AI 1: lab session 3b
Logic

Instructions: Formulate your answers clearly: we cannot give points for things that are intended but not written down. Read the book and use the information from the lectures. Read the questions carefully and feel free to ask questions if something is not clear.

Submission: For this lab session, you are not required to write a report! You only need to hand in codes and give answers to the posed questions. Please send your codes and answers (together with the first part about CSPs) digitally to `artint1617@gmail.com`. Supply the names and student numbers of both authors (you work in pairs). Also clearly write down in which lab group you are. For the programming assignments, always supply all relevant (self-written) source codes such that the teaching assistants can test them. It is not allowed to use code supplied by others. If we detect plagiarism, the exam committee will immediately be notified! The deadlines for each lab session can be found on Nestor.

Grading: Every session is graded by the teaching assistants. The average of all lab sessions is your grade for the labs. Note that the deadlines are strict: we subtract 2^{n-1} grading points for a report that is n days late!

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.c`. The file compiles, but it will not result in a complete program. A complete Linux executable solution is also available. This is the complete solution of the exercise (in executable machinecode), so you can check whether your solution produces the same results. Beware, this executable only runs on 64 bit Linux systems (like the ones in the lab rooms)!

   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 as set of propositional sentences given a model. An example routine `evaluateRandomModel(cntidents)` is available to show you how to use these evaluation routines. The routine `evaluateRandomModel(cntidents)` generates a random model (random assignment of boolean values) for `cntidents` variables 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:

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

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

```
$ gcc -Wall model.c
$ ./a.out < model1
KB = [
  ((p + (q*r)) <=> ((p + q)*(p + r))) ;
  ((p => q) <=> (!p + q)) ;
  (p => q) ;
  (q => r) ;
  p
]
INFER = [
  q ;
  r ;
  (q*r) ;
  true
]
Random chosen model: [p=true,q=false,r=true]
  KB evaluates to false
  INFER evaluates to false

THE FUNCTION checkAllModels IS NOT IMPLEMENTED YET
PLEASE IMPLEMENT IT YOURSELF!
THIS FUNCTION CURRENTLY ALWAYS RETURNS 1.

KB entails INFER
```

The program chose the random model `[p=true,q=false,r=true]` and then evaluated `KB` and `INFER`. Both evaluations returned false. Implement the routine int `checkAllModels(int modelSize)` that determines whether `KB` $\models$ `INFER`. If this is not the case, then the program should print a counter example on the output.

2. **Resolution in propositional logic**

On Nestor, you can find the files `resolution.c` and `resolution`. The file `resolution.c` contains the source code of a complete program that performs resolution on a (propositional) KB in conjunctive normal form (CNF). The file `resolution` is a 64-bit Linux executable (so, it will not run on Windows or a Mac), that performs resolution and shows a proof tree on the standard output. This executable is available for reference purposes only.

In the file `resolution.c` 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:

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

The negation of the conclusion ($\neg\neg d = d$) is added to the KB in the last lines of the routine `init` of `resolution.c`.

Compile the code in `resolution.c` and run the resulting executable:

```
$ gcc -Wall resolution.c
$ ./a.out
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 ROUTINE recursivePrintProof 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. $a \vee \neg b \vee \neg c$) is inferred, but is not used in the proof of $\neg d$.

- Study the code in `resolution.c`. Extend the program such that after resolution, a proof is printed in the routine `recursivePrintProof`. You are not allowed to break the ADTs (Abstract Data Types) in the program. The output should be as follows (run the executable resolution):

```
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]]`

- Produce yourself a KB and a conclusion (goal) using at least 10 variables, for which the proof consists of at least 15 steps. Include this example (and the generated proof) in your submission.
- **Bonus exercise**: You can earn one extra grade point for this lab session, by extending the program such that it accepts the input files of exercise 4 (model checking). This means that you have to write yourself code that transforms a KB into CNF (which is a lot of work).

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. **Biblical family**
   Load the file `biblical.pl`, study the rules in the KB and answer the following 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?

4. **Arithmetic with natural numbers**
   Download the file `arith.pl`. In this file some operations on *Peano integers* are defined. Study the rules in the KB and answer the following 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)?

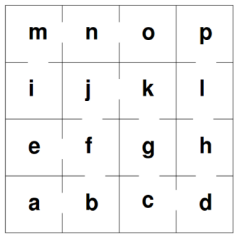
5. **Lists**
   In Prolog it is possible to use lists. For example, the following snippet of code determines the length of a list:

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

   - `member(X,L)` returns true if `X` is a member of the list `L`.
   - `concat(L,X,Y)` returns true if `L` is the concatenation of the lists `X` and `Y`.
   - `reverse(L,R)` returns true if `R` is the reversal of the list `L`.
   - `palindrome(L)` returns true if `L` is a palindrome.

6. **Maze**
   Write a Prolog KB that represents the following maze.



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