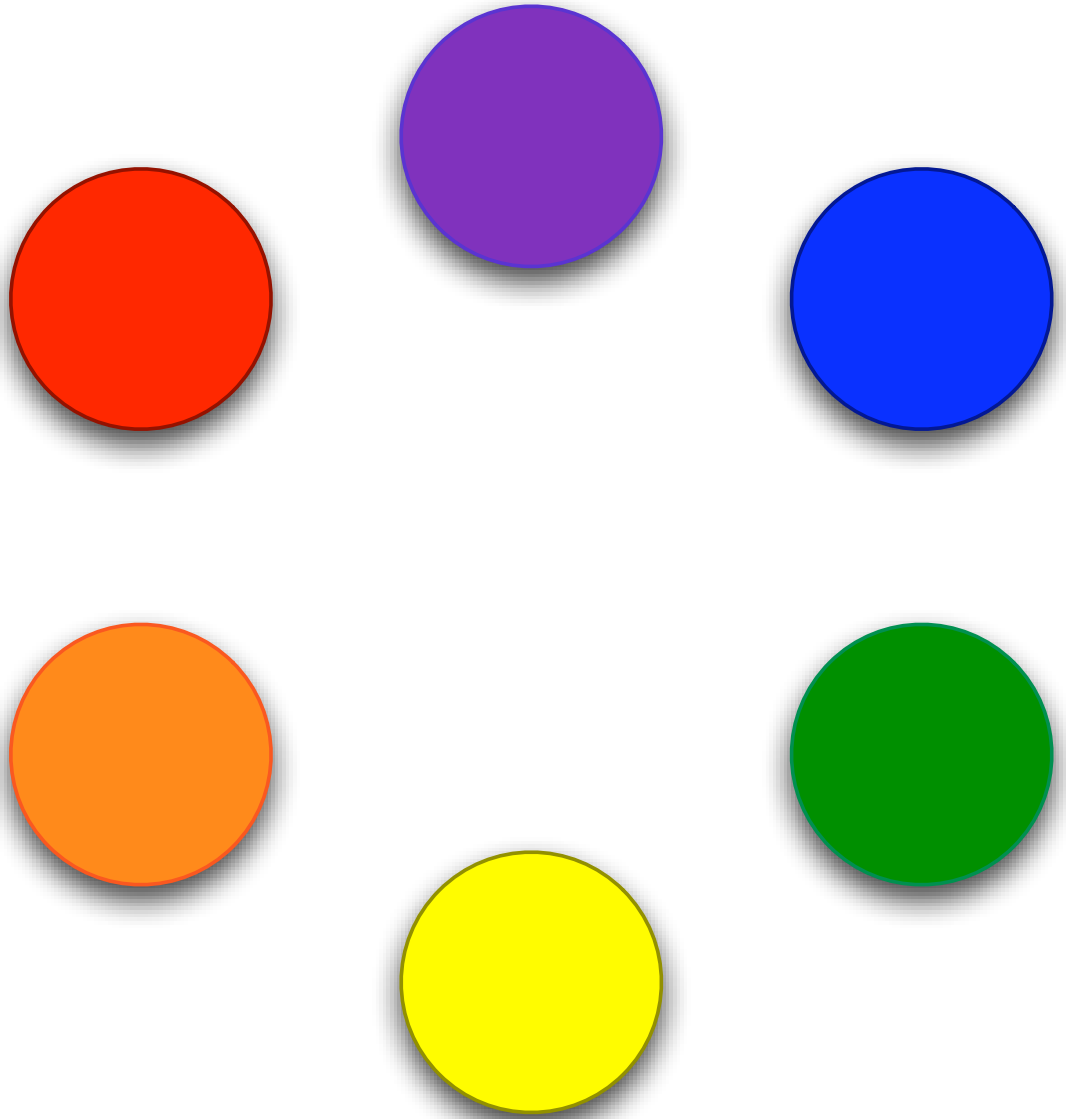
- Learning how to obtain different colors
- by mixing paint

Art 101: Mixing Paint

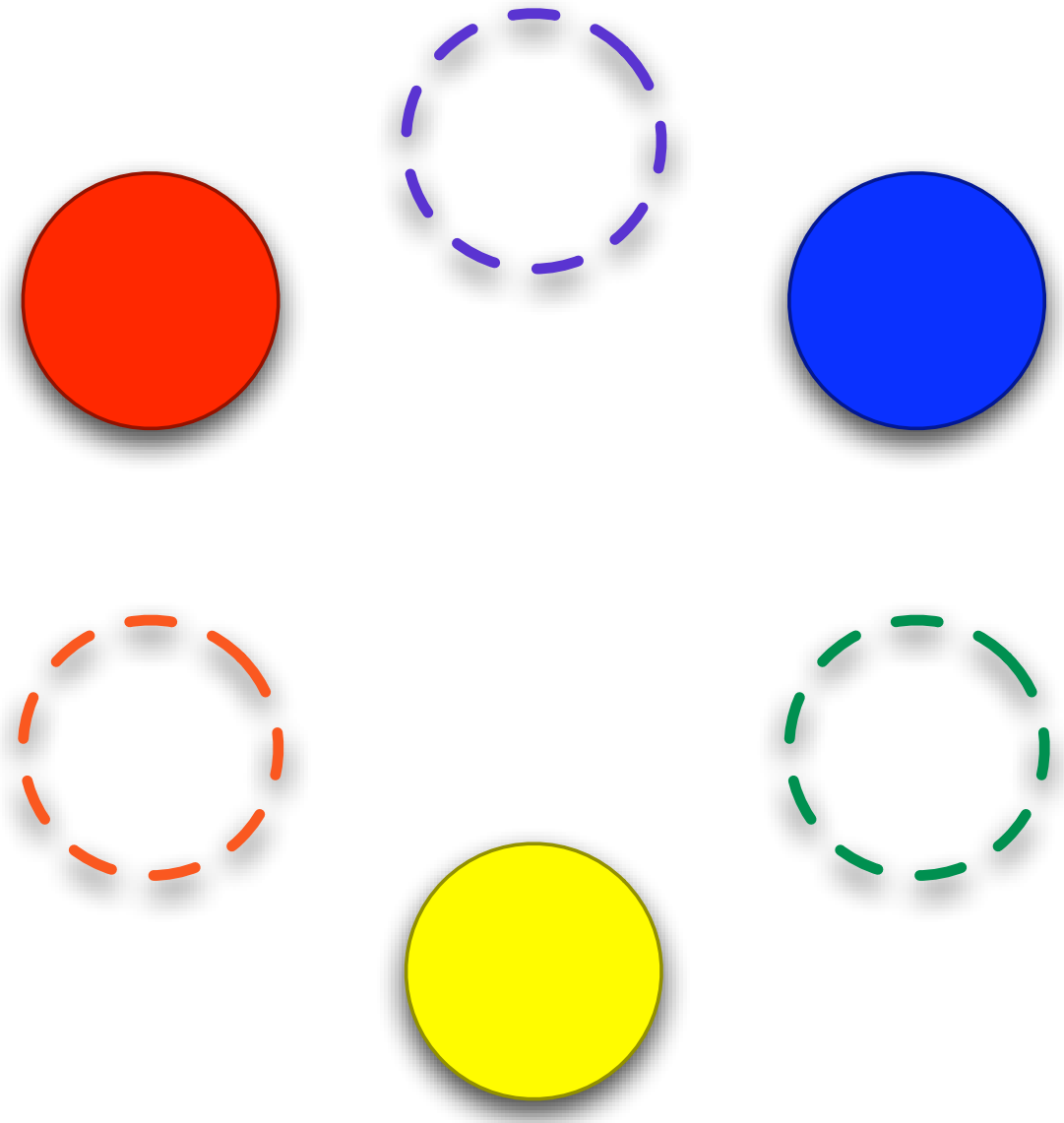
As novice, you have to start at the bottom of the ladder:

- Learning how to obtain different colors
- by mixing paint
- Made easy by the Paint Mixing Rules (PMR) language

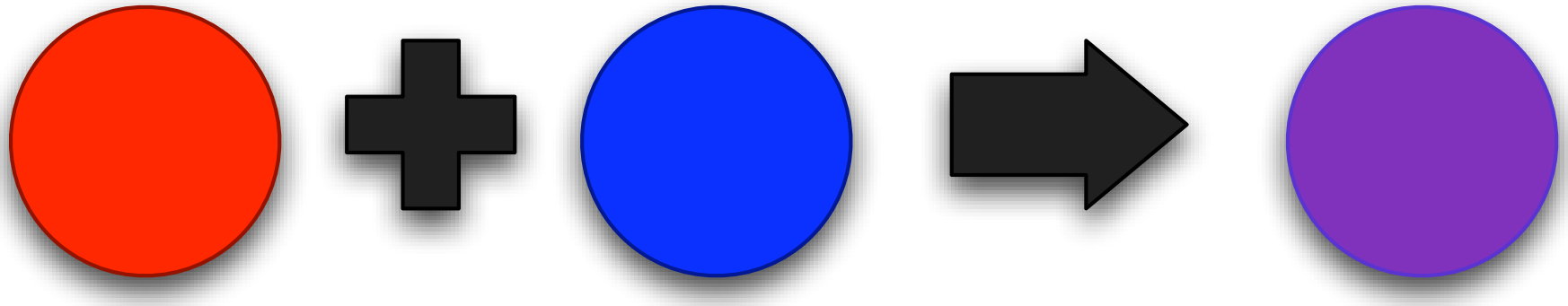
The Paint



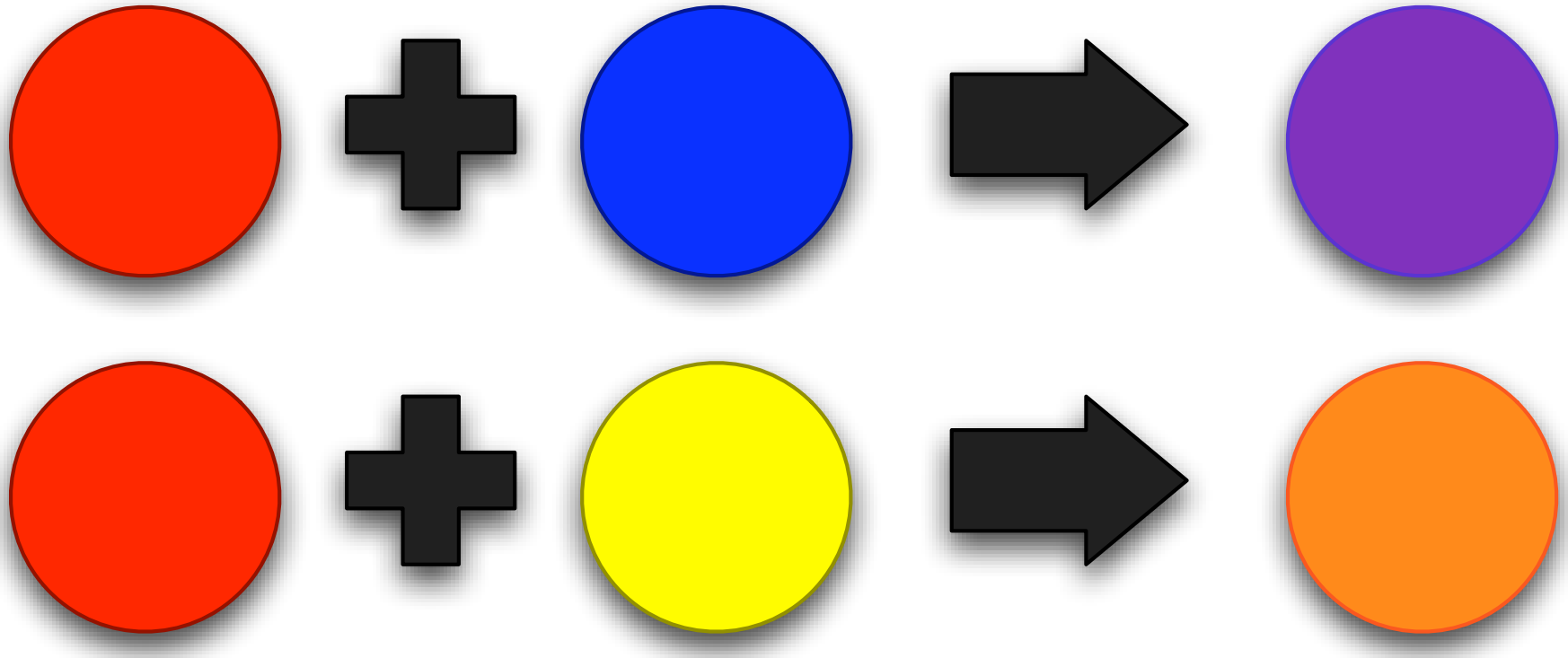
Fewer Paint



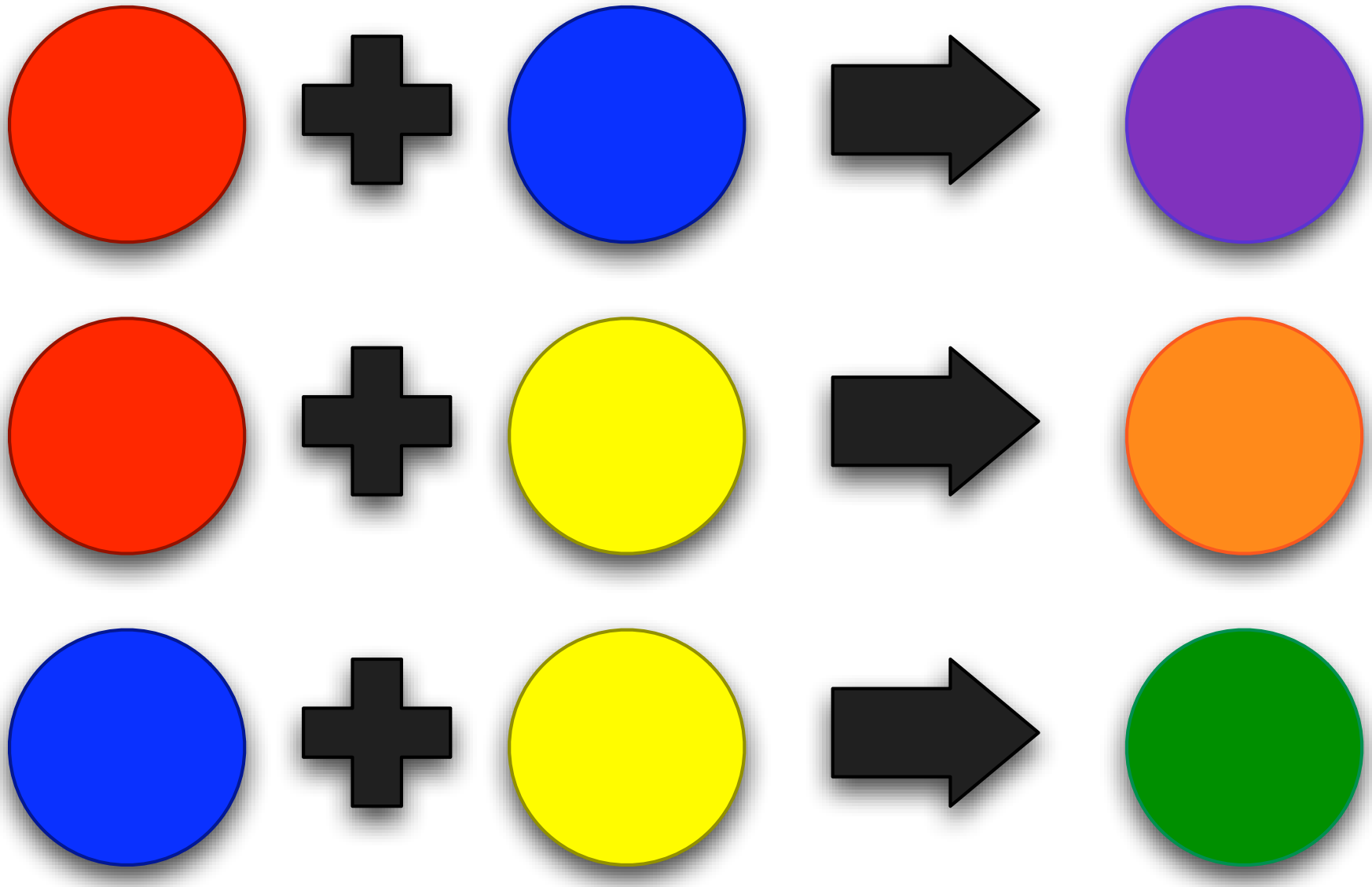
The Rules



The Rules



The Rules



PMR as CHR

Under the hood, PMR is of course CHR:

PMR	**CHR**
paints	constraints
rules	simplification rules (textual)
mixing bucket	constraint store

Paint as Constraints

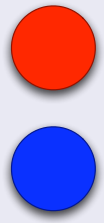
Declaring our paint colors, in textual form:

```
:- chr_constraint red.                    %
```

Paint as Constraints

Declaring our paint colors, in textual form:

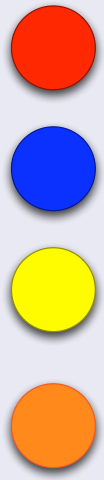
```
:- chr_constraint red.                    %
:- chr_constraint blue.                   %
```



Paint as Constraints

Declaring our paint colors, in textual form:

```
:- chr_constraint red.                    %
:- chr_constraint blue.                   %
:- chr_constraint yellow.                 %
:- chr_constraint orange.                 %

...
```



Paint as Constraints?

Are paint colors really “constraints”?

Paint as Constraints?

Are paint colors really "constraints"?

No: not in the Constraint Programming sense

Paint as Constraints?

Are paint colors really "constraints"?

No: not in the Constraint Programming sense

Yes: in the CHR sense

- 1st-class entities in CHR
- rewritten by rules

Paint as Constraints?

Are paint colors really "constraints"?

No: not in the Constraint Programming sense

Yes: in the CHR sense

- 1st-class entities in CHR
- rewritten by rules

(Are Prolog predicates really "predicates"?)

The Paint Rules in CHR

Simplification Rule: *head* <=> *body* .

The Paint Rules in CHR

Simplification Rule: *head* <=> *body* .

"*head* may be replaced with *body*"

The Paint Rules in CHR

Simplification Rule: *head* <=> *body* .

"*head* may be replaced with *body*"

% 

```
red, blue    <=> purple.
```

The Paint Rules in CHR

Simplification Rule: *head* <=> *body* .

"*head* may be replaced with *body*"

% 

```
red, blue     <=> purple.
```

% 

```
red, yellow   <=> orange.
```

The Paint Rules in CHR

Simplification Rule: *head* <=> *body* .

"*head* may be replaced with *body*"

% 

```
red, blue     <=> purple.
```

% 

```
red, yellow   <=> orange.
```

% 

```
blue, yellow <=> green.
```

Time For...

...a cocktail break!

Cocktail Break

You can have as many heads as you like:

```
vodka, martini <=> vodka_martini.
```

Cocktail Break

You can have as many heads as you like:

```
vodka, martini <=> vodka_martini.

tequila, cointreau, lime <=> margarita.
```

Cocktail Break

You can have as many heads as you like:

```
vodka, martini <=> vodka_martini.

tequila, cointreau, lime <=> margarita.

gin, cherry_brandy, cointreau, grenadine,
pineapple, lemon, angostura_bitters
  <=> singapore_sling.
```

but at least one!

Cocktail Break

Warning: too many heads may cause a headache!

Cocktail Break

Warning: too many heads may cause a headache!

```
vodka_martini, margarita, singapore_sling
        <=> hangover, blackout.
```

Cocktail Break

Warning: too many heads may cause a headache!

```
vodka_martini, margarita, singapore_sling
        <=> hangover, blackout.
```

You can have many constraints in the body too.

Running the Rules

Paint-Mixing Engine:

- Prolog (SWI-Prolog, . . .)
- Leuven CHR system

Running the Rules

Paint-Mixing Engine:

- Prolog (SWI-Prolog, . . .)
- Leuven CHR system

Everything in a file:

paint.pl

```
:- use_module(library(chr)).

:- chr_constraint red.
...

red, blue <=> purple.
...
```

Running the Rules

Paint-Mixing Engine:

- Prolog (SWI-Prolog, . . .)
- Leuven CHR system

Everything in a file:

paint.pl

```
:- use_module(library(chr)).

:- chr_constraint red.
...

red, blue <=> purple.
...
```

```
?- [paint].
```

Running the Rules

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

```
?- red, blue.
```

- queries
- answers

Running the Rules

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

- queries
- answers

```
?- red, blue.
purple

?- red, yellow.
```

Running the Rules

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

- queries
- answers

```
?- red, blue.
purple

?- red, yellow.
orange

?- blue, yellow.
```

Running the Rules

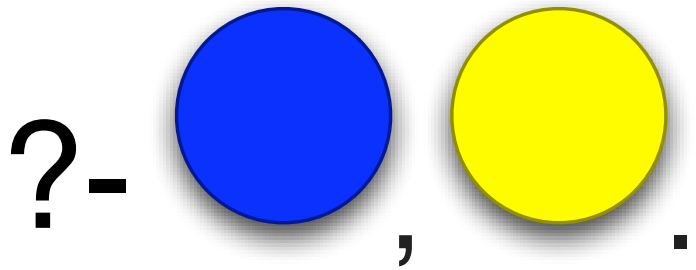
```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

- queries
- answers

```
?- red, blue.
purple

?- red, yellow.
orange

?- blue, yellow.
green
```

In Slow Motion

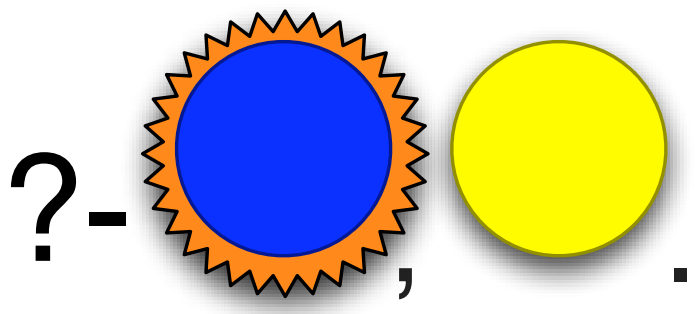
```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```



- query: left-to-right

In Slow Motion

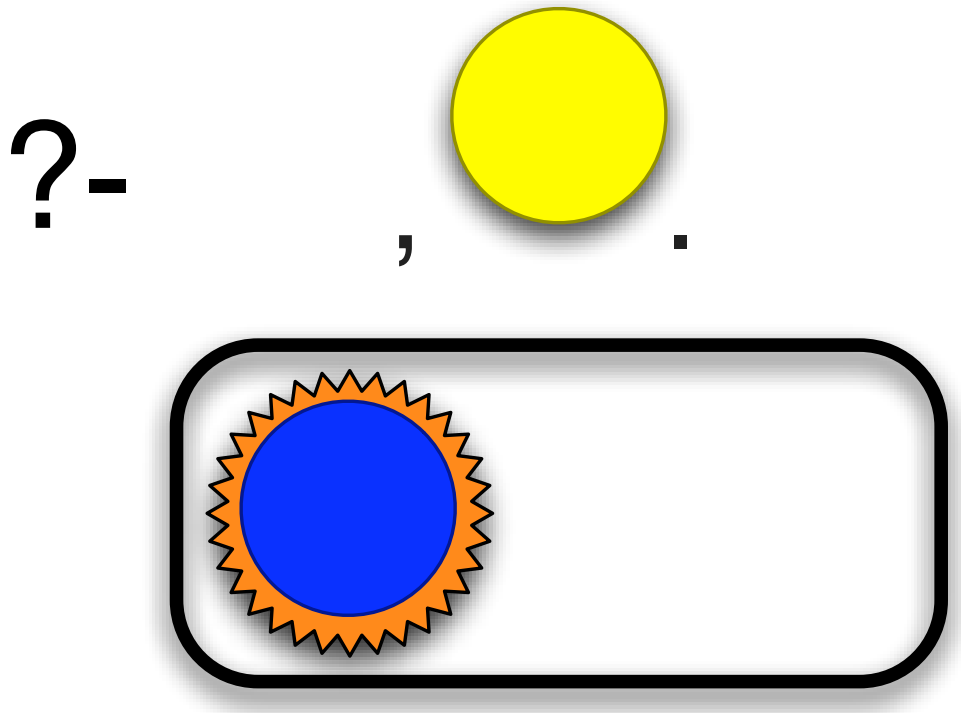
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```



- query: left-to-right
- active constraint: `blue`

In Slow Motion

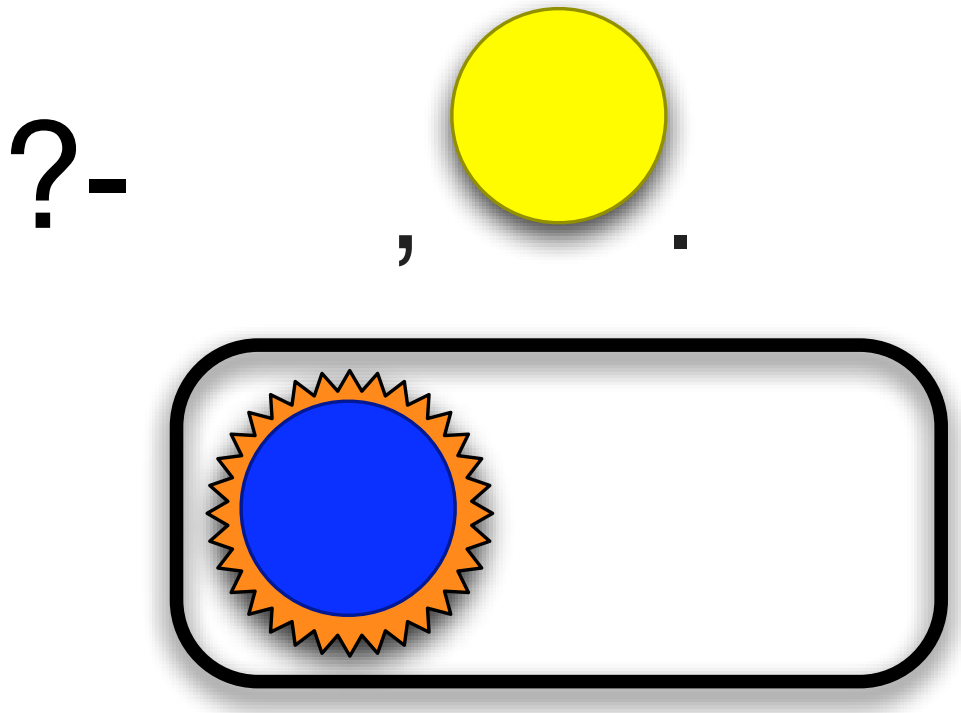
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```



- query: left-to-right
- active constraint: `blue`
- put in constraint store

In Slow Motion

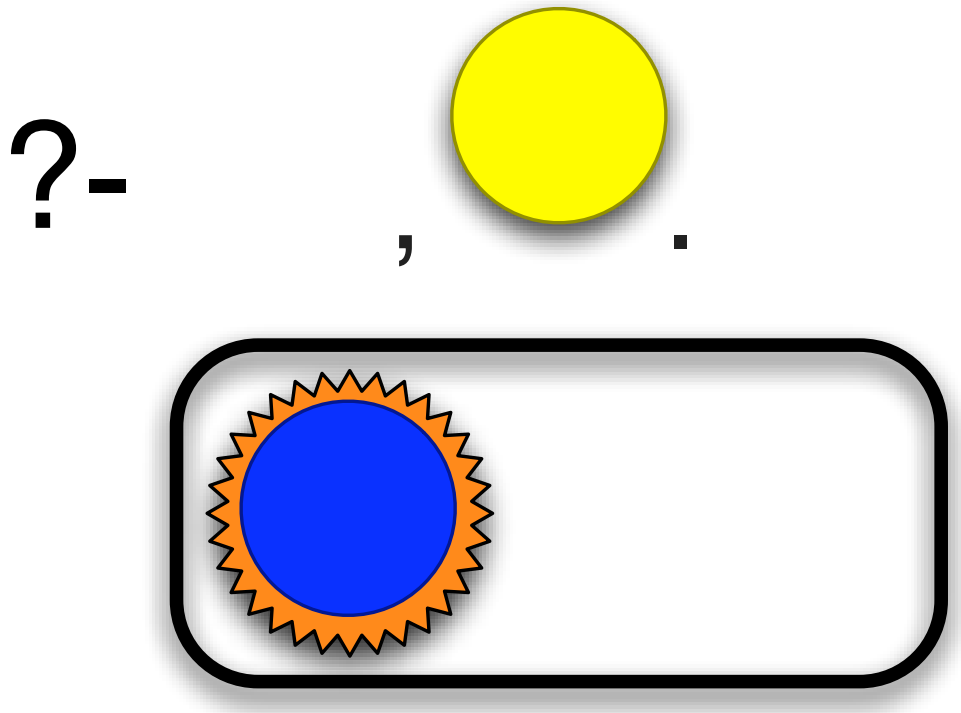
```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```



- active constraint: `blue`
- fire first rule?
- no partner `red` in constraint store

In Slow Motion

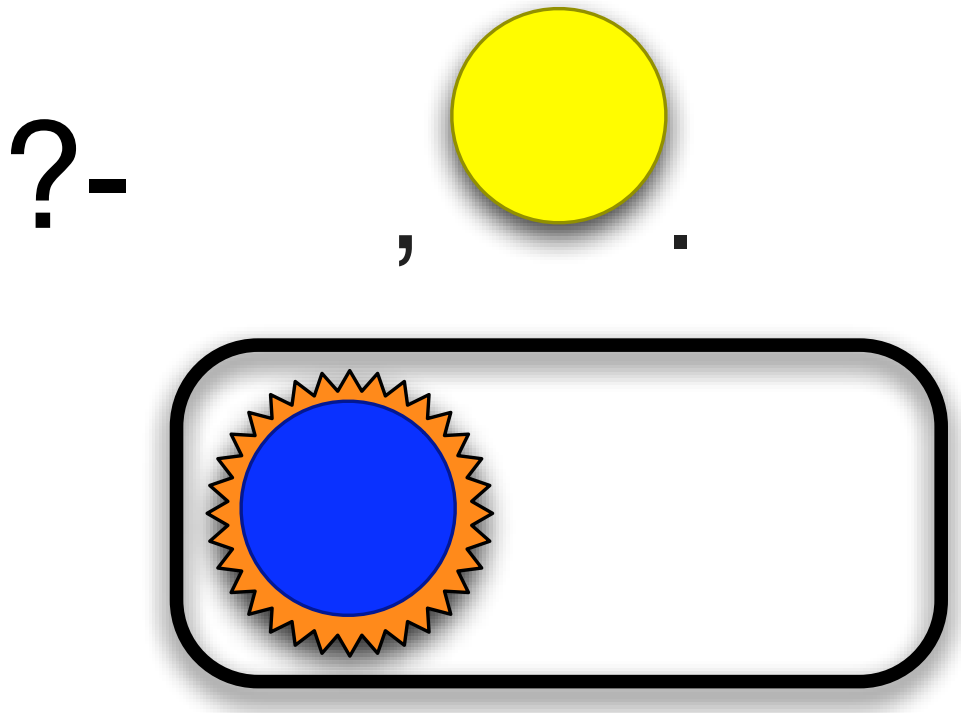
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```



- active constraint: `blue`
- fire second rule?
- no `blue` in head

In Slow Motion

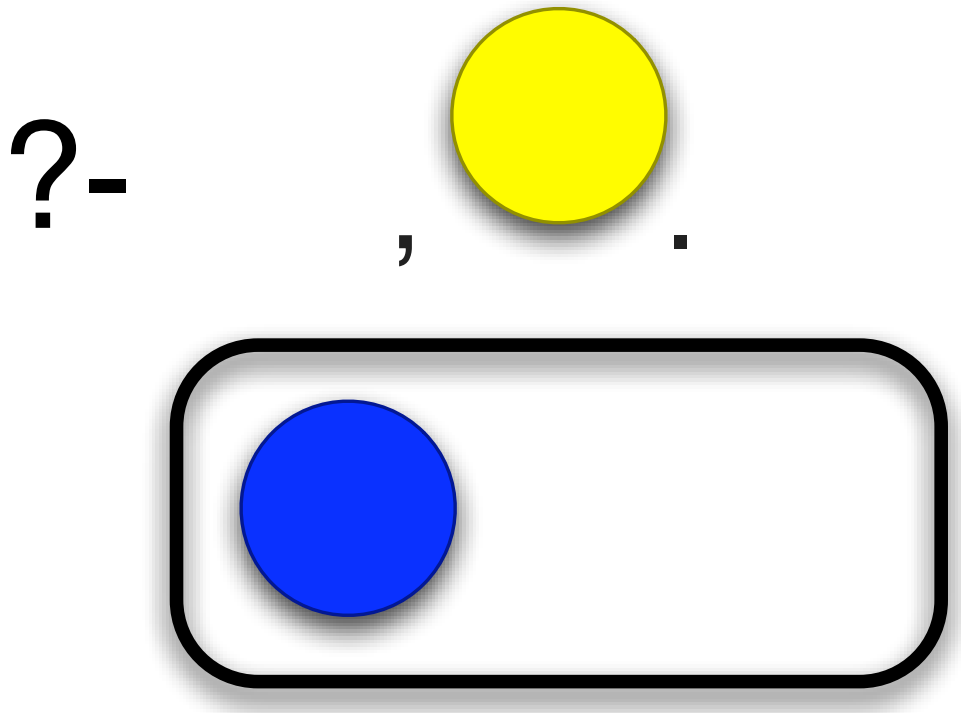
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```



- active constraint: `blue`
- fire third rule?
- no partner `yellow` in constraint store

In Slow Motion

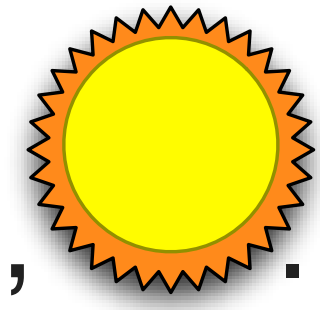
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

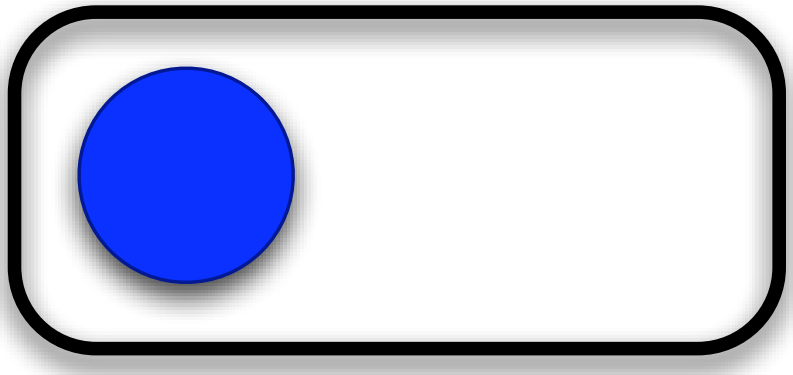


- deactivate constraint `blue`

In Slow Motion

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

?-  .

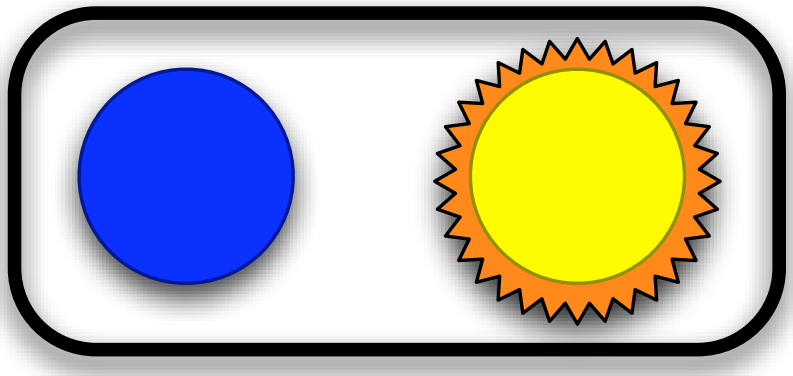


- active constraint: `yellow`

In Slow Motion

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

?- , .

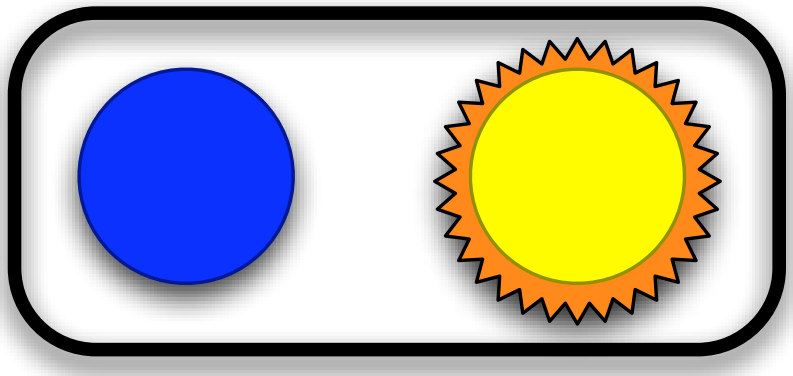


- active constraint: `yellow`
- put in constraint store

In Slow Motion

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

?- , .

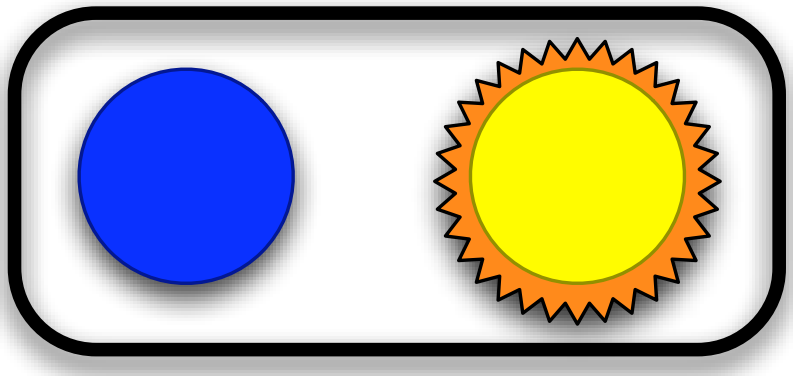


- active constraint: `yellow`
- fire first rule?
- no `yellow` in head

In Slow Motion

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

?- , .

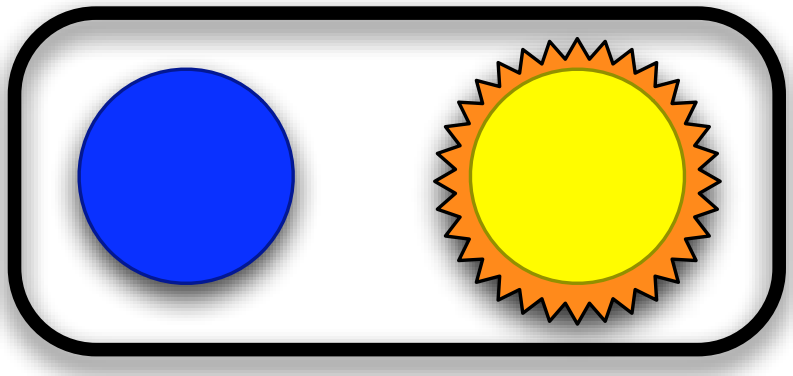


- active constraint: `yellow`
- fire second rule?
- no partner `red` in constraint store

In Slow Motion

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

?- , .

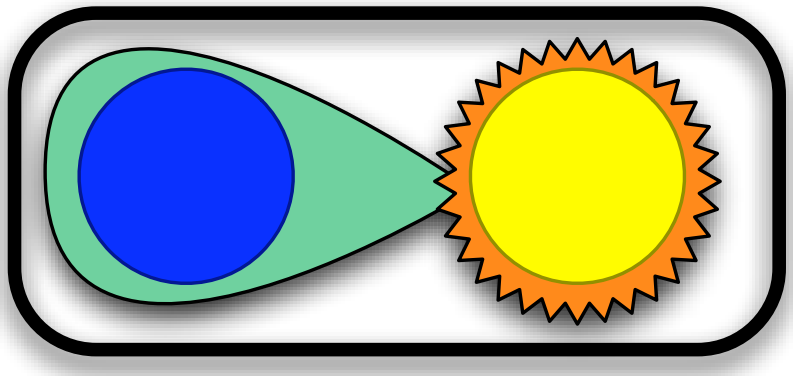


- active constraint: `yellow`
- fire third rule?

In Slow Motion

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

?- , .



- active constraint: `yellow`
- fire third rule!
- with partner `blue` in constraint store

In Slow Motion

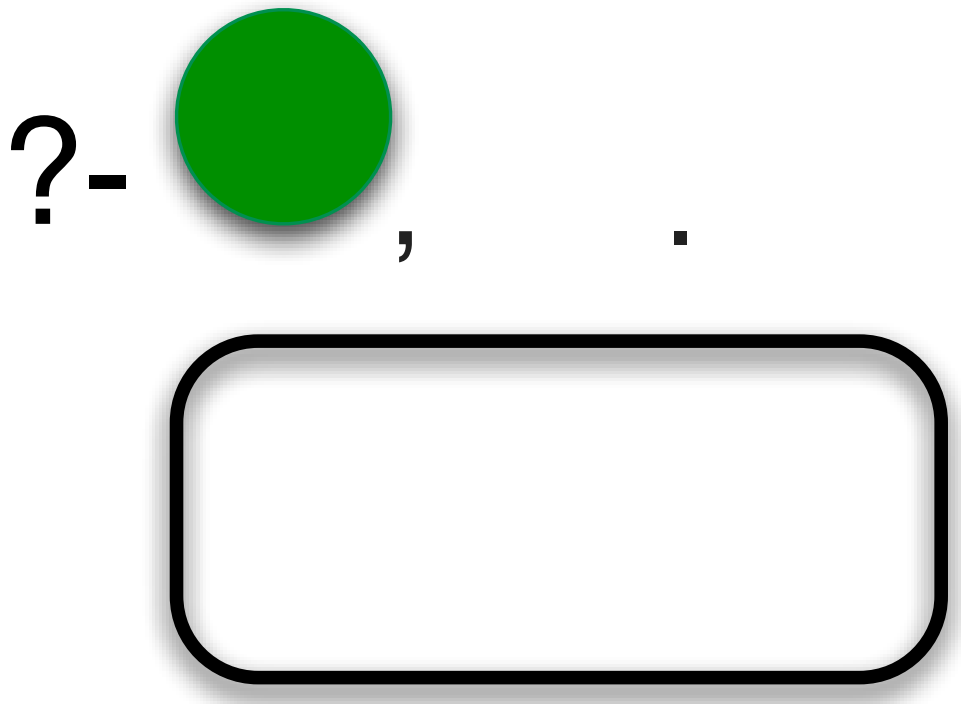
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

?- , .

- fire third rule!
- delete the matched head constraints

In Slow Motion

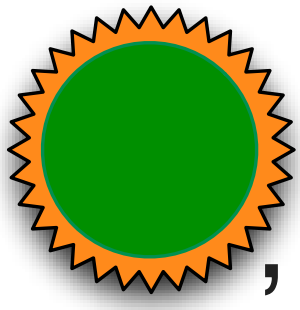
```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```



- fire third rule!
- add the body to (front of) query

In Slow Motion

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

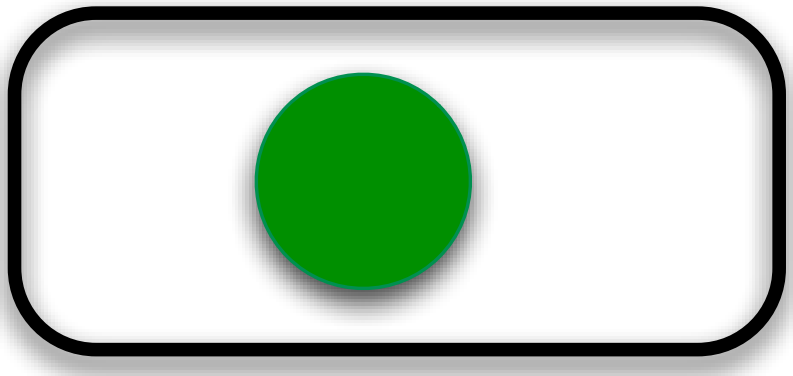
?-  , .

- active constraint: `green`

In Slow Motion

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

?- , .



to make a long story short:

- ...
- final constraint store

What Happens?

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

```
?- red, yellow.
orange
```

What Happens?

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

```
?- red, yellow.
orange

?- yellow, red.
orange
```

What Happens?

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

```
?- red.
```

What Happens?

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

```
?- red.
red
```

What Happens?

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

```
?- red, yellow, blue.
```

What Happens?

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

```
?- red, yellow, blue.
orange
blue
```

What Happens?

```
red, blue   <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.
```

```
?- red, blue, yellow.
```

What Happens?

```
red, blue    <=> purple.
red, yellow  <=> orange.
blue, yellow <=> green.
```

```
?- red, blue, yellow.
purple
yellow
```

Summary

Simplification Rule:

- *head* <=> *body* .
- replace head with body

Summary

Simplification Rule:

- *head* <=> *body* .
- replace head with body

Queries:

- processed left-to-right
- incrementally
- active constraint looks for partners

Brown

Brown stays brown.

```
brown, orange <=> brown.
brown, green  <=> brown.
brown, purple <=> brown.
...
```

Short Hand

Simpagation rule: $head_k \setminus head_r$ <=> $body$.

```
brown \ orange <=> true.
brown \ green  <=> true.
brown \ purple <=> true.
...
```

Short Hand

Simpagation rule: $head_k \setminus head_r$ <=> $body$.

```
brown \ orange <=> true.
brown \ green  <=> true.
brown \ purple <=> true.
...
```

- $head_k$: kept head (1 or more)

Short Hand

Simpagation rule: $head_k \setminus head_r$ <=> $body$.

```
brown \ orange <=> true.
brown \ green  <=> true.
brown \ purple <=> true.
...
```

- $head_k$: kept head (1 or more)
- $head_r$: removed head (1 or more)

Short Hand

Simpagation rule: $head_k \setminus head_r$ <=> $body$.

```
brown \ orange <=> true.
brown \ green  <=> true.
brown \ purple <=> true.
...
```

- $head_k$: kept head (1 or more)
- $head_r$: removed head (1 or more)
- `true` : Prolog's no-op

Lead to Gold

```
philosophers_stone \ lead
        <=> gold.
```

```
?- philosophers_stone, lead.
```

Lead to Gold

```
philosophers_stone \ lead
        <=> gold.
```

```
?- philosophers_stone, lead.
philosophers_stone
gold
```

Lead to Gold

```
philosophers_stone \ lead
        <=> gold.
```

```
?- philosophers_stone, lead.
philosophers_stone
gold

?- philosophers_stone, lead, lead.
```

Lead to Gold

```
philosophers_stone \ lead
        <=> gold.
```

```
?- philosophers_stone, lead.
philosophers_stone
gold

?- philosophers_stone, lead, lead.
philosophers_stone
gold
gold
```

All Combinations

```
philosophers_stone \ lead
        <=> gold.
```

```
?- lead, lead.
```

All Combinations

```
philosophers_stone \ lead
        <=> gold.
```

```
?- lead, lead.
lead
lead
```

All Combinations

```
philosophers_stone \ lead
        <=> gold.
```

```
?- lead, lead.
lead
lead

?- lead, lead, philosophers_stone.
```

All Combinations

```
philosophers_stone \ lead
        <=> gold.
```

```
?- lead, lead.
lead
lead

?- lead, lead, philosophers_stone.
philosophers_stone
gold
gold
```

All Combinations

```
philosophers_stone \ lead
        <=> gold.
```

```
?- lead, lead.
lead
lead

?- lead, lead, philosophers_stone.
philosophers_stone
gold
gold
```

Rule fires with all combinations!

Summary

Simplification Rule:

- *head* <=> *body* .
- replace head with body

Summary

Simplification Rule:

- *head* <=> *body* .
- replace head with body

Simpagation Rule:

- $head_k \setminus head_r$ <=> *body* .
- replace $head_r$ with *body*
- in the presence of $head_k$
- fires all possible combinations

Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

- constraints can have arguments

Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

- constraints can have arguments
- arity must be declared

Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

- constraints can have arguments
- arity must be declared
- Prolog terms as arguments

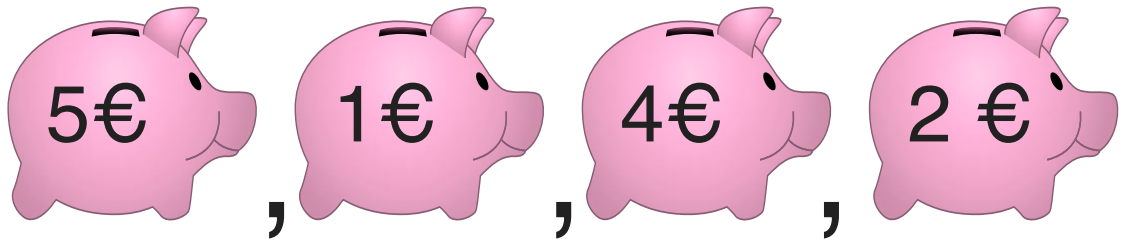
Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- piggy(5),piggy(1),piggy(4),piggy(2).

Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- 

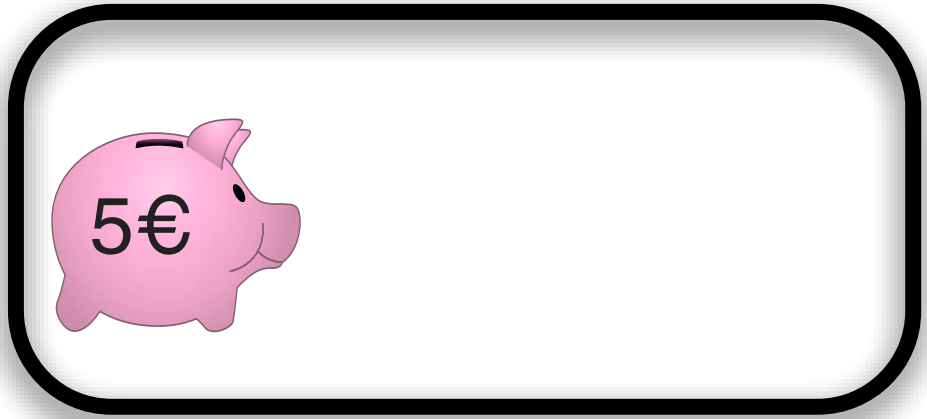
.

Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- 

.



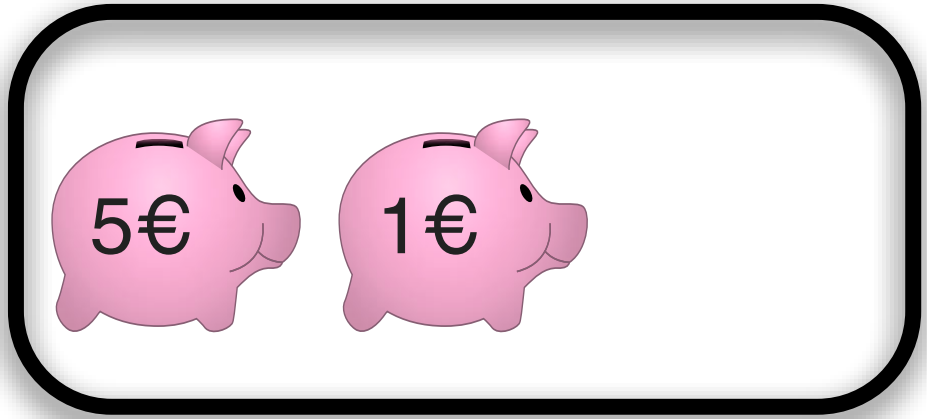
Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- , 

, 

.



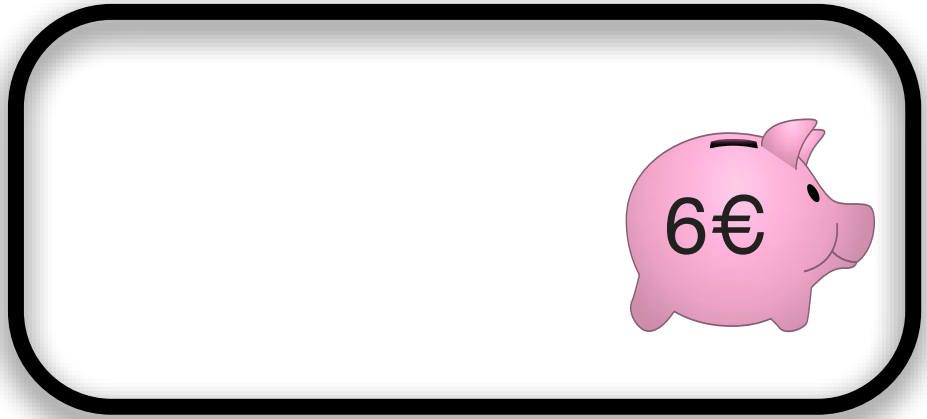
Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- , 

, 

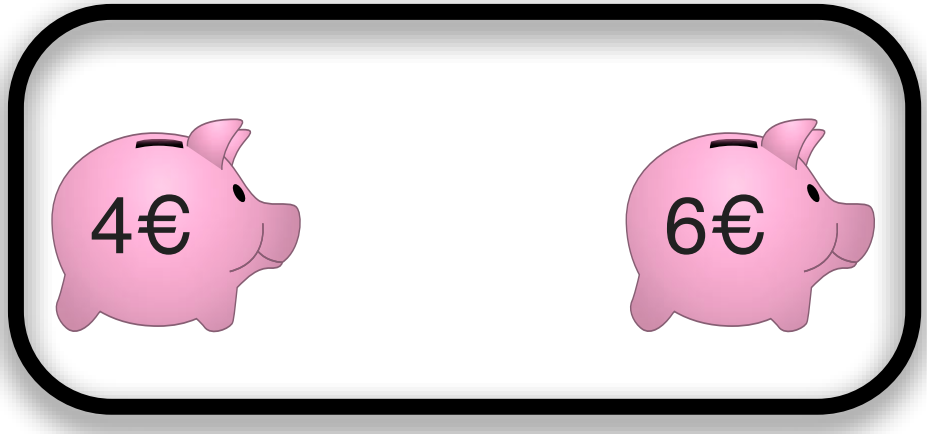
.



Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- , , , .

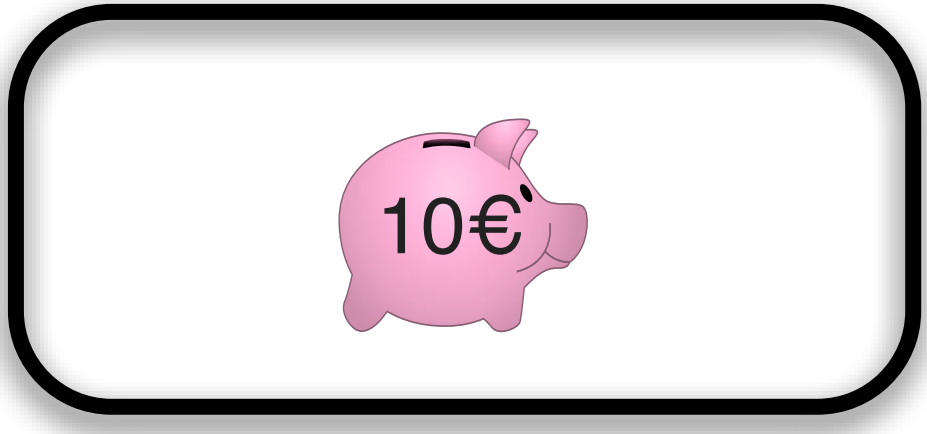


Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- , , , 

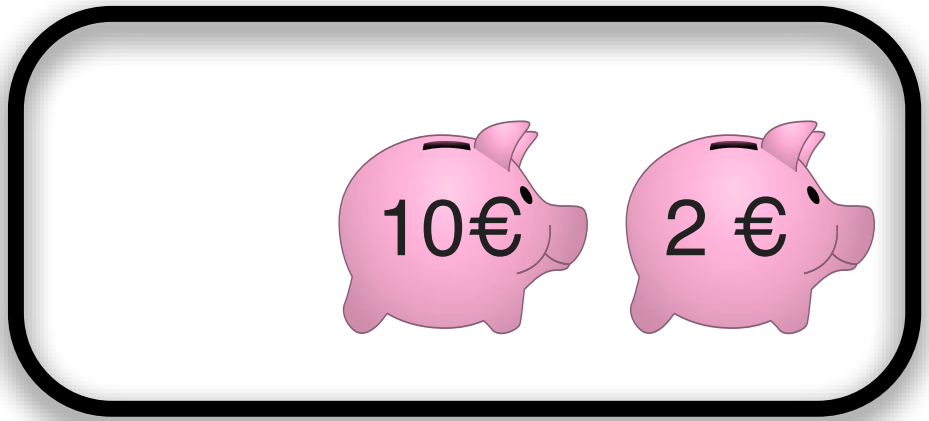
.



Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

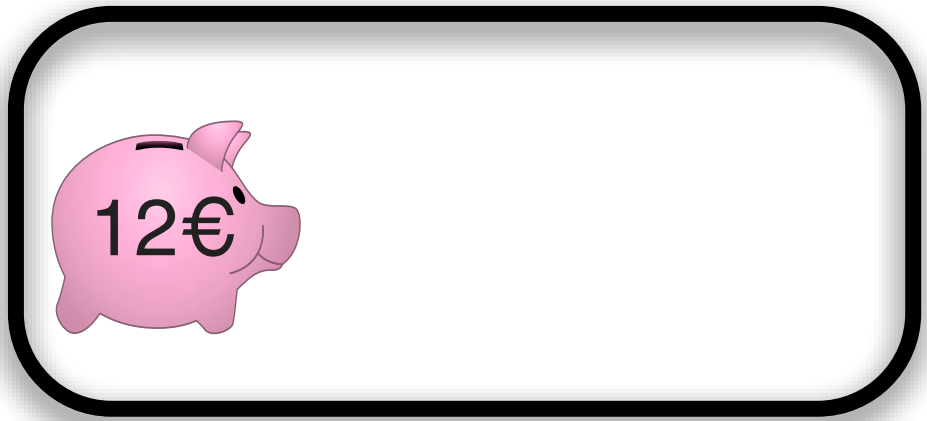
?- , , , .



Piggy Bank Merger

```
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).
```

?- , , , .



CHR($\mathcal{X}$)

CHR is an embedded language

- embedded in host language $\mathcal{X}$

CHR($\mathcal{X}$)

CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X}$ = Prolog

CHR($\mathcal{X}$)

CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X}$ = Prolog

2-way communication:

CHR($\mathcal{X}$)

CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X}$ = Prolog

2-way communication:

- Prolog calls CHR constraints
  e.g. all along from toplevel

CHR($\mathcal{X}$)

CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X}$ = Prolog

2-way communication:

- Prolog calls CHR constraints
  e.g. all along from toplevel
- CHR calls Prolog
  e.g. `K is I + J`

Not Just Numbers

```
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
```

Not Just Numbers

```
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
```

```
?- value([a]), value([b]).
```

Not Just Numbers

```
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
```

```
?- value([a]), value([b]).
value([b,a]).
```

Not Just Numbers

```
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
```

```
?- value([a]), value([b]).
value([b,a]).

?- value([f(a),g(b)]), value([h(i,j)]).
```

Not Just Numbers

```
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
```

```
?- value([a]), value([b]).
value([b,a]).

?- value([f(a),g(b)]), value([h(i,j)]).
value([h(i,j),f(a),g(b)]).
```

Not Just Numbers

```
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
```

```
?- value([a]), value([b]).
value([b,a]).

?- value([f(a),g(b)]), value([h(i,j)]).
value([h(i,j),f(a),g(b)]).
```

Any Prolog terms!

Summary

CHR constraints

- zero or more arguments
- Prolog terms

Summary

CHR constraints

- zero or more arguments
- Prolog terms

Bodies

- CHR constraints and
- Prolog predicate calls

Count Down

Problem: write a program to enumerate values:

```
?- generate(5).
```

Count Down

Problem: write a program to enumerate values:

```
?- generate(5).
value(5)
value(4)
value(3)
value(2)
value(1)
```

Count Down

Solution:

```
:- chr_constraint generate/1.
```

Count Down

Solution:

```
:- chr_constraint generate/1.
:- chr_constraint value/1.
```

Count Down

Solution:

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=>
  value(N), M is N - 1, generate(M).
```

Running the Program

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).
```

```
?- generate(5).
```

Running the Program

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).
```

```
?- generate(5).
    % waiting
```

Running the Program

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).
```

```
?- generate(5).
    % waiting
    % still waiting
```

Running the Program

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).
```

```
?- generate(5).
    % waiting
    % still waiting
ERROR: Out of local stack
```

Running the Program

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).
```

```
?- generate(5).
    % waiting
    % still waiting
ERROR: Out of local stack
```

Recursion without a base case!

Guarded Rules

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0  | value(N),
                         M is N - 1, generate(M).
```

Guarded Rules

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0  | value(N),
                         M is N - 1, generate(M).
```

Guarded rule:

- apply rule **if** guard succeeds

Guarded Rules

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0  | value(N),
                         M is N - 1, generate(M).
```

Guarded rule:

- apply rule **if** guard succeeds
- guard is optional

Guarded Rules

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0  | value(N),
                         M is N - 1, generate(M).
```

Guarded rule:

- apply rule **if** guard succeeds
- guard is optional
- guard may contain any Prolog, but may not bind any variables from the head (pure check)

Even Shorter

```
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

Even Shorter

```
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

```
?- generate(5).
```

Even Shorter

```
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

```
?- generate(5).
value(5)
value(4)
value(3)
value(2)
value(1)
generate(0)
```

Summary

Simplification Rule:

- *head* <=> *guard* | *body* .
- replace head with body
- if *guard* succeeds

Summary

Simplification Rule:

- *head* <=> *guard* | *body* .
- replace head with body
- if *guard* succeeds

Simpagation Rule:

- $head_k \setminus head_r$ <=> *guard* | *body* .
- replace $head_r$ with *body*
- in the presence of $head_k$
- if *guard* succeeds

CHR Style: Matching

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

Matching:

- "inline" notation for guard
- same meaning as explicit guard

Meaning of Matching

Maching	**Guard**
`c(X) <=> true.`	`c(X) <=> true	true.`

Meaning of Matching

Maching	**Guard**
`c(X) <=> true.`	`c(X) <=> true | true.`
`c(a) <=> true.`	`c(X) <=> X == a | true.`

Meaning of Matching

Maching	Guard
`c(X) <=> true.`	`c(X) <=> true | true.`
`c(a) <=> true.`	`c(X) <=> X == a | true.`
`c(f(A)) <=> true.`	`c(X) <=> nonvar(X), X = f(A) | true.`

Meaning of Matching

Maching	Guard
`c(X) <=> true.`	`c(X) <=> true | true.`
`c(a) <=> true.`	`c(X) <=> X == a | true.`
`c(f(A)) <=> true.`	`c(X) <=> nonvar(X), X = f(A) | true.`
`c(X,X) <=> true.`	`c(X,Y) <=> X == Y | true.`

Meaning of Matching

Maching	**Guard**
`c(X) <=> true.`	`c(X) <=> true | true.`
`c(a) <=> true.`	`c(X) <=> X == a | true.`
`c(f(A)) <=> true.`	`c(X) <=> nonvar(X), X = f(A) | true.`
`c(X,X) <=> true.`	`c(X,Y) <=> X == Y | true.`
`c(X), d(X) <=> true.`	`c(X), d(Y) <=> X == Y | true.`

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

```
?- c(world).
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

```
?- c(world).
Hello, World!
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

```
?- c(world).
Hello, World!

?- c(Free).
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

```
?- c(world).
Hello, World!

?- c(Free).
c(Free)
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

```
?- c(world).
Hello, World!

?- c(Free).
c(Free)



?- c(hello).
```

Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```

```
?- c(world).
Hello, World!

?- c(Free).
c(Free)


?- c(hello).
c(hello)
```

Summary

Matching

- short-hand for equality-based guards
- one-way unification
  - only instantiates head
  - does not instantiate constraints

Combining Rules

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

What's the outcome?

```
?- generate(4).
```

Combining Rules

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

What's the outcome?

```
?- generate(4).
value(10)
```

Combining Rules

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

What's the outcome?

```
?- generate(4).
value(10)

?- generate(5).
```

Combining Rules

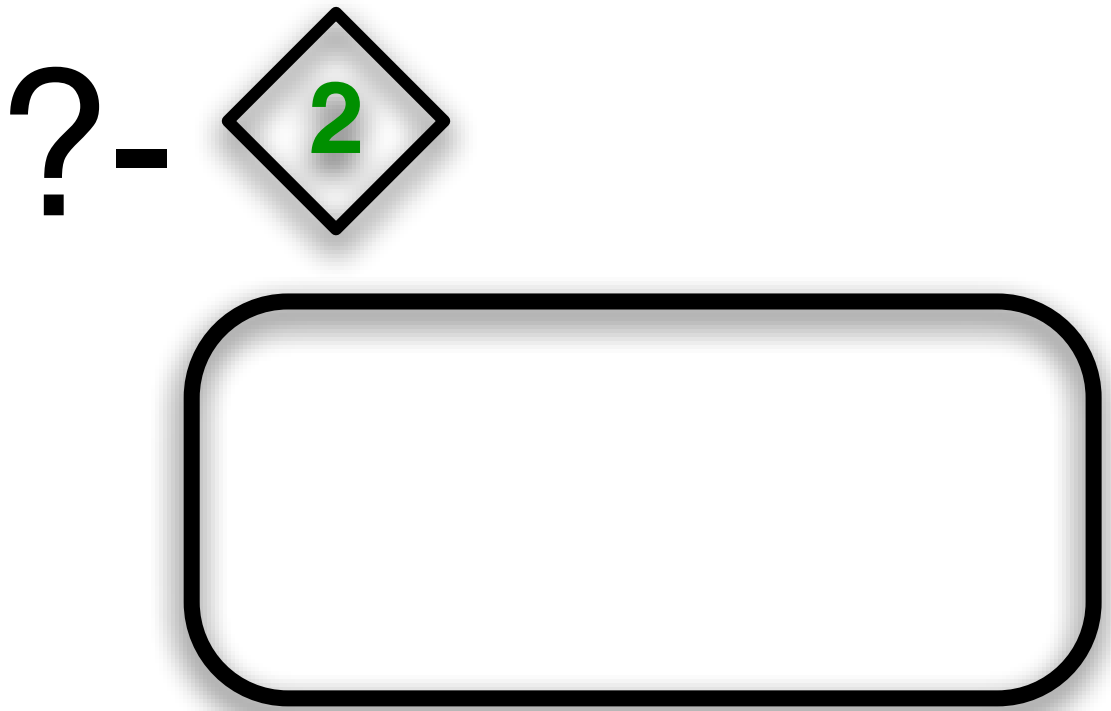
```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

What's the outcome?

```
?- generate(4).
value(10)

?- generate(5).
value(15)
```

In Slow Motion

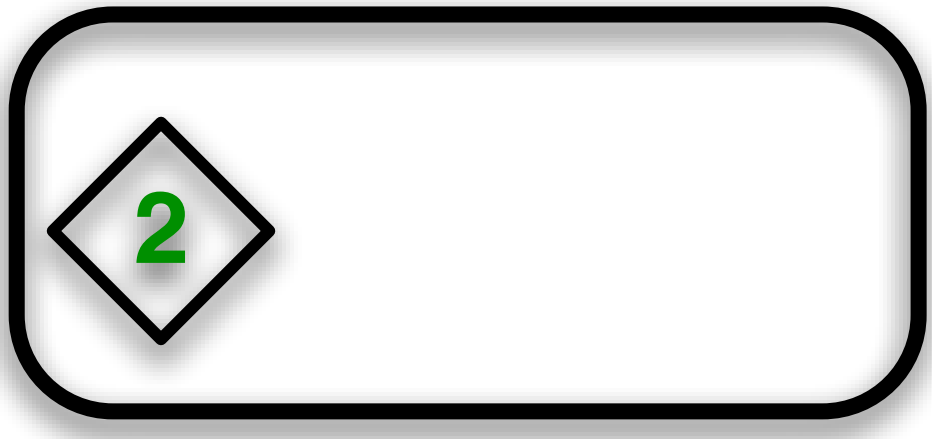
```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



In Slow Motion

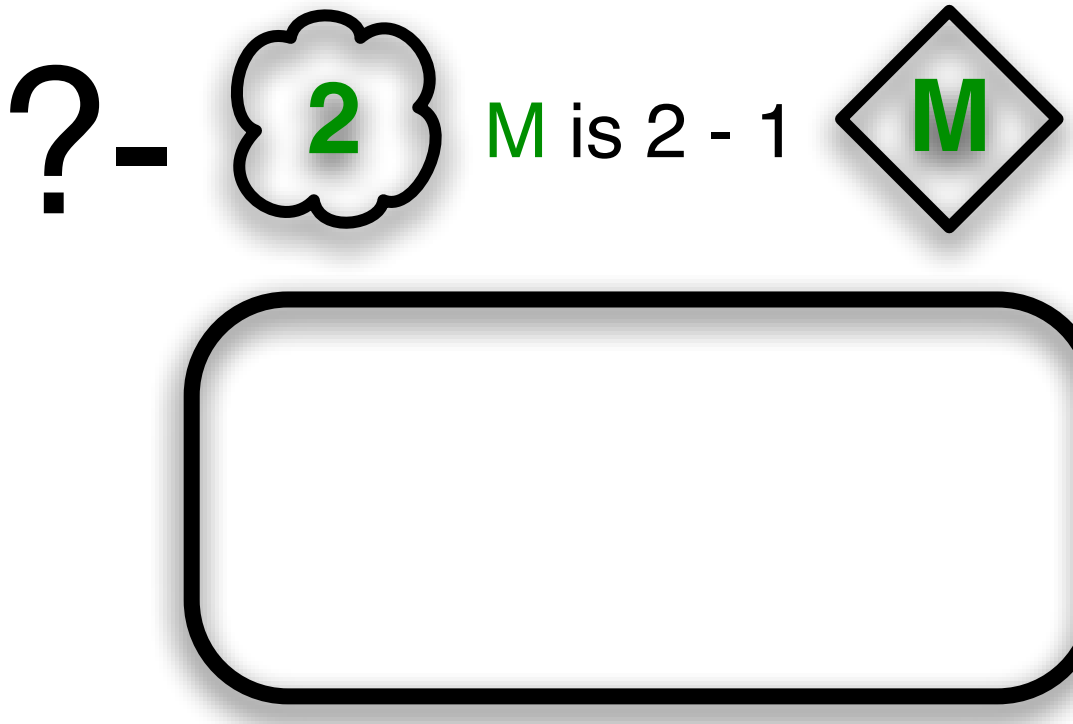
```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

?-



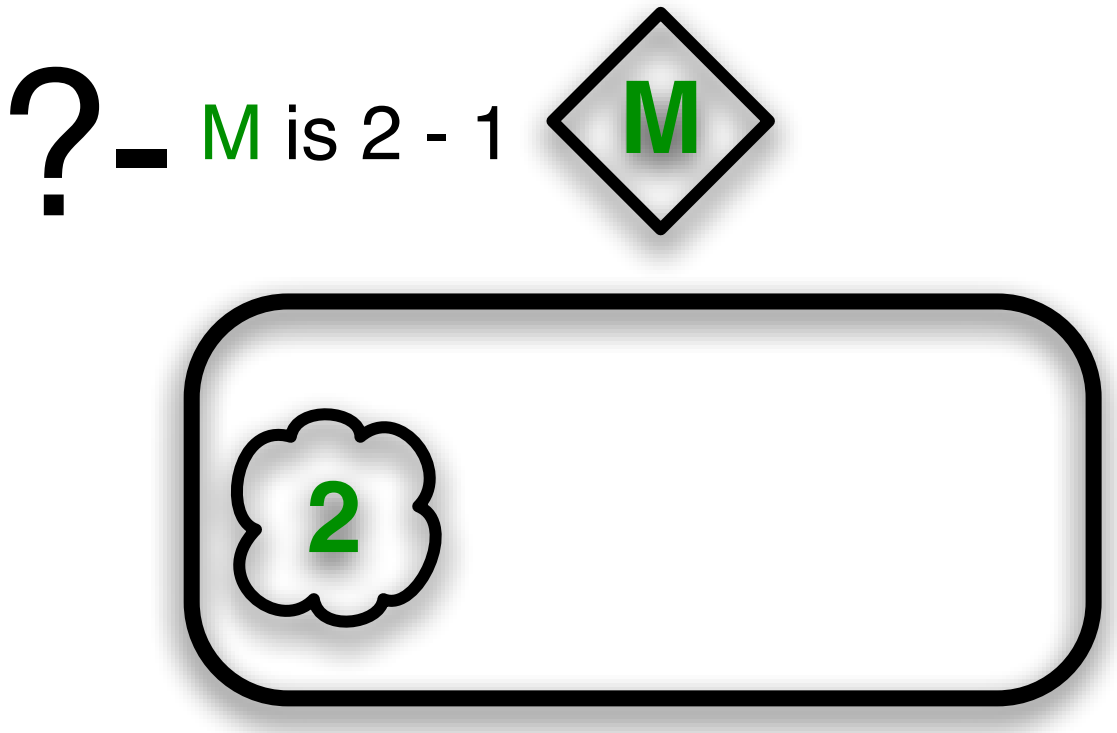
In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



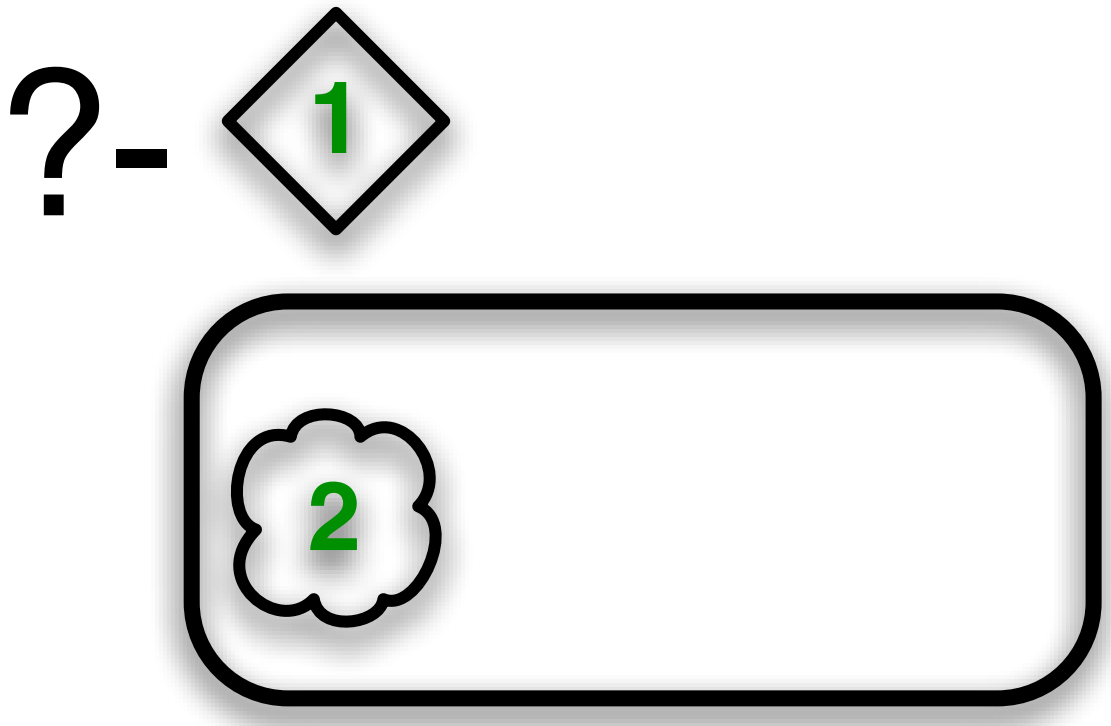
In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



In Slow Motion

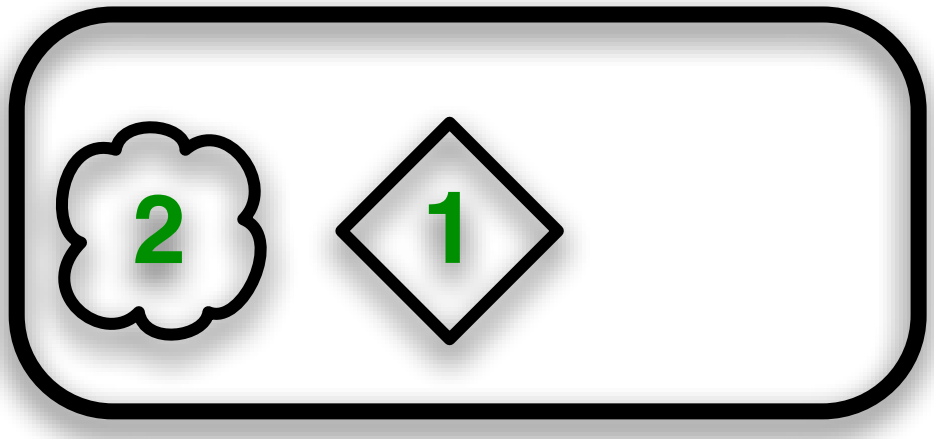
```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



In Slow Motion

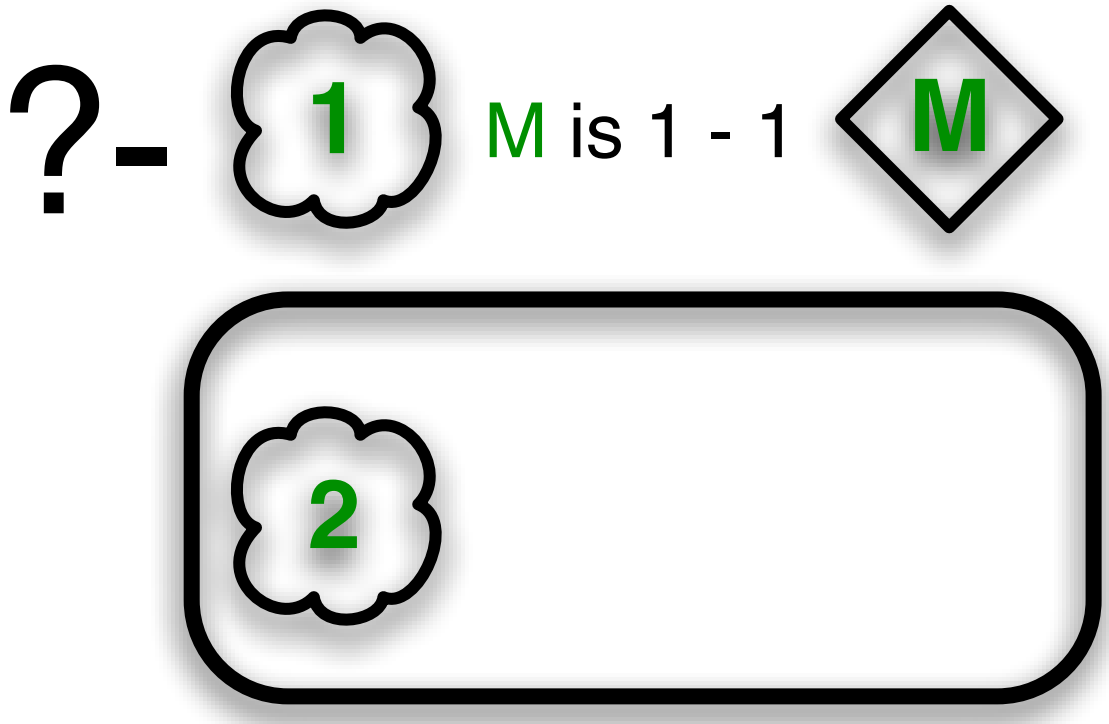
```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

?-



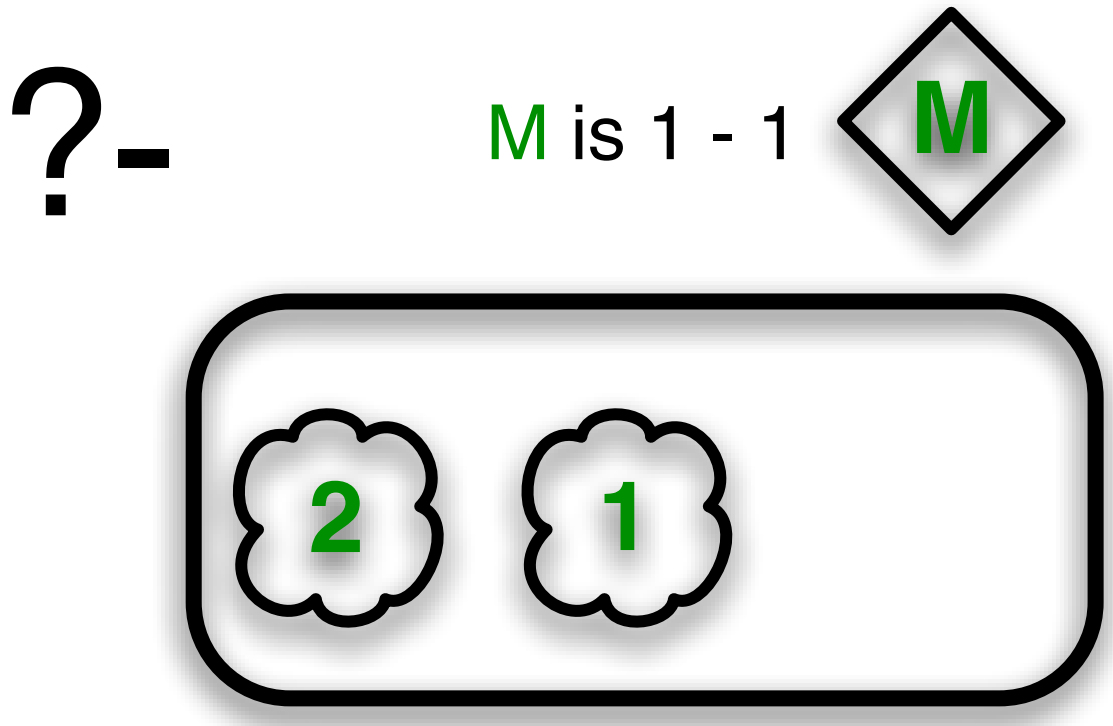
In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



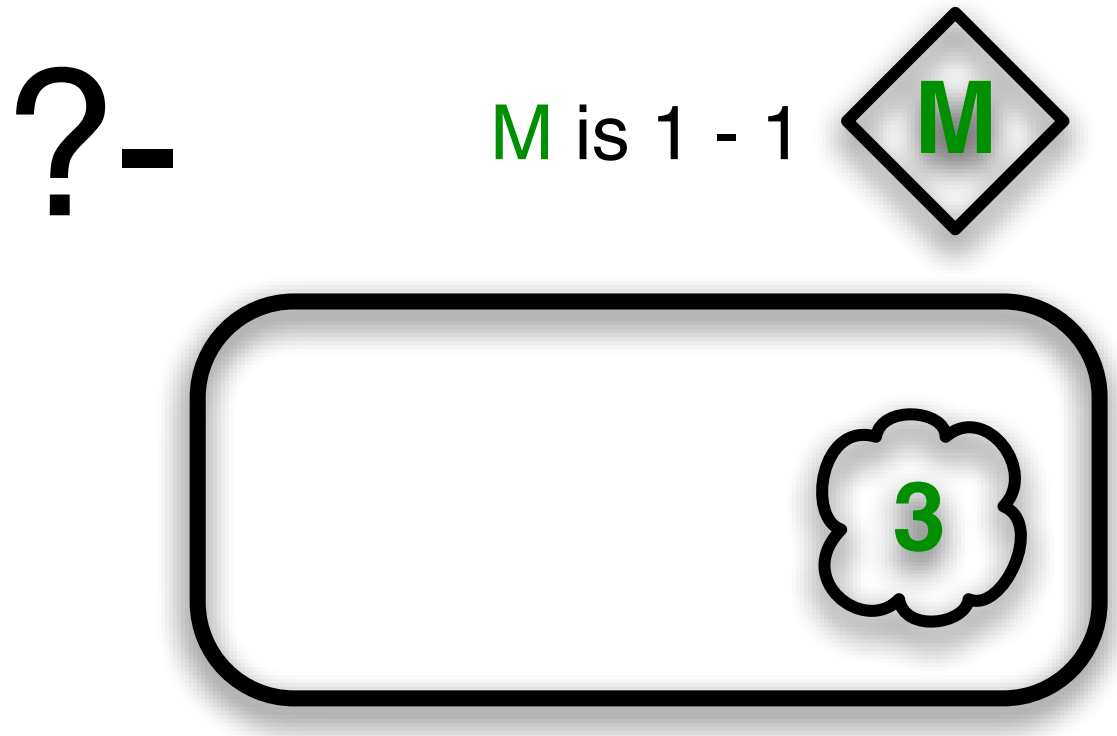
In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



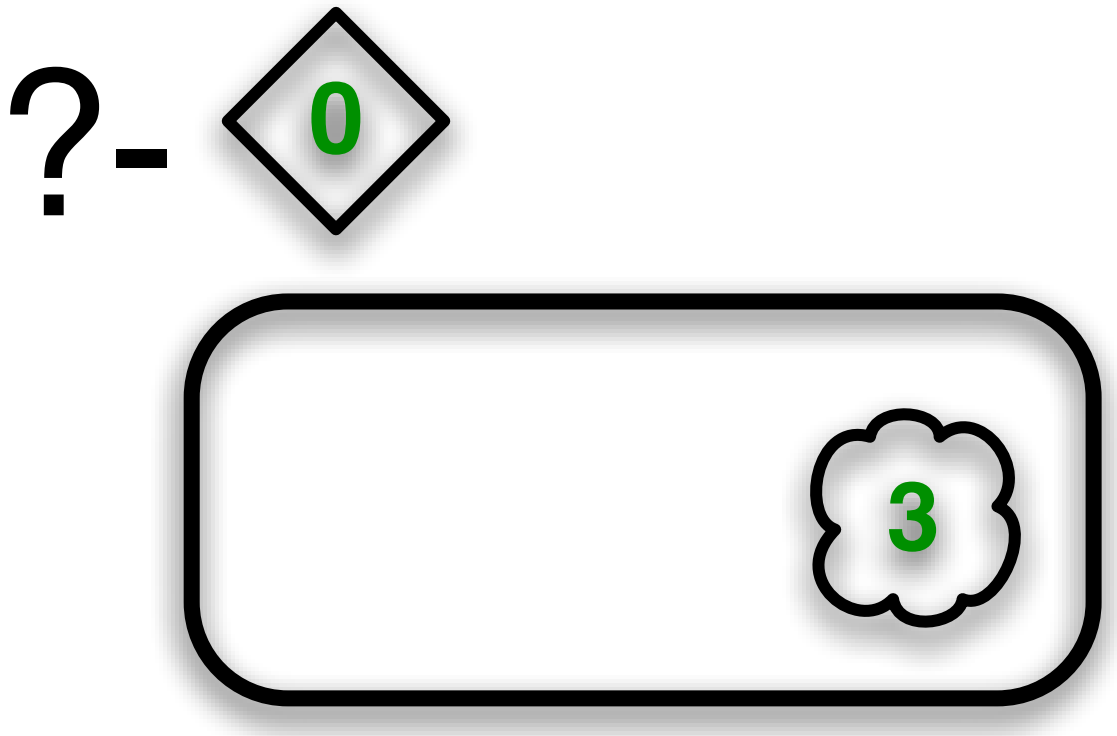
In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



In Slow Motion

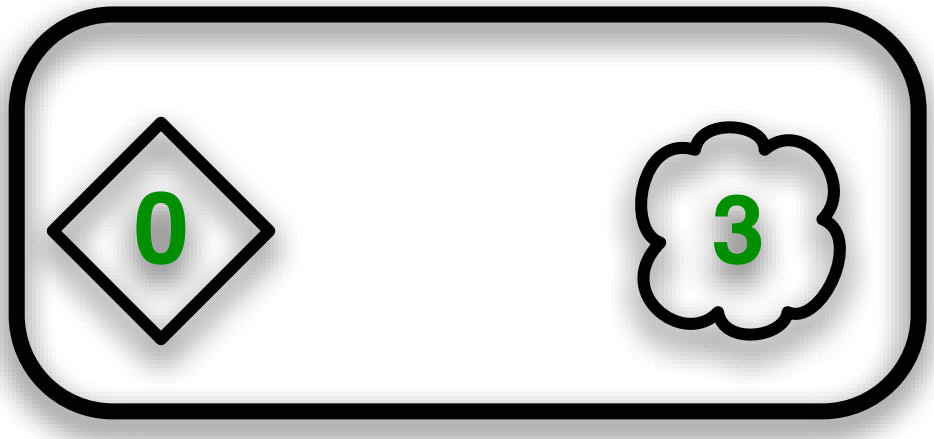
```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```



In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

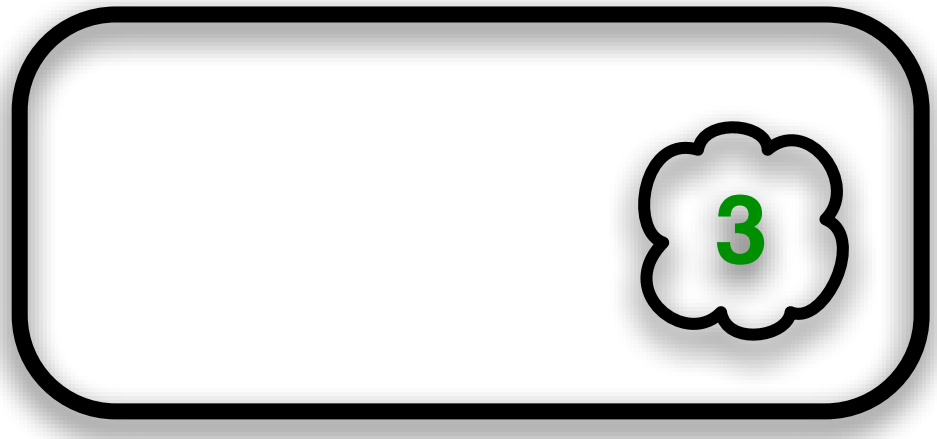
?-



In Slow Motion

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
```

?-



Combining Rules

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

What's the outcome?

```
?- generate(4).
```

Combining Rules

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

What's the outcome?

```
?- generate(4).
value(2)
value(3)
```

Combining Rules

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

What's the outcome?

```
?- generate(4).
value(2)
value(3)

?- generate(10).
```

Combining Rules

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

What's the outcome?

```
?- generate(4).
value(2)
value(3)

?- generate(10).
value(2)
value(3)
value(5)
value(7)
```

How Does It Work?

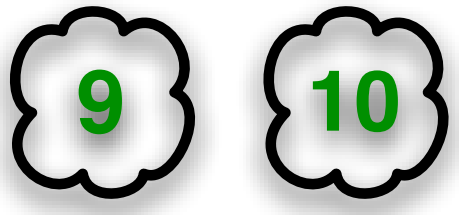
```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

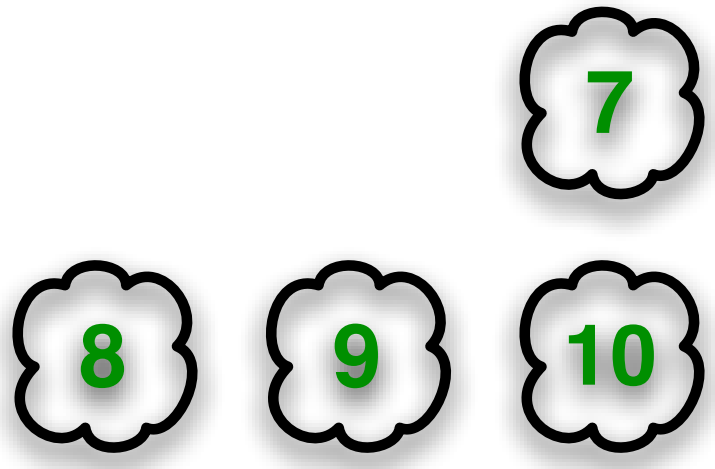


How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

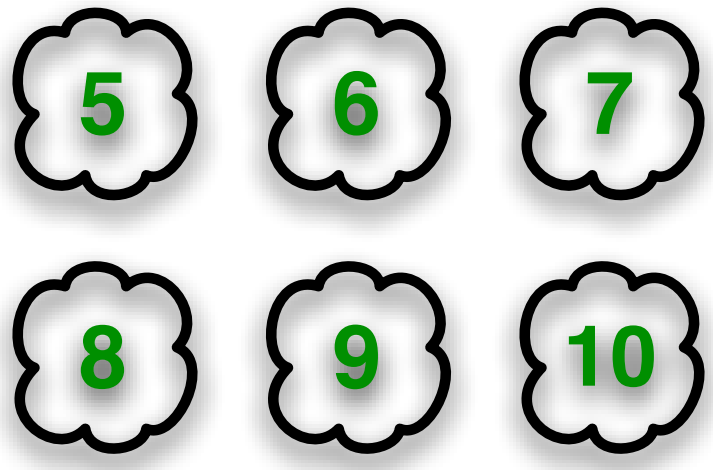


How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```



How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

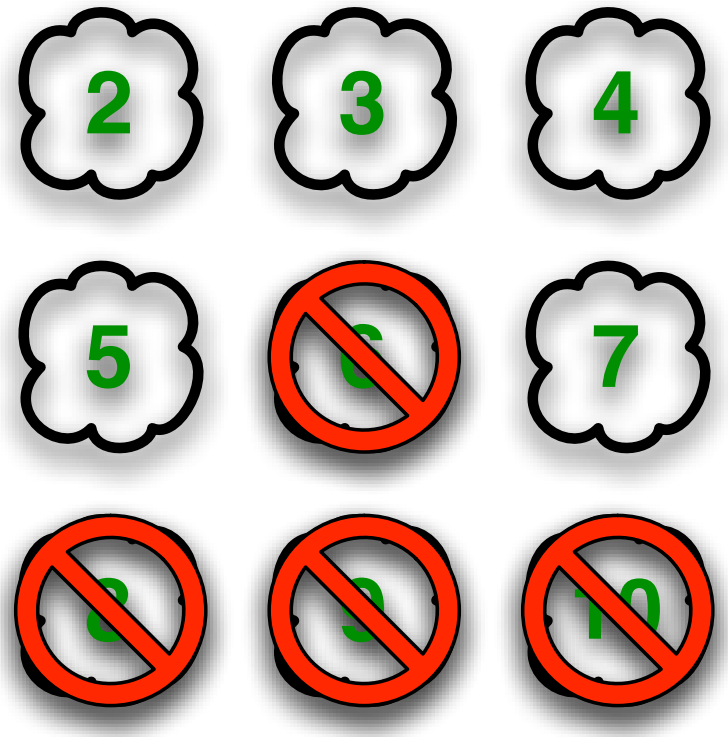
```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

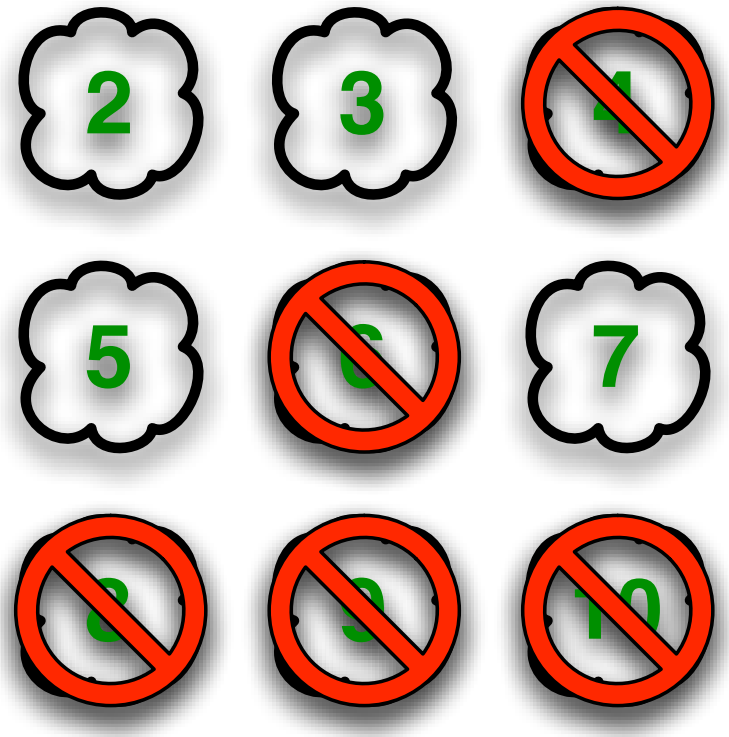
How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```



How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```



Alternatives?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Just added `value(3)`.
What now?

Alternatives?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Just added `value(3)`.
What now?

- remove `value(6)` ?

Alternatives?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Just added `value(3)`.
What now?

- remove `value(6)` ?
- remove `value(9)` ?

Alternatives?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Just added `value(3)`.
What now?

- remove `value(6)` ?
- remove `value(9)` ?
- unspecified in CHR!

Alternate Ending

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Alternate Ending

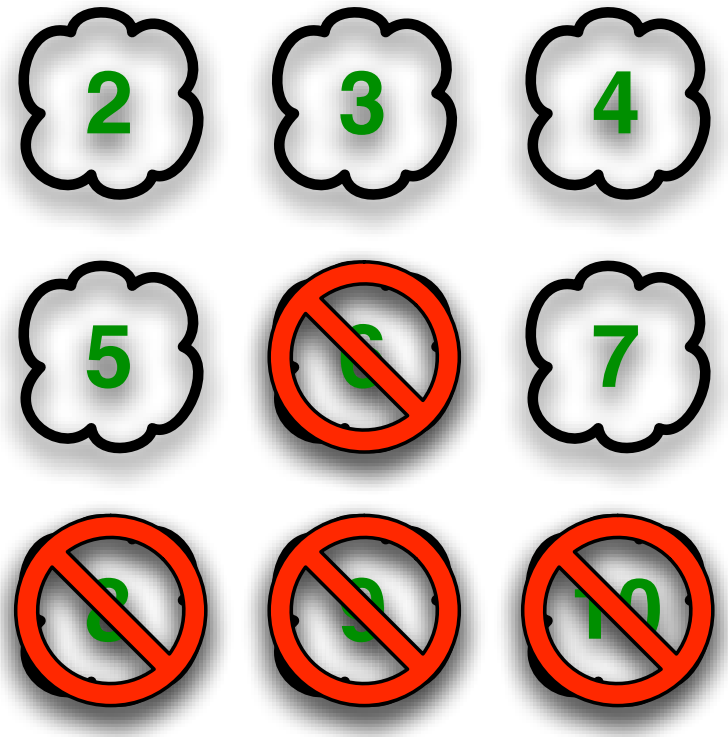
```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Alternate Ending

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

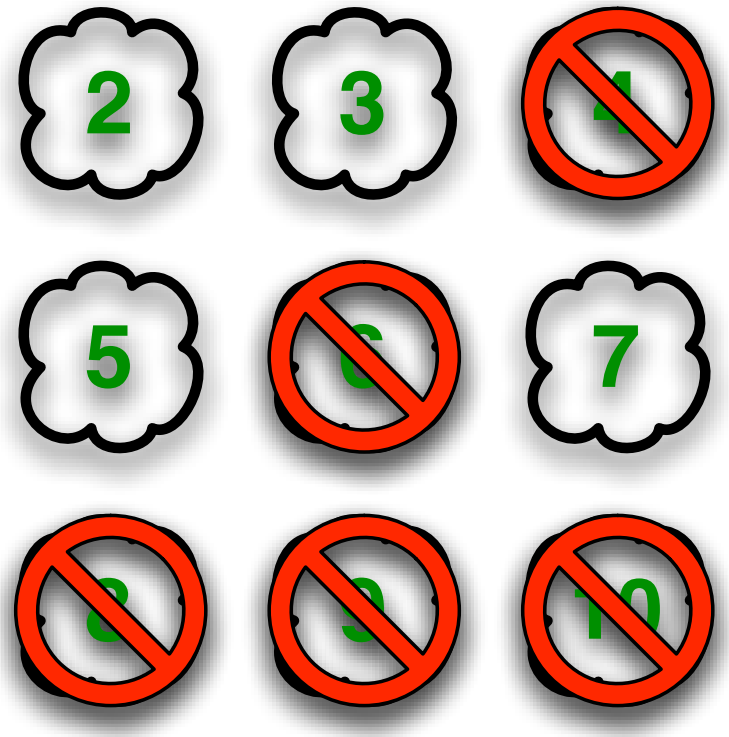
Alternate Ending

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```



Alternate Ending

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                        M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```



Summary

CHR Programs

- Programs consist of a sequence of rules

Summary

CHR Programs

- Programs consist of a sequence of rules
- Active constraint traverses the rules top-to-bottom to find any that fire

Summary

CHR Programs

- Programs consist of a sequence of rules
- Active constraint traverses the rules top-to-bottom to find any that fire
- Rest of query/body waits

Summary

CHR Programs

- Programs consist of a sequence of rules
- Active constraint traverses the rules top-to-bottom to find any that fire
- Rest of query/body waits
- Unspecified which partner is found

Simpagation with $H_k = \emptyset$

Simpagation rule

$$head_k \setminus head_r \mathrel{<=>} guard \mid body\,.$$

Simpagation with $H_k = \emptyset$

Simpagation rule

$head_k \setminus head_r$ <=> $guard \mid body$.

What if $head_k = \emptyset$?

Simpagation with $H_k = \emptyset$

Simpagation rule

$head_k \setminus head_r$ <=> $guard \mid body$.

What if $head_k = \emptyset$?

Simplification rule

$head_r$ <=> $guard \mid body$.

Simpagation with $H_r = \emptyset$

Simpagation rule

$$head_k \setminus head_r \Leftrightarrow guard \mid body\ .$$

Simpagation with $H_r = \emptyset$

Simpagation rule

$head_k \setminus head_r$ <=> $guard \mid body$.

What if $head_r = \emptyset$?

Simpagation with $H_r = \emptyset$

Simpagation rule

$head_k \setminus head_r <=> guard \mid body$.

What if $head_r = \emptyset$?

Propagation rule

$head_k ==> guard \mid body$.

- add *body*
- in the presence of $head_k$
- if *guard* holds

Propagation

```
a, b ==> c.
```

```
?- a, b.
```

Propagation

```
a, b ==> c.
```

```
?- a, b.
a
b
c
```

Propagation

```
a, b ==> c.
```

```
?- a, b.
a
b
c
?- a, b, b.
```

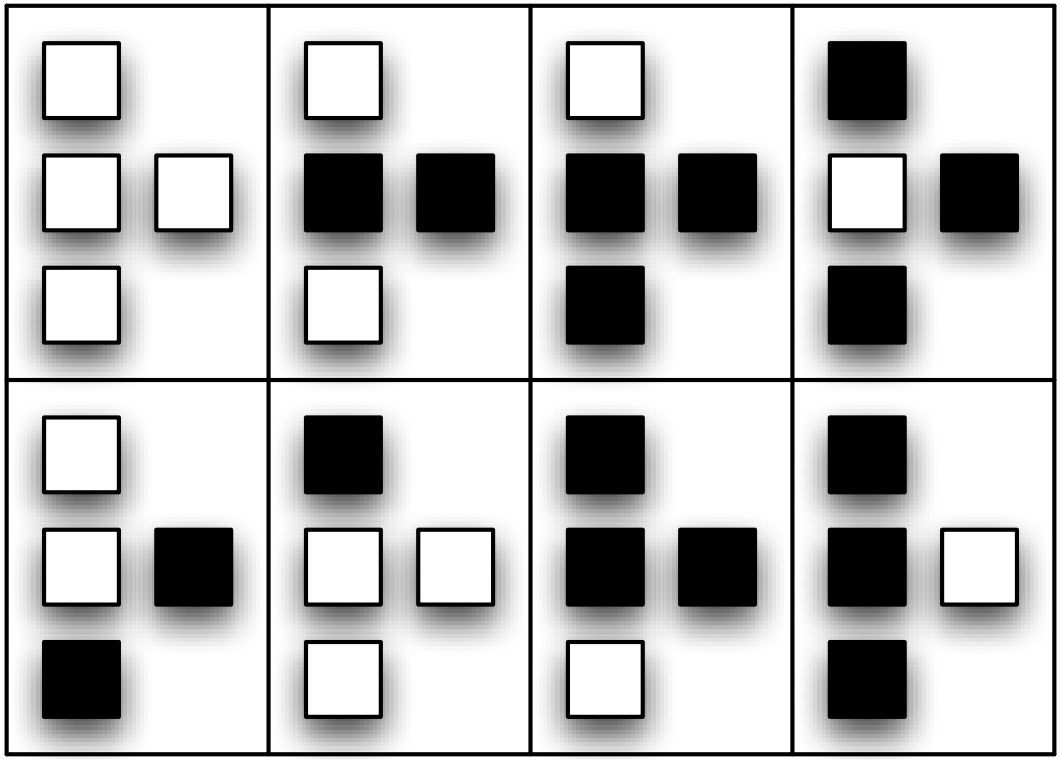
Propagation

```
a, b ==> c.
```

```
?- a, b.
a
b
c
?- a, b, b.
a
b
b
c
c
```

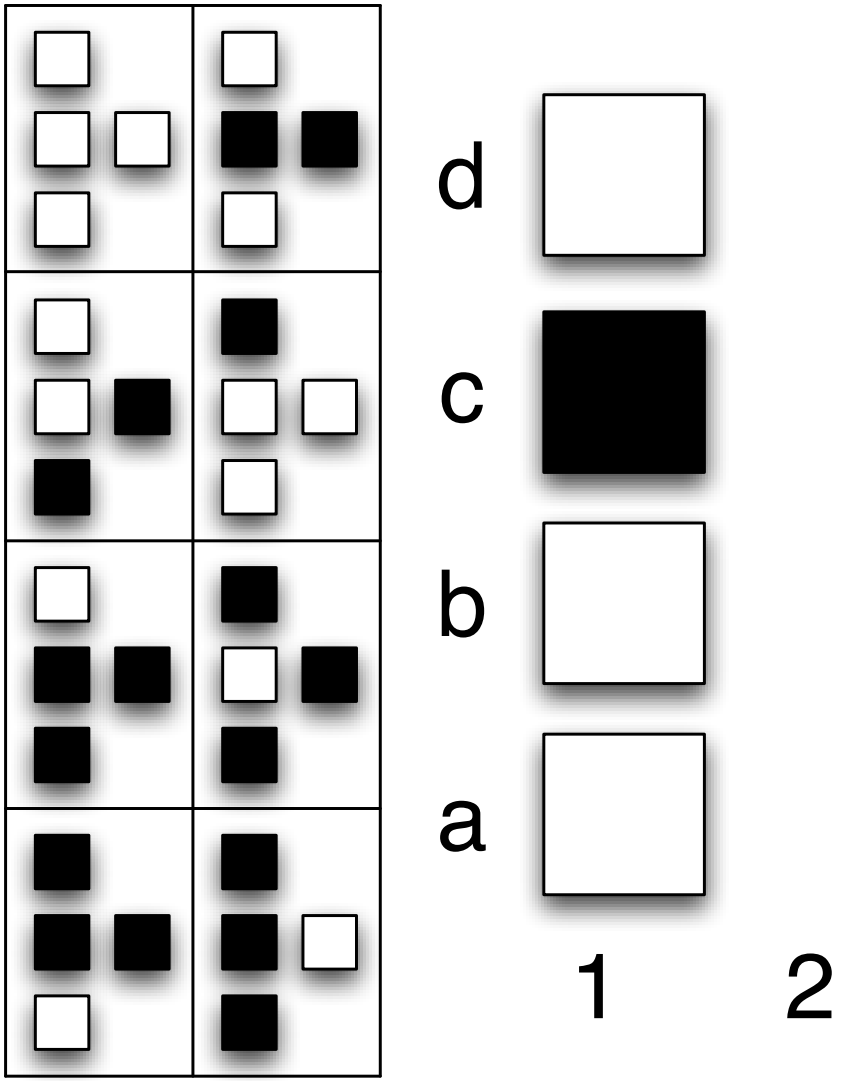
Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.



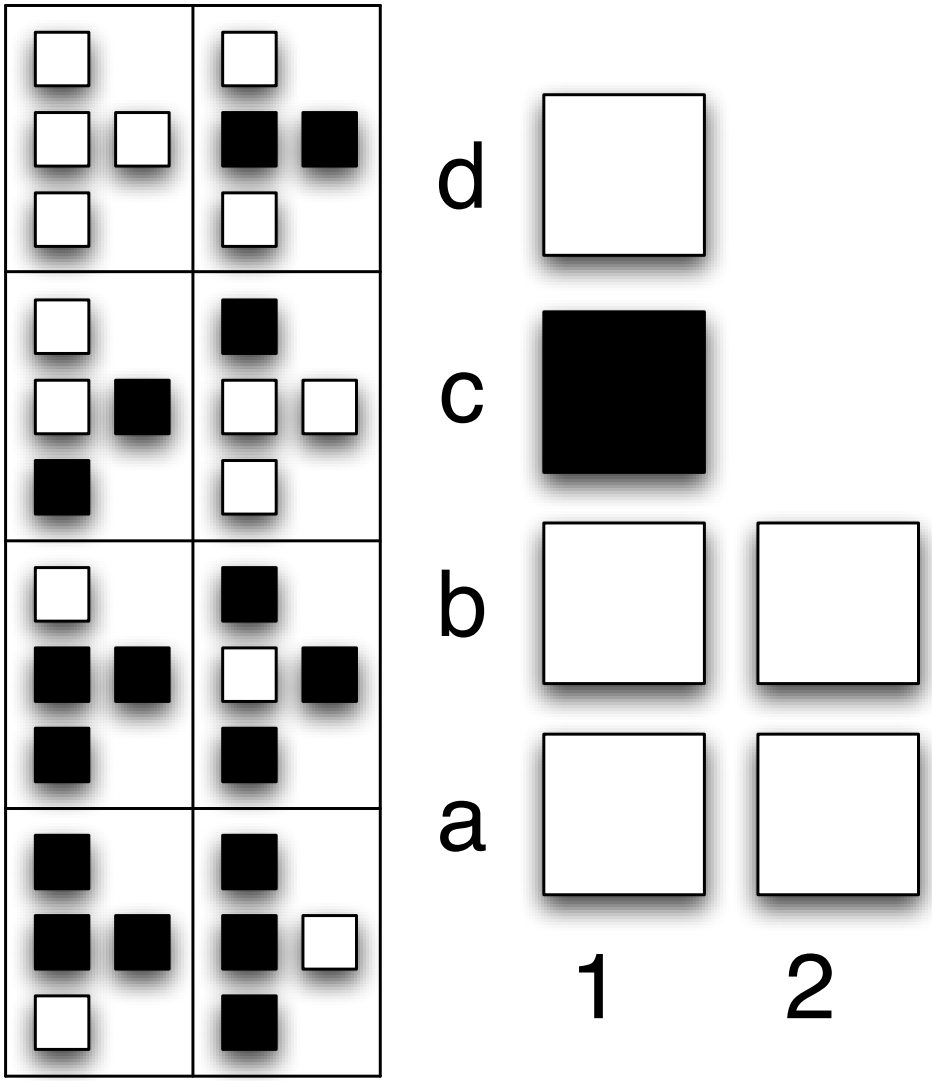
Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.



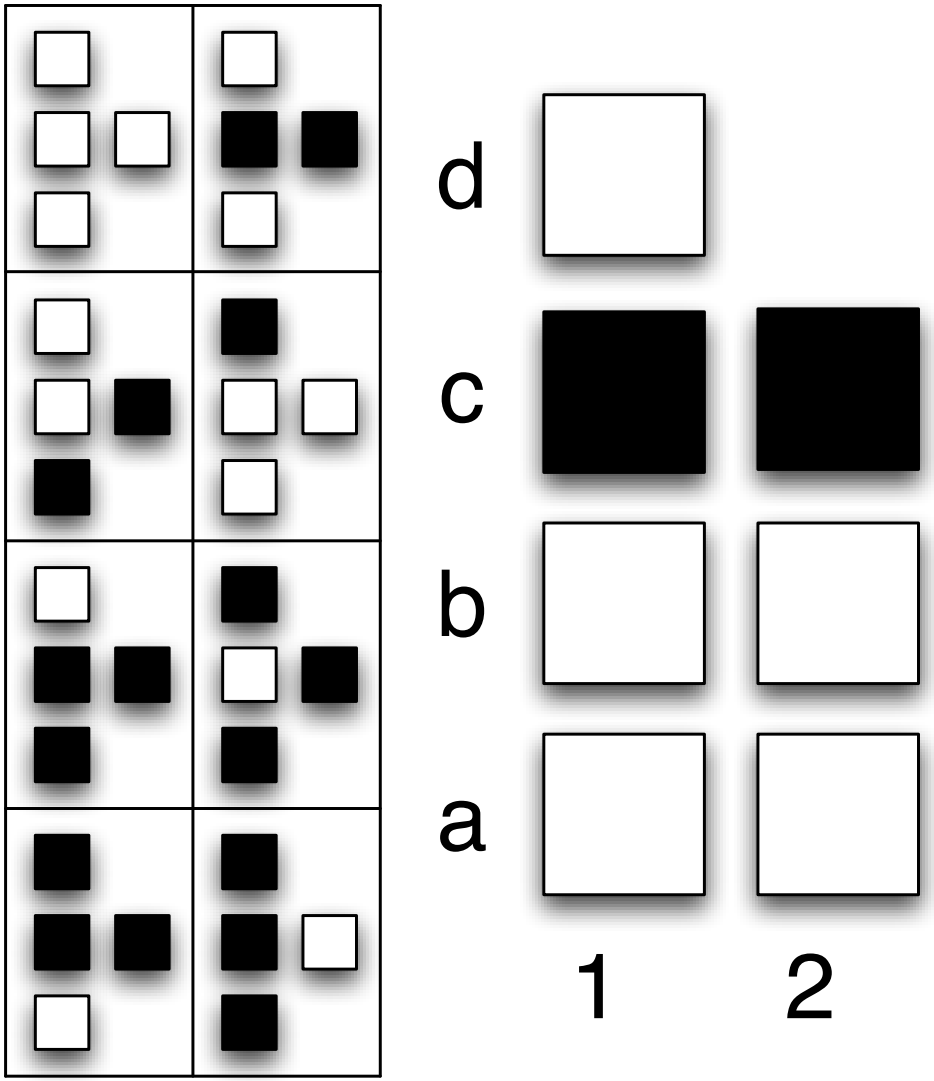
Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.



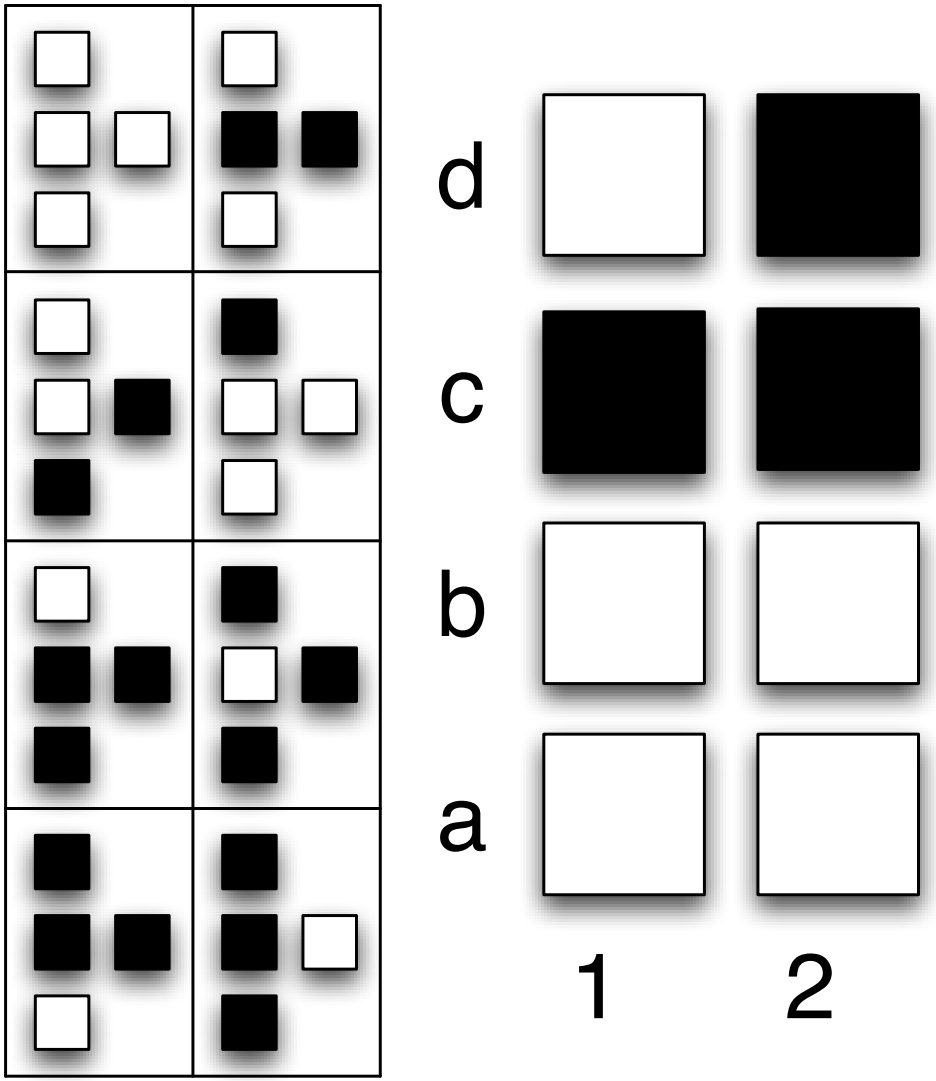
Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.



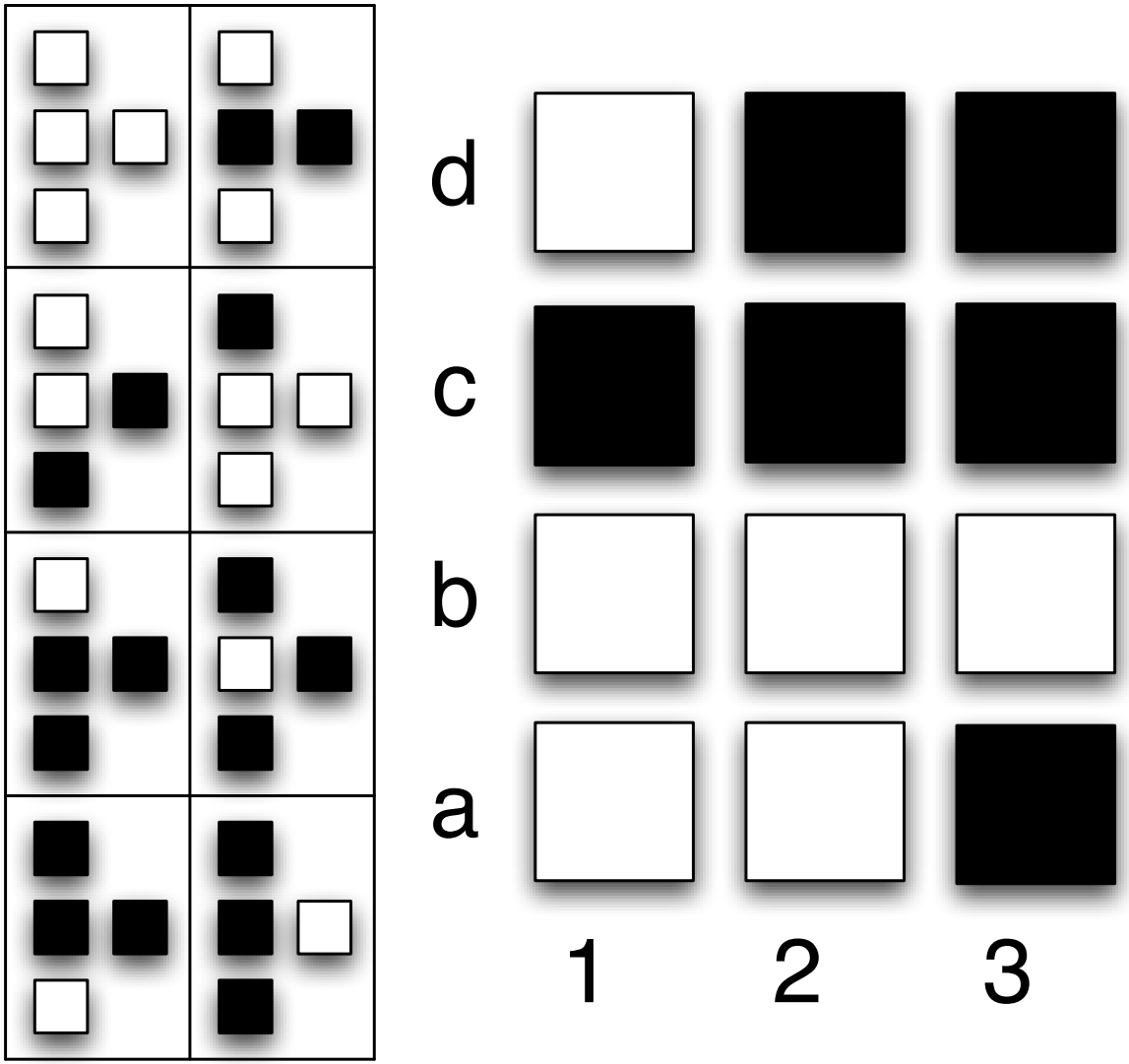
Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.



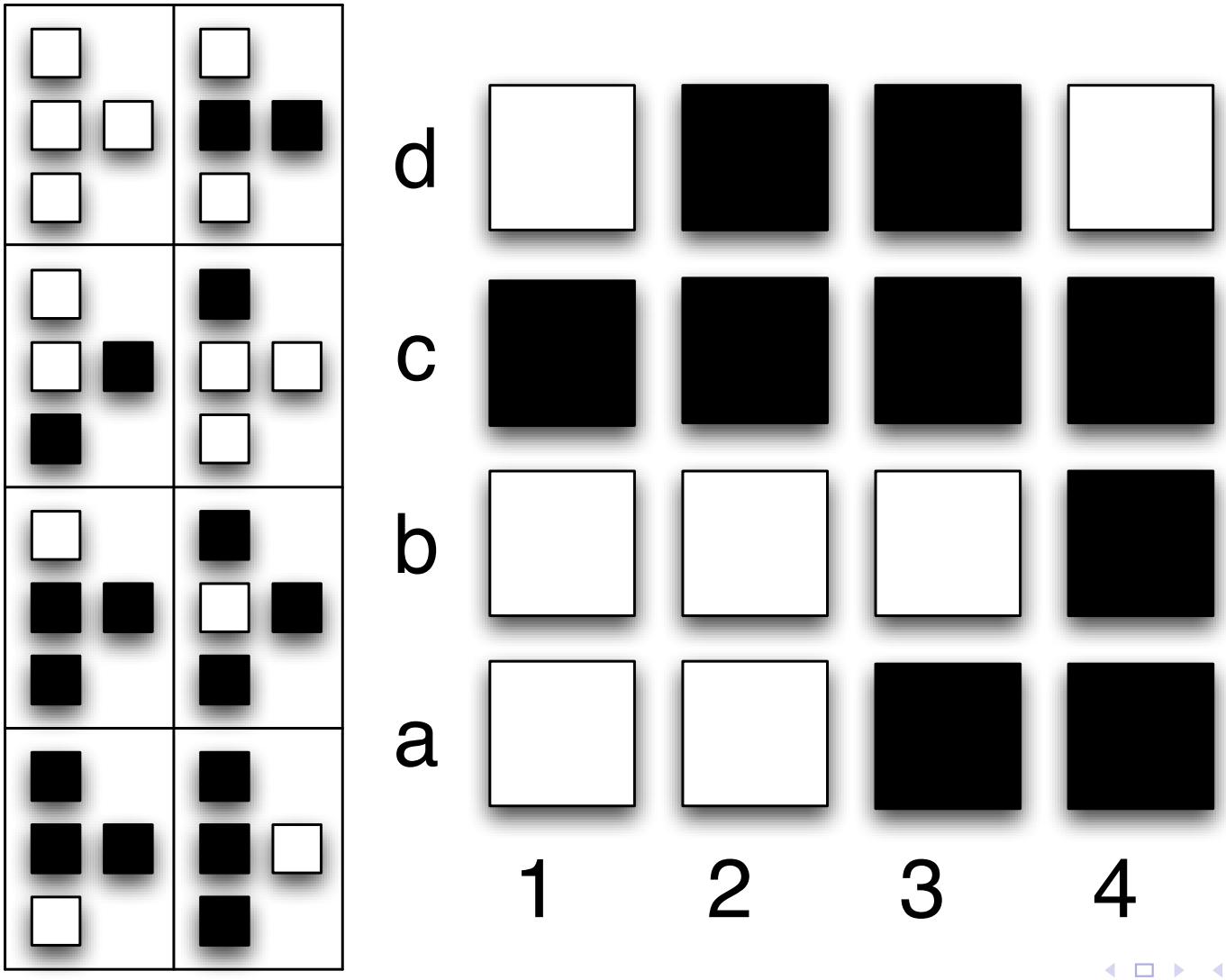
Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.



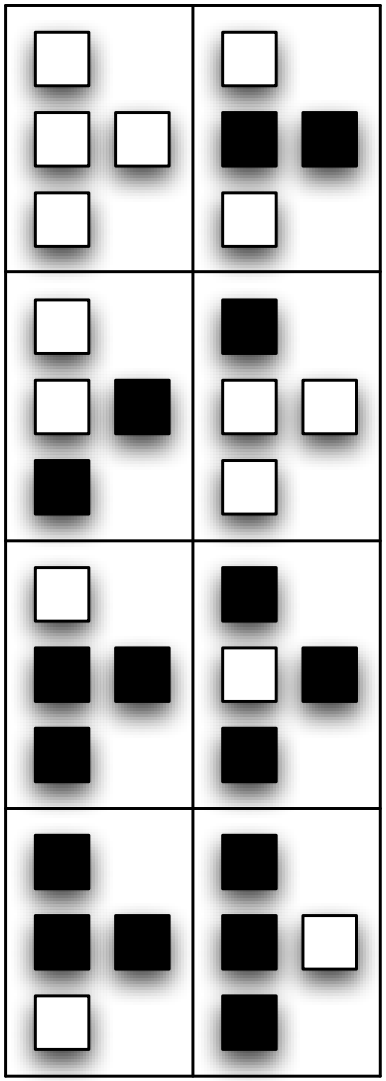
Cellular Automaton

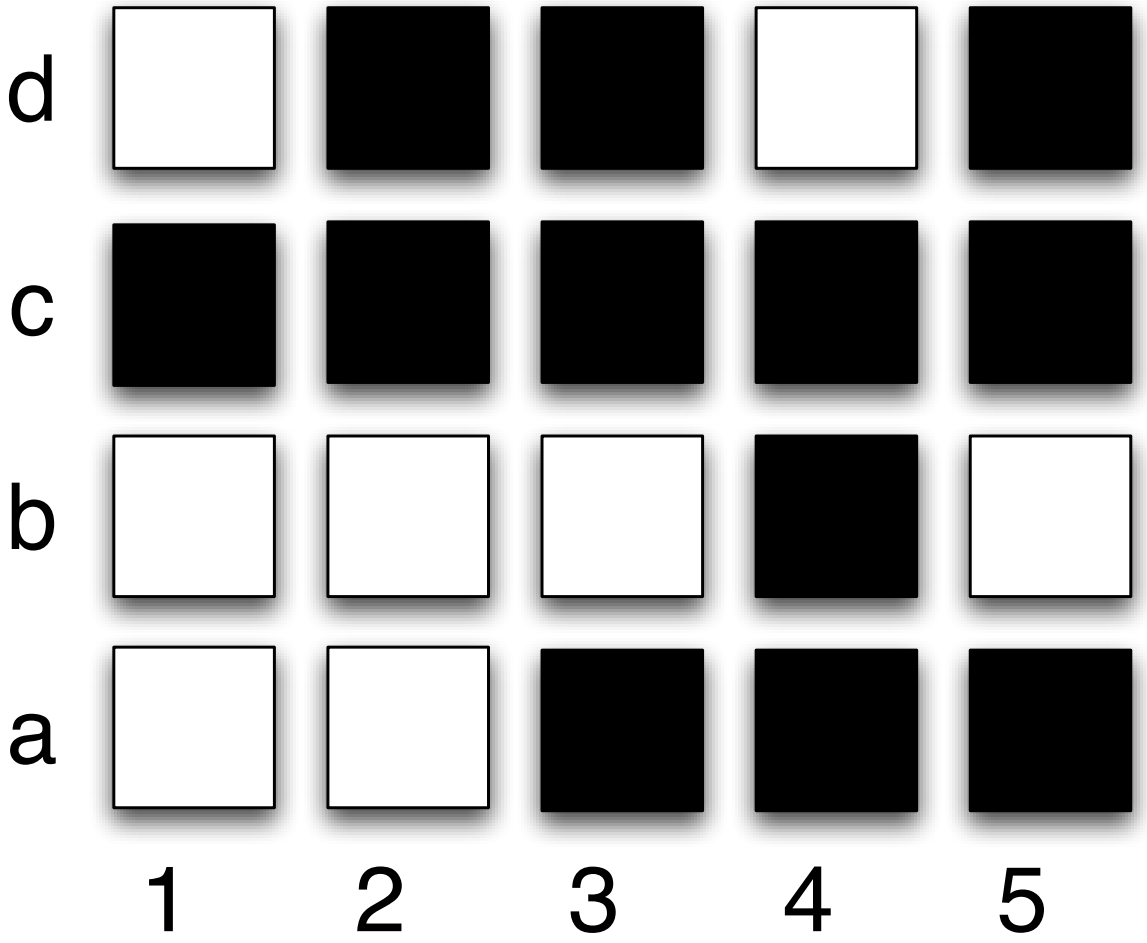
Writing Wolfram's celluar automaton Rule 110 in CHR.



Cellular Automaton

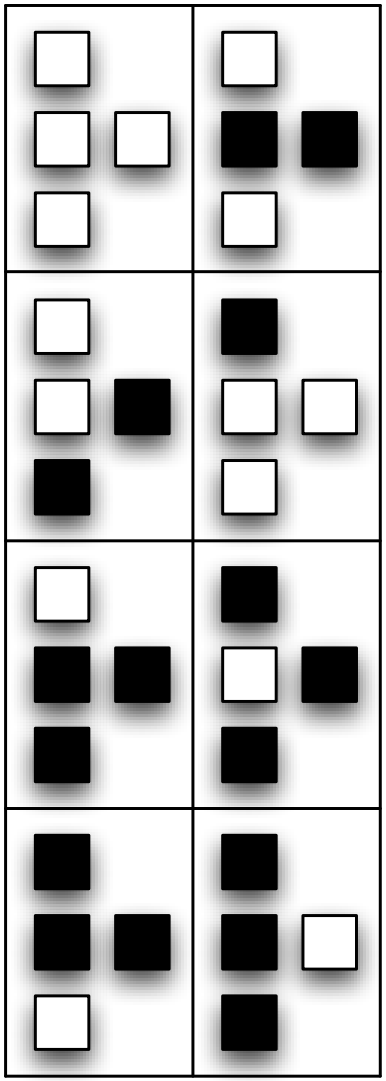
Writing Wolfram's celluar automaton Rule 110 in CHR.

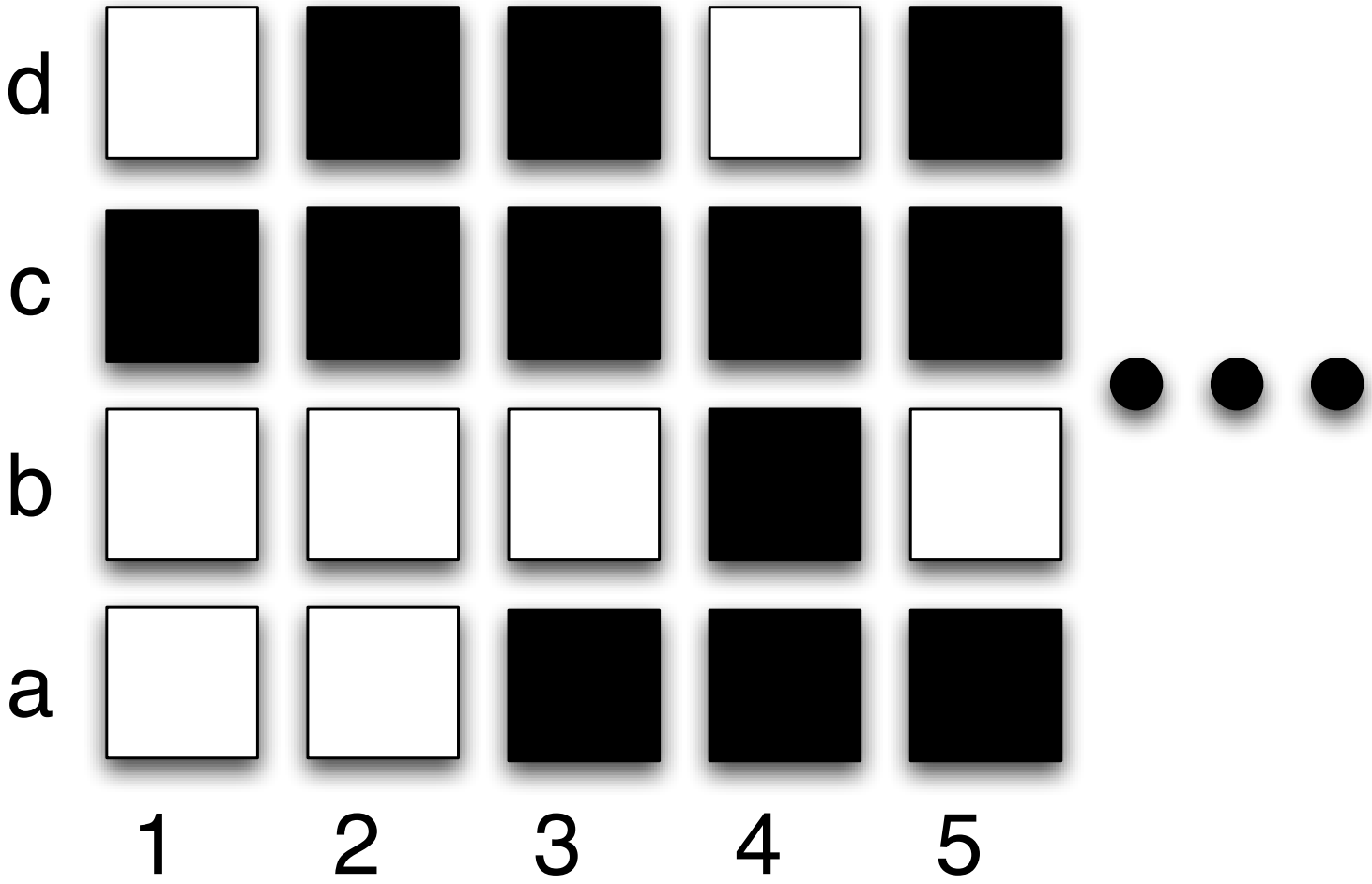




Cellular Automaton

Writing Wolfram's celluar automaton Rule 110 in CHR.





Cellular Automaton

CHR cells

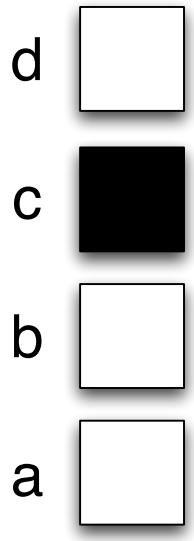
```
% black(Name,Below,Above,Generation).
:- chr_constraint black/4.

% white(Name,Below,Above,Generation).
:- chr_constraint white/4.
```

Cellular Automaton

CHR cells

```
% black(Name,Below,Above,Generation).
:- chr_constraint black/4.

% white(Name,Below,Above,Generation).
:- chr_constraint white/4.
```

```
?- white(d,c,a,1),
   black(c,b,d,1),
   white(b,a,c,1),
   white(a,d,b,1).
```



1

Automaton Rule

```
white(C,_,_,G) ,
black(B,A,C,G) ,
white(A,_,_,G)
   ==> G < 10 | NG is G + 1, black(B,A,C,NG).
```

Automaton Rule

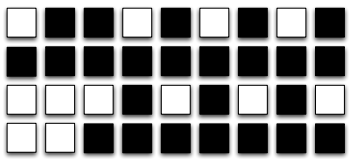
```
white(C,_,_,G) ,
black(B,A,C,G) ,
white(A,_,_,G)
  ==> G < 10 | NG is G + 1, black(B,A,C,NG).
```

```
?- white(d,c,a,1),
   black(c,b,d,1),
   white(b,a,c,1),
   white(a,d,b,1).
```

Automaton Rule

```
white(C,_,_,G) ,
black(B,A,C,G) ,
white(A,_,_,G)
  ==> G < 10 | NG is G + 1, black(B,A,C,NG).
```

```
?- white(d,c,a,1),
   black(c,b,d,1),
   white(b,a,c,1),
   white(a,d,b,1).
```



Another Example of Propagation

```
generate(N) ==> value(2).
```

Another Example of Propagation

```
generate(N) ==> value(2).

generate(N), value(I) ==>
        I < N | J is I + 1, value(J).
```

Another Example of Propagation

```
generate(N) ==> value(2).

generate(N), value(I) ==>
        I < N | J is I + 1, value(J).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

```
?- generate(10).
```

Another Example of Propagation

```
generate(N) ==> value(2).

generate(N), value(I) ==>
        I < N | J is I + 1, value(J).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

```
?- generate(10).
value(2)
value(3)
value(5)
value(7)
generate(10)
```

Summary

Simplification Rule: *head* <=> *guard* | *body* .

- replace head with body

Simpagation Rule: $head_k \backslash head_r$ <=> *guard* | *body* .

- replace $head_r$ with *body*
- in the presence of $head_k$

Propagation Rule: *head* ==> *guard* | *body* .

- add *body*
- in the presence of *head*
- fires once for each combination (propagation history)

Biased Coin?

```
coin <=> heads.
coin <=> tails.
```

What's the outcome?

```
?- coin.
```

Biased Coin?

```
coin <=> heads.
coin <=> tails.
```

What's the outcome?

```
?- coin.
heads
```

Biased Coin?

```
coin <=> heads.
coin <=> tails.
```

What's the outcome?

```
?- coin.
heads
```

Rules are tried in order!

Coin Flipping in Prolog

CHR	Prolog
`coin <=> heads.`	`p_coin :- heads.`
`coin <=> tails.`	`p_coin :- tails.`

What's the outcome?

Coin Flipping in Prolog

CHR	Prolog
`coin <=> heads.`	`p_coin :- heads.`
`coin <=> tails.`	`p_coin :- tails.`

What's the outcome?

```
?- coin.
heads
```

Coin Flipping in Prolog

CHR

```
coin <=> heads.
coin <=> tails.
```

Prolog

```
p_coin :- heads.
p_coin :- tails.
```

What's the outcome?

```
?- coin.
heads
```

```
?- p_coin.
heads
```

Coin Flipping in Prolog

CHR	Prolog
`coin <=> heads.`	`p_coin :- heads.`
`coin <=> tails.`	`p_coin :- tails.`

What's the outcome?

```
?- coin.
heads
```

```
?- p_coin.
heads ; y
```

Coin Flipping in Prolog

CHR	Prolog
`coin <=> heads.`	`p_coin :- heads.`
`coin <=> tails.`	`p_coin :- tails.`

What's the outcome?

```
?- coin.
heads
```

```
?- p_coin.
heads ; y

tails
```

Coin Flipping in Prolog

CHR	Prolog
`coin <=> heads.`	`p_coin :- heads.`
`coin <=> tails.`	`p_coin :- tails.`

What's the outcome?

```
?- coin.
heads
```

```
?- p_coin.
heads ; y

tails
```

- Prolog: backtracking
- CHR: **committed choice** for simplification rules

Propagation Coin Flipping

```
coin ==> heads.
coin ==> tails.
```

What's the outcome?

```
?- coin.
heads
tails
coin
```

Propagation Coin Flipping

```
coin ==> heads.
coin ==> tails.
```

What's the outcome?

```
?- coin.
heads
tails
coin
```

Rules are applied in sequence.

Propagation Coin Flipping

```
coin ==> heads.
coin <=> tails.
coin ==> side.
```

What's the outcome?

```
?- coin.
heads
tails
```

Propagation Coin Flipping

```
coin ==> heads.
coin <=> tails.
coin ==> side.
```

What's the outcome?

```
?- coin.
heads
tails
```

Rules are applied in sequence, until the active constraint is removed.

Exploiting Rule Order

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

Exploiting Rule Order

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

vs.

```
generate(0) <=> true.
generate(N) <=> value(N),
                M is N - 1, generate(M).
```

Exploiting Rule Order

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

vs.

```
generate(0) <=> true.
generate(N) <=> value(N),
                M is N - 1, generate(M).
```

for queries `?- generate(N).` with $N \geq 0$

Exploiting Rule Order

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

vs.

```
generate(0) <=> true.
generate(N) <=> value(N),
                M is N - 1, generate(M).
```

for queries `?- generate(N).` with $N \geq 0$

Better Style?

Exploiting Rule Order

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).
```

vs.

```
generate(0) <=> true.
generate(N) <=> value(N),
                M is N - 1, generate(M).
```

for queries `?- generate(N).` with $N \geq 0$

Better Style?

Compare to:

```
generate(0) :- !.
generate(N) :- value(N), M is N - 1, generate(M).
```

Summary

Rule Order

- rules are tried from top to bottom
- applied in sequence
- until the active constraint is removed
- committed choice: no alternatives explored for simplification rules

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
?- c(X).
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
?- c(X).
c(X)
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
?- c(X).
c(X)
?- c(X), X = hello.
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
?- c(X).
c(X)
?- c(X), X = hello.
hello
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
?- c(X).
c(X)
?- c(X), X = hello.
hello
?- c(X), c(world), X = hello.
```

Hello, World!

```
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).
```

```
?- c(hello), c(world).
hello
world
?- c(X).
c(X)
?- c(X), X = hello.
hello
?- c(X), c(world), X = hello.
world
hello
X = hello
```

Reactivation

- constraints suspend in the constraint store
- unification reactivates suspended constraints

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(A,B), A = B.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
No
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
No

?- neq(A,B), A = a, B = b.
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
No

?- neq(A,B), A = a, B = b.
Yes
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
No

?- neq(A,B), A = a, B = b.
Yes

?- neq(A,B), A = f(C), B = f(D).
```

A Constraint: Inequality

```
neq(X,X) <=> fail.
neq(X,Y) <=> X \= Y | true.
```

```
?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)
```

```
?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
No

?- neq(A,B), A = a, B = b.
Yes

?- neq(A,B), A = f(C), B = f(D).
neq(f(C),f(D))
```

Using Reactivation for Finite Domains

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

Using Reactivation for Finite Domains

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
```

Using Reactivation for Finite Domains

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
```

Using Reactivation for Finite Domains

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
```

Using Reactivation for Finite Domains

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
```

Using Reactivation for Finite Domains

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
```

Using Reactivation for Finite Domains

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
domain(X,[s,z])
```

Using Reactivation for Finite Domains

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
domain(X,[s,z])
?- domain(X,[d,f,h,s,z]), domain(X,[c,z]).
```

Using Reactivation for Finite Domains

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
```

```
?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
domain(X,[s,z])
?- domain(X,[d,f,h,s,z]), domain(X,[c,z]).
X = z
```

Summary

Reactivation

- rules may not fire because of lack of instantiation

Summary

Reactivation

- rules may not fire because of lack of instantiation
- upon instantiation, the rule may now fire

Summary

Reactivation

- rules may not fire because of lack of instantiation
- upon instantiation, the rule may now fire
- useful for implementing constraint solvers

Summary: CHR vs Prolog

	Prolog	CHR
heads	1	≥ 1
rule selection	unification	matching & guard
different rules	alternatives/backtracking	try all in sequence
no rule	failure	delay

CHR and Backtracking

```
p :- a.          a <=> c1.          b <=> d1.
p :- b.          a <=> c2.          b <=> d2.
```

```
?- p.
c1
```

CHR and Backtracking

```
p :- a.          a <=> c1.          b <=> d1.
p :- b.          a <=> c2.          b <=> d2.
```

```
?- p.
c1   ; y
d1
```

CHR and Backtracking

```
p :- a.          a <=> c1.          b <=> d1.
p :- b.          a <=> c2.          b <=> d2.
```

```
?- p.
c1   ; y
d1
```

- Prolog creates choicepoints
- CHR does not
- Prolog backtracking undoes CHR changes

Backtracking for Labeling

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

split(X) <=> ground(X) | true.
split(X), domain(X,[V|Vs]) <=> X = V ; domain(X,Vs).
```

```
?- domain(X,[d,z]), split(X).
```

Backtracking for Labeling

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

split(X) <=> ground(X) | true.
split(X), domain(X,[V|Vs]) <=> X = V ; domain(X,Vs).
```

```
?- domain(X,[d,z]), split(X).
X = d  ; y
X = z
```

Backtracking for Labeling

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

split(X) <=> ground(X) | true.
split(X), domain(X,[V|Vs]) <=> X = V ; domain(X,Vs).
```

```
?- domain(X,[d,z]), split(X).
X = d  ; y
X = z

?- domain(X,[d,f,z]), split(X).
```

Backtracking for Labeling

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

split(X) <=> ground(X) | true.
split(X), domain(X,[V|Vs]) <=> X = V ; domain(X,Vs).
```

```
?- domain(X,[d,z]), split(X).
X = d  ; y
X = z

?- domain(X,[d,f,z]), split(X).
X = d  ; y
domain(X,[f,z])
```

Backtracking for Labeling

```
domain(X,L)    <=> ground(X) | memberchk(X,L).
domain(X,[])   <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

indomain(X) <=> ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn]) <=> X = V1;...;X = Vn.
```

Backtracking for Labeling

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

indomain(X) <=> ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn]) <=> X = V1;...;X = Vn.
indomain(X), domain(X,Vs) <=> member(X,Vs).
```

Backtracking for Labeling

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

indomain(X) <=> ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn]) <=> X = V1;...;X = Vn.
indomain(X), domain(X,Vs) <=> member(X,Vs).
```

```
?- domain(X,[d,f,z]), indomain(X).
```

Backtracking for Labeling

```
domain(X,L)   <=> ground(X) | memberchk(X,L).
domain(X,[])  <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
        intersection(L1,L2,L), domain(X,L).

indomain(X) <=> ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn]) <=> X = V1;...;X = Vn.
indomain(X), domain(X,Vs) <=> member(X,Vs).
```

```
?- domain(X,[d,f,z]), indomain(X).
X = d  ; y
X = f  ; y
X = z
```

Summary

You should now have an understanding of:

Summary

You should now have an understanding of:

- how CHR works.

Summary

You should now have an understanding of:

- how CHR works.
- how CHR differs from Prolog.

Summary

You should now have an understanding of:

- how CHR works.
- how CHR differs from Prolog.
- that CHR is not exclusively about constraints .

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

2002-2005 **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

2002-2005 **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers

2004 **refined semantics**, Gregory Duck et al.

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

2002-2005 **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers

2004 **refined semantics**, Gregory Duck et al.

2004 1st **CHR workshop**

History: Programming Highlights

1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

2002-2005 **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers

2004 **refined semantics**, Gregory Duck et al.

2004 1st **CHR workshop**

2005-2008 **computational complexity**, PhD Jon Sneyers

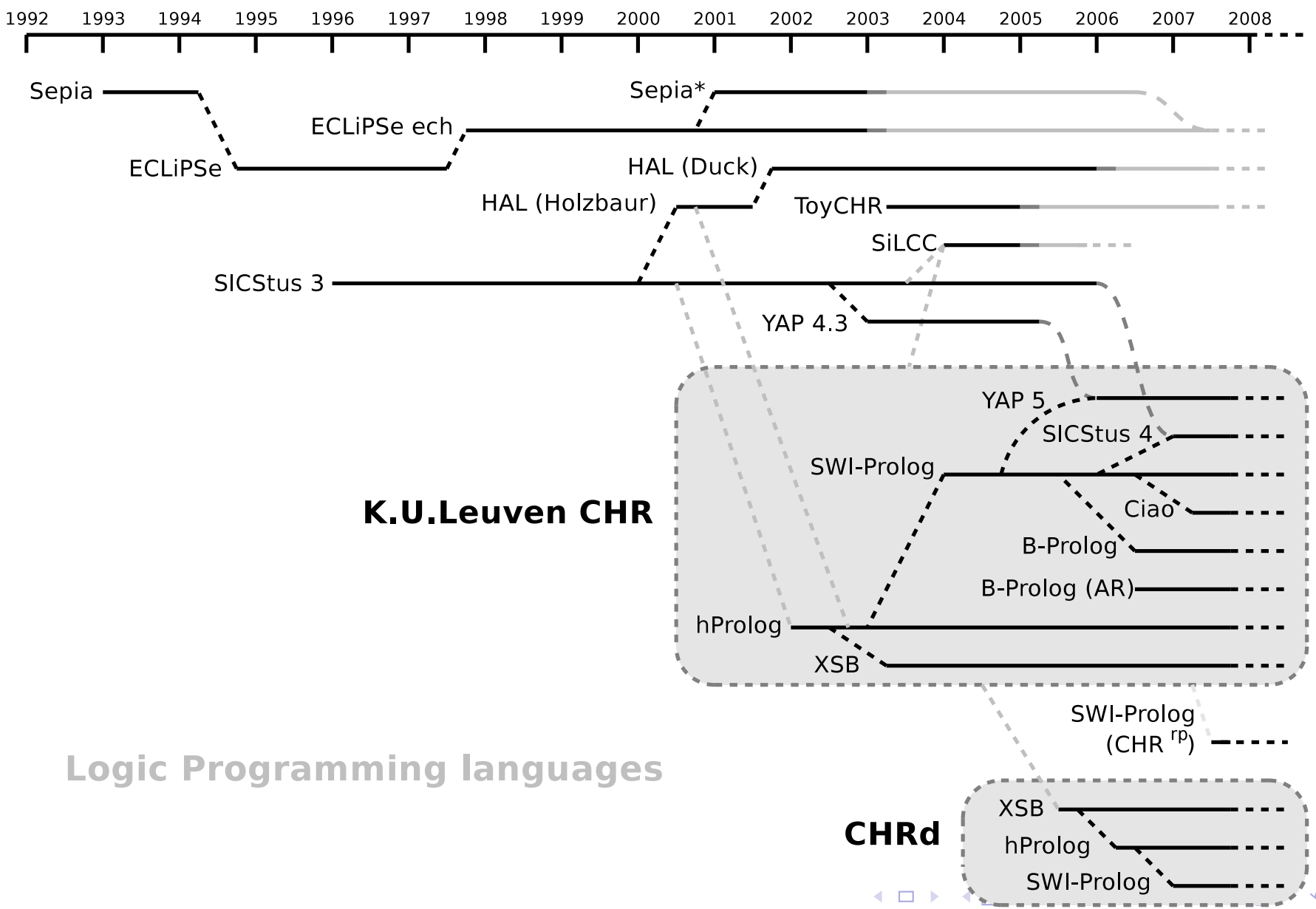
History: Programming Highlights

1991	**CHR** is born, Thom Frühwirth
1995	Christian Holzbaur implements **CHR(SICStus)**
2002	**Leuven CHR** is born
2002-2005	**optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers
2004	**refined semantics**, Gregory Duck et al.
2004	1st **CHR workshop**
2005-2008	**computational complexity**, PhD Jon Sneyers
2005-	Leuven **JCHR** (Java), Peter Van Weert

History: Programming Highlights

1991	**CHR** is born, Thom Frühwirth
1995	Christian Holzbaur implements **CHR(SICStus)**
2002	**Leuven CHR** is born
2002-2005	**optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers
2004	**refined semantics**, Gregory Duck et al.
2004	1st **CHR workshop**
2005-2008	**computational complexity**, PhD Jon Sneyers
2005-	Leuven **JCHR** (Java), Peter Van Weert
2007	first **concurrent system**, Sulzmann & Lam

CHR Systems


1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
Sepia
ECLiPSe
ECLiPSe ech
Sepia*
HAL (Duck)
HAL (Holzbaur)
ToyCHR
SiLCC
SICStus 3
YAP 4.3
K.U.Leuven CHR
YAP 5
SICStus 4
SWI-Prolog
Ciao
B-Prolog
B-Prolog (AR)
hProlog
XSB
SWI-Prolog
(CHR rp)
Logic Programming languages
CHRd
XSB
hProlog
SWI-Prolog

CHR Programs

Thom Frühwirth, with his students:

- lots of example programs, constraint solvers and others

Thom Frühwirth & Tom Schrijvers,

- union-find

Jon Sneyers,

- Fibonacci heaps and Dijkstra's shortest path
- Hopcroft's DFA minimization
- Turing and RAM machine simulators

Henning Christiansen,

- meta-programming in CHR

...

Declarative Semantics

- classical first-order logic (Frühwirth)
- linear logic (Betz)

CHR Theory & Applications

Declarative Semantics

- classical first-order logic (Frühwirth)
- linear logic (Betz)

Rewriting Properties

- confluence (Frühwirth;Duck;Haemmerle)
- completion (Frühwirth&Abdennadher)
- termination (Frühwirth;Voets&Pilozzi)
- complexity (Frühwirth;De Koninck)

CHR Theory & Applications

Declarative Semantics

- classical first-order logic (Frühwirth)
- linear logic (Betz)

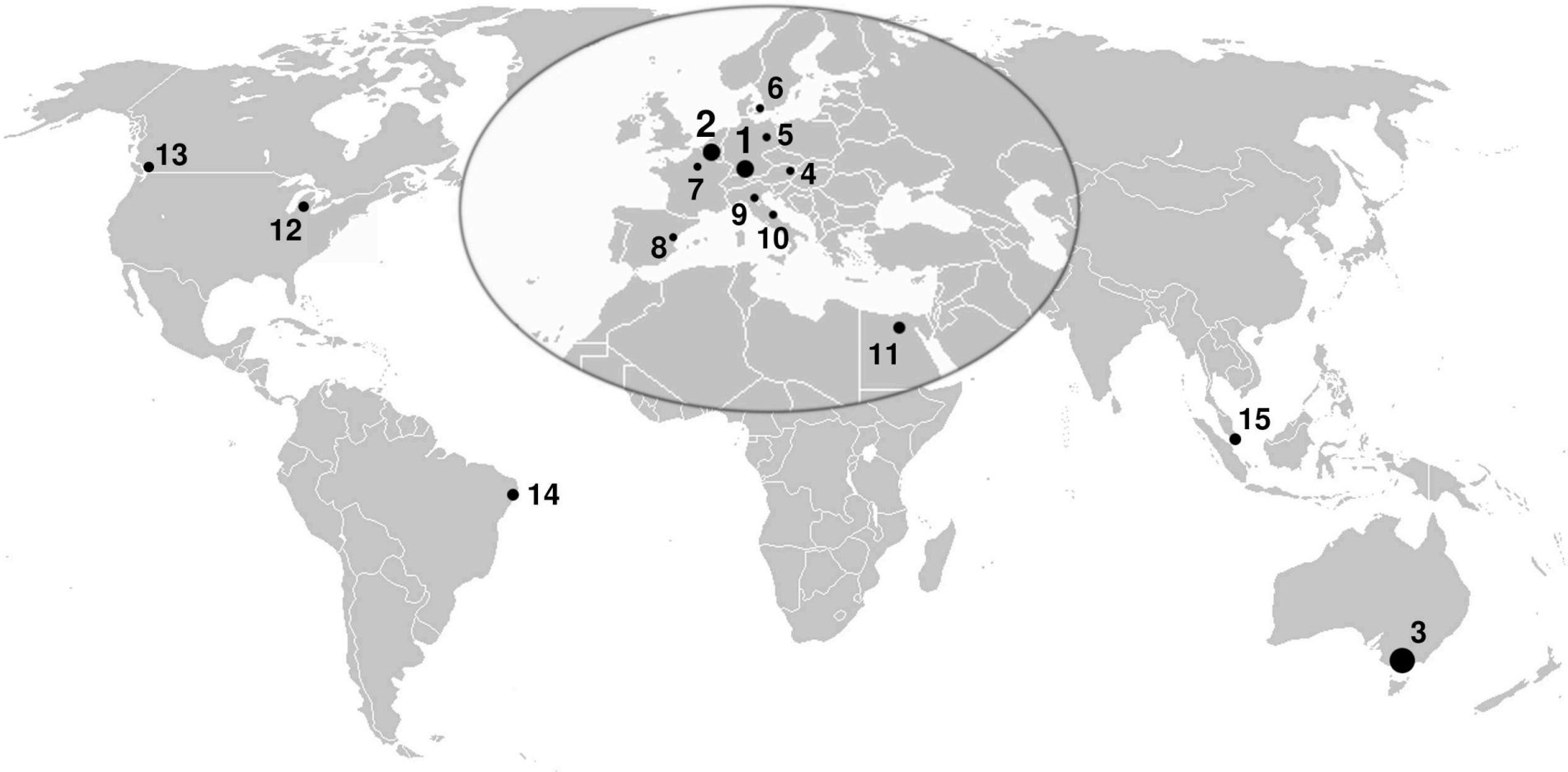
Rewriting Properties

- confluence (Frühwirth;Duck;Haemmerle)
- completion (Frühwirth&Abdennadher)
- termination (Frühwirth;Voets&Pilozzi)
- complexity (Frühwirth;De Koninck)

Applications

- type systems (Sulzmann & Stuckey)
- test case generation (Schrijvers)
- multi-agent systems (Alberti)
- ...

CHR Researchers around the World



CHR in Industry

- multi-headed business rules
- custom constraint solvers

CHR in Industry

- multi-headed business rules
- custom constraint solvers

- Scientific Software & Systems Ltd.
  - stock brokering software

- Cornerstone Technology Inc
  - injection mould design tool

- BSSE System and Software Engineering
  - test generation

- MITRE Corporation
  - optical network design

Further Reading

- CHR website (Google for)
  - programs
  - papers
- CHR Survey
  - As Time Goes By: Constraint Handling Rules – A Survey of CHR Research from 1998 to 2007, J. Sneyers, P. Van Weert, T. Schrijvers, L. De Koninck.

Thank You!

My Questions: Get the Quiz!

Your Questions?