Prof. Dr. Jürgen Giesl

Carsten Otto

Notes:

- To solve the programming exercises you can use the Prolog interpreter **SWI-Prolog**, available for free at `http://www.swi-prolog.org`. For Debian and Ubuntu it suffices to install the `swi-prolog` package. You can use the command "`swipl`" to start it and use "`[exercise11].`" to load the clauses from file **exercise11.pl** in the current directory.
- The solutions must be handed in **directly before (very latest: at the beginning of)** the exercise course on Wednesday, 17.07.2013, in lecture hall **AH 2**. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl's office (this box is emptied **a few minutes before** the exercise course starts).
- Please write the **names** and **immatriculation numbers** of all students on your solution. Also please staple the individual sheets!

Important: This sheet is only relevant for students attending the **V3B** version of the lecture.

Exercise 1 (Open Trees): (9 points)

Difference lists are lists where the end of the list is left "open" by using a variable. For example, the list `[a,b,c]` can be represented as the difference list `[a,b,c|X] - X`. This notation makes it easy to append difference lists by instantiating the variable with the list that should be appended.
In the example above, appending `[d,e,f|Y] - Y` to `[a,b,c|X] - X` can easily be done by instantiating `X` with `[d,e,f|Y]` and using the end of the appended list (`Y`) as the end of the resulting list:

`[a,b,c|[d,e,f|Y]] - Y` = `[a,b,c,d,e,f|Y] - Y`

The fact

```
app(Xs - Ys, Ys - Zs, Xs - Zs).
```

(as presented in the lecture) corresponds to this computation.
So the main idea behind appending difference lists basically is that instead of traversing the list, just a variable needs to be instantiated.

In this exercise we want to use the idea of creating the result by instantiating a variable (instead of constructing/traversing explicitly) to work on binary trees, represented using the symbol `t/3`. Here, leaves of trees are just numbers. Nodes are represented using `t/3`, where the first argument is the node's value (a number), the second argument is the left child, and the third argument is the right child.
As an example, the tree containing the value `1` at the root with a left child (a leaf) with value `2` and a right child (a leaf) with value `3` is represented as the term `t(1, 2, 3)`.

Write a predicate `replaceWithMin(Input, Output)`. Here, `Input` is a tree as described above, while `Output` is the input tree where the leaves are replaced by the minimum value found in the whole tree. **Your algorithm must traverse the input tree only once.** In other words, it is not allowed to first find the minimum and then (with that knowledge) traverse the input tree again to explicitly construct the resulting tree.

As an example, `replaceWithMin(t(0,t(4,2,5),1), X)` gives the answer substitution `X = t(0,t(4,0,0),0)`.

Hints:

- Think about how difference lists can be appended without the need to traverse the lists.
- Think about how the variable `X` is used in the `app` example above.

- To adapt these ideas to trees, implement an auxiliary predicate `replace(InputTree, X, M, OutputTree)` which holds if `InputTree` is a tree (e.g., `t(0,t(4,2,5),1)`), `M` is the minimum of the tree (i.e. 0) and `OutputTree` results from replacing all leaves in the `InputTree` by `X` (i.e., `t(0,t(4,X,X),X)`).

Exercise 2 (Definite Clause Grammars): (5 points)

Consider the following context free grammar $G = (N, T, S, P)$ with
$N = \{Program, Rule, Body, Atom, Identifier, Variable\}$,
$T = \{$`.`, `:-`, `','`, `(`, `)`, `a`, `b`, `c`, `X`, `Y`, `Z`$\}$,
$S = Program$,
P is defined as follows (ε represents the empty word).

Program → ε	*Atom* → *Identifier*
Program → *Rule* `.` *Program*	*Atom* → *Identifier* `(` *Body* `)`
Rule → *Atom*	*Identifier* → `a`
Rule → *Atom* `:-` *Body*	*Identifier* → `b`
Body → *Atom*	*Identifier* → `c`
Body → *Variable*	*Variable* → `X`
Body → *Atom* `,` *Body*	*Variable* → `Y`
Body → *Variable* `,` *Body*	*Variable* → `Z`

Please write a predicate `program/1` such that the query `?- program(W).` is true iff `W` is in $L(G)$. For example, `a :- b(X).` is in $L(G)$. In your program, it would be represented by the list `['a',':-','b','(','X',')','.']`. Your Prolog program must not contain the symbol `-->` and must not use predicates for list concatenation. Make use of difference lists as much as possible.