Wellesley College ◊ CS251 Programming Languages ◊ Spring, 2000

PROBLEM SET 5
Due Thursday, April 27, 2000

This is a revised version of PS5. The main revisions are:
(1) Numerous bugs in Problems 1, 2, and 3 have been fixed.
(2) What was problem 4 has now been made an extra credit problem.
(3) Points have been redistributed so that Problems 1, 2, and 3 are worth 100 points.
(4) Appendix B now contains detailed instructions on how to run the HOFLIPT/HOFLEPT type checkers and evaluators.

Reading

(5) The ML programming language*: Paulson's ML for the Working Programmer (MLWP),* Chapters 2, 3, 4, 5.1-5.11, 9.

(6) Handout on Standard ML of New Jersey (in preparation)

(7) Handout on Types (in preparation)

Problem 1 [20]: ML Types

On the next page are nineteen higher order ML functions. For each function, write down the type that would be reconstructed for it in ML.

For example, consider the following ML `length` function:

```
fun length [] = 0
        | length (_::xs) = 1 + (length xs)
```

The ML type of this function is:

```
val length : 'a list -> int
```

Note: you can check your answers by typing them into the ML interpreter. But please write down the answers first before you check them – otherwise you will not learn anything!

```
fun id x = x

fun compose f g x = (f (g x))

fun repeated n f =
    if (n = 0) then id else compose f (repeated (n - 1) f)

fun uncurry f (a,b) = (f a b)

fun curry f a b = f(a,b)

fun generate seed next done =
    if (done seed)
    then []
    else seed :: (generate (next seed) next done)

fun map f [] = []
  | map f (x::xs) = (f x) :: (map f xs)

fun filter pred [] = []
  | filter pred (x::xs) =
     if (pred x)
     then x::(filter pred xs)
     else (filter pred xs)

fun zip ([], _) = []
  | zip (_, []) = []
  | zip (x::xs, y::ys) = (x,y)::(zip(xs,ys))

fun unzip [] = ([], [])
  | unzip ((x,y)::xys) =
     let val (xs,ys) = unzip xys
      in (x::xs, y::ys)
     end

fun foldr binop init [] = init
  | foldr binop init (x::xs) =
      binop(x, foldr binop init xs)

fun foldr2 ternop init xs ys =
     foldr (fn ((x,y), ans) => ternop(x,y,ans)) init (zip(xs,ys))

fun flatten lst = foldr op@ [] lst

fun forall pred [] = true
  | forall pred (x::xs) =
     pred(x) andalso (forall pred xs)

fun forall2 pred lst1 lst2 = forall pred (zip(lst1,lst2))

fun exists pred [] = false
  | exists pred (x::xs) = (pred x) orelse (exists pred xs)

fun exists2 pred lst1 lst2 = exists pred (zip(lst1,lst2))

fun some pred [] = NONE
  | some pred (x::xs) = if (pred x) then SOME x else some pred xs

fun some2 pred lst1 lst2 = some pred (zip(lst1,lst2))
```

Problem 2 [40]: Explicit Types

Recall that HOFLIPT is a language with implicit polymorphic types, while HOFLEPT has explicit polymorphic types. If a HOFLIPT program is well typed, it is possible to translate it into a well-typed HOFLEPT program by adding appropriate type annotations. As an example, consider the following HOFLIPT program:

```
(program (hi)
  (bindrec ((map (abs (f lst)
                   (if (empty? lst)
                       (empty)
                       (prepend (f (head lst))
                                (map f (tail lst))))))
            (from-to (abs (lo)
                      (if (> lo hi)
                         (empty)
                         (prepend lo (from-to (+ lo 1)))))))
    (bind test-list (from-to 1)
      (prepend (map (abs (n) (prepend n (empty)))
                    (map (abs (x) (* x x)) test-list))
               (prepend (map (abs (b)
                                (if b
                                    (prepend 1 (empty))
                                    (prepend 0 (empty))))
                             (map (abs (y) (= (mod y 2) 0))
                                  test-list))
                        (prepend (map (abs (z)
                                       (prepend z
                                                (prepend (* 2 z)
                                                         (empty))))
                                      test-list)
                                 (empty)))))))
```

Below is the result of translating the above program to HOFLEPT. The explicit type annotations are shown in bold:

```
(program (hi)
  (bindrec
    ((map (forall (a b)
            (-> ((-> (a) b) (listof a))
                (listof b)))
      (pabs (a b)
        (abs ((f (-> (a) b)) (lst (listof a)))
          (if (empty? lst)
              (empty b)
              (prepend (f (head lst))
                       ((papp map a b) f (tail lst))))))))
     (from-to (-> (int) (listof int))
      (abs ((lo int))
        (if (> lo hi)
            (empty int)
            (prepend lo (from-to (+ lo 1))))))
     )
    (bind test-list (from-to 1)
      (prepend ((papp map int (listof int))
                (abs ((n int)) (prepend n (empty int)))
                ((papp map int int)
                 (abs ((x int)) (* x x))
                 test-list))
        (prepend ((papp map bool (listof int))
                  (abs ((b bool))
                    (if b
                        (prepend 1 (empty int))
                        (prepend 0 (empty int))))
                  ((papp map int bool)
                   (abs ((y int)) (= (mod y 2) 0))
                   test-list))
          (prepend ((papp map int (listof int))
                    (abs ((z int))
                      (prepend z
                        (prepend (* 2 z)
                          (empty int))))
                    test-list)
                   (empty (listof (listof int))))))))))
```

On the next page are three well-typed programs in the implicitly typed polymorphic HOFLIPT language. For each of the three programs, translate the program into the explicitly typed HOFLEPT language.

You can test your answers to this problem using the HOFLEPT type checker. To run the HOFLEPT type checker, follow the instructions in Appendix B.

```
(program (a)
  (bindrec ((sigma
             (abs (lo hi f)
               (if (> lo hi)
                   0
                   (+ (f lo) (sigma (+ lo 1) hi f))))))
    (sigma 1 a (abs (x) (* x x)))))

(program (b)
  (bindrec ((generate
             (abs (seed next done?)
               (if (done? seed)
                   (empty)
                   (prepend seed (generate (next seed) next done?)))))
            (foldr
              (abs (binop init lst)
                   (if (empty? lst)
                      init
                      (binop (head lst)
                             (foldr binop init (tail lst)))))))
    (bind lst (generate b (abs (x) (- x 1)) (abs (y) (= y 0)))
      (if (foldr (abs (x y) (scor (= x 3) y)) #f lst)
          (foldr (abs (x y) (+ x y)) 0 lst)
          (foldr (abs (x y) (* x y)) 1 lst)))))

(program (c)
  (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) c))
      (+ (nat (abs (h) (compose (thrice h) (thrice h))))
         (+ (nat (compose thrice thrice))
            (nat (thrice thrice)))))))
```

Problem 3 [40]: Type Derivations

Consider the following implicitly typed HOFLIPT abstraction:

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

Part a. Translate the above abstraction into an explicitly typed HOFLEPT abstraction. You may assume that `map` has the following type:

```
map: (forall (a b) (-> ((-> (a) b) (listof a)) (listof b)))
```

Part b. Give the type of the explicitly typed abstraction from part a.

Part c. Give a typing derivation that proves that the explicitly typed abstraction from Part a has the type from Part b. You may assume that the abstraction is typed relative to the following type environment:

A_1 = `{map: (forall (a b) (-> ((-> (a) b) (listof a)) (listof b)))}`

That is, the conclusion of your type derivation should be a typing judgement A_1 `|- E : T`, where `E` is the explicitly typed expression from Part a and `T` is the type from Part b.

Recall that a typing derivation is a proof tree – that is, a tree whose nodes are judgements, where each such node and its subnodes are, respectively, the conclusion and hypotheses of an instantiation of a typing rule. The typing rules for HOFLEPT are given in Appendix A.

To simplify the presentation of the type derivation, feel free to introduce abbreviations for type environments, expressions, and types where appropriate.

Extra Credit Problem 1 [40] : Tuple Types

NOTE: THIS PROBLEM IS STILL UNDER CONSTRUCTION. THE PROBLEM IS NOT LIKELY TO CHANGE, BUT DETAILS OF HOW TO RUN THE CODE AND MODIFY THE CODE STILL NEED TO BE ADDED.

In this problem you will extend HOFLEPT and HOFLIPT with constructs that support a record data type. You will get ML programming experience by extending the HOFLIPT evaluator, the HOFLEPT type checker, and the HOFLIPT type reconstructor to handle the new tuple constructs.

We introduce records in the implicitly typed HOFLIPT language via the following two constructs:

- `(tuple` E_1 `...` E_n`)` creates a tuple value with *n* component values, where the *i*th 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. 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.

To handle tuples in the explicitly typed HOFLEPT language, we extend the expression syntax with the above two new constructs, and the type syntax with the following new type constructor:

`(tupleof` T_1 `...` T_n`)`

The `tupleof` type represents the type of a tuple with n component values whose *i*th value (starting at index 1) has type T_i. As in ML, the types of the tuple components are independent. Note that no explicit type annotations are needed in the syntax for HOFLEPT record expressions.

For example, consider the following HOFLEPT abstraction:

```
(abs ((amount int)
      (entry (tupleof string bool int)))
  (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 following HOFLIPT program, which defines and uses `zip` and `unzip` as well as some other functions:

```
(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)))))))
```

In the following parts, you will be extending the ML implementations of HOFLIPT and HOFLEPT to support tuples. You should begin by making a local copy of the following folder:

```
/usr/users/cs251/download/typed-hofl
```

This folder contains the files of a working implementation of both the HOFLEPT and HOFLIPT languages. The expression and type syntax for these languages have been extended as follows to represent the tuple constructs:

TUPLE SYNTAX GOES HERE!

The parsers and unparsers for the two languages have been extended to handle the new constructs. Additionally, the type eraser that erases HOFLEPT programs into HOFLIPT programs has also been extended to handle the tuple constructs.

EXPLAIN TUPLE VALUES

To complete the following parts, you will need to modify the following files:

FILE LISTING GOES HERE!

For your hardcopy submission, you should turn in copies of your final versions of all of the above files.

HOW TO RUN THE VARIOUS SYSTEMS!

Part a. Modify the HOFLIPT interpreter in the file `Eval.sml` to handle the `tuple` and `match-tuple` constructs. You may assume that every HOFLIPT program is the type erasure of a well-typed HOFLEPT program, so you do not need to check for any type errors in the interpreter.

HOW TO TEST THE INTERPRETER!

Part b. Modify the HOFLEPT type checker in the file `TypeChecker.sml` to handle the `tuple` and `match-tuple` constructs.

DISCUSS EXCEPTIONS

HOW TO TEST THE TYPE CHECKER

Part c. Modify the HOFLIPT type reconstructor in the file `TypeReconstructor.sml` to handle the `tuple` and `match-tuple` constructs.

DISCUSS GENERICS

DISCUSS EXCEPTIONS

HOW TO TEST THE TYPE RECONSTRUCTOR

Extra Credit Problem 2 [50] Record Types

Many languages (such as ML, Pascal, and C) support records, which are similar to tuples except that the components are referred to by name rather than by position. In this problem, we consider extending HOFLEPT and HOFLIPT with constructs that support records.

We extend the expression syntax of both HOFLEPT and HOFLIPT with the following two new record constructs:

- `(record (`F_1 E_1`) ... (`F_n E_n`))` creates a record value that binds each field name F_i to the value of the corresponding field expression E_i.

- `(match-record ((`F_1 I_1`) ... (`F_n I_n`))` E_{rec} E_{body}`)` evaluates E_{rec} to the value V_{rec}, which should be a record value with field names F_1 ... F_n. . It returns the value of E_{body} in an environment where the names I_1 ... I_n are bound, respectively, to the components of V_{rec} with field names F_1 ... F_n and the meanings of all other names are determined by the lexical context in which the `match-rec` expression appears.

The HOFLEPT type syntax is extended with the following type constructor:

`(recordof (`F_1 T_1`) ... (`F_n T_n`))`

The `recordof` type represents the type of a record with n component values named F_1 ... F_n, where the component named by field name F_i has type T_i.

For example, here is one of the tuple examples from Problem 4 expressed using records:

```
(abs ((amount int)
      (entry (recordof (name string) (student? bool) (tuition int)))
  (match-record ((name n) (student? s) (tuition t)) entry
```

```
    (if s
        (record (name n) (student? s) (tuition (+ t amount)))
        entry)))
```

Supposing that the above abstraction is named `increase-tuition`, here is a sample invocation:

```
(increase-tuition 3000
                  (record (name "Abby Stracksen")
                          (student? true)
                          (tuition 25000)))
```

Note that the order of named fields within a record is irrelevant; only the association of names and values within `record` (or names and types within `recordof`) matters. For example, the `increase-tuition` abstraction could also be invoked as:

```
(increase-tuition 3000
                  (record (student? true)
                          (tuition 25000)
                          (name "Abby Stracksen")))
```

and it could be defined as follows without any change in meaning:

```
 (abs ((amount int)
       (entry (recordof (tuition int) (name string) (student? bool)))
  (match-record ((name n) (tuition t) (student? s)) entry
    (if s
        (record (student? s) (name n) (tuition (+ t amount)))
        entry)))
```

In the presence of records, it is no longer necessary to handle tuples specially because they can be desugared into records with field names that are integers:

`(tuple` E_1 `...` E_n`))` *desugars to* `(record (1` E_1`) ...` `(`n E_n`))`

`(match-tuple (`I_1 `...` I_n`)` E_{tup} E_{body}`)` *desugars to*
`(match-record ((1` I_1`) ...` `(`n I_n`))` E_{tup} E_{body}`)`

`(tupleof` T_1 `...` T_n`))` *desugars to* `(recordof (1` T_1`) ...` `(`n T_n`))`

Based on the above description, modify the following components of the ML implementations of HOFLEPT and HOFLIPT to handle records:

HOFLEPT:
- The term and type syntax
- The type checker
- The type eraser

HOFLIPT:
- The term syntax
- The interpreter
- The type reconstructor

Appendix A: HOFLEPT Typing Rules

In the following typing rules, we use the following metavariable conventions:

(8) A ranges over type environments
(9) B ranges over boolean literals
(10) E ranges over expressions
(11) I ranges over term variables
(12) J ranges over type variables
(13) S ranges over strings
(14) T ranges over types

A type environment maps term variables to types. If A is a type environment, I is a term variable, and T is a type, we will use the notation A(I) to denote the type bound to I in type environment A, and A[I1 → T1]… [In → Tn] to stand for the environment A extended with bindings between I1…In and T1…Tn, respectively.

If T0, T1, …, Tn are types and J1, …, Jn are type variables, we use the notation T0[T1,…,Tn/J1,…,Jn] to denote the result of simultaneously substituting the types T1,…,Tn for the variables J1,…,Jn in the type T0. For example, `(-> (a b) a)[(listof b), a/a, b]` denotes the type `(-> ((listof b) a) (listof b))`.

Recall that typing judgements are written as "A |-- E : T", which states that expression E has type T in type environment A.

A typing rule consists of a collection of typing judgements that are hypotheses (above the horizontal line) and a single typing judgement that is a conclusion (below the horizontal line). A typing derivation is a tree whose nodes are instantiated typing rules where the hypotheses of one node are the conclusions of its children. A typing derivation is a proof that the typing judgement at the root of the tree is correct.

Below are the typing rules for the explicitly typed language HOFLEPT. Typing rules for the implicitly typed language HOFLIPT can be obtained by erasing all the type annotations in the terms.

(int) ----------
A |-- N : `int`

(bool) ----------
A |-- B : `bool`

(string) ----------
A |-- B : `bool`

(var) ----------
A |-- I : A(I)

$$\text{(add)}\ \frac{A \vdash E1 : \texttt{int};\ A \vdash E2 : \texttt{int}}{A \vdash (+\ E1\ E2) : \texttt{int}}$$

There are analogous rules for (sub), (mul), (div) and (mod)

$$\text{(lt)}\ \frac{A \vdash E1 : \texttt{int};\ A \vdash E2 : \texttt{int}}{A \vdash (<\ E1\ E2) : \texttt{bool}}$$

There are analogous rules for (leq), (eq), (neq), (geq), and (gt)

$$\text{(band)}\ \frac{A \vdash E1 : \texttt{bool};\ A \vdash E2 : \texttt{bool}}{A \vdash (\texttt{band}\ E1\ E2) : \texttt{bool}}$$

There is an analogous rule for (bor)

$$\text{(not)}\ \frac{A \vdash E : \texttt{bool};}{A \vdash (\texttt{not}\ E) : \texttt{bool}}$$

$$\text{(empty)}\ \frac{}{A \vdash (\texttt{empty T}) : (\texttt{listof T})}$$

$$\text{(empty?)}\ \frac{A \vdash \texttt{E} : (\texttt{listof T})}{A \vdash (\texttt{empty? E}) : \texttt{bool}}$$

$$\text{(head)}\ \frac{A \vdash \texttt{E} : (\texttt{listof T})}{A \vdash (\texttt{head E}) : \texttt{T}}$$

$$\text{(tail)}\ \frac{A \vdash \texttt{E} : (\texttt{listof T})}{A \vdash (\texttt{tail E}) : (\texttt{listof T})}$$

$$\text{(prepend)}\ \frac{A \vdash \texttt{E1} : \texttt{T};\quad A \vdash \texttt{E2} : (\texttt{listof T})}{A \vdash (\texttt{prepend E1 E2}) : (\texttt{listof T})}$$

$$\text{(prepend)}\ \frac{A \vdash \texttt{E1} : \texttt{bool};\quad A \vdash \texttt{E2} : \texttt{T};\ A \vdash \texttt{E3} : \texttt{T}}{A \vdash (\texttt{if E1 E2 E3}) : \texttt{T}}$$

$$\frac{A[I1 \rightarrow T1]\ldots[In \rightarrow Tn] \vdash \texttt{E} : \texttt{T}}{A \vdash \texttt{(abs ((I1 T1)... (In Tn)) E) : (-> (T1 ... Tn) T)}}\ (\text{abs})$$

$$\frac{A \vdash \texttt{E0 : (-> (T1 ... Tn) T)};\ A \vdash \texttt{E1 : T1; ...;}\ A \vdash \texttt{En : Tn}}{A \vdash \texttt{(E0 E1 . . . En) : T}}\ (\text{app})$$

$$\frac{A \vdash \texttt{E1 : T1; ...;}\ A \vdash \texttt{En : Tn;}\ A[I1 \rightarrow T1]\ldots[In \rightarrow Tn] \vdash \texttt{E : T}}{A \vdash \texttt{(bindpar ((I1 E1)...(In En)) E) : T}}\ (\text{bindpar})$$

$$\frac{Arec \vdash \texttt{E1 : T1; ...;}\ Arec \vdash \texttt{En : Tn;}\ Arec \vdash \texttt{E : T}}{A \vdash \texttt{(bindrec ((I1 T1 E1)...(In Tn En)) E) : T}}\ (\text{bindrec})$$

where Arec = $A[I1 \rightarrow T1]\ldots[In \rightarrow Tn]$

$$\frac{A \vdash \texttt{E : T,}\ \text{where } \texttt{J1 ... Jn}\ \text{are not free in the types appearing in A}}{A \vdash \texttt{(pabs (J1 ... Jn) E) : (forall (J1 ... Jn) T)}}\ (\text{pabs})$$

$$\frac{A \vdash \texttt{E : (forall (J1 ... Jn) T)}}{A \vdash \texttt{(papp E T1 ... Tn) : T[T1,…,Tn/J1,…,Jn]}}\ (\text{papp})$$

Appendix B: How to Run the Parts of the Typed HOFL System

Step 1: Configure your `~/.emacs` file to interface properly with ML. (You only need to do this once; not every time you want to run ML.) If you haven't done so already, you will need to add the following lines to your `~/.emacs` file:

```
(setq load-path
 (append '("/usr/share/emacs/20.3/lisp/sml-mode-3.3"
           "/usr/share/emacs/site-lisp/sml-mode-3.3")
         load-path))

(require 'sml-site)

(add-hook 'sml-load-hook '(lambda () (require 'sml-font)))
```

Once you have added the above lines to your `~/.emacs` file, exit Emacs and relaunch it so that your changes will take effect. You will need to relaunch Emacs before going on to Step 2.

Step 2. Start SMLNJ within Emacs by typing

`M-x sml.` <*return*>

Here, `M-x` (pronounced "Meta -x") means pressing the "meta" and "x" keys at the same time. On the Linux workstation keyboards, the "meta" key is the one labelled "Alt".

Step 3. In Emacs, change the default directory for sml by typing

`M-x sml-cd` <*return*> <*dir*>

where <*dir*> is the name of the directory you wish to be the default directory for finding SML files. For Problems 2 and 3, you want <*dir*> to be

```
/usr/users/cs251/download/typed-hofl
```

For Problems 4 and 5, you want <*dir*> to be the name of your local directory containing your personal copy of the `typed-hofl` directory.

Step 4. Compile and load the subsystem you are interested in running. The `typed-hofl` directory contains several "load" files for compiling and loading the various subsystems you might want to run in this assignment. You load one of these load files by using ML's `use` command. Here's how you load and compile the specific subsystems of `typed-hofl`:

- For the type checker: `use("load-typecheck.sml");`
- For the HOFLIPT evaluator: `use("load-hoflipt-eval.sml");`
- For the HOFLEPT evaluator: `use("load-hoflept-eval.sml");`

Executing the above expressions will cause many lines of text to appear on the screen. Although some of the lines seem to indicate some sort of error, you can ignore these. For example:

```
[checking CM/x86-unix/HofleptEval.cm.stable ... not usable]
```

You know that everything has compiled OK if the lines of text ends with the following:

```
val it = () : unit
```

Step 5. Type the HOFLIPT/HOFLEPT program you want to manipulate into a file. You can only have one program per file. A number of HOFLIPT and HOFLEPT test files are included in the `test` subdirectory of `the typed-hofl` directory. By convention, HOFLIPT programs have a `.hip` extension and HOFLEPT programs have a `.hep` extension. If you have previously created files containing the HOFLIPT/HOFLEPT programs you want, then you can skip this step.

Step 6. Run the desired SML program on the HOFLIPT/HOFLEPT program. Here's how you run the SML programs that manipulate the HOFLIPT/HOFLEPT programs:

- To type check the HOFLEPT program in file <*file*>, execute:

  ```
  HofleptTypeChecker.typeCheckFile <file>;
  ```

  This returns the type of the body of the program. For example:

  ```
  - HofleptTypeChecker.typeCheckFile "test/fact.hep";
  val it = IntTy : Type.Type
  ```

- To run the HOFLIPT program in file <*file*> on arguments <*ints*>, execute:

  ```
  HofliptEval.runFile <file> <ints>;
  ```

  This returns the result of evaluating the HOFLIPT program. For example:

  ```
  - HofleptEval.runFile "test/fact.hep" [5];
  val it = IntVal 120 : Value.Val
  ```

- To run the HOFLEPT program in file <*file*> on arguments <*ints*>, execute:

  ```
  HofleptEval.runFile <file> <ints>;
  ```

  For example:

  ```
  - HofleptEval.runFile "test/fact.hep" [5];
  val it = IntVal 120 : Value.Val
  ```

In all of the above cases, you can reduce the amount of typing by "opening" ML structures. E.g., if you execute

```
open HofleptTypeChecker;
```

this makes all the components of the `HofleptTypeChecker` structure available without having to prefix them with "`HofleptTypeChecker.`" For example, fter executing the above line, you can then execute:

```
typeCheckFile "test/fact.hep";
```

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

CS251 Problem Set 5
Due Thursday, April 27, 2000

Name:

Date & Time Submitted *(only if late)*:

Collaborators (*anyone you collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the time you spent on the parts of this problem set. Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	Time	Score
General Reading		
Problem 1 [20]		
Problem 2 [40]		
Problem 3 [40]		
Extra Credit 1 [50]		
Extra Credit 2 [50]		
Total [100]		