CS411 Homework 5
Type inference *Due: 4 April 2007*

Submit the answers for the first two problems in a file `types.txt`. Submit the code for the last problem in a file `inference.ml`.

1. For each of the following, construct terms in the simply typed lambda calculus, or show why a type cannot be constructed:

   - $(\lambda x.x)\ (\lambda x.x)$
   - $\lambda x.\lambda y.x\ y\ x$
   - $\lambda b.\lambda t.\lambda f.(b\ t)\ f$

2. For each of the following expression, show the unification constraints, and the most general (polymorphic) type that can be inferred for each of them:

   - $\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$
   - λ`nul` . λ`tail` . λ`len` . λ`l` . `if (nul l) 0 (len(tail l) + 1)`
     where `if` has type $\forall\alpha.\texttt{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$.

3. Consider a simple functional language with conditionals, pairs, and let constructs. Your job is to write a program that performs type inference for this language.

```
type typ = IntType | BoolType
         | ArrowType of typ * typ | ProdType of typ * typ
         | TypeVar of string

type expr =
    Var of string                    (* variables *)
  | True | False                     (* boolean constants *)
  | Int of int                       (* integer constants *)
  | If of expr * expr * expr         (* if e1 then e2 else e3 *)
  | Plus of expr * expr              (* e1 + e2 *)
  | Pair of expr * expr              (* (e1, e2) *)
  | Fst of expr                      (* fst e *)
  | Snd of expr                      (* snd e *)
  | Apply of expr * expr             (* e1 (e2) *)
  | Lambda of string * expr          (* lambda x . e *)
  | Let of string * expr * expr      (* let x = e1 in e2 end *)
  | TypedLambda of string * typ * expr (* lambda x : type . e *)
```

   You have to write a function `infer_type` that takes an expression in this language and yields the inferred type:

```
val infer_type : expr -> typ
```

You may want implement the algorithm using two functions: a function `build` that builds the constraints, and a function `solve` that solves the constraints:

```
type var = string
type type_env = var -> typ
type constraints = (typ * typ) list

val build : (type_env * expr) -> (typ * constraints)
val solve : (typ * constraints) -> typ
```

If your inference system runs into an error, it must raise an exception with an explanatory message:

```
exception InferenceError of string;
... raise InferenceError("error message");
```

The initial type environment passed to `type_inference` should be the empty environment:

```
empty_env : var -> typ =
    fun _ -> raise (InferenceError "variable not in type environment")
```