CS251 Programming Languages **Handout # 36**
Prof. Lyn Turbak **Sunday, April 7**
Wellesley College

Problem Set 7

Due: Saturday, April 13, 2002

Reading:

- Handout #30: Type Checking
- Handout #35: Type Reconstruction

Code:

- Problems 2, 3, and 4 require you to do some SML programming. All of the code you need is in the `~/cs251/ps7` directory. To install this code, execute the following in a Linux shell:

```
cd ~cs251/p7
cvs update -d
```

- To test your modifications to the SML programs, create an SML buffer (via `M-x sml`) and make sure you are connected to the `~/cs251/ps7` directory (via `M-x sml-cd`).

Submission:

- Problem 1 is a pencil-and-paper problem that only needs to appear in your hardcopy submission.
- For Problems 2, 3, and 4, your softcopy submission should include a copy of your entire `ps7` directory.
- Your hardcopy submission for Problem 2 should be the files `~/cs251/ps7/hoflad/Eval.sml` and `~/cs251/ps7/hoflad/Parser.sml`.
- Your hardcopy submission for Problem 3 should be the file:

  `~/cs251/ps7/hofmad/TypeCheck.sml`.
- Your hardcopy submission for Problem 4 should be the file:

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

Problem 1 [20]: Monomorphic Type Reconstruction

Following the format of the type 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. You should use the type reconstruction algorithm presented in Handout #35.

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

Problem 2 [40]: Tuples and Variants

In this problem you will extend the SML implementation of HOFL with two new data structures: tuples and variants. We will call the resulting language HOFLAD (for HOFL And Data). This will give you some experience with programming language implementation in SML and will also expose you to some issues concerning data structures.

Tuples

An n-**tuple** is a value with n component values. Tuples are also known as **positional products** because they combine components that are referenced by their positions within the tuple. Tuples are closely related to **records**, also known as a **named products**, in which component values are indexed by names rather than positions. Most programming languages have some sort of tuple and/or record data structure. Examples of record facilities include SML and Haskell's records, C's `struct`, CLU's `record` and `struct`, Common Lisp's `defstruct`, and and Pascal's `record`. A Java class instance can even be viewed as a kind of record whose components are instance variables.

We introduce tuples into the HOFLAD language via the following two new kinds of expressions:

(tuple $E_1 \ \ldots \ E_n$**)**
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 ($I_1 \ \ldots \ I_n$**)** 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 $I_1 \ \ldots \ I_n$ 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 $I_1 \ \ldots \ I_n$ should be distinct.

For example, consider the following HOFLAD abstraction:

```
(abs (amount entry)
  (match-tuple (name student? tuition) entry
    (if student?
        (tuple name student? (+ tuition amount))
        entry)))
```

This abstraction 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.

As another example, consider the HOFL program in Fig. 1, which defines and uses `zip` and `unzip` as well as some other functions:

Variants

A **variant** data structure is a value that consists of an identifying tag and a single component value. Variants are supported by many typed programming languages; they are used to model situations in which the calculation performed on a value depends on the run-time type of the value. Variants are often combined with records into a single data structure that has a tag and multiple component values. Examples of this include Pascal's variant records, SML and Haskell's **sum-of-products datatypes**, and Java's objects (where the class of an object is its variant tag and the instance variables are the record components). It is rarer to see pure variants; examples include C's

```
(program (n)
  (bindrec ((down-from (abs (x)
                          (if (= x 0)
                              (empty)
                              (prepend x (down-from (- x 1))))))
            (map (abs (f lst)
                   (if (empty? lst)
                       lst
                       (prepend (f (head lst))
                                (map 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 (map (abs (x) (= 0 (mod x 2))) ints)
        (unzip (map (abs (tup)
                      (match-tuple (a b) tup
                        (if b (tuple a (* a a)) (tuple (- 0 a) a))))
                    (zip ints bools)))))))
```

Figure 1: A program illustrating tuples in the context of `zip` and `unzip`.

`union` (which must be combined with C's `struct` facility in order to create tags) and the `oneof` and `variant` constructs of the CLU language. Tagged data is common in dynamically typed languages like Scheme; see Sections 2.4 and 2.5 of *SICP* for a discussion.

Here we will consider pure variants and will make the single component of a variant a tuple when we want a variant with multiple components. We will introduce variants into HOFLAD via the following new kinds of expressions.

> **(variant *tag* E)**
> Creates a variant value whose tag is *tag* and whose component value is the value of E.
>
> **(tagcase** E_{disc} **(*tag*$_1$** I_1 E_{body_1}**)** ... **(*tag*$_n$** I_n E_{body_n}**))**
> Evaluates the **discriminant** expression E_{disc} to the value V_{disc}, which should be a variant value; otherwise `tagcase` signals an error. If the tag of V_{disc} is the same as the tag tag_i in the ith `tagcase` clause, the `tagcase` returns the value of E_{body_i} in a context where I_i is bound to the component value of V_{disc} and the meanings of all other names are determined by the lexical context in which the `tagcase` expression appears. If the tag of V_{disc} does not match any tag_i, then an error is signalled. Note that all the tag_i should be distinct.

As an example of variants, consider a simple system for manipulating geometric figures like squares, rectangles, and triangles. Each kind of figure is characterized by different information:

- a square has a side length;
- a rectangle has a width and height;
- a triangle has three side lengths (alternatively, it could be represented by combinations of side lengths and angles).

Here is an example of a HOFLAD expression manipulating figures:

```
(bind perim
    (abs (fig)
      (tagcase fig
        (sqr side (* 4 side))
        (rect width*height
          (match-tuple (width height) width*height
            (* 2 (+ width height))))
        (tri sides
          (match-tuple (s1 s2 s3) sides
            (+ s1 (+ s2 s3))))
        ))
  (prepend (perim (variant rect (tuple 2 3)))
    (prepend (perim (variant tri (tuple 4 1 6)))
      (prepend (perim (variant sqr 5))
        (empty)))))
```

The `perim` function uses a `tagcase` to discriminate on its figure argument and perform a perimeter calculation that depends on the kind of figure.

As another example, here is a function that scales each dimension of a figure by a given scaling factor `sf`.

```
(abs (sf fig)
  (tagcase fig
    (sqr side (variant sqr (* sf side)))
    (rect width*height
      (match-tuple (width height) width*height
        (variant rect (tuple (* sf width) (* sf height)))))
    (tri sides
      (match-tuple (s1 s2 s3) sides
        (variant tri (tuple (* sf s1) (* sf s2) (* sf s3)))))
    ))
```

Your Task

In the following parts, you will be extending the SML evaluator of HOFL that we studied in class to the HOFLAD language that supports tuples and variants. The abstract syntax, parser, and pretty-printers of HOFL have already been extended to support the `tuple`, `match-tuple`, `variant`, and `tagcase` constructs of HOFLAD. See Appendix A for a summary of HOFLAD.

a. [10]: Tuples

Modify the `eval` function in `~/cs251/ps7/hoflad/Eval.sml` to handle the the `tuple` and `match-tuple` constructs. See the notes in Appendix A for details concerning the evaluator and environments. Note in particular that all non-`bindrec`-bound values inserted into an environment must have the `NonRecVal` tag; see Section A.2 for details.

To test your solution, first evaluate

```
use("loadProb2.sml");
```

and then evaluate

```
testProb2a();
```

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

b. [15]: Variants

Modify the HOFL evaluator in `~/cs251/ps7/hoflad/Eval.sml` to evaluate the `tuple` and `match-tuple` constructs. You can test these as in Part (a), except that you should evaluate `testProb2b()` as your second expression.

c. [15]: Sum-of-products Data

As seen in the variant examples above, the proposed variant notation is rather clumsy. Typically, a variant will have a tuple as a component, and the `tuple` and `match-tuple` constructs for manipulating these are cumbersome. As noted above, many languages combine the notion of variants and tuples into a single sum-of-product data type. Here will will do this by adding the constructs `data` and `datacase`, whose syntax and meaning are introduced by the following examples:

```
(bind perim
    (abs (fig)
      (datacase fig
        (sqr (side) (* 4 side))
        (rect (width height)
          (* 2 (+ width height)))
        (tri (s1 s2 s3)
          (+ s1 (+ s2 s3)))
        ))
  (prepend (perim (data rect 2 3))
    (prepend (perim (data tri 4 1 6))
      (prepend (perim (data sqr 5))
        (empty)))))

(abs (sf fig)
  (datacase fig
    (sqr (side) (data sqr (* sf side)))
    (rect (width height)
      (data rect (* sf width) (* sf height)))
    (tri (s1 s2 s3)
      (data tri (* sf s1) (* sf s2) (* sf s3)))
    ))
```

Here is a more formal specification of `data` and `datacase`:

> **(data *tag* $E_1 \ldots E_n$)**
> Creates a sum-of-products variant value whose tag is *tag* and whose component is a tuple whose values are the value of $E_1 \ldots E_n$.
>
> **(datacase** E_{disc} **(*tag*$_1$** $(I_{(1,1)} \ldots I_{(1,k_1)})$ E_{body_1}**)** $\ldots$ **(*tag*$_n$** $(I_{(n,1)} \ldots I_{(n,k_n)})$ E_{body_n}**))**
> Evaluates the **discriminant** expression E_{disc} to the value V_{disc}, which should be a variant value; otherwise `datacase` signals an error. If the tag of V_{disc} is the same as the tag tag_i in the ith `datacase` clause, the `datacase` returns the value of E_{body_i} in a context where $I_{(i,1)} \ldots I_{(i,k_i)}$ are bound to the components of the tuple that is the component of the variant value V_{disc}, and the meanings of all other names are determined by the lexical context in which the `datacase` expression appears. If the tag of V_{disc} does not match any tag_i, or if the component of V_{disc} is not a tuple, or if the number of names $I_{(i,1)} \ldots I_{(i,k_i)}$ does not match the number of components in the tuple within V_{disc}, then an error is signalled. Note that all the tag_i should be distinct, and for any i, all the $I_{(i,1)} \ldots I_{(i,k_i)}$ should be distinct.

It would be possible to add `data` and `datacase` to HOFLAD by extending the abstract syntax type `Exp`. But a much easier way to achieve this result is to desugar `data` and `datacase` into appropriate variant and tuple commands. In this problem, you are to implement an appropriate desugaring for `data` and `datacase` by extending the desugaring section of `~/cs251/ps7/hoflad/Parser`. Before attempting this problem, you should (1) study the notes on HOFLAD parsing in Appendix A.3; (2) should study the parsers for `tuple`, `match-tuple`, `variant`, and `tagcase` in `~/cs251/ps7/hoflad/Parser`; and (3) should study the desugarings for `bind`, `bindseq`, `scand`, `scor`, and `funrec` in `~/cs251/ps7/hoflad/Parser`.

As part of your desugaring for `datacase`, you will need to introduce a name for the tuple value that is the component of a variant. Rather than using some sort of machinery for generating

fresh names, you should instead choose an arbitrary name that begins with the character #; because # is treated specially by the HOFLAD parser, such a name could not have been in the original HOFLAD program, and so cannot accidentally capture any variables from the original program. (You should convince yourself that one occurrence of the name you choose also cannot accidentally capture another occurence of the same name!)

To test your solution, first evaluate

```
use("loadProb2.sml");
```

and then evaluate

```
testProb2c();
```

Problem 3 [20]: Type Checking Tuples

In Problem 2, 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 two constructs. (It is also possible to extend HOFLEMT to handle variants, but this involves some complexities we wish to avoid, and so we ignore variants in the explicitly typed language.)

Extending the explicitly typed HOFLEMT with the `tuple` and `match-tuple` constructs 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. 2, which is an explicitly typed version of a similar example from Problem 2: 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 Fig. 3, which defines and uses versions of `zip`, `unzip`, and `map` that manipulate tuples. Convince yourself that the program is well-typed.

Your Task

a. [5]: Typing Rules

Write down typing rules for `tuple` and `match-tuple` in the style of the typing rules in figure 4 of Handout #30.

b. [15]: Type Checking Implementation

In this part, you will extend the SML type checker for HOFLEMT from Handout #30 that we studied in class to handle the `tuple` and `match-tuple` constructs of HOFMAD. An almost

```
(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`.

complete implementation of HOFMAD can be found in `~/cs251/ps7/hoflmad`. 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 B describes the resources you will need for this problem.

- The abstract syntax for HOFMAD expressions can be found in Fig. 13.
- The signature for HOFMAD types can be found in Fig. 14.
- The signature for the HOFMAD type checker can be found in Fig. 15.
- The signature for the type environments used by the type checker (`Ident.Env`) is given in Figs. 11 and 12 in App. A. In the case of type checking, the binding type `'b` is 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("loadProb3.sml");
```

and then evaluate

```
testProb3();
```

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 4 [20]: Reconstructing Tuple Types

In this problem, you will extend the type reconstructer for HOFLIMT presented in Handout #35 to handle the `tuple` and `match-tuple` constructs. Your extended type reconstructer should be able to transform the implicitly typed HOFLAD programs in Figs. 4 and 5 to the corresponding explicitly typed programs in Figs. 2 and 3.

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 the appendices describes the resources you will need for this problem.

- The abstract syntax for *implicitly typed* HOFLAD expressions can be found in Fig. 6 in appendix A. 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. 13 in appendix B. 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. 14 in appendix B.
- The signature for the HOFMAD type subsitutions can be found in Fig. 16 in appendix B. 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. 17 in Fig. 14 in appendix B.
- The signature for the type environments used by the type reconstructer (`Ident.Env`) is given in Figs. 11 and 12 in App. A. In the case of type reconstruction, the binding type `'b` is 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("loadProb4.sml");
```

and then evaluate

```
testProb4();
```

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. 2.

```
(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. 3.

A HOFLAD

A.1 Abstract Syntax

The abstract syntax, parser, and pretty-printers of HOFL have been extended to support the `tuple`, `match-tuple`, `variant`, and `tagcase` constructs of HOFLAD.

Fig. 6 presents the abstract syntax of HOFLAD. The first seven constructors have been inherited from HOFL; the last four constructors are new in HOFLAD.

```
  and Exp =
    Lit of Literal
  | 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 *)
  | Variant of Tag * Exp                    (* tag, component *)
  | TagCase of Exp * (Tag * Id * Exp) list  (* discriminant, clauses *)
```

Figure 6: The abstract syntax of HOFLAD expressions.

The HOFLAD syntax uses the types `Id` and `Tag`, which are from the `Ident` and `Tag` structures, respectively. The signatures for these structures are almost identical, and are given in Figs. 7 and 8.

```
signature IDENT = sig
  type Id                         (* abstract type of identifiers *)
  val toString : Id -> string     (* returns string for given identifier *)
  val fromString : string ->  Id (* returns identifier for given string *)
  val compare : Id * Id -> order (* compare two identifiers *)
  val equal : Id * Id -> bool     (* test equality of two identifiers *)
  structure Env : ENV             (* environment whose keys are identifiers *)
  sharing type Env.key = Id
end
```

Figure 7: Signature for HOFLAD identifiers.

```
signature TAG = sig
  type Tag                          (* abstract type of tags *)
  val toString : Tag -> string      (* returns string for given tag *)
  val fromString : string ->  Tag  (* returns tag for given string *)
  val compare : Tag * Tag -> order (* compare two tags *)
  val equal : Tag * Tag -> bool     (* test equality of two tags *)
  structure Env : ENV               (* environment whose keys are tags *)
  sharing type Env.key = Tag
end
```

Figure 8: Signature for HOFLAD tags.

A.2 Evaluation

The datatype `Val` of HOFL values has been extended in HOFLAD to support tuple and variant values; see Fig. 9. The signature for HOFLAD evaluation appears in Fig. 10 and the signature for environments appears in Figs. 11 and 12; these are the same as in HOFL. In the case of evaluation, environments map keys of type `Ident.Id` (which you can think of as a synonym for `string`) to values of type `EvalValue` (i.e., the type to which `'b` is instantiated in the environment signature). As in HOFL, the `EvalValue` type is the type of values stored in the environment of the evaluator. The `RecVal` constructor of `EvalValue` is used to implement the "knot-tying" aspect of `bindrec`. Environment bindings not introduced by `bindrec` need to be tagged with the `NonRecVal` constructor.

```
signature VALUE = sig

  datatype Val =
     UnitVal
   | IntVal of int
   | BoolVal of bool
   | SymVal of string
   | StringVal of string
   | ListVal of Val list
   | ClosureVal of Ident.Id list * AST.Exp * EvalValue Ident.Env.env
   (* new in HOFLAD *)
   | TupleVal of Val list
   | VariantVal of Tag.Tag * Val

   (* It would be nicer to have EvalValue hidden within Eval,
      but ML's types and restrictions on non-recursive modules
      require us to put it here. An alternative would be to
      define IdentEnv in a way that it did not require us
      to specify the type of the values bound in the environment. *)
   and EvalValue =
     NonRecVal of Val
   | RecVal of Val option ref (* use SML's mutable reference cells
                                 to "tie the knot" of recursion *)

   val equal : (Val * Val) -> bool
   (* test value equality *)

   val toString : Val -> string
   (* return a string representation of a value *)

end
```

Figure 9: Signature for HOFLAD values.

```
signature EVAL = sig

  val run : AST.Program -> int list ->  Value.Val
  (* Returns the result of running the given program on the given list
     of arguments *)

  val eval : AST.Exp -> Value.EvalValue Ident.Env.env -> Value.Val
  (* Returns the result of evaluating the given expression in the
     given environment *)

  val runString' : string -> int list -> Value.Val
  (* Returns the result of running the program that is the result of
     parsing the given string on the given argument list. *)

  val runString : string -> int list -> Value.Val
  (* Like runString', but runs the program in the context of a standard
     error handler. This is the version that should be called from
     top-level. *)

  val runFile' : string -> int list -> Value.Val
  (* Returns the result of running the program that is the contents of
     the file with the given name on the given argument list. *)

  val runFile : string -> int list -> Value.Val
  (* Like runString', but runs the program in the context of a standard
     error handler. This is the version that should be called 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 the exception
       or throwing it away (when 'a is unit) *)

end
```

Figure 10: Signature for HOFLAD evaluator.

```
signature IDENT_ENV = sig

    type key = Ident.Id

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

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

    val bind : (Ident.Id * '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 : (Ident.Id 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 : (Ident.Id * '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 : (Ident.Id * '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 : (Ident.Id 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 : (Ident.Id * '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 -> Ident.Id 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 11: 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 12: Environment signature, part 2.

A.3 Parsing

A **parser** is a function that translates a character-based representation of a program into an abstract syntax tree. The HOFLAD parser has a two phase structure:

1. In the first phase, the HOFLAD parser translates a character-based representation of a HOFLAD program into a tree for the symbolic expression datatype `Sexpr`. The `Sexpr` datatype models the tree structure of Scheme's parenthesized s-expressions:

```
datatype Sexpr =
  Intx of int
| Boolx of bool
| Charx of char
| Realx of real
| Symx of string
| Stringx of string
| Listx of Sexpr list
```

   There is one constructor for each kind of leaf (integer, boolean, character, real, symbol, and string) and a constructor for lists of s-expressions (which are denoted by paired parentheses in Scheme). Each constructor name ends with an `x` to emphasize that it is a constructor for s-eXpressions. For example, the HOFLAD program

```
(program (a b)
  (if (< a (* 2 b)) "foo" "bar"))
```

   would be parsed as the following `Sexpr` tree:

```
Listx [Symx("program"),
       Listx [Symx("a"), Symx("b")],
       Listx [Symx("if"),
              Listx [Symx("<"),
                     Symx("a"),
                     Listx [Symx("*"), Intx(2), Symx("b")]
                     ],
              Stringx("foo"),
              Stringx("bar")
              ]
       ]
```

2. In the second phase, the HOFLAD parser translates the `Sexpr` tree into a HOFLAD abstract syntax tree. The HOFLAD AST for the above sample program is:

```
Prog([Ident.fromString("a"), Ident.fromString("b")],
     If(PrimApp(LT,
                VarRef(Ident.fromString("a")),
                PrimApp(Mul,
                        Lit(IntLit(2)),
                        VarRef(Ident.fromString("b")))),
        Lit(StringLit("foo")),
        Lit(StringLit("bar"))))
```

Using a two-phase parser allows the `Sexpr` parser to be shared with other languages. In fact, HOFL and HOFLEMT use the very same `Sexpr` parser as HOFLAD.

Expression parsing is accomplished via the `toExp` function in the file `~/cs251/ps7/hoflad/Parser`. For instance, here are the clauses responsible for parsing an `Sexpr` into an HOFLAD `abs` or `if` expression:

```
(* parse abstraction *)
| toExp (Listx([Symx "abs", Listx(formalsx), bodyx])) =
    Abs(map (fn (Symx(fml)) => Ident.fromString(fml)
              | s => parseErr ("toExp: improper abs formal",s))
            formalsx,
        toExp bodyx)
| toExp (x as (Listx((Symx "abs") :: _))) =
    parseErr ("toExp: Ill-formed abstraction", x)

(* parse conditional *)
| toExp (Listx[(Symx "if"), x1, x2, x3]) =
    If(toExp(x1), toExp(x2), toExp(x3))
| toExp (x as (Listx((Symx "if") :: _))) =
    parseErr ("toExp: Ill-formed if", x)
```

The second clause of each pair of clauses treats as an error any s-expression beginning with the keyword (i.e., `abs` or `if`) that does not have the expected structure.

The `toExp` function also performs desugaring. For example, the desugaring of `scand` is implemented by the following clauses:

```
(* desugar scand *)
| toExp (Listx([Symx("scand"),x1,x2])) =
    If(toExp x1, toExp x2, Lit(BoolLit(false)))
| toExp (x as (Listx((Symx("scand")) :: _))) =
    parseErr ("toExp: Ill-formed scand ", x)
```

B HOFMAD Documentation

This section documents the HOFMAD language. In particular:

- Fig. 13 gives the abstract syntax for HOFMAD, which is the same as that for HOFLAD except for the removal of the `Variant` and `TagCase` constructors.
- Fig. 14 gives the types for HOFMAD, which extends the types of of HOFLEMT with the `TupleTy` constructor.
- Fig. 15 gives the signature for the HOFMAD type checker.
- Fig. 16 gives the signature for type substitutions in HOFMAD.
- Fig. 17 gives the signature for the HOFMAD type reconstructor.

```
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 13: 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 14: 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 15: Signature for the HOFMAD type checker.

```
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 16: Signature for type substitutions in HOFMAD. More details on these functions can be found in Handout #35.

```
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 17: 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 Saturday, April 13

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 [40]		
Problem 3 [20]		
Problem 4 [20]		
Total		