Declaratieve Talen

Prolog 4

1 Junior interpreter

Extend the mini-meta-interpreter from the lectures with support for the built-ins `>/2` and `is/2` that are required to interpret the `fib/2` predicate.

```
interpret((G1,G2)) :-
    !,
    interpret(G1),
    interpret(G2).

interpret(true) :- !.

interpret(Head) :-
    clause(Head,Body),
    interpret(Body).
```

The predicate `fib/2` is defined as follows:

```
fib(0,1).
fib(1,1).
fib(N,F) :-
    N > 1,
    N2 is N - 2,
    fib(N2,F2),
    N1 is N - 1,
    fib(N1,F1),
    F is F1 + F2.
```

Assignments

1. Extend the interpreter with support for the built-ins used by `fib/2`.
2. Extend the interpreter further with support for memoization as follows:
    - The interpreter has a memory that is initially empty.

- Each time the interpreter needs to call a predicate, it first verifies whether the answer is in its memory:
  - If the result of the call was saved in memory, return this answer.
  - If the result of the call was not saved in memory, call the predicate and save the answer to the memory before returning it.

Questions

- What is the time complexity of `fib/2` in SWI-Prolog?
- What is the time complexity of `fib/2` in your mini-meta-interpreter **without** as well as **with** memoization?
- How does your mini-meta-interpreter behave during backtracking? Or when the answers contain variables?

2 Holiday lights

The European government has decided to festively light major European highways during the holiday season. There will be differently colored lights, however each highway will have exactly one color. During the holiday season you plan some family visits, and you like some variety while driving. Therefore you decide to drive each highway exactly once, but two consecutive highways you drive can not be lit by the same color.

A more abstract representation of this problem is as follows. Assume M bidirectional connections that each connect two places, and are lit by a specific color. Assume N places, numbered from 1 through N. You start at place 1. Now you have to plan a trip such that each connection is used exactly once, no two consecutive connections have the same color, and your trip should end back at place 1.

Consider the following example. There are three places. There is a yellow highway between places 1 and 2, a blue highway between 2 and 3, and a yellow highway between 1 and 3. You can drive a yellow highway from 1 to 2, a blue highway from 2 to 3, and another yellow highway from 3 back to 1.

Such a graph can be represented by Prolog facts. For our example we can write:

```
highway(1,2,yellow).
highway(2,3,blue).
highway(1,3,yellow).
```

Before you start planning your trip, you need to check the following two conditions:

1. The nodes (i.e. places) have an even number of connections,

2. If a node K, such that $1 < K \leq N$, has X connections lit by the same color, then the node K should have at least X connections with other colors.

Write the predicate `check` which will check these two conditions for a network of highways given by `highway/3` facts. This predicate succeeds if both conditions are met, and fails otherwise.
Hint Decompose the check into two separate predicates, `check_even_at(N)` and `check_colors_at(N)`, verifying the first and second conditions at node `N`, respectively.
Hint Try to use negation to quickly check the conditions for every node. Remember that write the negation of a conjunction, you must use double parentheses (e.g., `\+( (p(X),q(Y)) )`).

Some examples:

```
highway(1,2,yellow).
highway(2,3,yellow).
highway(1,3,blue).

?- check.
fail.

highway(1,2,yellow).
highway(2,3,blue).
highway(1,3,yellow).

?-check.
true.
```

Write the predicate `tour(T)` which will first check the conditions listed above for a network of highways given by `highway/3` facts, and then calculates a trip. Our example has two possible trips: `T = [2-yellow, 3-blue, 1-yellow]` and `T = [3-yellow, 2-blue, 1-yellow]`. In such a case you only return the smallest trip: `T = [2-yellow, 3-blue, 1-yellow]`. We will use the order defined by `@</2`. Notice that this query `?- [2-yellow] @< [3-yellow]` succeeds.

The solution `T = [2-yellow, 3-blue, 1-yellow]` tells us we will drive a yellow highway from 1 to 2, then a blue highway from 2 to 3, and finally a yellow highway from 3 to 1.

The predicate `tour` will fail if `check` fails. Some examples:

```
highway(1,2,yellow).
highway(2,3,blue).
highway(1,3,yellow).

?- tour(T).
```

```
T = [2-yellow, 3-blue, 1-yellow]


highway(1,2,c).
highway(2,3,a).
highway(1,3,b).
highway(3,5,a).
highway(3,4,c).
highway(5,4,d).

?- tour(T).
T = [2-c, 3-a, 4-c, 5-d, 3-a, 1-b] ;
```

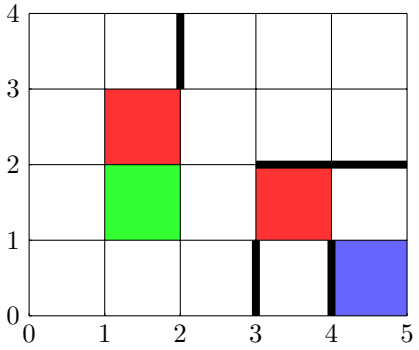
3 Extra: Tapis plein

Consider a factory that makes room-sized carpets for people's homes. These room-sized carpets are also called "tapis plein" in Flanders. Customers give a blueprint of their house and want the factory to produce carpet for the entire floor. The blueprint is given using Prolog facts `house/2`, `wall/4`, and `color/3`. The `house(xSize,ySize)` predicate describes the dimension of the house.

```
house(5,4).
```

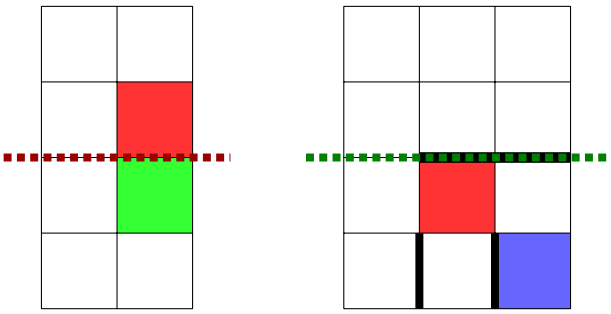
The `wall/4` predicate describes where the walls are: `wall(v,xCo,yCo1,yCo2)` facts describe vertical walls, `wall(h,yCo,xCo1,xCo2)` facts describe horizontal walls. The `color(xCo,yCo,color)` predicate describes which color of carpet is desired for some locations. An example of a blueprint is given in Figure 1.

```
house(5,4).

wall(v,2,3,4).
wall(v,3,0,1).
wall(v,4,0,1).
wall(h,2,3,5).

color(1,1,green).
color(1,2,red).
color(3,1,red).
color(4,0,blue).
```



Figuur 1: Example of a house blueprint

Since the carpet factory can only produce carpets of a rectangular form (any size), the house needs to be divided into rectangular parts. We'll call these divisions "cuts". Performing a cut is represented using `cut(v,xCo,yCo1,yCo2)`



Invalid cut Valid cut

Figuur 2: Resulting sub-blueprints after performing cut(v,2,0,4) on the example. Performing cut(h,2,0,2) has become invalid, since no wall is present anymore along this line in the left sub-blueprint

for vertical cuts and using `cut(h,yCo,xCo1,xCo2)`. Cutting a blueprint divides it up into two separate blueprints. Cuts always have to be performed along the entire (sub-)blueprint. Cuts can only be performed along a wall in the current (sub-)blueprint. Cuts are ordered:

- When a vertical cut is made, all cuts in the *left* sub-blueprint are done before cutting the *right* sub-blueprint.
- When a horizontal cut is made, all cuts in the *bottom* sub-blueprint are done before cutting the *top* sub-blueprint.

Note that there are cuts that become impossible after performing some cuts first. See the example in Figure 2 where for the house in Figure 1 a vertical cut along $x = 2$ is performed. Cutting horizontally for $y = 2$ in the left sub-blueprint has become impossible now, whilst it would have been possible in the original blueprint.
Performing the cuts results in a set of *parts* (rectangles) that cover the entire floor of the house. These parts are represented as a sorted (using `sort/2`) list of `part(xCo1,yCo1,xCo2,yCo2)` terms, where `(xCo1,yCo1)` represents the bottom left corner coordinate of the part and `(xCo2,yCo2)` represents the upper right corner.

Assignment 1 : write a predicate `wall_in_part(part(xCo1,yCo1,xCo2,yCo2),Wall)` such that for a given part (first argument), `Wall` unifies with a wall present in that part. Note: a wall that lies on the edge of a part does not count.

Assignment 2 : write a predicate `color_in_part(part(xCo1,yCo1,xCo2,yCo2),Color)` such that for a given part (first argument), `Color` unifies with a color preference present in that part.

Assignment 3 : based on the previous two assignments, write, for a given `Part`, the predicates `single_color_present(Part)` (succeeds if the part contains color preferences of at most a single color) and `part_has_wall(Part)` (succeeds if the part has at least one wall in it).

Assignment 4 : write a predicate `cut_plan(CuttingPlan,Parts)` that constructs a valid cutting plan that results in the subdivision in `Parts` for the house blueprint given in the facts. A cutting plan is represented as an ordered list of cuts, using the representation we described above.

```
CuttingPlan = [
  cut(h,2,0,5),
  cut(v,3,0,2),
  cut(v,4,0,2),
  cut(v,2,2,4)
]
```

A valid cutting plan has the following properties.

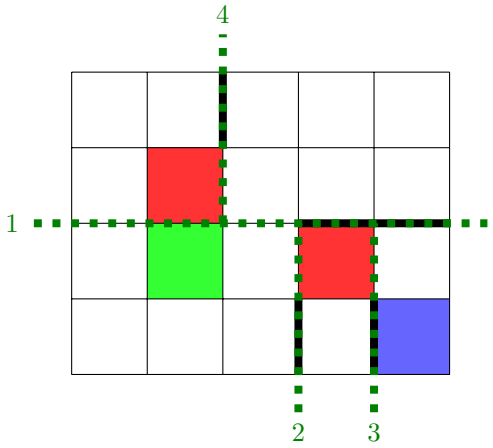
- Cuts go along the entire current (sub-)blueprint.
- Cuts go along at least one wall in the current (sub-)blueprint.
- Cuts are ordered with respect to the ordering discussed above.

Performing the cuts in `CuttingPlan` in-order results in a list of *parts* (rectangles) that cover the entire floor of the house. These parts are represented as a sorted (using `sort/2`) list of `part(xCo1,yCo1,xCo2,yCo2)` terms. `(xCo1,yCo1)` represent the bottom left corner coordinate of the part and `(xCo2,yCo2)` represent the upper right corner.

```
Parts = [
  part(0,0,3,2),
  part(0,2,2,4),
  part(2,2,5,4),
  part(3,0,4,2),
  part(4,0,5,2)
]
```

These parts have to satisfy the following constraints:

- Each part contains color preferences of at most a single color.
- Each part contains no walls.



Figuur 3: A valid cutting for the example. Cuts 2 and 3 have to be performed before cut 4, because of the order we imposed. The order of cut 2 and 3 does not matter and these can be swapped