CS251 Programming Languages **Handout # 34**
Prof. Lyn Turbak **Sunday, April 22**
Wellesley College

Problem Set 7
Due: Tuesday, May 1, 2001

Note:

This is the penultimate problem set. The final problem set (Problem Set 8) will be handed out on Tuesday, May 1, and will be due on Tuesday, May 8. It will cover the final topics in the course: imperative programming, parameter passing, laziness, control, and object-oriented programming.

Reading:

- Handouts 28 (Intro to Types), 29 (Type Checking), 32 (Polymorphism), 33 (Type Reconstruction)

Submission:

- Problems 2 and 3 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- For Problems 1, 4, and 5, your softcopy submission should include a copy of your entire `ps7` directory.
- Your hardcopy submission for Problem 1 should be the files `~/cs251/ps7/prob1`*x*`.hep`, where *x* ranges over `a`, and `b`.
- Your hardcopy submission for Problem 4 should be the file:

  `~/cs251/ps7/hofmad/TypeCheck.sml`.
- Your hardcopy submission for Problem 5 should be the file:

  `~/cs251/ps7/hofmad/Recon.sml`.

Problem 1 [20]: Explicit Polymorphic Types

For each of the two HOFL programs in Fig. **??**, annotate the program with type information so that it becomes a well-typed HOFLEPT program. Recall that HOFLEPT supports polymorphism via the `pabs` and `papp` expressions and `forall` type. Unlike in Problem 2 of Problem Set 6, here it is not necessary to duplicate any function definitions. Rather, for function definitions that must be duplicated for HOFLEMT, HOFLEPT gives such functions polymorphic types and uses `papp` to instantiate the polymorphic type differently for different uses.

You can find the programs in Fig. **??** in the files `~/cs251/ps7/problem-1x.hfl` and in the files `~/cs251/ps7/problem-1x.hep`, where x ranges over `a` and `b`. (The `.hfl` extension is for HOFL programs and `.hep` is for HOFLEPT programs.) You should annotate the `.hep` version of each file so that it is a well-typed HOFLEPT program.

To test your solutions, first launch SML connected to `~/cs251/ps7`, and then evaluate (one time only)

```
use("loadProb1.sml");
```

and then evaluate

```
testProb1();
```

You are additionally encouraged to experiment with the HOFLEPT type checker on additional files of your own choosing. To do this, evaluate

```
use("loadHofleptTypeCheck.sml");
```

and then evaluate

```
TypeCheck.checkFile filename;
```

where filename is the name of a file containing the HOFLEPT program you wish to type check.

a. [10]

```
(program (a)
  (bindrec ((generate
              (abs (seed next done?)
                 (if (done? seed)
                     (empty)
                     (prepend seed
                       (generate (next seed) next done?)))))x
            (foldr
              (abs (binop init xs)
                (if (empty? xs)
                    init
                    (binop (head xs)
                           (foldr binop init (tail xs))))))
             )
     (bind lst (generate a (abs (x) (- x 1)) (abs (y) (= y 0)))
       (if (foldr (abs (x y) (bor (> x 5) y)) false lst)
           (foldr (abs (x y) (+ x y)) 0 lst)
           (foldr (abs (x y) (* x y)) 1 lst)))))
```

b. [10]

```
(program (b)
  (bindpar ((inc (abs (x) (+ x 1)))
            (compose (abs (f g)
                       (abs (x) (f (g x)))))
            (thrice (abs (f)
                      (abs (x) (f (f (f x)))))))
    (bind nat (abs (g) ((g inc) b))
      (+ (nat (abs (h) (compose (thrice h) (thrice h))))
         (+ (nat (compose thrice thrice))
            (nat (thrice thrice)))))))
```

Figure 1: Annotate these HOFL programs to make them well-typed HOFLEPT programs.

Problem 2 [20]: Polymorphic Type Derivations

Consider the following HOFL abstraction:

```
(abs (fs xs)
  (map (abs (f) (map (abs (x) (f x))
                     xs))
       fs))
```

a. **[5]** Translate the above expression into an explicitly typed HOFLEPT expression. You will need to use HOFLEPT's polymorphic features.

b. **[15]** Give a typing derivation that proves that the explicitly typed HOFLEPT expression from Part (a) is well-typed in the following type environment:

$$A_1 = \{\texttt{map} \mapsto \texttt{(forall (a b) (}\rightarrow\texttt{ ((}\rightarrow\texttt{ (a) b) (listof a)) (listof b)))}\}$$

As explained in Handout #29, a typing derivation is an upside-down tree in which every node is a type judgement of the form $A \vdash E : T$, where A is a type environment, E is an expression, and T is a type. Each node of the tree is the conclusion of the instantiation of one of the typing rules from from figure 1 of Handout #29 or from figure 3 of Handout #32, and the children of a node are the hypotheses of the rule instantiation.

Because the traditional tree-shaped (horizontal format) derivation would be *very* wide, you should use the vertical format for type derivations explained in Handout #29. You may abbreviate expressions and types to make your derivation more readable as long as you give explicit definitions for each abbreviation. Make sure to label each rule by its name.

Problem 3 [20]: Monomorphic Type Reconstruction

Following the format of the reconstruction example given in class, give a derivation for reconstructing an explicitly typed HOFLEMT expression from the following implicitly typed HOFLIMT expression:

```
((abs (f) (f 5))
 (abs (n) (abs (x) (> x n))))
```

For your derivation, you should draw an abstract syntax tree for the above expression and should annotate each node with (1) the type environment A that would be used for reconstructing the node and (2) the triple (E, T, σ) that results from calling `reconExp` on the node, where E is the explicitly type expression corresponding to the node, T is the type reconstructed for the node, and σ is the type substitution reconstructed for the node.

To make the annotated tree more compact, you are encouraged to name complex entities and give the definitions of these names separately.

Note that the type reconstruction algorithm presented in Handout #33 differs from the one presented in class in that the handout version does *not* perform any substitutions for type variables in types or expressions in the triples returned by `reconExp`. Rather, such substitutions are performed only at the program node at the root of the abstract syntax tree. You should follow the reconstruction rules detailed in the handout in preference to the ones given in class.

Problem 4 [20]: Type Checking Tuples

In Problem Set 6, you extended the evaluator in the SML implementation of the dynamically typed HOFL language to handle the `tuple` and `match-tuple` constructs. Here, you will extend the type checker for the SML implementation of the statically typed, explicitly typed HOFLEMT language to handle these same two constructs.

Recall that tuples are supported via the following two constructs:

> **(tuple** E_1 ... E_n**)**
> Creates a tuple value with n component values, where the ith component value (indices start at 1) is the value of the expression E_i.
>
> **(match-tuple (**I_1 ... I_n**)** E_{tup} E_{body}**)**
> Evaluates E_{tup} to the value V_{tup}, which should be a tuple value with n component values; otherwise, `match-tuple` signals an error. It returns the value of E_{body} in an environment where the names I_1 ... I_n are bound, in order, to the n component values of V_{tup}, and the meanings of all other names are determined by the lexical context in which the `match-tuple` expression appears. Note that all the I_1 ... I_n should be distinct.

Extending the explicitly typed HOFLEMT with these two expressions yields the HOFMAD language. To handle tuple values, the type system of HOFMAD must extend the types of HOFLEMT with the following type:

> **(tupleof** T_1 ... T_n**)**
> Denotes the type of a tuple value whose n component values have the types T_1 ... T_n.

As a simple example of manipulating tuples in HOFMAD, consider the program in Fig. **??**, which is an explicitly typed version of a similar example from Problem Set 6: The funcion `f` denotes a function that takes two arguments (1) an integer named `amount` and (2) a tuple named `entry` with three component values: a string, a boolean, and an integer. If the boolean is true, the function returns a similar tuple where the integer component has been incremented by `amount`; otherwise the function returns the original tuple.

```
(program (n)
  (bind f (abs ((amount int) (entry (tupleof string bool int)))
            (match-tuple (name student? tuition) entry
              (if student?
                  (tuple name student? (+ tuition amount))
                  entry)))
    (f 1 (f 20 (f 300 (tuple "Alyssa Hacker" true n))))))
```

Figure 2: A program illustrating HOFMAD tuples.

Note that, as usual in an explicitly typed language, the formal parameters of the function `f` are annotated with explicit types. However, the names declared by `match-tuple` are *not* annotated with any type information. Intuitively, this is because the types of these names can be deduced from the type of the tuple expression being deconstructed. This is similar to the reason why the names declared in HOFLEMT's `bindpar` need not have explicit type annotations.

As another example, study the HOFMAD program in figure **??**, which defines and uses versions of `zip`, `unzip`, and `map` that manipulate tuples. Convince yourself that the program is well-typed.

Your Task

```
(program (n)
  (bindrec ((down-from (-> (int) (listof int))
              (abs ((x int))
                     (if (= x 0)
                   (empty int)
                   (prepend x (down-from (- x 1))))))
            (map1 (-> ((-> (int) bool) (listof int)) (listof bool))
              (abs ((f (-> (int) bool))
                    (lst (listof int)))
                (if (empty? lst)
                    (empty bool)
                    (prepend (f (head lst))
                      (map1 f (tail lst))))))
            (map2 (-> ((-> ((tupleof int bool)) (tupleof int int))
                       (listof (tupleof int bool)))
                      (listof (tupleof int int)))
              (abs ((f (-> ((tupleof int bool)) (tupleof int int)))
                    (lst (listof (tupleof int bool))))
                (if (empty? lst)
                    (empty (tupleof int int))
                    (prepend (f (head lst))
                      (map2 f (tail lst))))))
            (zip (-> ((tupleof (listof int) (listof bool)))
                     (listof (tupleof int bool)))
              (abs ((duple-of-lists (tupleof (listof int) (listof bool))))
                (match-tuple (L1 L2) duple-of-lists
                  (if (scor (empty? L1) (empty? L2))
                      (empty (tupleof int bool))
                      (prepend (tuple (head L1) (head L2))
                        (zip (tuple (tail L1) (tail L2))))))))
            (unzip (-> ((listof (tupleof int int)))
                       (tupleof (listof int) (listof int)))
              (abs ((list-of-duples (listof (tupleof int int))))
                (if (empty? list-of-duples)
                    (tuple (empty int) (empty int))
                    (match-tuple (tl1 tl2) (unzip (tail list-of-duples))
                      (match-tuple (hd1 hd2) (head list-of-duples)
                        (tuple (prepend hd1 tl1)
                               (prepend hd2 tl2))))))))
            )
    (bind ints (down-from n)
      (bind bools (map1 (abs ((x int)) (= 0 (mod x 2))) ints)
        (unzip (map2 (abs ((tup (tupleof int bool)))
                       (match-tuple (a b) tup
                         (if b (tuple a (* a a)) (tuple (- 0 a) a))))
                     (zip (tuple ints bools))))))))
```

Figure 3: A program illustrating HOFMAD tuples in the context of `zip` and `unzip`.

a. [5]: Typing Rules

Write down typing rules for `tuple` and `match-tuple` in the style of the typing rules in figure 1 of Handout #29.

b. [15]: Type Checking Implementation

In this part, you will extend the SML type checker for HOFLEMT from Handout #29 that we studied in class to handle the `tuple` and `match-tuple` constructs of HOFMAD. An almost complete implementation of HOFMAD can be found in `~/cs251/ps7/hofmad`. The abstract syntax, types, values, parser, and pretty-printers of HOFMAD have already been extended to support the `tuple` and `match-tuple` expressions as well as the `tupleof` type.

Modify the `checkExp` function in `~/cs251/ps7/hofmad/TypeCheck.sml` to handle the the `tuple` and `match-tuple` constructs. You should carefully study the existing clause of `checkExp` before you write your new ones.

The following documentation in appendix **??** describes the resources you will need for this problem.

- The abstract syntax for HOFMAD expressions can be found in Fig. **??**.
- The signature for HOFMAD types can be found in Fig. **??**.
- The signature for the HOFMAD type checker can be found in Fig. **??**.
- The signature for the type environments used by the type checker (`Ident.Env`) is the same as that for the string environment in Figs. **??**–**??**, except that the key type `string` should be replaced by `Ident.Id` and the binding type `'b` should be instantiated to `Type.Ty`. The `Ident.Env` structure is abbreviated as `TEnv` in within the implementation of the `TypeCheck` structure.

To test your solution, first evaluate

```
use("loadProb4.sml");
```

and then evaluate

```
testProb4();
```

You will need to reevaluate *both* of these expressions any time you make a change to `TypeCheck.sml`. Note that you are very likely to encounter numerous SML type checking errors when evaluating first expression; you must fix these before you proceed. Be aware that it will typically take *many* attempts to fix all the type errors!

You'll know that a test case succeeds if it concludes by printing `OK!`. Otherwise, an error message will be displayed indicated why the test failed.

Problem 5 [20]: Reconstructing Tuple Types

In this problem, you will extend the type reconstructer for HOFLIMT presented in Handout #33 to handle the `tuple` and `match-tuple` constructs. Your extended type reconstructer should be able to transform the implicitly typed HOFLAD programs in Figs. **??**–**??** to the corresponding explicitly typed programs in Figs. **??**–**??**.

Modify the `reconExp` function in `~/cs251/ps7/hofmad/Recon.sml` to handle the the `tuple` and `match-tuple` constructs. You should carefully study the existing `reconExp` clauses before you write your new ones.

The following documentation in appendix **??** describes the resources you will need for this problem.

- The abstract syntax for *implicitly typed* HOFLAD expressions can be found in Fig. **??**. This syntax is contained in the `Hoflad.AST` structure, which is abbreviated `I` (for "implicit") in `Recon.sml`.
- The abstract syntax for *explicitly typed* HOFMAD expressions can be found in Fig. **??**. This syntax is contained in the `AST` structure, which is abbreviated `E` (for "explicit") in `Recon.sml`.
- The signature for HOFMAD types can be found in Fig. **??**.
- The signature for the HOFMAD type subsitutions can be found in Fig. **??**. Note that the `Subst` structure with this signature has already been extended to correctly handle the `tuple` and `match-tuple` expressions as well as the `tupleof` type.
- The signature for the HOFMAD type reconstructer can be found in Fig. **??**.
- The signature for the type environments used by the type reconstructer (`Ident.Env`) is the same as that for the string environment in Figs. **??**–**??**, except that the key type `string` should be replaced by `Ident.Id` and the binding type `'b` should be instantiated to `Type.Ty`. The `Ident.Env` structure is abbreviated as `TEnv` in within the implementation of the `Recon` structure.

To test your solution, first evaluate

```
use("loadProb5.sml");
```

and then evaluate

```
testProb5();
```

You will need to reevaluate *both* of these expressions any time you make a change to `Recon.sml`.

You'll know that a test case succeeds if it concludes by printing `OK!`. Otherwise, an error message will be displayed indicated why the test failed.

```
(program (n)
  (bind f (abs (amount entry)
             (match-tuple (name student? tuition) entry
               (if student?
                   (tuple name student? (+ tuition amount))
                   entry)))
    (f 1 (f 20 (f 300 (tuple "Alyssa Hacker" true n))))))
```

Figure 4: An implicitly typed version of the program in Fig. **??**.

```
(program (n)
  (bindrec ((down-from (abs (x)
                         (if (= x 0)
                             (empty)
                             (prepend x (down-from (- x 1))))))
            (map1 (abs (f lst)
                   (if (empty? lst)
                       (empty)
                       (prepend (f (head lst))
                                (map1 f (tail lst))))))
            (map2 (abs (f lst)
                   (if (empty? lst)
                       (empty)
                       (prepend (f (head lst))
                                (map2 f (tail lst))))))
            (zip (abs (duple-of-lists)
                   (match-tuple (L1 L2) duple-of-lists
                     (if (scor (empty? L1) (empty? L2))
                         (empty)
                         (prepend (tuple (head L1) (head L2))
                                  (zip (tuple (tail L1) (tail L2)))))))))
            (unzip
              (abs (list-of-duples)
                (if (empty? list-of-duples)
                    (tuple (empty) (empty))
                    (match-tuple (tl1 tl2) (unzip (tail list-of-duples))
                      (match-tuple (hd1 hd2) (head list-of-duples)
                        (tuple (prepend hd1 tl1)
                               (prepend hd2 tl2)))))))
            )
    (bind ints (down-from n)
      (bind bools (map1 (abs (x) (= 0 (mod x 2))) ints)
        (unzip (map2 (abs (tup)
                       (match-tuple (a b) tup
                         (if b (tuple a (* a a)) (tuple (- 0 a) a))))
                     (zip (tuple ints bools))))))))
```

Figure 5: An implicitly typed version of the program in Fig. **??**.

A HOFMAD Documentation

```
structure AST = struct

   type Id = Ident.Id    (* local abbreviation *)

   datatype Exp =
     Lit of Literal.Lit
   | VarRef of Id
   | PrimApp of Primitive.Primop * Exp list (* rator, rands *)
   | PrimEmpty of Type.Ty                   (* special exp for typed empty list *)
   | If of Exp * Exp * Exp                  (* test, then, else *)
   | Abs of Id list * Type.Ty list * Exp    (* formals, formalTypes, body *)
   | FunApp of Exp * Exp list               (* names, defns, body *)
   | BindPar of Id list * Exp list * Exp    (* names, defns, body *)
   | BindRec of Id list * Type.Ty list
                * Exp list * Exp            (* names, types, defns, body *)
   (* new in HOFMAD *)
   | Tuple of Exp list                      (* tuple components *)
   | MatchTuple of Id list * Exp * Exp      (* names, tuple, body *)

  datatype Program = Prog of Id list * Exp  (* formals, body *)

end
```

Figure 6: Structure specifying abstract syntax trees for the explicitly typed HOFMAD language.

```
signature TYPE = sig

  datatype Ty =
    UnitTy | IntTy | BoolTy | SymTy | StringTy (* base types *)
  | ListTy of Ty                               (* component type *)
  | ArrowTy of Ty list * Ty                    (* arg types, result type *)
  (* New for HOFMAD *)
  | TupleTy of Ty list                         (* tuple types *)
  | TyVar of TypeVar.TVar                      (* type variables *)

  val equal : Ty * Ty -> bool     (* are two types equal? *)
  val toSexpr : Ty -> Sexpr.Sexpr (* S-expression representation of type *)
  val toString : Ty -> string     (* string representation of type *)

  val TVs : Ty -> TypeVar.Set.set (* set of all type variables appearing in type *)
  val TVsList : Ty list  -> TypeVar.Set.set
    (* Set of all type variables appearing in a list of types. *)

end
```

Figure 7: Signature for the HOFMAD types.

```
signature TYPE_CHECK = sig

  exception TypeCheckError of string
    (* Exception raised when type checking error encountered *)

  val checkProg : AST.Program -> Type.Ty
    (* Returns the type of a well-typed program. Raises TypeCheckError
       if the program is not well-typed. *)

  val checkExp : AST.Exp -> Type.Ty Ident.Env.env -> Type.Ty
    (* Returns the type of a well-typed expression relative to the
       given type environment. Raises TypeCheckError if the expression
       is not well-typed relative to the type environment *)

  val checkString' : string -> Type.Ty
    (* Returns the type of the program parsed from the input string.
       Raises TypeCheckError if the program is not well-typed. *)

  val checkString : string -> Type.Ty
    (* Like checkString', but converts all TypeCheckError exceptions to
       Fail exceptions. This is the better function to call from top-level. *)

  val checkFile' : string -> Type.Ty
    (* Returns the type of the program parsed from file with the given name.
       Raises TypeCheckError if the program is not well-typed. *)

  val checkFile : string -> Type.Ty
    (* Like checkFile', but converts all TypeCheckError exceptions to
       Fail exceptions. This is the better function to call from top-level. *)

  val withStandardHandler : (exn -> 'a) -> (unit -> 'a) ->  'a
    (* Wraps the thunk (second argument) in a standard exception handler
       for TypeCheckError. The first argument allows reraising exception
       or throwing it away (when 'a is unit) *)

end
```

Figure 8: Signature for the HOFMAD type checker.

```
signature STRING_ENV = sig

    type key = string

    type 'b env
    (* Type of env that maps string keys to values of type 'b *)

    val empty : 'b env
    (* empty denotes an empty env *)

    val bind : (string * 'b * 'b env) -> 'b env
    (* bind(key,value,tbl) returns a new env that includes the
       binding key->value in addition to  all bindings of tbl.
       Any existing binding for key in tbl is shadowed by key->value. *)

    val extend : (string list * 'b list * 'b env) -> 'b env
    (* extend(keys,values,tbl) returns a new env that includes
       corresponding bindings between keys and values in addition to
       all bindings of tbl. Any  existing binding for keys in tbl
       are shadowed by the new bindings. *)

    val lookup : (string * 'b env) -> 'b option
    (* lookup(key,tbl) returns SOME(value) if tbl contains the binding
       key->value. If there is no binding for key, lookup returns NONE. *)

    val unbind : (string * 'b env) -> 'b env
    (* unbind(key,tbl) returns a new env that includes all
       bindings of tbl except for a binding for key. *)

    val remove : (string list * 'b env) -> 'b env
    (* unbind(keys,tbl) returns a new env that includes all
       bindings of tbl except for bindings for keys. *)

    val bindingsToEnv : (string * 'b) list -> 'b env
    (* bindingsToEnv(keyValuePairs) returns a env whose bindings consist
       of all the bindings specified by the list of pairs keyValuePairs. *)

    val keys : 'b env -> string list
    (* keys(tbl) returns a list of all keys for bindings in tbl.
       Each key is mentioned only once. *)

    val values : 'b env -> 'b list
    (* values(tbl) returns a list of all values for bindings in tbl.
       The values are in the same order as the keys returned by
       keys(tbl). Because the same value may be bound to more than
       one key, the result may contain duplicates. *)
```

Figure 9: Environment signature, part 1.

```
    val map : ('a -> 'b) -> 'a env -> 'b env
    (* (map f env) returns an environment of bindings {k -> f(v) |
       (k -> v)} is a binding in env. *)

    val mapi : ((key * 'a) -> 'b) -> 'a env -> 'b env
    (* (map f env) returns an environment of bindings {k -> f(k,v) |
       (k -> v)} is a binding in env. *)

    val foldr : (('a * 'b) -> 'b) -> 'b -> 'a env -> 'b
    (* (foldr f z env) returns the result of folding f from right-to-left
       starting with z over the values of env (ordered by key). *)

    val foldri : ((key * 'a * 'b) -> 'b) -> 'b -> 'a env -> 'b
    (* Like foldr, but f takes the key as well as the value and answer *)

    val foldl : (('a * 'b) -> 'b) -> 'b -> 'a env -> 'b
    (* Like foldr, but accumulates values left-to-right *)

    val foldli : ((key* 'a * 'b) -> 'b) -> 'b -> 'a env -> 'b
    (* Like foldl, but f takes the key as well as the value and answer *)
  end
```

Figure 10: Environment signature, part 2.

```
structure AST = struct

  type Id = Ident.Id (* local abbreviation *)
  type Tag = Tag.Tag (* local abbreviation *)

  datatype Program = Prog of Id list * Exp (* formals, body *)

  and Exp =
    Lit of Literal.Lit
  | VarRef of Id
  | Abs of Id list * Exp                    (* formals, body *)
  | FunApp of Exp * Exp list                (* rator, rands *)
  | PrimApp of Primitive.Primop * Exp list  (* rator, rands *)
  | If of Exp * Exp * Exp                   (* test, then, else *)
  | BindRec of Id list * Exp list * Exp     (* names, defns, body *)
  (* new in HOFLAD *)
  | Tuple of Exp list                       (* tuple components *)
  | MatchTuple of Id list * Exp * Exp       (* names, tuple, body *)

end
```

Figure 11: Structure specifying abstract syntax trees for the dynamically typed HOFLAD language.

```
signature SUBST =

sig
    type subst
    exception UnifyError of string
    val empty: subst
    val unify : (Type.Ty * Type.Ty) -> subst
    val unifyList : (Type.Ty list * Type.Ty list) -> subst
    val union : (subst * subst) -> subst
    val unionList : subst list -> subst
    val subTy : subst -> Type.Ty -> Type.Ty
    val subTys : subst -> Type.Ty list -> Type.Ty list
    (* The ASTs here are HOFMAD (not HOFLAD)  ASTs *)
    val subExp : subst -> AST.Exp -> AST.Exp
    val subExps : subst -> AST.Exp list -> AST.Exp list
    val toString : subst -> string
end
```

Figure 12: Signature for type substitutions in HOFMAD. More details on these functions can be found in Handout #33.

```
signature RECON = sig

    val reconProg: Hoflad.AST.Program -> AST.Program
    (* Returns an explicitly typed (HOFMAD) program that is
       a well-typed type-annotated version of the given
       untyped (HOFLAD) program. Raises ReconError if
       no such well-typed program exists. *)

    val reconExp: Hoflad.AST.Exp -> Type.Ty Ident.Env.env
                  -> (AST.Exp * Type.Ty * Subst.subst)
    (* Returns an explicitly typed (HOFMAD) expression that is
       a well-typed type-annotated version of the given
       untyped (HOFLAD) expression in the given type environment.
        Raises ReconError if no such well-typed expression exists. *)

    val reconString' : string -> AST.Program
    (* Returns the explicitly typed program that is the reconstructed
       version of the implicitly typed program in the given string *)

    val reconString : string -> AST.Program
    (* Like reconString', but converts all UnifyError and ReconError exceptions to
       Fail exceptions. This is the better function to call from top-level. *)

    val reconFile' : string -> AST.Program
    (* Returns the explicitly typed program that is the reconstructed
       version of the implicitly typed program in the given filename *)

    val reconFile : string -> AST.Program
    (* Like reconFile', but converts all UnifyError and ReconError exceptions to
       Fail exceptions. This is the better function to call from top-level. *)

    val withStandardHandler : (exn -> 'a) -> (unit -> 'a) ->  'a
    (* Wraps the thunk (second argument) in a standard exception handler
       for ReconError and UnifyError. The first argument allows reraising exception
       or throwing it away (when 'a is unit) *)

    exception ReconError of string

end
```

Figure 13: Signature for the HOFMAD type reconstructer.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 7
Due Tuesday, May 1

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading you problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [20]		
Problem 2 [20]		
Problem 3 [20]		
Problem 4 [20]		
Problem 5 [20]		
Total		