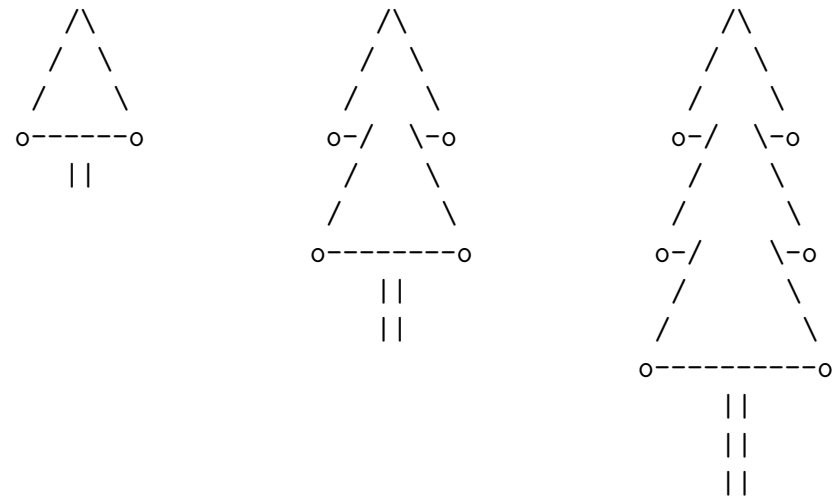
Prolog Programming Contest: 2 October 2005, Sitges

Prologue

There are 6 problems. The name of the file that contains your submitted solution must be the same as given in the header of the problem - this file must be readable for the organisers. So, for the first problem, you make a file named `xmas.pl`.
The *input* to your program will be in the form of Prolog facts, or as one or more arguments to the predicate you must write. Please do not include such Prolog facts in your submission.
Each correct submission (at most one for each problem) earns you one point.
Efficiency of your programs is not important (except in the Turing problem), but if your program fails to finish in a reasonable time, it will be considered incorrect. You can earn a bonus for the problem named **The Efficient Turing Machine**: in case of a tie otherwise, the speed of your solution will break it.
Do not start predicate names with iclp05 - do not use the dynamic database!

(1) The Christmas Tree. *(xmas.pl)*

Christmas trees come in different sizes. Below you see a size 1, a size 2 and a size 3 Christmas tree:



You guessed it: you must write a predicate `xmas/1`, which when called with a positive integer N, draws a xmas tree of size N.

(2) The Maximal Non-commutative Subset. *(maxnoncom.pl)*

In group theory, the problem of characterising the subsets whose elements commute (or do not commute) is quite well researched. Erdős proved that every group contains a commuting subset of size at least ... sorry, this was not supposed to be a group theory class. Here is your assignment: write a predicate `maxnoncom/1` which when called with a free argument unifies it with a list of group elements that is *maximal* (i.e. you can't add another element without violating the next constraints) and whose elements pairwise do not commute. The internal group operation is specified as a set of `internal/3` facts, whose arguments are atomic. For the integers with addition, `internal(5,3,8)` would mean `5+3=8`. Here is an example and a query:

```
internal(a,a,a).        internal(a,b,b).
internal(a,c,c).        internal(a,d,d).
internal(b,a,b).        internal(b,b,a).
internal(b,c,a).        internal(b,d,d).
internal(c,a,c).        internal(c,b,a).
internal(c,c,b).        internal(c,d,a).
internal(d,a,d).        internal(d,b,c).
internal(d,c,d).        internal(d,d,b).

?- maxnoncom(MNC).
MNC = [b,d]
```

You may rely on the `internal/3` facts really specifying a group. The order in the answer to `maxnoncom/1` is not important. You might think that for some groups there is more than one possible answer ... well, whatever you think, any correct answer suffices!
You might not remember all the details about groups ... a group is a set with an internal operation, often denoted by $*$. A group has an identity, every element has an inverse and the operation is associative. The fact `internal(c,b,a)` denotes the identity: $c * b = a$. Two elements a and b commute iff $a * b = b * a$.
Note added afterwards: we wanted to exclude answers with only one element, because otherwise trivially for every group the set with only the unity would be maximally non-commutative. We forgot this however.

(3) The Efficient Turing Machine. *(turing.pl)*

When specifying his machine, Alan used Occam's razor to such an extent that his machine is easy to implement in any programming language ... oh well, you pompous sod, we meant in any Turing complete language of course, and thus a (universal) Turing machine simulator in Prolog is possible. But Alan also cared about efficiency. So we want you to implement as efficiently as you can a Turing machine simulator and you must name it `alan/0`. As input you get facts `rule/5` which denote the program rules of a TM. Here is one such fact:

```
rule(   q0 , 1 ,  q1 , x, right ).
```

The arguments are: current state, current symbol under the reading head, next state, symbol to be written, direction of movement of the head. The directions can be `right`, `left` and `stay` - with obvious meaning. The start state is always `q0`, the final state always `qf`. The blanc symbol is represented by `b`. You can represent the tape whichever way you want, but when the machine has finished a computation, we want you to write out the symbol on every square of the tape the execution has visited and in the order from leftmost to rightmost. Ah, the input on the tape ... you get it as one `intape/1` fact whose argument is a list of symbols: the head of the Turing machine is initially reading the first element of that list and the other elements are the symbols to the right of the head. Here is a complete example:

```
rule(q0, 1, q0, b, right).
rule(q0, b, qf, 1, stay).

intape([1,1,1,1,1]).

?- alan.
bbbbb1
```

Correctness is most important of course, but in case of a tie, (time) efficiency will be measured and can decide about the winner for this contest.
You are free to use to your advantage the linear speedup theorem :-)

(4) Who did it? *(didit.pl)*

There are a number of people involved in an event of an unspecified nature - it would distract you if we told you the details, so we don't - and the investigation related to this event are meant to identify *who-did-it*. The abstract form of the investigation consists of Prolog facts for the predicate `says/3`: they represent what the different participants in the event say. The first argument is the name of a person (a Prolog atom). The second argument is an identifier of this particular utterance: it can be used in other utterances. It is also atomic. The third argument is the utterance itself. Because of the abstraction and the filtering capabilities of the investigators, the utterances are of the following kind only

- `true(ut_id)`: this means that the statement says that *ut_id* is a true statement
- `false(ut_id)`: this means that the statement says that *ut_id* is a false statement
- `didit(person)`: this means that the statement says that *person* did it

We also know that one of the persons saying something did it. And we assume that every statement is either true or false.
Here is an example:

```
says(bart,1,true(3)).
says(bart,2,didit(john)).
says(john,3,false(2)).
```

Clearly, not all three statements can be true, because `john` says that 2 is false. That is typical for an investigation: people contradict each other; sometimes they contradict themselves. So it is not always easy to find out who did it. Fortunately, experience has shown that the truth can be discovered by assuming that the least number of people said something untrue while still being able to pinpoint one person. Let's analyse the above example: it is impossible that `bart` and `john` always spoke the truth, because 2 and 3 cannot both be true. So there is at least one person who lies (but perhaps not all the time): if `john` tells the truth, then 2 is false, and we can conclude that `bart` did it. No other assumption that exactly person lies leads to an identification of who did it: it is indeed impossible that `john` lies and `bart` always tells the truth.
Unfortunately, the minimality assumption might still not lead to an unambiguous identification of who did it. See later for an example.
You must be very close to guessing what we want you to do: write a predicate `didit/1` which unifies its argument with a list of names of everybody who could have done it, under the assumption that a minimal number of people have told a lie.

An example in which there are two potential answers for who did it:

```
says(bart,1,didit(john)).
says(john,2,didit(bart)).

?- didit(L).
L = [bart, john]
```

Both `bart` and `john` can't be speaking the truth - so either of them lies. So either of them has done it.
The order in the answer to the query is unimportant.
Three more examples:

```
says(bart,1,didit(bart)). | says(bart,1,didit(bart)). | says(bart,1,didit(will)).
says(bart,2,false(1)).    | says(bart,2,false(1)).    | says(bart,2,didit(john)).
says(john,3,false(1)).    | says(john,3,false(1)).    | says(john,3,true(3)).
                          | says(will,4,true(3)).     | says(will,4,true(4)).
?- didit(L).              |                           |
L = [jonh]                | ?- didit(L).              | ?- didit(L).
                          | L = [bart]                | L = [bart,john,will]
```

The first example shows that even though nobody said that `john` did it, `john` in fact must have done it, because obviously `bart` lies (his two statements are contradictory), so if `john` does not lie, then `bart` lies when saying that he did it himself. This leaves only `john` as the the one who did it.
The second example shows the entanglement of the lies: if `will` speaks the truth, then `bart` didn't do it, and one can't decide whether `john` or `will` did it. So the assumption that we must be able to single out one do-er leads to the conclusion that all three lie, and then only `bart` can have done it.
The third example might surprise you: clearly `bart` lies at least once. Assuming only his first statement is false, results in the unambiguous identification of `john`. Assuming only his second statement is false, results in the unambiguous identification of `will`. Assuming both `bart`s statements are false, results in the unambiguous identification of `bart`. Every time, there was only one lier ... with different lies.

(5) A Pebble Graph Game. *(pebble.pl)*

A *pebble graph game* consists in a graph with an initial configuration of pebbles on the nodes and rules on how pebbles can be moved from one node to another, and added to or removed from the graph. Such games have been the topic of study by famous algorithmists like R. Tarjan in the 70-ties. Here is the variant that concerns us: there are two kinds of pebbles; one kind is made of ordinary matter, the other kind is made of anti-matter. We will name them pro-pebbles and anti-pebbles. The graphs we consider are similar to transport networks: directed, acyclic, with one source and one sink, and not necessarily simple. All edges have the same length. All pebbles move at the same constant speed. Pro-pebbles can only move in the graph by following the direction of the edges; anti-pebbles go only against the edge direction. In the initial configuration, there are N pro-pebbles in the source, and N anti-pebbles in the sink. And then the clock starts ticking ... At each tick of the clock, all pebbles must make a move - except of course the *surviving* pebbles: a pro-pebble that arrives in the sink, or an anti-pebble that arrives in the source.

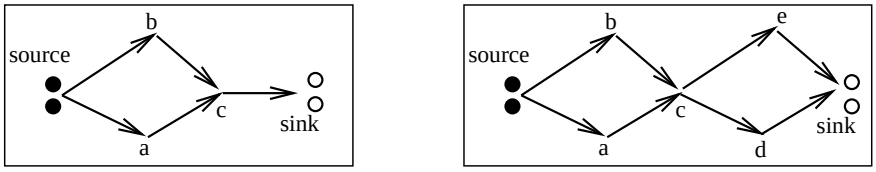
No two pro-pebbles can move along the same edge at the same moment. No two anti-pebbles can move along the same edge at the same moment.

This means that when there are say 7 pro-pebbles in a node with out-degree 5, there are 2 of these pebbles that can't move: they are taken away from the graph. Note that you can't take 3 pebbles away!

When a pro-pebble meets an anti-pebble, obviously they annihilate and release an enormous amount of energy. Such meeting can happen in the middle of an edge: the released energy blows away the edge. Or such meeting can happen at a node: nodes are much stronger than edges, so it's just the two involved pebbles that disappear. To be more precise: if I pro-pebbles and J anti-pebbles arrive at the same node X and if $I \geq J$, then only $I - J$ pro-pebbles remain on X ...

Now you know the setting, here is the problem ... for a given N, a given graph is an *N-pebble eater* if it is possible that starting with N pro-pebbles in the source and N anti-pebbles in the sink, it is possible for the pebbles to move in such a way that they all disappear - because they get annihilated or removed from the graph.

Not every graph is an N-pebble eater for every N: have a look at Figure 1(a).



(a) Not a 2-pebble eater

(b) A 2-pebble eater

Figure 1: Two pebble graphs.

The pro-pebbles are the black dots, the anti-pebbles the circles. One pro-pebble certainly survives: the graph in Figure 1(a) is not a 2-pebble eater. The graph in Figure 1(b) is. It is an N-pebble eater for every N in fact.
Write a predicate `pebble_eater/1` which is called with a positive integer N and which succeeds iff the given graph is an N-pebble eater.
The graph itself is given as a set of `edge/2` facts of as for instance for the graph in Figure 1(a):

```
edge(source,a).
edge(source,b).
edge(a,c).
edge(b,c).
edge(c,sink).
```

Here are some queries for this graph:

```
?- pebble_eater(1).
Yes

?- pebble_eater(2).
No

?- pebble_eater(3).
No
```

The name of the source node is always `source` and the name of the sink node is always `sink`.

(6) Conjunctive Grammars. *(cong.pl)*

Alexander Okhotin invented them, and since Prolog and grammars are two hands on one belly [1], you'll program them!
In a conjunctive grammar the rules have the form:

```
A ==> T1 & T2 & ... & Tn
```

where each Ti is a sequence of terminals and non-terminals.
As a recognising device, this rule specifies that a sentence is recognised by the grammar as an A if it is recognised as a $T1$ **AND** as a $T2$ ... **AND** as a Tn.
As an example (we use capital letters for non-terminals here, but this is not necessarily true in the input for your program later):

```
S ==> AB & DC
A ==> aA                    C ==> cC
A ==>                       C ==>
B ==> bBc                   D ==> aDb
B ==>                       D ==>
```

recognises the set $\{a^n b^n c^n | n \geq 0\}$: AB recognises $\{a^i b^j c^j\}$ and DC recognises $\{a^i b^i c^j\}$.
Conjunctive grammars resemble CFGs, but as the example shows, they describe also some non-context-free languages.
Of course one can use a conjunctive grammar as a generating device, i.e. to generate words in the language recognised by the grammar.
Your task is to write a predicate `cong/2` whose first argument is a positive number N and which unifies its second argument (by backtracking) with every string accepted by the given grammar with a derivation that uses N or less times a grammar rule [2].
Duplicates are allowed (there is no guarantee that the given grammar is unambiguous), but looping is not allowed.
The grammar is given as one `grammar/1` fact. For the above grammar it could be:

```
grammar([start ==> ( (nA, nB) & (nD, nC) ),
         nA ==> (a, nA),
         nA,
         nB ==> (b, nB, c),
         nB,
         nC ==> (c, nC),
         nC,
         nD ==> (a, nD, b),
         nD]).
```

[1] Flemish expression

[2] This confused people: if grammar rule X and grammar rule Y are used 3 and 4 times respectively, the total is 7, so this 7 must be less than or equal to N

All terminals and non-terminals are atomic.
And here is an example query with answers for this grammar:

```
?- cong(17,L).
L = [a,a,a,b,b,b,c,c,c]
L = [a,a,b,b,c,c]
L = [a,b,c]
L = []
```

The order of answers is not important.
The notion of *using a rule* ... the analogy with logical inference is so strong that you as Prolog experts shouldn't need further clarification, but if you are off by one, that's fine.
You might want to define : $-op(800, xfx, \&), op(900, xfx, ==>)$. The start symbol is always named *start*. A symbol in the left hand side of a rule is a non-terminal.
Alternatives are expressed by having more than one rule for a non-terminal; the usual symbol for denoting alternatives in a grammar is |, but is not used in our rules.