Hints:

• Please solve these exercises in groups of two!

• The solutions must be handed in directly before (very latest: at the beginning of) the exercise course on Wednesday, December 22nd, 2010, in lecture hall AH 3. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl’s office (until the exercise course starts).

• Please write the names and immatriculation numbers of all (two) students on your solution. Also please staple the individual sheets!

• Exercises or exercise parts marked with a star are voluntary challenge exercises with advanced difficulty. However, they do not contribute to the overall maximum number of points that you can obtain by this exercise sheet. (i.e., you can obtain at most 17 points for this exercise sheet).

• As the topics related to the exercises on this sheet are relevant for all versions of the lecture, all exercises on this sheet are relevant for all students.

Exercise 1 (SLD Tree): (4 + 1 + 1 = 6 points)

Consider the following logic program \( P \):

```prolog
edge(a, b, 1).
edge(a, e, 7).
edge(b, c, 4).
edge(c, e, 1).
edge(d, e, 2).
p\text{athCost}(X, Y, Z) :- edge(X, Y, Z).
p\text{athCost}(X, Y, Z) :- p\text{athCost}(A, Y, B), edge(X, A, C), Z is B + C.
```

The predicate edge defines the following weighted graph \( G \):

![Graph](https://via.placeholder.com/150)

Furthermore, \( p\text{athCost}(X, Y, Z) \) is true iff there is a path from \( X \) to \( Y \) in \( G \) where the sum of all weights along the path is \( Z \). As an example, \( ?- p\text{athCost}(b, e, Z) \) gives the single solution \( Z = 5 \).

**a)** Please give a graphical representation of the SLD tree for the query \( ?- p\text{athCost}(a, e, Z) \). To indicate the infinite parts of the SLD tree please write “…” for nodes that would contain at least six atoms (and do not continue with the subtrees of such nodes).

**b)** Improve the order of the literals in \( P \) such that for each query the resulting SLD tree is finite.

**c)** Is it possible to modify the edge facts in the improved program of part **b** (i.e., by changing the represented graph \( G \)) such that there is a query with an infinite SLD tree? Give a short explanation of your answer.
Exercise 2 (Built-in Arithmetic and Lists): (6 points)

Implement a predicate \texttt{sumSplit/3} in Prolog which computes for a given list of integers all possible distributions of the list elements in two lists such that the sum of all elements in the resulting two lists is equal. More precisely, \texttt{sumSplit(Xs,Ys,Zs)} is true iff \(Xs\), \(Ys\), and \(Zs\) are lists of integers where the concatenation of \(Ys\) and \(Zs\) is a permutation of \(Xs\) and the sum of all integers in \(Ys\) is equal to the sum of all integers in \(Zs\).

For example, \(\texttt{?– sumSplit([1,2,3],Ys,Zs)}\) should compute exactly the following four answer substitutions (not necessarily in the same order):

\begin{align*}
Ys &= [1,2], \\
Zs &= [3]; \\
Ys &= [2,1], \\
Zs &= [3]; \\
Ys &= [3], \\
Zs &= [1,2]; \\
Ys &= [3], \\
Zs &= [2,1]
\end{align*}

Exercise 3 (Equalities): (5 points)

In Prolog there are the following five built-in predicates of arity 2 computing some kind of equality:

- \texttt{=}
- \texttt{==}
- \texttt{=:=}
- \texttt{is}
- \texttt{unify_with_occurs_check}

For each combination of two of these predicates, give an example of two terms where the application of the first predicate succeeds while the application of the second predicate fails or results in an error. If this is not possible for some combination, please explain why.

Exercise 4\textsuperscript{*} (Occurs-Check and Rational Terms): (6\textsuperscript{*} points)

Omitting the occurs-check is not only useful to reduce the complexity of unification, but it can also be used to create rational terms. These terms can again be used for elegant programming solutions. Consider for example the usual way to define a predicate \texttt{odd/1} which should be true iff its argument is an odd natural number in Peano notation (where the number \(n\) is represented by the term \(s(\ldots s(0)\ldots))\):

\begin{verbatim}
odd(s(X)) :- even(X).

even(0).
even(s(X)) :- odd(X).
\end{verbatim}

This declaration uses mutual recursion with another predicate \texttt{even/1}. Using unification without occurs-check, we can define the predicate \texttt{odd/1} without mutual recursion as follows:
odd(X) :- L = [o,e|L], odd(X,L).
odd(0,[e|_]).
odd(s(X),[M|MS]) :- odd(X,MS).

This technique is especially useful to implement context-free grammars (CFGs). Consider the following CFG defining the language of all palindromes over the letters a and b. As usual, ε denotes the empty word and a palindrome is a word which is the same no matter whether it is read from left to right or from right to left. Examples for palindromes are abba and aba whereas baba is no palindrome.

\[
S \rightarrow \varepsilon
\]
\[
S \rightarrow a
\]
\[
S \rightarrow b
\]
\[
S \rightarrow aSa
\]
\[
S \rightarrow bSb
\]

Please complete the following Prolog program by replacing the %IMPLEMENT ME comment such that the predicate palindrome/1 is true iff its argument contains a list representing a palindrome over the letters a and b (e.g. palindrome([a,b,b,a]) and palindrome([a,b,a]) are true while palindrome([b,a,b,a]) is false).

\[
palindrome(Word) :- grammar(Grammar), terminalstring(Word), produces(Grammar,Word).\]

\[
terminalstring([\]).\]
\[
terminalstring([X|XS]) :- terminalstring(XS), terminal(X).\]

\[
terminal(a).\]
\[
terminal(b).\]

\[
produces(Grammar,Word) :- gives(Grammar,Word,[]).\]

\[
gives(epsilon,Processed,Processed).\]
\[
gives(or(X,Y),Prefix,Processed) :- gives(X,Prefix,Processed).\]
\[
gives(or(X,Y),Prefix,Processed) :- gives(Y,Prefix,Processed).\]
\[
gives(and(X,Y),Prefix,Processed) :- gives(X,Prefix,Remainder),gives(Y,Remainder,Processed).\]
\[
gives(T, [T|Processed],Processed) :- terminal(T).\]

\[
grammar(S) :- %IMPLEMENT ME\]