CS251 Programming Languages
Prof. Lyn Turbak
Wellesley College

Handout # 30
Sunday, April 8

Problem Set 6

Due: Wednesday, April 18, 2001

This is the final version of Problem Set 6.

Reading:

- Handouts 25 (Intro to ML slides), 27 (Standard ML of New Jersey), 28 (Intro to Types), 29 (Type Rules)
- Paulson's ML for the Working Programmer (*MLWP*), Chapters 2, 3, 4, 5.1-5.11, 9.

Submission:

- Problems 1 and 2 are pencil-and-paper problems that only need to appear in your hardcopy submission.
- For Problems 3 and 4 your softcopy submission should include a copy of your entire `ps6` directory.
- Your hardcopy submission for Problem 3 should be the files `~/cs251/ps6/prob3`*x*`.hem`, where *x* ranges over `a`, `b`, and `c`.
- Your hardcopy submission for Problem 4 should be the files `~/cs251/ps6/hoflad/Eval.sml` and `~/cs251/ps6/hoflad/Parser.sml`.

Problem 1 [20]: ML Types

Figures 1–2 contain twenty higher-order ML functions. For each function, write down the type that would be inferred for it in SML. For example, consider the following SML `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 churchPair x y f = f x y

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 product fs xs =
      map (fn f => map (fn x => (f x)) xs) fs
```

Figure 1: A sampler of higher-order functions in ML, part 2.

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

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

fun oneListOpToTwoListOp f =
      let fun twoListOp binop xs ys = f binop (zip(xs,ys))
       in twoListOp
      end

fun some2 pred = oneListOpToTwoListOp some pred
```

Figure 2: A sampler of higher-order functions in ML, part 2.

Problem 2 [20]: Type Derivations

Consider the following HOFL expression:

```
(((abs (a b)
    (abs (f)
      (f b a)))
  1 true)
 (abs (x y)
   (if x y 0)))
```

a. **[5]** Translate the above expression into an explicitly typed HOFLEMT expression.

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

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; 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]: Explicit Types

For each of the HOFL programs in figure 3 annotate the program with type information so that it becomes a well-typed HOFLEMT program. In cases where a single function is used at more than one type, you will need to make separate copies of the function for each type at which it is used.

You can find the programs in figure 3 in the files `~/cs251/ps6/problem-3`*x*`.hfl` and in the files `~/cs251/ps6/problem-3x.hem`, where *x* ranges over `a`, `b`, and `c`. (The `.hfl` extension is for HOFL programs and `.hem` is for HOFLEMT programs.) You should annotate the `.hem` version of each file so that it is a well-typed HOFLEMT program.

To test your solutions, follow the directions for running SML in Appendix A. When you running SML connected to `~/cs251/ps6`, first evaluate

```
use("loadProb3.sml");
```

(you only need to do this once per session) and then evaluate

```
testProb3();
```

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

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

and then evaluate

`TypeCheck.checkFile` *filename*;

where filename is the name of a file containing the HOFLEMT program you wish to type check. Make sure the filename is relative to the current directory; e.g. `"foo.hfl"` must be in the current directory, `"test/bar.hfl"` must be in the subdirectory `test` of the current directory, and `"../baz.hfl"` must be in the parent directory of the current directory.

a. [5]

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

b. [5]

```
(program (b)
  (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 b (abs (x) (- x 1)) (abs (y) (= y 0)))
       (if (foldr1 (abs (x y) (bor (> x 5) y)) false lst)
           (foldr2 (abs (x y) (+ x y)) 0 lst)
           (foldr2 (abs (x y) (* x y)) 1 lst)))))
```

c. [10]

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

Figure 3: Annotate these HOFL programs to make them well-typed HOFLEMT programs.

Problem 4 [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 programming 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 C's `struct`, CLU's `record` and `struct`, Common Lisp's `defstruct`, and and Pascal's `record`. SML and Haskell support both tuples and records. A Java class instance can even be viewed as a kind of record.

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

(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.

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 figure 4, 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

```
(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 4: A program illustrating tuples in the context of `zip` and `unzip`.

component values. Examples of this include Pascal's variant records, C's `union` (which must be combined with the `struct` facility in order to create tags), 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; the `oneof` and `variant` constructs of CLU are an example. 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 B for a summary of HOFLAD.

a. [10]: Tuples

Modify the `eval` function in `~/cs251/ps6/hoflad/Eval.sml` to handled the the `tuple` and `match-tuple` constructs. See the notes in Appendix B for details concerning the evaluator and environments. To test your solution, first evaluate

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

and then evaluate

```
testProb4a();
```

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/ps6/hoflad/Eval.sml` to evaluate the `tuple` and `match-tuple` constructs. You can test these as in Part (a), except that you should evaluate `testProb4b()` 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} $(tag_1\ (I_{(1,1)} \ldots I_{(1,k_1)})\ E_{\text{body}_1}) \ldots (tag_n\ (I_{(n,1)} \ldots I_{(n,k_n)})\ E_{\text{body}_n})$**)**
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/ps6/hoflad/Parser`. Before attempting this problem, you should (1) study the notes on HOFLAD parsing in Appendix B.3; (2) should study the parsers for `tuple`, `match-tuple`, `variant`, and `tagcase` in `~/cs251/ps6/hoflad/Parser`. ; and (3) should study the desugarings for `bind`, `bindseq`, `scand`, `scor`, and `funrec` in `~/cs251/ps6/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("loadProb4.sml");
```

and then evaluate

```
testProb4c();
```

A Using SML

Here are the steps you need to follow to use SML:

1. In a shell in the `~/cs251` directory, perform a `cvs update -d` to grab all relevant files.

2. Configure your `~/.emacs` file to interface properly with SML by adding the code in Figure 5. You need not type in the code; you can find it in `~/cs251/sml/emacs.txt`. You only need to update your `~/.emacs` file once, not every time you want to run SML.

   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 3.

3. Start SMLNJ within Emacs by typing `M-x sml` ENTER.

4. Go to the SMLNJ interpreter buffer via `C-x b *sml*` ENTER.

5. In Emacs, change the default directory used by `sml` by typing `M-x sml-cd` ENTER *dir*, where *dir* is the name of the directory you wish to be the default directory for finding SML files. For this problem set, you *dir* to be `~/cs251/ps6`.

```
(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)))

(setq auto-mode-alist
      (append auto-mode-alist
              '(
                ("\.itx$"  . scheme-mode) ;; INTEX
                ("\.bdx$"  . scheme-mode) ;; BINDEX
                ("\.ibx$"  . scheme-mode) ;; IBEX
                ("\.ffl$"  . scheme-mode) ;; FOFL
                ("\.fbs$"  . scheme-mode) ;; FOBS
                ("\.hfl$"  . scheme-mode) ;; HOFL
                ("\.hem$"  . scheme-mode) ;; HOFLEMT
                ("\.him$"  . scheme-mode) ;; HOFLIMT
                ("\.hep$"  . scheme-mode) ;; HOFLEPT
                ("\.hip$"  . scheme-mode) ;; HOFLIPT
                ("\.hfd$"  . scheme-mode) ;; HOFLAD
                ("\.hfd$"  . scheme-mode) ;; HOFLAD
                )))

(put 'program 'scheme-indent-hook 1)
(put 'abs 'scheme-indent-hook 1)
(put 'bind 'scheme-indent-hook 2)
(put 'bindpar 'scheme-indent-hook 1)
(put 'bindseq 'scheme-indent-hook 1)
(put 'bindrec 'scheme-indent-hook 1)
(put 'funrec 'scheme-indent-hook 1)
(put 'prepend 'scheme-indent-hook 1)
(put 'tagcase 'scheme-indent-hook 1)
(put 'datacase 'scheme-indent-hook 1)
(put 'match-tuple 'scheme-indent-hook 2)
```

Figure 5: Code to add to your `.emacs` file for SML.

6. Compile and load the subsystem you are interested in running. The `ps6` directory 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 SML's `use` command. Here's how you load and compile the specific subsystems for this assignment:

   - For testing problem 3: `use("loadProb3.sml");`
   - For testing problem 4: `use("loadProb4.sml");`
   - For experimenting with the HOFL evaluator: `use("loadHoflEval.sml");`
   - For experimenting with the HOFLEMT type checker: `use("loadHoflemtTypeCheck.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. Here's an example of something you can safely ignore:

   ```
   [checking ../sml/hoflemt/CM/x86-unix/Pretty.cm.stable ... not usable]
   ```

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

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

7. Run the desired SML function:

   - For testing problem 3: `testProb3();`
   - For testing problem 4: `testProb4`*x*`();` (where *x* ranges over `a`, `b`, and `c`)
   - For experimenting with the HOFL evaluator: `Eval.runFile` *filename args*, where *filename* is a string naming the file in which the HOFL program resides and *args* is an SML list of integer arguments for the program. Make sure the filename is relative to the current directory; e.g. `"foo.hfl"` must be in the current directory, `"test/bar.hfl"` must be in the subdirectory `test` of the current directory, and `"../baz.hfl"` must be in the parent directory of the current directory.
   - For experimenting with the HOFLEMT type checker: `TypeCheck.checkFile` *filename*, where *filename* is a string naming the file in which the HOFLEMT program resides.

   In the last two cases, you can reduce the amount of typing by "opening" SML structures. E.g., if you execute

   ```
   open TypeCheck;
   ```

   this makes all the components of the `TypeCheck` structure available without having to prefix them with `"TypeCheck."`. For example, after executing the above line, you can then execute `checkFile` *filename*.

Handy Tidbits:

- In the SML intepreter, typing `M-p` cycles backwards through previous expressions typed at the interpreter, and typing `M-n` cycles forwards. Use these often to avoid unnecessary typing!
- If you see printed entities in angle brackets, such as `<Sexpr>`, or ellipses (`...`), then your **print depth** may be too low. You can reset it to a number n as follows:

  ```
  Compiler.Control.Print.printDepth := n
  ```

  A good value for n is 1000.

B HOFLAD

B.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. In particular, the `Exp` datatype of HOFL has been extended to be the following:

```
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 *)
```

The first seven constructors have been inherited from HOFL; the last four constructors are new in HOFLAD.

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 figures 6–7.

B.2 Evaluation

The datatype `Val` of HOFL values has been extended in HOFLAD to support tuple and variant values; see figure 8. The signature for HOFLAD evaluation appears in figure 9 and the signature for environments appears in figure 10; these are the same as in HOFL, 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 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 6: 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 7: Signature for HOFLAD tags.

```
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 8: 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 9: Signature for HOFLAD evaluator.

```
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. *)

  end
```

Figure 10: Signature for HOFLAD evaluator.

B.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 `hofl`.

Expression parsing is accomplished via the `toExp` function in the file `~/cs251/ps6/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)
```

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

CS251 Problem Set 6
Due Wednesday, April 18

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 [40]		
Total		