Prolog
Programming in Logic

Lecture #2

Ian Lewis, Andrew Rice

Video/Lesson Recap

Lecture 1:

- **Video 1: Prolog Basics**
  - *Style* (Imperative, Functional, Logic)
  - *Facts*
  - *Queries*
  - *Terms* (constants/atoms, Variables, compound)
  - *Unification*

Lecture 2:

- **Video 2: Logic Puzzle** (zebra) - 5 houses, patterns
  Facts + *Unification*++
- **Video 3: Rules:** Head, Body, Recursion.
- **Video 4: Lists:** [ ], [a], [a|T], [a,b|T]

Any questions from the FIRST lecture and video?

1. Interacting with the Prolog interpreter e.g. **[consult].** , ‘**,**’ and, ‘**;**’ or/next, ‘**.**’ stop.
2. The succeed/true, fail/false Closed-World of Prolog.
3. Prolog terms (atoms, variables, compound).
4. Unification.

Course Outline

1. Introduction, terms, facts, unification
2. Unification. Rules. Lists.
3. Arithmetic, Accumulators, Backtracking
4. Generate and Test
5. Extra-logical predicates (cut, negation, assert)
6. Graph Search
7. Difference Lists
8. Wrap Up.

Today's discussion

Videos:

- Solving a logic puzzle
- Prolog rules
- Lists

Where’s the Zebra ?

There are five houses.
The Englishman lives in the red house.
The Spaniard owns the dog.
Coffee is drunk in the green house.
The Ukrainian drinks tea.
The green house is immediately to the right of the ivory house.
The Old Gold smoker owns snails.
Kools are smoked in the yellow house.
Milk is drunk in the middle house.
The Norwegian lives in the first house.
The man who smokes Chesterfields lives in the house next to the man with the fox.
Kools are smoked in the house next to the house where the horse is kept.
The Lucky Strike smoker drinks orange juice.
The Japanese smokes Parliaments.
The Norwegian lives next to the blue house.

Where's the Zebra ?

Represent houses as 5-tuple **(A,B,C,D,E)**.

Represent each house as **house(Nation,Pet,Smokes,Drinks,Colour)**

The Englishman lives in the red house.

can be represented with:

house(british, _, _, _, red).

Note we are structuring our COMPOUND TERMS here, not defining facts/rules. The similarity (and possible confusion) results from Prolog's symmetry between a PROGRAM and a TERM.

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
:- exists(house(british,_,_,_,red),Houses),
exists(house(spanish,dog,_,_,_),Houses),
exists(house(_,_,_,coffee,green),Houses),
exists(house(ukrainian,_,_,tea,_),Houses),
rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
exists(house(_,snail,oldgold,_,_),Houses),
exists(house(_,_,kools,_,yellow),Houses),
middleHouse(house(_,_,_,milk,_),Houses),
firstHouse(house(norwegian,_,_,_,_),Houses),
nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
exists(house(_,_,luckystrike,orangejuice,_),Houses),
exists(house(japanese,_,parliaments,_,_),Houses),
nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
exists(house(WaterDrinker,_,_,water,_),Houses),
exists(house(ZebraOwner,zebra,_,_,_),Houses),
print(ZebraOwner),nl,
print(WaterDrinker),nl.
```

Zebra puzzle

(If you haven't watched the video you'll be confused at this point)

1. You're not expected to be able to write that program **yet**.
2. The example uses only **facts** and **UNIFICATION**, without **lists** and **rules**.
3. Typical query term: The Spaniard owns the dog:
   exists(house(spanish,dog,Smokes,Drinks,Colour),Houses).

This 'exists' relation provides essential backtracking.

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
:- exists(house(british,_,_,_,red),Houses),
   exists(house(spanish,dog,_,_,_),Houses),
   exists(house(_,_,_,coffee,green),Houses),
   exists(house(ukranian,_,_,tea,_),Houses),
   rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
   exists(house(_,snail,oldgold,_,_),Houses),
   exists(house(_,_,kools,_,yellow),Houses),
   middleHouse(house(_,_,_,milk,_),Houses),
   firstHouse(house(norwegian,_,_,_,_),Houses),
   nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
   nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
   exists(house(_,_,luckystrike,orangejuice,_),Houses),
   exists(house(japanese,_,parliaments,_,_),Houses),
   nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
   exists(house(WaterDrinker,_,_,water,_),Houses),
   exists(house(ZebraOwner,zebra,_,_,_),Houses),
   print(ZebraOwner),nl,
   print(WaterDrinker),nl.
```

Zebra puzzle

```
exists(A, (A,_,_,_,_)).
exists(A, (_,A,_,_,_)).
exists(A, (_,_,A,_,_)).
exists(A, (_,_,_,A,_)).
exists(A, (_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
    exists(house(spanish,dog,_,_,_),Houses),
      ...
```

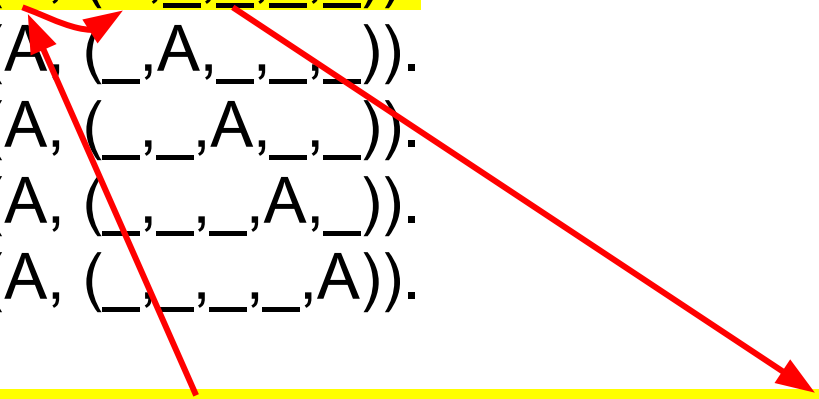
Zebra puzzle

```
exists(A, (A,_,_,_,_)).
exists(A, (_,A,_,_,_)).
exists(A, (_,_,A,_,_)).
exists(A, (_,_,_,A,_)).
exists(A, (_,_,_,_,A)).
```

```
?-  exists(house(british,_,_,_,red), Houses).
Houses = (house(british,_,_,_,red),_,_,_,_)
```

Zebra puzzle

```
exists(A, (A,_,_,_,_)).
exists(A, (_,A,_,_,_)).
exists(A, (_,_,A,_,_)).
exists(A, (_,_,_,A,_)).
exists(A, (_,_,_,_,A)).
```



```
:-  exists(house(british,_,_,_,red), Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
```

SUCCESS !!

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),Houses),
```

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),(house(british,_,_,_,red),_,_,_,_)),
```

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),(house(british,_,_,_,red),_,_,_,_)),
```

FAIL !!

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).
```

BACKTRACK / RETRY

```
:-  exists(house(british,_,_,_,red),Houses),
A = house(british,_,_,_,red),
Houses = (house(british,_,_,_,red),_,_,_,_)
    exists(house(spanish,dog,_,_,_),Houses),
    exists(house(spanish,dog,_,_,_), (house(british,_,_,_,red),_,_,_,_))
    exists(house(spanish,dog,_,_,_), (house(british,_,_,_,red), house(spanish,dog,_,_,_),_,_,_))
```

SUCCESS !!

Backtracking

Note that Prolog backtracked and retried the ‘Spanish’ house assignment, not the ‘British’.

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
:- exists(house(british,_,_,_,red),Houses),
   exists(house(spanish,dog,_,_,_),Houses),
   exists(house(_,_,_,coffee,green),Houses),
   exists(house(ukranian,_,_,tea,_),Houses),
   rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
   exists(house(_,snail,oldgold,_,_),Houses),
   exists(house(_,_,kools,_,yellow),Houses),
   middleHouse(house(_,_,_,milk,_),Houses),
   firstHouse(house(norwegian,_,_,_,_),Houses),
   nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
   nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
   exists(house(_,_,luckystrike,orangejuice,_),Houses),
   exists(house(japanese,_,parliaments,_,_),Houses),
   nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
   exists(house(WaterDrinker,_,_,water,_),Houses),
   exists(house(ZebraOwner,zebra,_,_,_),Houses),
   print(ZebraOwner),nl,
   print(WaterDrinker),nl.
```

Zebra puzzle

```
exists(A,(A,_,_,_,_)).
exists(A,(_,A,_,_,_)).
exists(A,(_,_,A,_,_)).
exists(A,(_,_,_,A,_)).
exists(A,(_,_,_,_,A)).

rightOf(A,B,(B,A,_,_,_)).
rightOf(A,B,(_,B,A,_,_)).
rightOf(A,B,(_,_,B,A,_)).
rightOf(A,B,(_,_,_,B,A)).

middleHouse(A,(_,_,A,_,_)).

firstHouse(A,(A,_,_,_,_)).

nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

GENERATE

```
:- exists(house(british,_,_,_,red),Houses),
   exists(house(spanish,dog,_,_,_),Houses),
   exists(house(_,_,_,coffee,green),Houses),
   exists(house(ukranian,_,_,tea,_),Houses),
   exists(house(_,snail,oldgold,_,_),Houses),
   exists(house(_,_,kools,_,yellow),Houses),
   exists(house(_,_,luckystrike,orangejuice,_),Houses),
   exists(house(japanese,_,parliaments,_,_),Houses),
   exists(house(WaterDrinker,_,_,water,_),Houses),
   exists(house(ZebraOwner,zebra,_,_,_),Houses),
```

TEST

```
   rightOf(house(_,_,_,_,green),house(_,_,_,_,ivory),Houses),
   middleHouse(house(_,_,_,milk,_),Houses),
   firstHouse(house(norwegian,_,_,_,_),Houses),
   nextTo(house(_,_,chesterfields,_,_),house(_,fox,_,_,_),Houses),
   nextTo(house(_,_,kools,_,_),house(_,horse,_,_,_),Houses),
   nextTo(house(norwegian,_,_,_,_),house(_,_,_,_,blue),Houses),
```

Course Outline

1. Introduction, terms, facts, unification
2. Unification. Rules. Lists.
3. Backtracking
4. Generate and Test
5. Extra-logical predicates (cut, negation, assert)
6. Graph Search
7. Difference Lists
8. Wrap Up.

Rules

Q: In the Zebra puzzle, why isn't the `rightOf` fact used help define the `nextTo` fact?

Improving on nextTo

```
nextTo(A,B,(A,B,_,_,_)).
nextTo(A,B,(_,A,B,_,_)).
nextTo(A,B,(_,_,A,B,_)).
nextTo(A,B,(_,_,_,A,B)).
nextTo(A,B,(B,A,_,_,_)).
nextTo(A,B,(_,B,A,_,_)).
nextTo(A,B,(_,_,B,A,_)).
nextTo(A,B,(_,_,_,B,A)).
```

```
nextTo(A,B,Houses) :- rightOf(A,B,Houses).

nextTo(A,B,Houses) :- rightOf(B,A,Houses).
```

Unification recap

Which of these are true statements

1. _ unifies with anything
2. 1+1 unifies with 2
3. prolog unifies with prolog
4. prolog unifies with java

Unification recap

Which of these are true statements

1. _ unifies with anything
2. 1+1 unifies with 2
3. prolog unifies with prolog
4. prolog unifies with java

What's the result of unifying:

cons(1,cons(X)) with

cons(1,cons(2,cons(Y)))

1. False: they don't unify
2. True: they unify
3. True: X is now cons(2,cons(Y))
4. True: X is now cons(1,cons(2,cons(Y)))

What's the result of unifying:

cons(1,cons(X)) with
cons(1,cons(2,cons(Y)))

1. False: they don't unify
2. True: they unify
3. True: X is now cons(2,cons(Y))
4. True: X is now cons(1,cons(2,cons(Y)))

cons(X) cannot unify with cons(2,cons(Y))

for the same reason, cons(X) cannot unify with cons(2,3)

Which of these is a list containing the numbers 1,2,3

1. [1 , 2 , 3]
2. [1 | [2 , 3]]
3. [1 | 2 , 3]
4. [1 , 2 | 3]
5. [1 , 2 | [3]]
6. [1 , 2 , 3 | []]

Which of these is a list containing the numbers 1,2,3

1. [1 , 2 , 3]
2. [1 | [2 , 3]]
3. [1 | 2 , 3]
4. [1 , 2 | 3]
5. [1 , 2 | [3]]
6. [1 , 2 , 3 | []]

Lists, Unification, and program termination

Q: I often write logically-correct code which doesn't terminate. What heuristics can I apply to see if this will happen without running the code?

Q: I often write logically-correct code which doesn't terminate. What heuristics can I apply to see if this will happen without running the code?

A: Its quite hard to do this without using things like arithmetic, but let's look at some examples now and then some more next time.

Does this program terminate?

```
a(X) :- a(X).
```

Does this program terminate?

```
a(X) :- a(X).
```

Yes! Trick question. This program doesn't have any queries in it...

Does this program terminate?

```
a(X) :- a(X).

:- a(1).
```

Does this program terminate?

```
a(X) :- a(X).

:- a(1).
```

NO.
In trying to ‘solve’ or ‘prove’ a(1), Prolog will unify X=1 in the single rule, and then try and prove a(1)...

Does this program terminate?

```
a([]).

a([_|T]) :- a(T).

:- X = <any_finite_list>, a(X).
```

Does this program terminate?

```
a([]).

a([_|T]) :- a(T).

:- X = <any_finite_list>, a(X).
```

Does this program terminate?

```
a([]).

a([_|T]) :- a(T).

:- X = <any_finite_list>, a(X).
```

YES. Recursive call is with shorter list.

More interesting query: `:- a(X).`

What does this print?

```
a([],R) :- print(R), a(R,[]).

a([H|T],R) :- a(T,[H|R]).

:- a([1,2,3],[]).
```

Does this terminate?

```
a([]) :- a([1|X]).

:- a([]).
```

Does this terminate?

```
a([]) :- a([1|X]).

:- a([]).
```

ABSOLUTELY! With fail/false.

In trying to prove a([]), Prolog tries to prove a([1|X]), and that fails to unify with any fact or rule.

Super-Heuristic - Determinism

```
last([H], H).

last([_|T], H) :- last(H,T).
```

(1) Call with ?- last([a,b,c],H).
    H = c.
(2) Call with ?- last(L, a).

Super-Heuristic - Determinism

```
last([H], H).

last([_|T], H) :- last(H,T).
```

(1) Call with ?- last([a,b,c],H).
H = c.

(2) Call with ?- last(L, a).
L = [a] ;
L = [_222, a] ;
L = [_333, _222, a] ...

Super-Heuristic - Determinism

```
len([], 0).

len([_|T], N) :- len(T,M), N is M + 1.
```

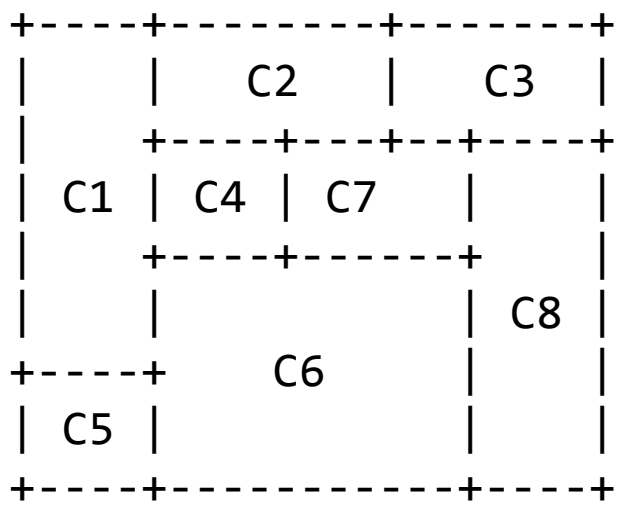
(1) Call with ?- len([a,b,c],N).
N = 3.

(2) Call with ?- len(L, 0).
L = [ ] ;
?

...

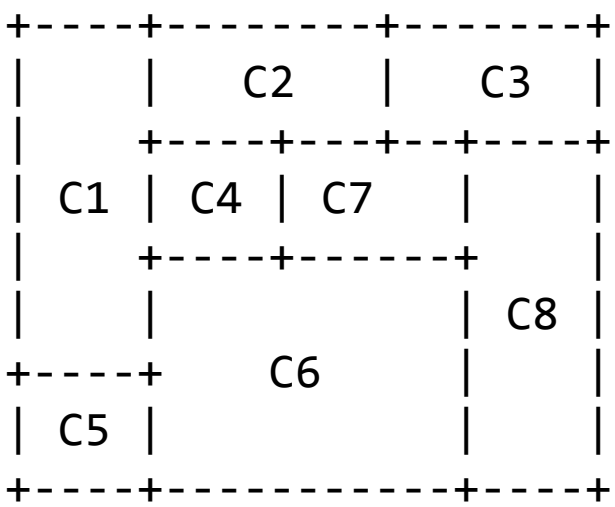
Today's programming challenge - Map colouring

Colour the regions shown below using four different colours so that no touching regions have the same colour.



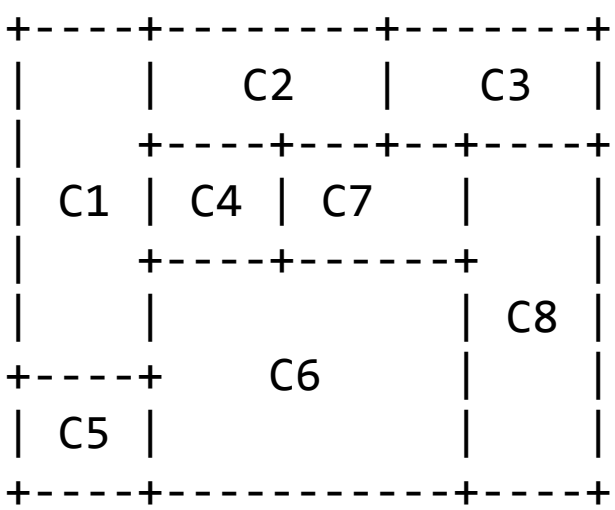
Hint 1: Write down what is true...

You have 4 colours and they are all different…



Hint 1: Write down what is true...

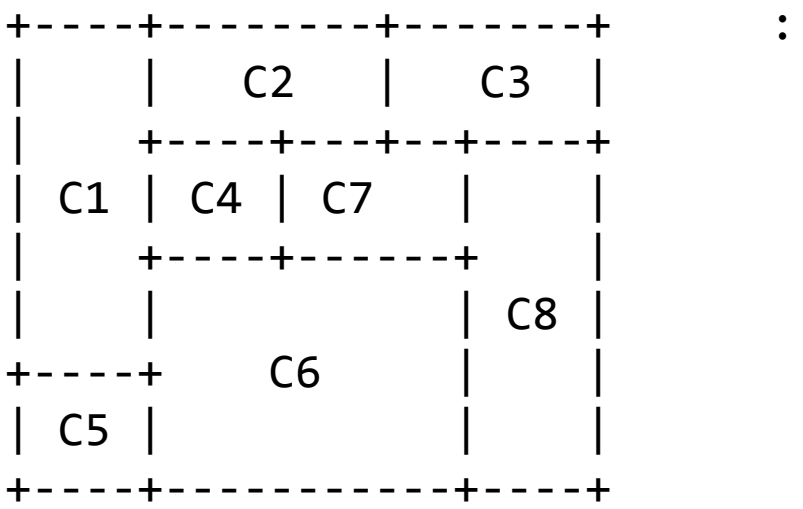
You have 4 colours and they are all different…




```
diff(red,green).
diff(red,blue).
diff(red,yellow).
diff(green,red).
diff(green,blue).
diff(green,yellow).
```
...etc...

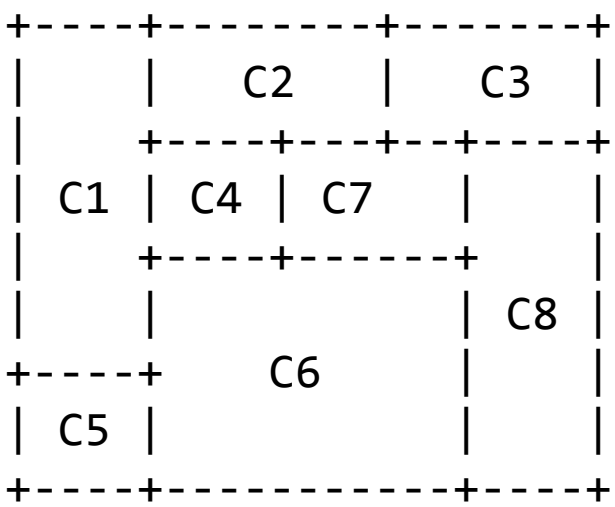
Hint 2: Ask for the answer

What colour does each region need to be so its different to its neighbours



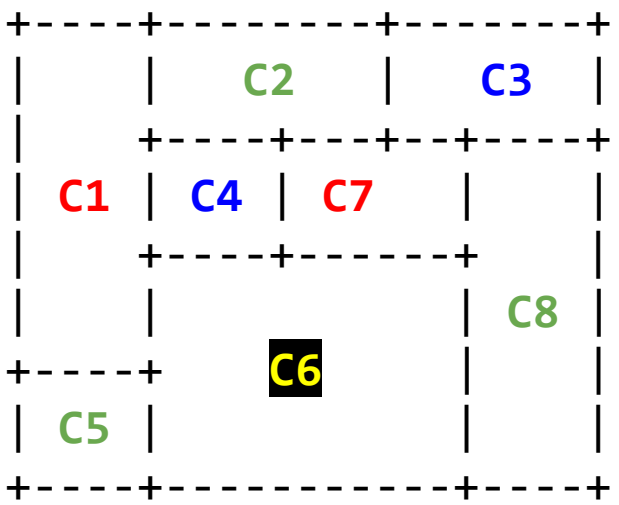
Hint 2: Ask for the answer

What colour does each region need to be so its different to its neighbours




```
:- diff(C1,C5),
   diff(C1,C2),
   diff(C1,C4),
   diff(C1,C6),
   diff(C2,C4),
   diff(C2,C7),
   ...etc...
```

Coloured map



Map colours

```
diff(X,Y) :- X \= Y.
color(red).
color(blue).
color(green).
color(yellow).
```

```
ans :- color(C1), color(C2), color(C3), color(C4), color(C5), color(C6), color(C7), color(C8),
  diff(C1,C5),
  diff(C1,C2),
  diff(C1,C4),
  diff(C1,C6),
  diff(C2,C3),
  diff(C2,C4),
  diff(C2,C7),
  diff(C3,C7),
  diff(C3,C8),
  diff(C4,C6),
  diff(C4,C7),
  diff(C5,C6),
  diff(C6,C7),
  diff(C6,C8),
  diff(C7,C8),
  print([C1,C2,C3,C4,C5,C6,C7,C8]).
```

Next time

Videos

- Arithmetic
- Backtracking