General Computer Science II (320201) Spring 2006

Michael Kohlhase
Jacobs University Bremen
For Course Purposes Only

April 8, 2013

Contents

Assignment 1: Resolution Calculus
(Given Feb. 8., Due Feb. 15.)

Problem 1.1: Prove in the resolution calculus using derived rules:

$$\models A \wedge (B \vee C) \Rightarrow A \wedge B \vee A \wedge C$$

Solution: Clause Normal Form transformation

$$\dfrac{\dfrac{\dfrac{A \wedge (B \vee C) \Rightarrow A \wedge B \vee A \wedge C^{\mathsf{F}}}{A \wedge (B \vee C)^{\mathsf{T}}; A \wedge B \vee A \wedge C^{\mathsf{F}}}}{A^{\mathsf{T}}; B^{\mathsf{T}} \vee C^{\mathsf{T}}; A \wedge B^{\mathsf{F}}; A \wedge C^{\mathsf{F}}}}{A^{\mathsf{T}}; B^{\mathsf{T}} \vee C^{\mathsf{T}}; A^{\mathsf{F}} \vee B^{\mathsf{F}}; A^{\mathsf{F}} \vee C^{\mathsf{F}}}$$

Resolution Proof

1	A^{T}	initial
2	$B^{\mathsf{T}} \vee C^{\mathsf{T}}$	initial
3	$A^{\mathsf{F}} \vee B^{\mathsf{F}}$	initial
4	$A^{\mathsf{F}} \vee C^{\mathsf{F}}$	initial
5	B^{F}	with 1 and 3
6	C^{F}	with 1 and 4
7	C^{T}	with 2 and 5
8	$\square$	with 6 and 7

Problem 1.2: Consider the following two formulae where the first one is in conjunctive normal form and the second in disjunctive normal form 25pt

1. $(P \vee \neg P) \wedge (Q \vee \neg Q)$
2. $P \wedge Q \vee (\neg P \vee \neg Q)$

Try to find the shortest proofs of both formulae using the resolution method as well as the tableau method. Describe your observations concerning the proof length in dependency on the normal form and proof method.

Solution: For the first forumla we have the tableau

$$
\begin{array}{c|c}
\multicolumn{2}{c}{(P \vee \neg P) \wedge (Q \vee \neg Q)^{\mathsf{F}}} \\
P \vee \neg P^{\mathsf{F}} & Q \vee \neg Q^{\mathsf{F}} \\
P^{\mathsf{F}} & Q^{\mathsf{F}} \\
\neg P^{\mathsf{F}} & \neg Q^{\mathsf{F}} \\
P^{\mathsf{T}} \bot & Q^{\mathsf{T}} \bot
\end{array}
$$

For the resolution proof we first have to convert into clause normal form.

$$
\begin{array}{c}
((P \Rightarrow Q) \wedge (Q \Rightarrow R) \Rightarrow \neg(\neg R \wedge P))^{\mathsf{F}} \\
(P \Rightarrow Q) \wedge (Q \Rightarrow R)^{\mathsf{T}}; \neg(\neg R \wedge P)^{\mathsf{F}} \\
(P \Rightarrow Q)^{\mathsf{T}}; (Q \Rightarrow R)^{\mathsf{T}}; \neg(\neg R \wedge P)^{\mathsf{F}} \\
P^{\mathsf{F}} \vee Q^{\mathsf{T}}; Q^{\mathsf{F}} \vee R^{\mathsf{T}}; \neg R^{\mathsf{T}} P^{\mathsf{T}} \\
P^{\mathsf{F}} \vee Q^{\mathsf{T}}; Q^{\mathsf{F}} \vee R^{\mathsf{T}}; R^{\mathsf{F}}; P^{\mathsf{T}}
\end{array}
$$

then we have the resolution derivation

$$
\begin{array}{c|c}
P^{\mathsf{F}} \vee Q^{\mathsf{T}} & \textit{initial} \\
Q^{\mathsf{F}} \vee R^{\mathsf{T}} & \textit{initial} \\
R^{\mathsf{F}} & \textit{initial} \\
P^{\mathsf{T}} & \textit{initial} \\
Q^{\mathsf{T}} & \textit{resolved} \\
Q^{\mathsf{F}} & \textit{resolved} \\
\Box & \textit{resolved}
\end{array}
$$

Now to the next formula; here we have the tableau

$$
\begin{array}{c|c}
\multicolumn{2}{c}{P \wedge Q \vee (\neg P \vee \neg Q)^{\mathsf{F}}} \\
\multicolumn{2}{c}{P \wedge Q^{\mathsf{F}}} \\
\multicolumn{2}{c}{\neg P \vee \neg Q^{\mathsf{F}}} \\
\multicolumn{2}{c}{\neg P^{\mathsf{F}}} \\
\multicolumn{2}{c}{\neg Q^{\mathsf{F}}} \\
\multicolumn{2}{c}{P^{\mathsf{T}}} \\
\multicolumn{2}{c}{Q^{\mathsf{T}}} \\
p^{\mathsf{F}} & Q^{\mathsf{F}} \\
\bot & \bot
\end{array}
$$

For the resolution proof we convert to clause normal form:

$$(P \vee \neg P) \wedge (Q \vee \neg Q)^{\mathsf{F}}$$
$$(P \vee \neg P)^{\mathsf{F}} \vee (Q \vee \neg Q)^{\mathsf{F}}$$
$$P^{\mathsf{F}} \vee (Q \vee \neg Q); \neg P(Q \vee \neg Q)^{\mathsf{F}}$$
$$P^{\mathsf{F}} \vee Q^{\mathsf{F}}; P^{\mathsf{F}} \vee Q^{\mathsf{T}}; P^{\mathsf{T}} \vee Q^{\mathsf{F}}; P^{\mathsf{T}} \vee Q^{\mathsf{T}}$$

So we have the resolution derivation

$P^{\mathsf{F}} \vee Q^{\mathsf{F}}$	*initial*
$P^{\mathsf{F}} \vee Q^{\mathsf{T}}$	*initial*
$P^{\mathsf{T}} \vee Q^{\mathsf{F}}$	*initial*
$P^{\mathsf{T}} \vee Q^{\mathsf{T}}$	*initial*
Q^{T}	*resolved*
Q^{F}	*resolved*
$\square$	*resolved*

We note that for the formula in DNF the shortest method is the tableaux and for the one in CNF it is the resolution method. This is not particularly surprising, since the Resolution medthod is CNF-based (we construct the CNF in for clause normal form first), whereas Tableau is DNF-based.

Assignment 2: Graphs and Trees
(Given Feb. 14., Due Feb. 22.)

15pt

Problem 2.3 (Tree of Paths of a Graph)
Let G be a directed acyclic graph (DAG). Given a node p_1 in G, the set Π_{p_1} of paths in G that start with p_1 can be arranged in a tree: The root of this tree is labeled by the empty path $\langle p_1 \rangle$ and a node labeled with a path $\pi = \langle p_1, \ldots, p_n \rangle$ has children labeled with paths $\langle p_1, \ldots, p_n, p_{n+1} \rangle$.

- Draw the tree of paths for the graph

$$G = \langle \{a, b, c, d, e\}, \{\langle a, b\rangle, \langle a, c\rangle, \langle b, c\rangle, \langle c, d\rangle, \langle a, d\rangle, \langle d, e\rangle\}\rangle$$

- Show that for a DAG with n nodes, the tree of paths can have at most 2^{n-1} nodes. Exhibit a set of graphs G_n that obtain this maximum.

Solution:

1. We prove the

Problem 2.4 (Operations on Binary Trees) 999pt

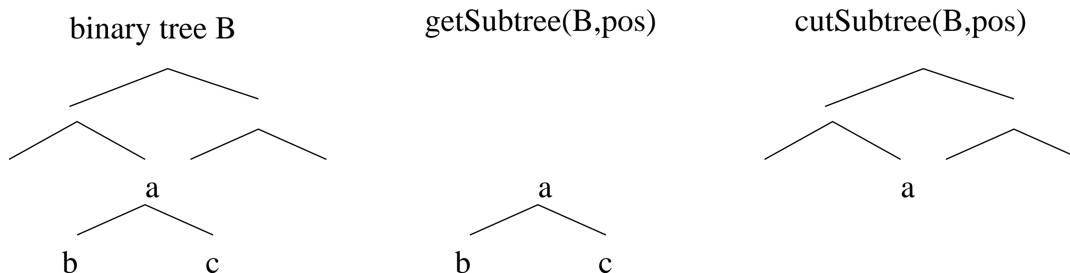
Given the SML datatype btree for binary trees and position for a position pointer into a binary tree:

```
datatype btree = leaf | parent of btree * btree;
datatype position = stop | right of position | left of position;
```

The interpretation of a position right(left(stop)) is like a reversed path: start from root follow the right branch then the left and then stop.

Write two SML functions:

- getSubtree that takes a binary tree and a position and returns the subtree of the that binary at the corresponding position.
- cutSubtree that takes a binary tree and a position and returns the binary tree where the subtree at the corresponding position is cut off; i.e replaced by a leaf.



In both cases an exception should be raised if the position exceeds the observed binary tree.

Solution:

```
datatype btree = leaf | parent of btree * btree;
datatype position = stop | right of position | left of position;

exception TreeExceeded;

fun getSubtree (parent(l, r), right(pos)) = getSubtree(r, pos)
  | getSubtree (parent(l, r), left(pos)) = getSubtree(l, pos)
  | getSubtree (tree, stop) = tree
  | getSubtree (_, _) = raise TreeExceeded;

fun cutSubtree (parent(l, r), right(pos)) = parent(l, cutSubtree(r, pos))
  | cutSubtree (parent(l, r), left(pos)) = parent(cutSubtree(l, pos), r)
  | cutSubtree (tree, stop) = leaf
  | cutSubtree (_, _) = raise TreeExceeded;

val testTree = parent(parent(leaf, parent(leaf, leaf)),parent(leaf, leaf));

val testPos = left(right(stop));
```

```
(* test get Subtree:
- getSubtree(testTree, testPos);
val it = parent (leaf,leaf) : btree
- cutSubtree(testTree, testPos);
val it = parent (parent (leaf,leaf),parent (leaf,leaf)) : btree
*)
```

Problem 2.5: How many edges can a directed graph of size n (i.e. with n vertices) have maximally. How many can it have if it is acyclic? Justify your answers (prove them). 4pt

Solution:

Theorem 1 *A directed graph with n vertices has at most n^2 edges.*

Proof:

P.1 Let $G = \langle V, E \rangle$.

P.2 By definition $E \subseteq V^2$, so maximally $\#(V) = n^2$. □

Theorem 2 *A DAG with n vertices has at most $(n(n-1))/2$ edges.*

Proof: by induction on n

P.1.1 If $n = 1$:

P.1.1.1 then $G_1 = \langle \{c\}, E \rangle$, since the only possible edge $\langle c, c \rangle$ is a cycle. □

P.1.2 If $n > 1$:

P.1.2.1 Let $G_n = \langle V_n, E_n \rangle$ be a maximal DAG,

P.1.2.2 then the graph $G_{n-1} = \langle commiV_{n-1}, E_{n-1} \rangle$ which we obtain from G_n by deleting an arbitrary vertex c and all the edges c in must be a maximal DAG with $n-1$ nodes (otherwise we could add an edge to G_n).

P.1.2.3 Thus $\#(E_{n-1}) = ((n-1)(n-2))/2$. Now, there can be at most $n-1$ edges in V_n, which c occurs in without cycles (G_{n-1} has $n-1$ vertices).

P.1.2.4 Therefore, $\#(V_n) = ((n-1)(n-2))/2 + (n-1) = (n-1)(n-2) + (2(n-1))/2 = ((n-1)n)/2$. □

□

Problem 2.6 (Parse Tree) 35pt

Given the data type prop for formulae

```
datatype prop = tru | fals (* true and false *)
              | por of prop * prop (* disjunction *)
              | pand of prop * prop (* conjunction *)
              | pimpl of prop * prop (* implication *)
              | piff of prop * prop (* biconditional *)
              | pnot of prop (* negation *)
              | var of int (* variables *)
```

Write an SML function that computes the parse tree for a formula. The output format should be

- a list of integers for the set of vertices,
- a list of pairs of integers for the set of edges,
- and for the labeling function a list of pairs where the first component is an integer and the second a string (the label).

Solution:

```
datatype prop = tru | fals (* true and false *)
| por of prop * prop (* disjunction *)
| pand of prop * prop (* conjunction *)
| pimpl of prop * prop (* implication *)
| piff of prop * prop (* biconditional *)
| pnot of prop (* negation *)
| var of int (* variables *)
```

The output is a triple (vertices, edges, labeling_list) where

- vertices is simply an integer (so the vertices are represented as integers from 1 to that number)
- edges is a list of pairs of integers that are the vertices between which there are edges
- labeling_list: a list of (vertex:integer, label:string) pairs that will be used to make a labeling function which takes the index of a vertex and returns its label, e.g. a string "impl" Input: A pair (root, p)
- root is a name for the root node (an integer), see the function maketree
- p a variable of datatype prop in which a boolean expression is stored

```
fun totree(mroot, pnot(be)) = let val (verout, edges, lblpairs) = totree(mroot+1, be)
                              in (verout, [mroot, mroot+1] :: edges, (mroot, "−") ::
                              end |
    totree(mroot, tru) = (mroot, nil, [(mroot, "T")])|
    totree(mroot, fals) = (mroot, nil, [(mroot, "F")])|
```

```
totree(mroot, var v) = (mroot, nil, [(mroot, Int.toString(v))])|
totree(mroot, two_var) = let val ax = fn (por(be1, be2)) => ("OR", be1, be2) |
                                                                        (pand(be1, be2)) =>
                                                                        (pimpl(be1, be2)) =>
                                                                        (piff(be1, be2)) => ("
                                             val (lbl, be1, be2) = ax two_var
                                             val (verout1, edges1, lblpairs1) = totree(mroot+
                                             val (verout2, edges2, lblpairs2) = totree(verout1
                                        in (verout2, [[mroot, mroot+1],[mroot, verout1+1]] @ ec
                                        end
```

Now we have an optional wrapper function that

- eliminates the need for the index of the root (default value is 1)
- converts the labeling_list into labeling function that takes a vertex and returns its label

```
local
     fun findString(num, hd::tl) = let
          val (a,b) = hd
     in
          if a = num then b
          else findString(num, tl)
     end

in
     fun maketree(t) = let
          val (lastnode, edges, lblpairs) = totree(1, t)
     in (lastnode, edges, fn num => findString(num, lblpairs))
     end
end
```

Finallly: an example of how the program can be tested:

```
totree(1, por(por(piff(pnot(var 4), fals), pimpl(tru, pand(var 2, var 3))), var 9));
val it =
(12,[[1,2],[1,12],[2,3],[2,7],[3,4],[3,6],[4,5],[7,8],[7,9],[9,10],[9,11]],
[(1,"OR"),(2,"OR"),(3,"<=>"),(4,"-"),(5,"4"),(6,"F"),(7,"=>"),(8,"T"),
(9,"AND"),(10,"2"),(11,"3"),(12,"9")])
: int * int list list * (int * string) list *)
```

Problem 2.7 (Spanning Tree of a Graph) 20pt

A spanning tree of a graph is a tree containing all nodes of the graph as well as a subset of its edges (but no additional edges!).

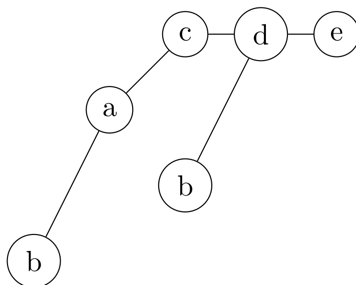
We call an (undirected) graph connected, iff for any two different nodes n_1 and n_2 there is a path starting at n_1 and ending at n_2.

- Draw the following undirected graph together with one corresponding spanning tree

$$G = \langle \{a, b, c, d, e\}, \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}, \{c\}, \{b, d\}, \{d, e\}\} \rangle$$

- Prove by induction or refute that every undirected, connected, and non-empty graph has a spanning tree.

Solution: The graph G:



A spanning tree of G:

Proof by induction on the number K of Nodes.

Problem 2.8 (Combinational Circuit for a Boolean Function) 10pt

Draw a combinational circuit that is defined by the following Boolean function:

$$f\colon \{0,1\}^3 \to \{0,1\}^2; \langle i_1, i_2, i_3\rangle \mapsto \langle i_1 + i_3\rangle, \overline{i_1 + i_2} * i_3$$

Assignment 3: Combinatorial Circuits
(Given Feb. 22., Due March. 1.)

20pt

Problem 3.9 (Binary Number Conversion)
Write an SML function binary that converts decimal numbers into binary strings and an inverse decimal that converts binary strings into decimal numbers. Use the positive integers (of built-in type int) as a representation for decimal numbers. binary should raise an exception, if applied to a negative integer.

Solution:

```
exception NegInteger
fun binary 0 = "0" |
    binary 1 = "1" |
    binary n = if (n<0) raise NegInteger
                        else binary(n div 2)^(Int.toString(n mod 2))

fun decimal s = let
    fun bintodec nil = 0
      | bintodec sa = foldl (fn (x, y) => 2*y+(if x = #"0" then 0 else 1)) 0 sa
in bintodec(explode(s))
end
```

Problem 3.10 (DNF Circuit with Quine McCluskey) 25pt

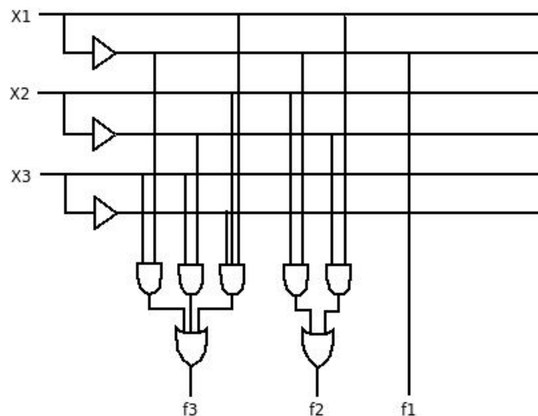
Design a combinational circuit for the following Boolean function:

X_3	X_2	X_1	$f_3(X)$	$f_2(X)$	$f_1(X)$
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Solution: After applying the QMC we get something like:

$$
\begin{aligned}
f_1(X) &= (X_3 \wedge \neg X_1 \vee X_3 \wedge \neg X_2) \vee \neg X_3 \wedge (X_2 \wedge X_1) \\
f_2(X) &= \neg X_1 \wedge X_2 \vee X_1 \wedge \neg X_2 \\
f_3(X) &= X_1 \wedge X_2 \vee X_3
\end{aligned}
$$

Hence the circuit:



Problem 3.11 (Carry Chain and Conditional Sum Adder) 35pt

Draw an explicit combinational circuit of a 4-bit Carry Chain Adder and a 4-bit Conditional Sum Adder. Do not use abbreviations, but only NOT, AND, OR, XOR gates. Demonstrate the addition of the two binary numbers $\langle 1,0,1,1\rangle$ and $\langle 0,0,1,1\rangle$ on both adders; i.e. annotate the output of each logic gate of your adders with the result bit for the given two binary numbers as input of the whole adder.

Assignment 5: Virtual Machine
(Given March 16., Due March 22.)

50pt

Problem 5.12 (Comparing Imperative and Functional Style via VM)
Write a $\mathcal{L}(\mathtt{VM})$ program that implements DIV and MOD functions, using imperative access to your variables (i.e. with `peek` and `poke` instructions). Your program expects numbers $A \geq 0$ and $B \geq 0$ on the stack (A is on the top), and upon termination leaves only numbers C and D on the stack (C is on the top), where $C := A\,div\,B$ and $D := A\,mod\,B$. If $B = 0$, leave only an error code -1 on the stack. Furthermore

1. Show the evolution of the stack for $A = 7$ and $B = 3$. Include all intermediate steps.
2. Is it possible to solve a) using only instructions of functional programming? (i.e. all but `peek` and `poke`). If yes, solve a) again without `peek` and `poke` If no, explain why by arguing about fundamental differences between imperative (C-like) and functional (SML-like) programming languages.

Solution: Darko says (in a mail to Christoph, 2007-Mar-20):

> It is indeed not possible to program divmod without PEEK and POKE. One of the reasons is that all other instructions in the imperative version of VM destroy the data once they access it. For example, ADD would take the two topmost elements and replace them with the sum. However, PEEK can read memory without destroying it, so we can access memory outside the top of the stack. One might ask how is then possible that functional programming languages despite that have the same power? Well, by the use of recursion which is supported by four additional instructions (PROC, ARG, RET and CALL) we enable the recursion pattern and make PEEK and POKE superfluous. This is the functional version of VM, which they learn later in the course. It performs calculations on stack, with all the variables being temporary.

Problem 5.13 (Convert Highlevel Code to $\mathcal{L}(\mathtt{VM})$ Code) 20pt

Given is an array A[0..10] and the following piece of imperative code:

```
for j := 1 to 5 do
 for i := j to 10−j do
  A[i] := A[i−j] + A[i+j];
```

Suppose the array is loaded on stack (top value being A[10]). Convert the code into $\mathcal{L}(\mathtt{VM})$ code.

30pt

Problem 5.14 (Compiling If-construct)
Write an SML function simpleif that takes a string of form

IF \RMdatastore{X} rel Y THEN Z ELSE W

where X, Y, Z, W are non-negative integers, $\sim$ an element of $\{\leq, \geq, >, <, =, \neq\}$, and returns the appropriate $\mathcal{L}(\texttt{VM})$ language piece of code. For example, given IF $\RMdatastore{2}$ $>$ 4 THEN 200 ELS your function should return an array of $\mathcal{L}(\texttt{VM})$ instructions that reads the second cell of the stack and pushes 200 if its content is greater than 4 and 3 otherwise. Note that your program should treat any (non-empty) concatenation of white-space as a single white-space.

Raise exception if the input string is not valid.

Assignment 8: A^* Search
(Given April 26., Due May 3.)

60pt

Problem 8.15 (Associating Students to Tutorials)
This is a real world problem you all know: At the beginning of a term all students of a course have to choose a tutorial that doesn't conflict with his/her time schedules.

Obviously this can be viewed as a search problem. For this exercise we want to simplify this real world problem slightly: Let us assume 65 students each of them having at least one and at most five free time slots for taking a tutorial. We assume that students commit to their favored time slots at the very beginning of the term (in real world this is unfortunately not always the case). For the 65 students 7 tutorials are offered for each day of the week one. Each tutorial accepts at most 10 students.

Implement in SML an A^* search algorithm that tries to solve this problem for a given dataset of students and their free time slots.

A state is a set of 65 (student $*$ tutorial) pairs meaning that the student is associated with this tutorial. The tutorials should be represented by

```
datatype tutorial = Mo | Tu | We | Th | Fr | Sa | Su | None
```

Thereby the None tutorial is used to express the set of students that aren't associated with any (real) tutorial. So in the initial state all students are in the "None" tutorial whereas in the goal state this None tutorial must be empty.

Possible actions are to move one student per step from one to another tutorial. Only those moves are allowed which are compatible with the student's free time slots. Moving a student to the None tutorial is always allowed. The definition of the step cost is up to you.

An example dataset is the following:

```
(* Example dataset: the second component of each pair in the dataset
   lists the free time slots of the corresponding student *)

datatype tutorial = Mo | Tu | We | Th | Fr | Sa | So | None
datatype student = s of int;
dataset =
[(s 1,[Mo,We]),
 (s 2,[Mo,Tu,Fr]),
 (s 3,[Mo]),
 (s 4,[Mo,Fr,Sa]),
 (s 5,[Mo,We,So]),
 (s 6,[Mo,Tu,We,Fr,Sa]),
 (s 7,[Mo,Tu]),
 (s 8,[Mo,We]),
 (s 9,[Mo]),
 (s 10,[Mo,Tu,Fr]),
 (s 11,[Mo,Tu,Sa]),
```

(s 12,[Tu,Fr]),
(s 13,[Tu]),
(s 14,[Tu,We,Th,Sa,So]),
(s 15,[Mo,Tu,Fr]),
(s 16,[Tu,We]),
(s 17,[Tu,Th,Fr,So]),
(s 18,[Mo,Tu]),
(s 19,[Tu]),
(s 20,[Mo,We,Th]),
(s 21,[Tu,We,Fr]),
(s 22,[Mo,We,Sa]),
(s 23,[We]),
(s 24,[Mo,We]),
(s 25,[Tu,We,Th]),
(s 26,[We,Th]),
(s 27,[We,Sa,So]),
(s 28,[Tu,We,Sa]),
(s 29,[Mo,We]),
(s 30,[Mo,We,Th]),
(s 31,[Mo,Th,Fr]),
(s 32,[Tu,Th]),
(s 33,[Th,Fr]),
(s 34,[Th,Sa]),
(s 35,[Mo,Tu,Th,Fr]),
(s 36,[Sa,Th,Fr]),
(s 37,[Mo,Th]),
(s 38,[Tu,Th,Fr]),
(s 39,[We,Th,Sa]),
(s 40,[Mo,Tu,Fr]),
(s 41,[Mo,Fr]),
(s 42,[Fr]),
(s 43,[Tu,Fr]),
(s 44,[Mo,Fr]),
(s 45,[Fr]),
(s 46,[Mo,Tu,We,Fr,Sa]),
(s 47,[We,Fr]),
(s 48,[Tu,Fr]),
(s 49,[Fr,Sa]),
(s 50,[Th,Fr,Sa]),
(s 51,[Sa,So]),
(s 52,[Mo,Tu,Sa]),
(s 53,[Tu, We,Sa]),
(s 54,[Mo,Sa,So]),
(s 55,[Tu,We,ThSa]),
(s 56,[Sa]),
(s 57,[Fr,Sa]),
(s 58,[Fr,So]),

```
 (s 59,[Mo,Tu,So]),
 (s 60,[So]),
 (s 61,[Fr,So]),
 (s 62,[Sa,So]),
 (s 63,[Mo,We,So]),
 (s 64,[Th,Fr,So]),
 (s 65,[Tu,So])
];
```

Assignment 9: Prolog Programming

(Given May 3., Due May 10.)

20pt

Problem 9.16 (Sort)

Write a ProLog program that sorts a list of integers; i.e. a predicate sort(L1,L2) that evaluates to true iff the list L2 is the sorted version of L1. For instance,

?− sort([5,2,6,1],X)

should return X=[1,2,5,6].

Problem 9.17 (Querying the Greek Mythology Family Tree) 30pt

Who is related to whom in the large family of Greek gods that is the question in this problem. An overview can be found at `http://en.wikipedia.org/wiki/Family_tree_of_the_Greek_gods`. Take a fragment of this tree and encode it in `ProLog` as facts in terms of mother/2, father/2 relations. (Note: in `ProLog` notation p/n means that p is a predicate of arity n).

Add predicates all_sibling_of, all_cousins_of, all_ancestors_of, all_descendants_of and rules such that predicate(God,List_of_Gods) hold for each predicate. For instance all_sibling_of(zeus,S) should return S=[hera,demeter, hades, poseidon]. read(Δ,t)oΓ

Problem 9.18 (Derivation of Arithmetic Expressions) 40pt

Write a ProLog program that "computes" the derivative of an arithmetic expression; i. e. write facts and rules such that derivative(E1,x,E2) is satisfied if and only if the arithmetic expression E2 is the derivative of E1 with respect to x (where x is an atomic expression of course). The constructors of arithmetic expressions should be any integer or lower case letter as atomic expressions together with mul/2 and add/2 for building compound arithmetic expressions.

For instance, derivative(mul(x,add(x,2)),x,add(mul(1,add(x,2)),mul(x,add(1,0)))) should evaluate to true.

Note: You do not need to implement term rewriting, i. e. you need not be recognize *all* possible ways of writing expressions equivalent to E2.

Assignment 10: Prolog Programming 2
(Given May 11., Due May 17.)

This assignment is intended to make you aware how the order of literals and rules affect the program execution. 10pt

Problem 10.19 (Trace of a ProLog Program)

With the trace command in ProLog you can look at the execution of a given program step by step. Try this command on the program below and explain the trace output.

```
a(X,Y):-b(X,Y),c(Y).
b(X,Y):-d(X,Y),e(Y).
b(X,_):-f(X).
c(4).
d(1,3).
d(2,4).
e(3).
f(4).
```

Problem 10.20 (Relevance of Rule and Literal Order for Termination) 20pt

Whether a program terminates may depend on the order of rules and even on the order of literals within one rule. Write two `ProLog` programs to demonstrate this; i.e. both programs should terminate for some query, but after a permutation of the rules in the first program and a permutation of some literals in the second program neither the first nor the second should terminate for the same query.

Moreover give an explanation why the programs don't terminate in the latter case.

Problem 10.21 (Relevance of Rule and Literal Order for Efficiency) 30pt

The order of rules can affect the efficiency of ProLog programs. Demonstrate this on the example program below: Compose two variants of this program by reordering the rules such that you get three different degrees of efficiency for the query ?− a([1,2,3],2) from the three programs. Each program variant should terminate as the original does! Rank the three programs with respect to their efficiency and explain the cause of the efficiency differences. Also give quantitative evidence using trace.

```
b(_,_).
a([A],C):−b(A,C).
a([A,B|R],C):−b(A,C),b(B,C),a([B|R],C).
a([A,B|R],C):−b(B,C),a([A|R],C),a([B|R],C).
a([A,B|_],_):−b(A,B).
```

Apart from the rules order the order of literals can be relevant for efficiency. To demonstrate this invent two programs differing only in the order of some literals and specify a query for testing the efficiency difference of your programs. Again both programs should terminate on this query.