Technische Universität München | *Summer Semester 08*
Fakultät für Informatik | *Vesal Vojdani*
Prof. Dr. H. Seidl | *vojdanig@in.tum.de*

Compiler Construction

Exercise Sheet 9

Deadline: 2. July 2008, at the lecture, in room 02.07.053, or by e-mail.

Exercise 1: *Type-equivalence* *3+3 Points*

Check with the methods presented in the lecture whether the following two types are semantically equivalent:

a)

```
class Tree{          class Tree1{     class Tree2{
  int n;               int n;           int n;
  Tree l,r;            Tree1 r;         Tree1 l,r;
                       Tree2 l;
}                    }                }
```

b)

```
class Tree{          class Tree1{     class Tree2{
  int n;               int n;           Tree1 l,r;
  Tree l,r;            Tree2 t;
}                    }                }
```

Exercise 2: *Type inference* *6 Points*

Consider the following expressions in the functional language of the lecture. Infer their types!

a) `letrec r = fn x ⇒ if x = 0 then 0 else x + r (x-1)`

b) `letrec f = fn t ⇒`
`case t of [] → [] | [(l,x,r)] → [((f l),0,(f r))]`

Exercise 3: *Extending type inference* *10 Points*

Let's extend the language and its type system to allow user defined data types. Consider the following example of a binary tree:

```
datatype tree t = Leaf t | Node (tree t,tree t)
```

Here, t is a type variable, which can be instantiated with for example integers to obtain the type of integer trees `tree int`. A concrete element of this type is for example `Node(Leaf 1,Leaf 2)`. We call `tree` a type constructor, while `Leaf` and `Node` are data constructors. In general, a type definition has the following form:

```
datatype <Type-identifier> (t1,...,tm) =
    <Constructor_1> [<Type-expression_1> ]
  | ....
  | <Constructor_n> [<Type-expression_n> ]
```

where the type-variables `t1,...,tn` as well as the type identifier itself may be used in the body of the type expression:

```
datatype list t = Nil | Cons (t, list t)
datatype rose t = Rose (t, list (Rose t))
```

The `case`-expression is expanded to allow pattern matching over the different alternative constructors of the data type:

```
letrec
  count = fn tree =>
              case tree of
                 Leaf _ -> 1
               | Node(x,y) -> (count x)+(count y)
in count (Node(Leaf 1,Leaf 2))
```

This function computes the number of leaves in the tree `Node(Leaf 1,Leaf 2)`, which is 2. Now we expand the type system to deal with these new constructs.

1. Extend the typing rules for dealing with data constructors and pattern-matching.
2. Show how the system of type-equalities are now generated.
3. Extend the algorithm $\mathcal{W}$.
4. Use your algorithm to compute the type of the function `count` defined above.