AI 1: lab session 4

Computational Logic

Instructions

- Answer all questions and ***formulate your answers concisely and clearly***.
- Read the book and use the information from the lectures. Read the questions carefully and feel free to ask for clarifications if something is not clear.
- You can use the helpdesk email `airug1819@gmail.com` to ask questions. Make sure you read the instructions to contact the helpdesk that are provided on Nestor. Only emails sent according to the instructions will be replied to.
- For the programming assignments, ***it is not allowed to use code supplied by others***. If we suspect plagiarism or collusion, the exam committee will be notified.

Rules for submission

- Submit, electronically via Nestor (Submissions section), a .zip file containing:
  1. ***All relevant source codes***, so that we can test them.
  2. A digital version of your team's report. ***Reports need to be written in LaTeX***. A template is available on Nestor.
- Comply with the deadline provided on Nestor. ***Deadlines are strict*** (penalties are applied for late submissions, see below).
- Supply the names and student numbers of all members of your team on the report. Also clearly ***write the learning community*** of each member in the group.

Grading Every assignment is graded by the teaching assistants based on grading schemes agreed with the lecturer. The grade of this lab assignment will count for ***10% of your final grade***. We ***subtract*** 2^{n-1} grade points for a submission that is between $n-1$ and n days late ($n \geq 1$).

1 Model checking in propositional logic

In this programming exercise, we will implement model checking for propositional logic. A major part of the code has already been prepared for you, and can be found in the file `model.py`. The script runs without error, but it will not result in a complete program.

The program reads from the input two sets of sentences in propositional logic. The first set is the KB, and the second is a set of propositional sentences that we want to infer (if possible) from the KB. The parsing of the input has been implemented completely: you need not change anything in this part of the code, nor are you requested to understand the parsing of the input. There are also routines to evaluate a set of propositional sentences given a model. An example routine `evaluate_random_model` is available to show you how to use these evaluation routines. The routine `evaluate_random_model` generates a random model (random assignment of boolean values) for all identifiers (atoms) and then evaluates the truth value of the sets `KB` and `INFER`. Study this routine carefully.

An example input for the program can be found in the file `model1`:

```
{'KB': [
  'p+(q*r) <=> (p+q)*(p+r)',
  'p=>q <=> !p+q',
  'p=>q',
  'q=>r',
  'p'
],
'INFER': [
  'q',
  'r',
  'q*r',
  'true'
]
}
```

This example input represents the following sets `KB` and `INFER`:

$$\begin{aligned} \text{KB} &= \{(p \vee (q \wedge r)) \Leftrightarrow ((p \vee q) \wedge (p \vee r)), (p \Rightarrow q) \Leftrightarrow (\neg p \vee q), p \Rightarrow q, q \Rightarrow r, p\} \\ \text{INFER} &= \{q, r, q \wedge r, \text{true}\} \end{aligned}$$

Compile the program and run it using the above input. Your session should look like:

```
$python model.py model1
Complete input:  {'KB': ['p+(q*r) <=> (p+q)*(p+r)', 'p=>q <=> !p+q', 'p=>q',
'q=>r', 'p'],
'INFER': ['q', 'r', 'q*r', 'true']}
Identifiers:  ['p', 'q', 'r']
===============
KB:
((p + (q * r)) <=> ((p + q) * (p + r)))
((p => q) <=> (!p + q))
(p => q)
(q => r)
```

```
p

INFER:
q
r
(q * r)
True
===============

Randomly chosen model:  {'p': True, 'q': False, 'r': False}
     KB evaluates to:  False
     INFER evaluates to:  False

KB entails INFER

The function check_all_models is not implemented yet
The goal of this assignment is to implement this yourself
Currently, this function always returns True
```

The program chose the random model `[p=true,q=false,r=false]` and then evaluated `KB` and `INFER`. Both evaluations returned false.

Implement the routine `check_all_models()` that determines whether `KB` $\models$ `INFER`. If `KB` $\models$ `INFER` is not the case, then the program should print a counter example on the output.

Questions:

a) Run your code on the files model1 model2 and model3 and report the results

2 Resolution in propositional logic

On Nestor, you can find the files `resolution.py`. The file `resolution.py` contains the source code of a complete program that performs resolution on a (propositional) KB in conjunctive normal form (CNF).
In the file `resolution.py` the following KB is hard coded in the routine `init`:

$$KB = \{a \vee \neg b \vee \neg c \vee \neg d, c \vee \neg d, \neg a \vee \neg b, b \vee \neg d\}$$

Note that this KB is the CNF equivalent of:

$$KB = \{\neg a \Rightarrow (\neg b \vee \neg c \vee \neg d), \neg c \Rightarrow \neg d, a \Rightarrow \neg b, \neg b \Rightarrow \neg d\}$$

From the latter KB, it is easy to see that $KB \models \neg d$:

- Assume a: From $a \Rightarrow \neg b$ we conclude $\neg b$. Using $\neg b \Rightarrow \neg d$ we conclude $\neg d$.
- Assume $\neg a$: From $\neg a \Rightarrow (\neg b \vee \neg c \vee \neg d)$ we conclude $\neg b \vee \neg c \vee \neg d$. Since $\neg b \Rightarrow \neg d$ and $\neg c \Rightarrow \neg d$, we can conclude $\neg d \vee \neg d \vee \neg d = \neg d$.

So, if we add $\neg\neg d = d$ to the KB, we should be able to infer the empty clause (i.e. false) using resolution. The resolution proof tree is given in the following figure:

$$\dfrac{\dfrac{\dfrac{\dfrac{a \lor \neg b \lor \neg c \lor \neg d \qquad c \lor \neg d}{a \lor \neg b \lor \neg d} \qquad \neg a \lor \neg b}{\neg b \lor \neg d} \qquad b \lor \neg d}{\neg d} \qquad d}{\emptyset}$$

The negation of the conclusion ($\neg\neg d = d$) is added to the KB in the last lines of the routine `init` of `resolution.py`.
Running the script `resolution.py` gives you the following output:

```
$ python resolution.py
KB={[~a,~b], [a,~b,~c,~d], [b,~d], [c,~d], [d]}
KB after resolution={[~a,~b], [a,~b,~c,~d], [b,~d], [c,~d], [d], [~b,~c,~d],
[~a,~d], [a,~c,~d], [a,~b,~d], [a,~b,~c], [b], [c], [~b,~d], [~b,~c], [~a],
[~c,~d], [a,~d], [a,~c], [a,~b], [~b], [~d], [~c], [a], []=FALSE}
Resolution proof completed.

Proof:
Implement the function recursive_print_proof() yourself!
```

As you can see, the program generates all possible clauses that can be inferred from the KB. However, it generates many clauses that are not needed at all in the proof of the goal $\neg d$. For example, the clause `[a,~b,~c]`(i.e. $a \lor \neg b \lor \neg c$) is inferred, but is not used in the proof of $\neg d$.

Study the code in `resolution.py`. Extend the program such that after resolution, a proof is printed in the routine `recursive_print_proof`. For your solution, it should not be necessary to change anything in the Clause class or in the helper functions provided. The output should be as follows:

```
Proof:
[a,~b,~d] is inferred from [a,~b,~c,~d] and [c,~d].
[~b,~d] is inferred from [~a,~b] and [a,~b,~d].
[~d] is inferred from [b,~d] and [~b,~d].
[]=FALSE is inferred from [d] and [~d].
```

Change the routine `init` such that your program can read a KB from standard input. The input for the above KB would be:

```
KB=[[~a,~b],[a,~b,~c,~d],[b,~d],[c,~d],[d]]
```

Questions:

a) Produce a KB and a conclusion (goal) using at least 10 variables yourself, for which the proof consists of at least 15 steps. Include this example (and the proof generated by your code) in your submission.

3 Prolog assignments

For the following exercises we will be using `swipl`, which is a freely available prolog implementation. It is already installed on the lab computers. For documentation, visit `http://www.swi-prolog.org`.

3.1 Biblical family

Load the file `biblical.pl`, study the rules in the KB and answer the following questions.

Questions:

a) Which Prolog query determines who is the grandfather of Lot? What is the answer?

b) Which Prolog query determines all grandsons of Terach? What is the answer?

3.2 Arithmetic with natural numbers

Download the file `arith.pl`. In this file some operations on so-called *Peano integers* are defined. Study the rules in the KB and answer the following questions.

Questions:

a) What is a suitable query to ask the system whether 3+2=5? What does the system answer?

b) What is a suitable query to ask the system whether 3+2=6? What does the system answer?

c) Add predicates `even(N)` and `odd(N)`, that determine whether `N` is even or odd.

d) Add a predicate `div2(N,D)` that determines whether the integer division `N/2` is equal to `D`. Your solution should not use the predicate `times`. Test your predicate for some even and odd arguments.

e) Add a predicate `divi2(N,D)`, that computes the same result as `div2(N,D)`, but now using the predicate `times`. Of course, you need to test this predicate as well.

f) Find an n such that $2^n = 8$ using a suitable query. Add a predicate `log(X,B,N)` that determines whether $B^N = X$.

g) Extend the KB with a predicate `fib(X,Y)`, where `fib` denotes the Fibonacci function. The predicate returns true if and only if $fib(X) = Y$. [Note: $fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)$]

h) In the course *Imperative programming* you learned that B^N can be computed in $O(\log N)$ steps using the rules: $a^{2b} = (a^2)^b$ and $a^{2b+1} = a \cdot a^{2b}$. Extend the KB with the predicate `power(X,N,Y)`, based on these rules. The predicate returns true if $X^N = Y$. Is this predicate an improvement over a direct $O(N)$ computation? Why (not)?

3.3 Lists

In Prolog it is possible to use lists. For example, the following snippet of code determines the length of a list:

```
len([],0).
len([H|T],N) :- len(T,N1), N is N1+1.
```

Note that lists are not types, i.e. the elements of a list can be anything. For example, the query `len([1,2,[artificial,intelligence]],X).` will be answered with `X=3`.

Questions:

Download the file `arith.pl`, and extend it with the following functionality:

a) `member(X,L)` returns true if `X` is a member of the list `L`.

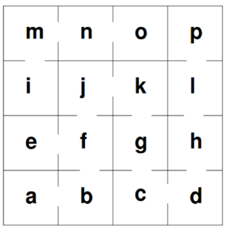
b) `concat(L,X,Y)` returns true if `L` is the concatenation of the lists `X` and `Y`.

c) `reverse(L,R)` returns true if `R` is the reversal of the list `L`.

d) `palindrome(L)` returns true if `L` is a palindrome.

3.4 Maze

Consider the following maze:



Questions:

a) Write a Prolog KB that represents the maze.

b) Extend the KB with a predicate `path(X,Y)` that returns true if there exists a path from `X` to `Y`. So, the query `path(a,p)` should succeed.

c) Try also `path(a,m)`. What is the result?