National Technical University of Athens
School of Electrical & Computer Engineering
Department of Computer Science
http://courses.softlab.ntua.gr/pl2/

Programming Languages II

Solutions to the exercises are to be handed in to the instructors in electronic form. Deadlines are firm. You may hand in at most one late exercise.

Exercise 2 Type Inference

Due date: 3/11/2010

In the programming language of your choice, implement type inference for the simply typed λ-calculus. For your convenience, we suggest that you choose a functional language: Haskell, Standard ML or OCaml.

Input and output. Your program must read data from the standard input and write the results to the standard output.

The first line of the input will contain a natural number N. Each of the following N lines will contain one λ-term. The grammar that describes the λ-terms is given below. The letter "λ" in abstractions is denoted by the character "\" (backslash). Variable names will start with a letter and consist of letters and digits. For your convenience, you may assume that there is exactly one space character after the dot in abstractions, and between the two terms in application. Parentheses will be used exactly where specified by the grammar.

Your program must output N lines, each of which must contain the type of the corresponding λ-term, or the message "type error", in case the type inference fails. Types must be output in normal form as follows. Type variables must be denoted by `@0`, `@1`, `@2`, and so on. Function arrows must be denoted by the string " `->` ", with exactly one space around the arrow. Parentheses must be used only when necessary. The choice of variables must be such that the resulting type is "lexicographically" smaller than any other equivalent type, which can be obtained by renaming variables. For example, the type

$$\beta \to (\beta \to \alpha) \to \alpha$$

must be printed as

```
@0 -> (@0 -> @1) -> @1
```

and not in any of the following equivalent forms:

```
@1 -> (@1 -> @0) -> @0
@7 -> (@7 -> @42) -> @42
```

Notice: the use of "lexicographically" above is not entirely right. According to lexicographical order, "`@2`" > "`@10`". In contrast, here the variable `@2` must be prefered over `@10`. In short, the variables that appear in a type, in order of appearance, must be `@0`, `@1`, `@2`, and so on.

Brief description of the solution. The following grammar describes the syntax of λ-terms (M, N) and types (σ, τ).

$$\begin{aligned} M, N &::= x \mid (\lambda x.M) \mid (M\,N) \\ \sigma, \tau &::= \alpha \mid (\sigma \to \tau) \end{aligned}$$

To construct the set of constraints that is the result of a λ-term's semantic analysis, you may use the following rules, which are equivalent to the standard typing rules à-la Curry for the simply typed λ-calculus.

The typing relation $\Gamma \vdash e : \tau \mid C$ means that in the environment Γ, expression e has type τ, provided that the constraints in set C are satisfied.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau \mid \varnothing} \qquad \frac{\alpha \text{ fresh type variable} \quad \Gamma, x : \alpha \vdash e : \tau \mid C}{\Gamma \vdash \lambda x.\, e : \alpha \to \tau \mid C}$$

$$\frac{\Gamma \vdash e_1 : \sigma \mid C_1 \quad \Gamma \vdash e_2 : \tau \mid C_2 \quad \alpha \text{ fresh type variable}}{\Gamma \vdash e_1\, e_2 : \alpha \mid C_1 \cup C_2 \cup \{\sigma = \tau \to \alpha\}}$$

The type checker that you have to implement will produce the constraints. It will take as input Γ and e and will produce as output τ and C. The algorithm W for calculating the most general unifier for C is given e.g. in `http://en.wikipedia.org/wiki/Algorithm_W`.

Input example.

```
5
(\x. x)
(\x. (\y. y))
(\x. (\y. x))
(\x. (x x))
(\x. (\y. (\z. ((y z) x))))
```

Output example.

```
@0 -> @0
@0 -> @1 -> @1
@0 -> @1 -> @0
type error
@0 -> (@1 -> @0 -> @2) -> @1 -> @2
```