Computer Science 131, Spring 2002
Assignment 13: Type Inference

Out: Monday, April 22, 2002

Due: Monday, April 29, start of class

Important: For this assignment you must use a special version of the SML/NJ compiler; the command on turing is `/cs/cs131/bin/sml-cs131-a13` instead of the usual `sml`. You should get the file `assign13.sml` (and the test files) from the web page, complete this file, and submit it via `cs131submit`.

For this assignment, you are to implement type *inference* for the same simple functional language for which you wrote a typechecker in Assignment 6. The only change to the concrete and abstract syntaxes (shown in Figures 1 and 2) are that the type annotations on recursive and non-recursive functions have been removed.

---

exp ::=	*n*	(integers)
	var	
	`tt` | `ff`	
	`(`*var*`) =>` *exp*	
	`rec` var_1`(`var_2`)=>` *exp*	
	`(`exp_1`,`exp_2`)`	
	exp_1 `+` exp_2 | exp_1 `-` exp_2 | exp_1 `*` exp_2	
	exp_1 `==` exp_2 | exp_1 `<` exp_2 | exp_1 `<=` exp_2 | ...	
	exp_1 `?` exp_2 `:` exp_3	
	`let` *var* `be` exp_1 `in` exp_2	
	exp_1 exp_2	
	`fst` *exp*	
	`snd` *exp*	
type ::=	`int`	
	`bool`	
	$type_1$`->`$type_2$	
	$type_1$`*`$type_2$	

Figure 1: Concrete Syntax for this Assignment

```
sig
  (* Representation of types in the Simple Functional Language
   *)
  datatype ty = Int_t
              | Bool_t
              | Arrow_t of ty * ty
              | Times_t of ty * ty
              | Metavar_t of ty Metavar.metavar  (* See Question 1 *)

  datatype aop = Plus | Minus | Times | Div
  datatype cop = Less | Equal | LessEq | Greater | GreaterEq | NotEq

  type varname = string

  datatype absyn =
      Num      of int
    | Bool     of bool
    | Var      of varname
    | Arith    of absyn * aop * absyn
    | Compare  of absyn * cop * absyn
    | Cond     of absyn * absyn * absyn
    | Let      of varname * absyn * absyn
    | FnVal    of varname * absyn            (* Types omitted *)
    | RecFnVal of varname * varname * absyn  (* Types omitted *)
    | Apply    of absyn * absyn
    | Pair     of absyn * absyn
    | Fst      of absyn
    | Snd      of absyn


  val ty_toString  : ty    -> string
  val toString     : absyn -> string
end
```

Figure 2: Signature of the structure `A`

1 Implementing Unification (50%)

In most formal mathematical treatments, unification is the procedure which, given two "phrases" u_1 and u_2 from some fixed language that includes metavariables, attempts to find a substitution σ of terms for metavariables such that $u_1\sigma = u_2\sigma$. (Here $u_1\sigma$ means to apply the substitution σ to the term u_1.)

In this assignment, we will take a more "imperative" view, which is commonly used

in implementations of type inference. Metavariables are mutable (implemented using SML `ref`s), and the unification procedure directly updates the definitions of metavariables, rather than carrying around and applying substitutions.

In `sml-cs131-a13` you are using, a structure `Metavar` has been predefined for you, with the following signature:

```
sig
  type 'a metavar
  val new   : unit -> 'a metavar
  val set   : 'a metavar * 'a -> unit
  val get   : 'a metavar -> 'a option
  val equal : 'a metavar * 'a metavar -> bool
end
```

A value of type `'a Metavar.metavar` can be thought of as a box which is either empty (i.e., it is an unset metavariable) or contains a value of type `'a`. For this assignment our metavariables will range over the abstract syntax of types, so all the metavariables we deal with in this assignment will have type `A.ty Metavar.metavar`.

The other functions `Metavar.new`, `Metavar.set`, `Metavar.get`, and `Metavar.equal` respectively are used to create fresh metavariables, set the value of a metavariable, get the current value (if any) of a metavariable, and check whether two metavariables are equal.

Each metavariable is internally identified by a unique number, so if the n-th metavariable appears in a type passed to `A.ty_toString` is is displayed as $\{n$`:`$t\}$ if it has been defined equal to some type t, and as $\{n$`:UNSET`$\}$ otherwise.

1. The `assign8.sml` function contains the definition of a function

   ```
   follow : A.ty -> A.ty
   ```

   This function takes a type and returns an equivalent type that is not a metavariable with a definition. Given any non-metavariable type or an unset metavariable, this function simply returns its argument; given a metavariable with a definition, this function takes the definition and recursively applies `follow` (because it might turn out that the definition itself is a metavariable with a definition).

   Unification often generates chains of metavariables whose definitions are other metavariables. When doing a lot of unifications, it can be expensive to repeatedly traverse long chains. Begin by modifying the function `follow` to do "path compression". That is, the result of the function should not change, but after the function returns all the intermediate metavariables in the chain should be set directly to the final result. For example, if M_1 is set to M_2, and M_2 is set to M_3, and M_3 is set to M_4, and M_4 is unset, then not only should `follow` applied to M_1 return M_4, but afterwards M_1, M_2, and M_3 should all be set to M_4 as well. Thus later calls to `follow` applied to the type M_1 (and m_2 for that matter) will execute more quickly.

2. Write a function

```
occurs : A.ty * (A.ty Metavar.metavar) -> bool
```

that determines whether the given metavariable is used anywhere in the given type *including inside the definitions of defined metavariables that appear in that type.* You may assume that the metavariable you are checking for does not have a definition yet.

3. The `assign8.sml` file also contains a definition for the function

```
unify : A.ty * A.ty -> unit
```

which takes its two arguments, calls `follow` on each type, and then passes the result to a function `unify'` which actually does the unification and gives values to any unset metavariables if necessary to make the two types equal. Complete the definition of `unify'`. This function should raise the exception `Unify` if the arguments cannot be unified.

2 Monomorphic Type Inference (50%)

Implement type inference for a *monomorphic* type system (no special treatment of `let`). Complete the function

```
minfer : (A.ty env * A.exp) -> A.ty
```

using the simple implementation of environments provided in `assign13.sml`.

It may be useful to model your code for `minfer` on your typechecker from Assignment 6. The main difference is that constraints between types must be checked by calling `unify` instead of by checking for equality of types, and that along the way you can create metavariables to stand for missing type information.

If the given expression cannot be made to typecheck, `minfer` should raise an exception.

As in Assignment 6 for debugging purposes you can use `print : string -> unit` along with the functions

```
A.toString     : A.absyn -> string
A.ty_toString  : A.ty    -> string
```

and the function

```
M.parse : string -> A.exp
```

which takes a filename and parses the contents of the file, returning the corresponding abstract syntax. These are all used by the handy function `test` in `assign13`, which takes a filename (a string) and displays the results of applying your `minfer` to the code in that file.