**CS251 Programming Languages** **Handout # ??**
**Prof. Lyn Turbak** **??, 2008**
**Wellesley College**

# Problem Set 5
**Due: ??**

**Overview:**

The individual problem on this assignment tests your understanding of first-class functions, higher-order list operations, and modules. The group problems on this assignment will give you practice with writing an interpreter and understanding the tree-based nature of programs and the scope of names in a program.

**Reading:**

- Handout #22: John Backus's Turing Award Lecture (Section 11)
- Handout #27: INTEX: An Introduction to Program Manipulation
- Handout #30: BINDEX: An Introduction to Naming

**Individual Problem Submission:**

Each student should turn in a hardcopy submission packet for the individual problem by slipping it under Lyn's office door by ??. The packet should include:

1. an individual problem header sheet;
2. your final version of `ListMatrix.ml` from Problem 1a;
3. a transcript showing the result of running `testListMatrix()`;
4. your final version of `FunMatrix.ml` from Problem 1b;
5. a transcript showing the result of running `testFunMatrix()`;

Each student should also submit a softcopy (consisting of your final `ps5-individual` directory) to the drop directory `~cs251/drop/ps5/`*username*.

**Working Together:**

If you want to work with a partner on this assignment, try to find a different partner than you worked with on a previous assignment. If this is not possible, you may choose a partner from before. But try not to choose the same partner you chose last week!

**Group Problem Submission:**

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by ??. The packet should include:

1. a team header sheet indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your pencil-and-paper solution to Group Problem 1.
3. your final version of `MiniFPInterp.ml` from Group Problem 2a;
4. your final version of `MiniFP.ml` from Group Problem 2b;
5. a transcript of running test cases from Group Problem 2b;

Each team should also submit a single softcopy (consisting of your final `ps5-new-group` directory) to the drop directory `~cs251/drop/ps5/`*username*, where *username* is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet).

**Individual Problem [40]: Enter the Matrix!**

*Matrix Definitions*

A matrix is a two dimensional array of elements. A matrix $M$ with $r$ rows and $c$ columns is called an $r \times c$ matrix and is often depicted as follows:

$$\begin{array}{cccc} M_{(1,1)} & M_{(1,2)} & \cdots & M_{(1,c)} \\ M_{(2,1)} & M_{(2,2)} & \cdots & M_{(2,c)} \\ \vdots & \vdots & \ddots & \vdots \\ M_{(r,1)} & M_{(r,2)} & \cdots & M_{(r,c)} \end{array}$$

The notation $M_{(i,j)}$ denotes the element in row $i$ and column $j$ of the matrix. Row and column indices start at 1. **We will assume throughout this problem that matrices are never empty; i.e., they always have at least one row and one column.**

Here is an example of a $3 \times 4$ matrix $A$ with integer elements:

$$\begin{array}{cccc} 61 & 64 & 69 & 76 \\ 71 & 74 & 79 & 86 \\ 81 & 84 & 89 & 96 \end{array}$$

In this example, $A_{(2,3)}$ is 79 and $A_{(3,2)}$ is 84. If $M$ is an $r \times c$ matrix and $i$ is outside the range $[1 \ldots r]$ or $j$ is outside the range $[1 \ldots c]$, then the notation $M_{(i,j)}$ is undefined.

The **transpose** of an $r \times c$ matrix $M$ is a $c \times r$ matrix $M'$ such that $M'_{(i,j)} = M_{(j,i)}$. For example, the transpose of $A$ is the following $4 \times 3$ matrix:

$$\begin{array}{ccc} 61 & 71 & 81 \\ 64 & 74 & 84 \\ 69 & 79 & 89 \\ 76 & 86 & 96 \end{array}$$

OCAML *MATRIX Signature*

Fig. 1 presents an OCAML signature `MATRIX` for an immutable matrix abstract data type (ADT). The contract for the functions in the `MATRIX` signature is presented in Fig. 2. Examples illustrating the behavior of the matrix functions are shown in Fig. 3.

```
module type MATRIX = sig
  type 'a matrix
  val make : int -> int -> (int -> int -> 'a) -> 'a matrix
  val dimensions : 'a matrix -> (int * int)
  val get : int -> int -> 'a matrix -> 'a option
  val put : int -> int -> 'a -> 'a matrix -> 'a matrix
  val transpose : 'a matrix -> 'a matrix
  val toLists : 'a matrix -> 'a list list
  val map : ('a -> 'b) -> 'a matrix -> 'b matrix
end
```

Figure 1: The `MATRIX` signature.

**val** `make: int -> int -> (int -> int -> 'a) -> 'a matrix`
Given positive integers $r$ and $c$ and a binary function $f$, `make` $r$ $c$ $f$ returns an $r \times c$ matrix $m$ such that $m_{(i,j)}$ is the result of the function call $f\ i\ j$. If $r$ or $c$ is $\leq 0$, the meaning of this function is undefined.

**val** `dimensions: 'a matrix -> (int * int)`
`dimensions` $m$ returns a pair (*rows*, *cols*), where *rows* is the number of rows in $m$ and *cols* is the number of cols in $m$.

**val** `get: int -> int -> 'a matrix -> 'a option`
If $m$ is an $r \times c$ matrix, $1 \leq i \leq r$, and $1 \leq j \leq c$ then `get` $i$ $j$ $m$ returns `Some` $v$, where $v$ is the value $m_{(i,j)}$ Otherwise, `get` $i$ $j$ $m$ returns `None`.

**val** `put: int -> int -> 'a -> 'a matrix -> 'a matrix`
If $m$ is an $r \times c$ matrix, $1 \leq i \leq r$, and $1 \leq j \leq c$ , then `put` $i$ $j$ $v$ $m$ returns a new matrix $m'$ such that $m'_{(i,j)} = v$ and $m'_{(i',j')} = m_{(i',j')}$ if $i' \neq i$ or $j' \neq j$; the original matrix $m$ is unchanged. Otherwise, `put` $i$ $j$ $v$ $m$ returns $m$ unchanged.

**val** `transpose: 'a matrix -> 'a matrix`
Given an $r \times c$ matrix $m$, `transpose` $m$ returns a new $c \times r$ matrix $m'$ such that $m'_{(i,j)} = m_{(j,i)}$; the original matrix $m$ is unchanged.

**val** `toLists: 'a matrix -> 'a list list`
Given an $r \times c$ matrix $m$, `toLists` $m$ returns a length-$r$ list of length-$c$ lists. The $k$th element of the returned list is a list of the elements in columns 1 through $c$ of row $k$ of $m$.

**val** `map: ('a -> 'b) -> 'a matrix -> 'b matrix`
Given a function $f$ and a $r \times c$ matrix $m$, `map` $f$ $m$ returns a new $r \times c$ matrix $m'$ such that $m'_{(i,j)} = f(m_{(i,j)})$; the original matrix $m$ is unchanged.

Figure 2: The contract for the seven OCAML matrix-manipulation functions.

*Matrix Implementations*

In this problem, your task is to implement the `MATRIX` signature using two very different concrete representations for matrices.

### a. [20]: Matrices as Lists

A straightforward way to implement an $r \times c$ matrix is as a list of $r$ elements, the $i$th of which is a list of the $c$ elements in the $i$th row of the matrix. In this case, the type `'a matrix` is implemented as `'a list list`. For instance, in this representation, the example matrix $A$ above would be represented as:

```
[[61; 64; 69; 76];
 [71; 74; 79; 86];
 [81; 84; 89; 96]]
```

In this part you are to implement the seven functions in the `MATRIX` signature using the lists-of-lists representation of matrices. But there's a catch: **You are not allowed to use recursion in any of your definitions.** Instead, you should define all functions using the higher order list operations in the `ListUtils` module.

To do this part, go to `~/cs251/ps5-individual/ListMatrix.ml` and flesh out the seven function definitions in the `ListMatrix` module in this file. You should load this module in OCAML via

```
#use "load-list-matrix.ml";;
```

and can test it by evaluating

```
# let m1 = make 3 4 (fun i j -> 10*(i+5) + j*j);;
val m1 : int M.matrix = <abstr>
# dimensions m1;;
- : int * int = (3, 4)
# toLists m1;;
- : int list list = [[61; 64; 69; 76]; [71; 74; 79; 86]; [81; 84; 89; 96]]
# ListUtils.map (fun i -> (ListUtils.map (fun j -> get i j m1)
                                          (ListUtils.range 0 5)))
                (ListUtils.range 0 4);;
- : int option list list =
[[None; None; None; None; None; None];
 [None; Some 61; Some 64; Some 69; Some 76; None];
 [None; Some 71; Some 74; Some 79; Some 86; None];
 [None; Some 81; Some 84; Some 89; Some 96; None];
 [None; None; None; None; None; None]]
# let m2 = transpose m1;;
val m2 : int M.matrix = <abstr>
# dimensions m2;;
- : int * int = (4, 3)
# toLists m2;;
- : int list list = [[61; 71; 81]; [64; 74; 84]; [69; 79; 89]; [76; 86; 96]]
# let m3 = map char_of_int m1;;
val m3 : char M.matrix = <abstr>
# dimensions m3;;
- : int * int = (3, 4)
# toLists m3;;
- : char list list =
[['='; '@'; 'E'; 'L']; ['G'; 'J'; 'O'; 'V']; ['Q'; 'T'; 'Y'; '`']]
# let m4 = put 1 3 13 (put 3 1 31 m1);;
val m4 : int M.matrix = <abstr>
# dimensions m4;;
- : int * int = (3, 4)
# toLists m4;;
- : int list list = [[61; 64; 13; 76]; [71; 74; 79; 86]; [31; 84; 89; 96]]
# let m5 = put (-2) (-3) (-23) (put 0 0 0 (put 1 5 15 (put 5 1 51 m1)));;
val m5 : int M.matrix = <abstr>
# dimensions m5;;
- : int * int = (3, 4)
# toLists m5;;
- : int list list = [[61; 64; 69; 76]; [71; 74; 79; 86]; [81; 84; 89; 96]]
```

Figure 3: Examples illustrating the behavior of the matrix functions for a module `M` implementing the `MATRIX` signature.

```
testListMatrix();;
```

Feel free to define any helper functions you find useful. However, you may *not* use recursion in your helper functions. You may assume that every matrix has at least one row and one column.

**b. [20]: Matrices as Functions**

An alternative representation of a `'a matrix` is as a function with type `int -> int -> 'a option` that takes two arguments, the coordinates $i$ and $j$, and returns an option (`Some` or `None`) of the element at these coordinates.

In this part you are to implement the seven matrix functions using this functional representation of matrices. As in the list representation, **you are not allowed to use recursion in any of**

**your definitions.** Additionally, **you are not allowed to use any list operations in this part *except* in the implementation of `toLists`.**

To do this part, go to `~/cs251/ps5-individual/FunMatrix.ml` and flesh out the seven function definitions in the `FunMatrix` module in this file. You should load this module in OCaml as follows:

```
#use "load-fun-matrix.ml";;
```

and can test it by evaluating

```
testFunMatrix();;
```

*Notes:*

- Feel free to define any helper functions you find useful.
- You may assume that every matrix has at least one row and one column.
- Think carefully about where the bounds checking of matrix indices should occur.
- Try to make your solutions as simple and as efficient as you can. Even if your solution "works", in terms of returning the correct results, some points may be deducted for unnecessarily complex and inefficient code. A solution is considered to be inefficient if it has an asymptotic runnning time that is larger than necessary – e.g., it has a $\Theta(n^2)$ running time when a $\Theta(n)$ running time can be easily achieved. For example, in the functional representation of matrices, many operations can be implemented using `make` in conjunction with `get`, but these implementations are not very efficient.
- A challenging aspect of this problem is determining the number of rows and columns in a matrix represented by a function. *Hint:* Use the fact that the `get` function returns `None` for out-of-bounds indices.

## Group Problems

*Note: you can do the two group problems in any order. Since Problem 1 requires knowledge of* BINDEX *and Problem 2 does not, you may want to delay starting Problem 1 and instead start working on Problem 2 until* BINDEX *is covered in lecture.*

### Group Problem 1 [20]: Abstract Syntax Trees

*Background*

This problem involves reasoning about abstract syntax trees (ASTs) in the BINDEX language described in Handout #30. Fig. 4 shows the AST for the following BINDEX averaging program:

```
(bindex (a b)
  (bind c (+ a b)
    (/ c 2)))
```

Suppose we annotate each node of the abstract syntax tree with a triple (*fv*, *env*, *val*) of the following pieces of information:

1. *fv*: the set of free variables of the program or expression rooted at the node.
2. *env*: The environment in which the node would be evaluated if the program were run on the actual parameters `a` = 3 and `b` = 8. (Write environments as sets of bindings of the form *key*↦*value*.)
3. *val*: The value that would result from evaluating the node in the environment *env*.

Fig. 5 shows the AST for the averaging program annotated with this information. The name *e0* abbreviates the environment $\{\mathrm{a} \mapsto 3, \mathrm{b} \mapsto 8\}$ and *e1* abbreviates the environment $\{\mathrm{a} \mapsto 3, \mathrm{b} \mapsto 8, \mathrm{c} \mapsto 11\}$.

*Your Task*

In this problem, your task is to draw a similar annotated AST for the following BINDEX program:

```
(bindex (a b c)
  (* (bind d (* a c)
       (bind e (- d b)
         (/ (* b d) (+ e a))))
     (bind e (bind b (* 8 a)
               (- b c))
       (+ e b))))
```

You should annotate each node of the abstract syntax tree with the same three pieces of information used in the average example above. In this case, assume that the program is run on the actual parameters `a` = 2, `b` = 3, and `c` = 5.

*Note:* for this problem, you will need to use a large sheet of paper and/or to write very small. It is strongly recommended that you write the solution using pencil (not pen, so you can erase) and paper. Don't waste your time attempting to format it on a computer with a drawing program.

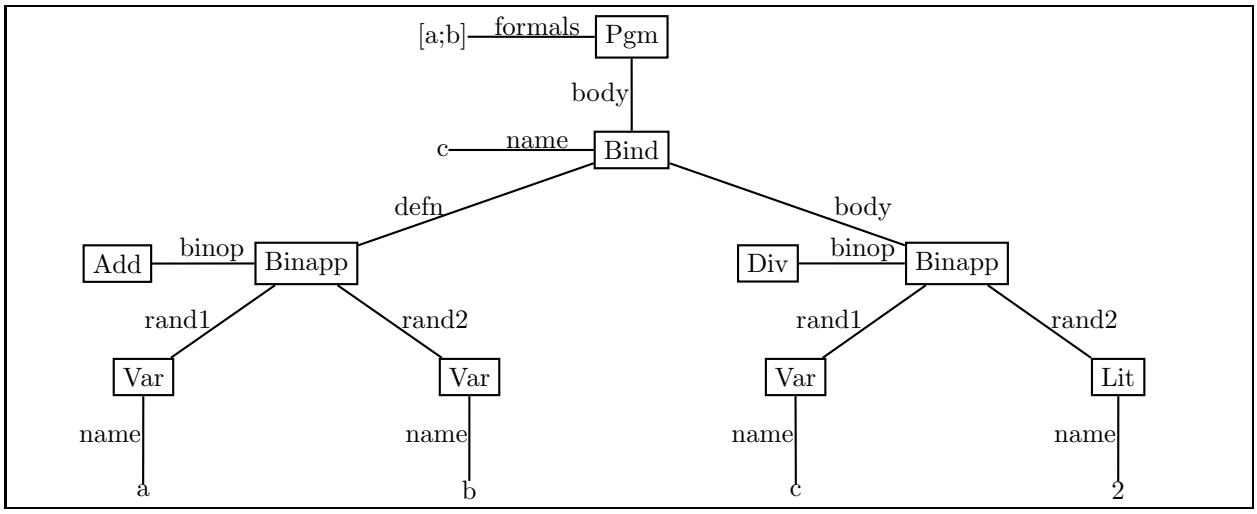


Figure 4: An AST for the `avg` program.

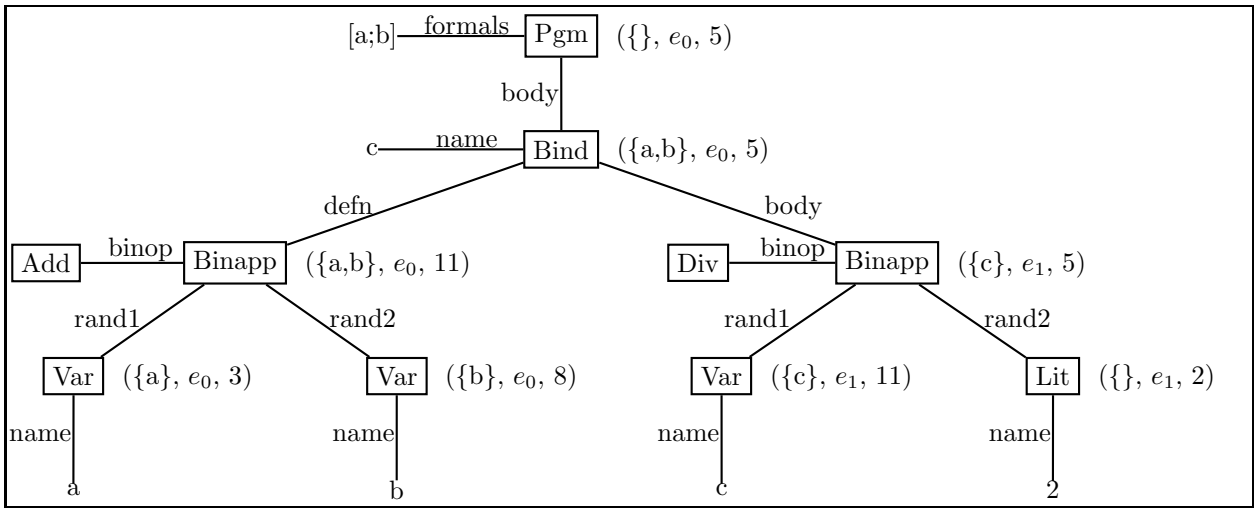


Figure 5: `avg` program AST annotated with free variable, environment, and value information.

**Group Problem 2 [80]: A MiniFP Interpreter**

In Section 11 of his Turing Award lecture paper, Backus describes an applicative lanuage named FP, which you studied for PS4. In this problem, you will implement an interpreter for MiniFP, a subset of FP that implements many of of FP's features. The fact that functional forms in FP (and MiniFP) have implicit parameters (rather than parameters with explicitly names) simplifies the structure of the interpreter. In particular, the environment structures used to handle naming in Bindex are not necessary in MiniFP.

MiniFP *Objects*

The objects manipulated by MiniFP programs are specified by the the `obj` data type in Fig. 6. MiniFP objects are either (1) integers (introduced by the `Int` constructor) or (2) sequences of objects (introduced by the `Seq` constructor).[1]

```
type obj =
    Int of int (* integer objects *)
  | Seq of obj list (* sequence of objects *)
```

Figure 6: Ocaml data type for MiniFP objects.

The following table shows how some objects in FP notation can be expressed as instances of the `obj` datatype.

| FP **Notation** | Ocaml `obj` **Notation** |
|---|---|
| 17 | `Int 17` |
| $\langle 8, -3, 6\rangle$ | `Seq[Int 8; Int (-3); Int 6]` |
| $\langle 17, \langle 8, -3, 6\rangle\rangle$ | `Seq[Int 17; Seq[Int 8; Int (-3); Int 6]]` |
| $\langle\langle 8, -3, 6\rangle, \langle 5, 1, 7\rangle\rangle$ | `Seq[Seq[Int 8; Int (-3); Int 6];Seq[Int 5; Int 1; Int 7]]` |

Here are some definitions of MiniFP objects in the file `MiniFP.ml` that we will use in later examples:

```
let vector1 = Seq[Int 2; Int 3; Int 5]

let vector2 = Seq[Int 10; Int 20; Int 30]

let vectors = Seq[vector1; vector2] (* A pair of vectors or a 2x3 matrix *)

let matrix = Seq[Seq[Int 1; Int 4]; Seq[Int 8; Int 6]; Seq[Int 7; Int 9]] (* A 3x2 matrix *)

let matrices1 = Seq[vectors; matrix] (* A pair of a 2x3 matrix and a 3x2 matrix *)

let matrices2 = Seq[matrix; vectors] (* A pair of a 3x2 matrix and a 2x3 matrix *)
```

MiniFP *Functional Forms*

MiniFP programs are **functional forms**, which include primitive functions like `id`, `+`, and `distr` as well as combining forms like composition ($\circ$), mapping ($\alpha$), and reduction (/). Fig. 7 presents the Ocaml data type `funForm` specifying MiniFP functional forms, and Fig. 8 shows the correspondence between FP notation for functional forms and Ocaml data type notation.

[1] Although the `Int` and `Seq` constructors have the same names as constructors in the `Sexp` module, the MiniFP constructors are different. They are defined in the `MiniFP` module in the file `MiniFP.ml` in the `ps5-new-group` directory. Explicit qualification can be used to disambiguate the constructors from these modules — e.g., `Sexp.Int` vs. `MiniFP.Int`.

```
type funForm =
    Id    (* identity *)
  | Add   (* addition *)
  | Sub   (* subtraction *)
  | Mul   (* multiplication *)
  | Div   (* division *)
  | Distl (* ditribute from left *)
  | Distr (* ditribute from right *)
  | Trans (* transpose *)
  | Sel of int    (* selection *)
  | Const of obj  (* constant function *)
  | Map of funForm  (* alpha = map *)
  | Reduce of funForm (* / = reduce *)
  | BinaryToUnary of funForm * obj (* bu *)
  | Compose of funForm * funForm  (* o = composition *)
  | FunSeq of funForm list (* [...] = function sequence *)
```

Figure 7: OCAML data type for MINIFP functional forms.

| FP **Notation** | OCAML `funForm` **Notation** |
|---|---|
| id | `Id` |
| $+$ | `Add` |
| $-$ | `Sub` |
| $\times$ | `Mul` |
| $\div$ | `Div` |
| distl | `Distl` |
| distr | `Distr` |
| trans | `Trans` |
| $i$ | `Sel` $i$ |
| $\overline{x}$ | `Const` $x$ |
| $\alpha f$ | `Map` $f$ |
| $/f$ | `Reduce` $f$ |
| bu $f$ $x$ | `BinaryToUnary(`$f$`,`$x$`)` |
| $f_1 \circ f_2$ | `Compose(`$f_1$`,` $f_2$`)` |
| $[f_1, \ldots, f_n]$ | `FunSeq[`$f_1$`; ...;` $f_n$`]` |

Figure 8: Correspondence between FP notation and OCAML notation for MINIFP functional forms. $i$ denotes an integer, $x$ denotes an object, and $f$ denotes a functional form.

For example, consider the innner product and matrix multiplication examples from Sections 11.3.2 and 11.3.3 (p. 622) of Backus's paper:

$$IP \equiv (/+) \circ (\alpha\times) \circ \text{trans}$$
$$MM \equiv (\alpha\alpha\text{IP}) \circ (\alpha\text{distl}) \circ \text{distr} \circ [1, \text{trans} \circ 2]$$

These can be expressed in OCAML via the following definitions (from `MiniFP.ml`):

```
let ip = Compose(Reduce Add, Compose(Map Mul, Trans))

let mm = Compose(Map(Map ip),
                 Compose(Map Distl,
                         Compose(Distr,
                                 FunSeq[Sel 1; Compose(Trans, Sel 2)])))
```

### a. [5]: A MiniFP Program

In Group Problem 1 of PS4, you manipulated the following functional form $F$:

$$F \equiv \alpha/+ \circ \alpha\alpha\times \circ \alpha\text{distl} \circ \text{distr} \circ [\text{id}, \text{id}]$$

In the file `MiniFP.ml`, express this functional form in OCaml via a definition of the form

```
let f = ...
```

### b. [55]: A MiniFP Interpreter

A MiniFP interpreter specifies the object that results from applying a program (i.e., functional form) to an object. So a MiniFP interpreter is simply an `apply` function with the following signature:

```
apply: MiniFP.funForm -> MiniFP.obj -> MiniFP.obj
```

The semantics (meaning) of all MiniFP functional forms is given in Sections 11.2.3 and 11.2.4 of Backus's paper. The only difference between MiniFP and FP semantics is that in MiniFP, the `Reduce` functional form is defined only on nonempty sequences. So applying `Reduce Add` to `Seq[]` is an error in MiniFP but not in FP (where it denotes 0). Fig. 9 shows numerous examples of the `apply` function.

In this problem, your goal is to implement a complete MiniFP interpreter by fleshing out the following skeleton of the `apply` function in the file `MiniFPInterp.ml`:

```
let rec apply funform obj =
  match (funform, obj) with
    (Add, Seq[Int x; Int y]) -> Int(x+y)
    (* flesh out the remaining clauses below *)
  | _ -> raise (EvalError ("Ill-formed application: apply "
                           ^ (funFormToString funform)
                           ^ " "
                           ^ (objToString obj)))
```

The clause for `Add` has been written for you. Your task is to flesh out the clauses for all the other MiniFP functional forms.

*Notes:*

- As usual, start this problem set by performing a `cvs update -d`, and perform an update every time you log in to work on this problem set.
- After launching OCaml, connect to the `ps5-new-group` directory via `#cd "/students/`*your-account-name*`/cs251/ps5-new-group"`.
- To load all the appropriate files into OCaml, execute `#use "load-minifp.ml"`. You will need to execute this every time you want to test changes to your code. This loads many files, including numerous utility files, `MiniFP.ml`, `MiniFPInterp.ml`, and `MiniFPTest.ml`. Carefully look at the output of the `#use` command each time you invoke it. Even if it finds and reports an error in `MiniFP.ml`, say, it will continue to load the other files, and it's easy to miss the error, especially if the error message has scrolled off the screen.
- Use `apply` to test your interpreter on examples similar to those in Fig. 9. For instance:

```
# apply Id vector1;;
- : MiniFP.obj = Seq [Int 2; Int 3; Int 5]

# apply (Const (Int 17)) vector1;;
- : MiniFP.obj = Int 17

# apply (Sel 1) vector1;;
- : MiniFP.obj = Int 2

# apply (Sel 2) vector1;;
- : MiniFP.obj = Int 3

# apply (Sel 3) vector1;;
- : MiniFP.obj = Int 5

# apply (Sel 4) vector1;;
Exception: MiniFPInterp.EvalError "Selection index 4 out of bounds [1..3]"

# apply Distl (Seq[Int 7;vector1]);;
- : MiniFP.obj = Seq [Seq [Int 7; Int 2]; Seq [Int 7; Int 3]; Seq [Int 7; Int 5]]

# apply Distr (Seq[vector1;Int 7]);;
- : MiniFP.obj = Seq [Seq [Int 2; Int 7]; Seq [Int 3; Int 7]; Seq [Int 5; Int 7]]

# apply (Map (Const (Int 17))) vector1;;
- : MiniFP.obj = Seq [Int 17; Int 17; Int 17]

# apply (Map (BinaryToUnary (Mul, Int 2))) vector1;;
- : MiniFP.obj = Seq [Int 4; Int 6; Int 10]

# apply (Reduce Add) vector1;;
- : MiniFP.obj = Int 10

# apply (Reduce Mul) vector1;;
- : MiniFP.obj = Int 30

# apply (Reduce Add) (Seq[]);;
Exception: MiniFPInterp.EvalError "Reduction of empty sequence".

# apply (Compose(Map (BinaryToUnary (Add, Int 1)), Map (BinaryToUnary (Mul, Int 2)))) vector1;;
- : MiniFP.obj = Seq [Int 5; Int 7; Int 11]

# apply (FunSeq[Const (Int 17); Id; Map (BinaryToUnary (Add, Int 1))]) vector1;;
- : MiniFP.obj = Seq [Int 17; Seq [Int 2; Int 3; Int 5]; Seq [Int 3; Int 4; Int 6]]

# apply Trans matrix;;
- : MiniFP.obj = Seq [Seq [Int 1; Int 8; Int 7]; Seq [Int 4; Int 6; Int 9]]

# apply (Compose(Trans,Trans)) matrix;;
- : MiniFP.obj = Seq [Seq [Int 1; Int 4]; Seq [Int 8; Int 6]; Seq [Int 7; Int 9]]

# apply ip vectors;;
- : MiniFP.obj = Int 230

# apply ip matrix;;
Exception: MiniFPInterp.EvalError "Ill-formed application: apply x (1 8 7)".

# apply mm matrices1;;
- : MiniFP.obj = Seq [Seq [Int 61; Int 71]; Seq [Int 380; Int 430]]

# apply mm matrices2;;
- : MiniFP.obj = Seq[Seq [Int 42; Int 83; Int 125];
                     Seq [Int 76; Int 144; Int 220];
                     Seq [Int 104; Int 201; Int 305]]

# apply f vector1;;
- : MiniFP.obj = Seq [Int 20; Int 30; Int 50]
```

Figure 9: MiniFP interpreter examples.

| FP **Notation** | OCAML `obj` **Notation** | **S-expression Notation** |
|---|---|---|
| $i$ | `Int` $i$ | $i$ |
| $\langle x_1, \ldots, x_n \rangle$ | `Seq[`$x_1$`; ...; `$x_n$`]` | `(`$x_1$ ... $x_n$`)` |

Figure 10: S-expression notation for MINIFP objects, $i$ denotes an integer and $x$ denotes an object.

```
# apply Add (Seq[Int 2; Int 3]);;
- : MiniFP.obj = Int 5

# apply Mul (Seq[Int 2; Int 3]);;
- : MiniFP.obj = Int 6
```

Note that if you have not yet implemented a clause for a functional form, testing it via `apply` will yield the following error:

```
Exception: MiniFPInterp.EvalError "Ill-formed application: apply <functional form> <object>".
```

Here, `<functional form>` and `<object>` are placeholders for the particular functional form and object in the ill-formed application. You will flesh out these placeholders in part c of this problem.

You needn't turn in a transcript of your testing examples, because the tests in this part will be superseded by the more extensive testing of your interpreter in the part c of this problem.

- You *should* use any standard library functions from the standard modules (e.g., `List`, `ListUtils`, `FunUtils`) that you find helpful.
- Pattern matching is your friend here. Use it to simplify your definitions.
- In addition to defining `apply` you may want to define some auxiliary functions.
- To simplify the handling of the `Trans` form, you have been provided with a `transposeSeqs` function that transposes the elements in a list of sequences. For example:

```
# transposeSeqs [Seq[Int 1;Int 2]; Seq[Int 3;Int 4]; Seq[Int 5;Int 6]];;
- : MiniFP.obj list = [Seq [Int 1; Int 3; Int 5]; Seq [Int 2; Int 4; Int 6]]
```

- To indicate an evaluation error in your interpreter, raise an `EvalError` exception as follows:

```
raise (EvalError string-stating-error-message)
```

- It's a good idea to implement, test, and debug a few commands at a time rather than attempting to handle all the commands at once.

### c. [20]: An S-Expression Syntax for MINIFP

In the final part of this problem, your task is to implement an s-expression syntax for MINIFP objects and functional forms. The s-expression notation for objects is shown in Fig. 10 and the s-expression notation for functional forms is shown in Fig. 11.

For example, our `matrices1` example, which in FP would be written

$$\langle\langle\langle 2,3,5\rangle, \langle 10,20,30\rangle\rangle, \langle\langle 1,4\rangle, \langle 8,6\rangle, \langle 7,9\rangle\rangle\rangle$$

is written in MINIFP using the following s-expression:

```
(((2 3 5) (10 20 30)) ((1 4) (8 6) (7 9)))
```

And the functional forms

$$IP \equiv (/+) \circ (\alpha\times) \circ \text{trans}$$
$$MM \equiv (\alpha\alpha\text{IP}) \circ (\alpha\text{distl}) \circ \text{distr} \circ [1, \text{trans} \circ 2]$$

| FP **Notation** | OCAML `funForm` **Notation** | **S-expression Notation** |
|---|---|---|
| id | `Id` | `id` |
| $+$ | `Add` | `+` |
| $-$ | `Sub` | `-` |
| $\times$ | `Mul` | `x` |
| $\div$ | `Div` | `-:` |
| distl | `Distl` | `distl` |
| distr | `Distr` | `distr` |
| trans | `Trans` | `trans` |
| $i$ | `Sel` $i$ | $i$ |
| $\overline{x}$ | `Const` $x$ | `($` $x$`)` |
| $\alpha f$ | `Map` $f$ | `(a` $f$`)` |
| $/f$ | `Reduce` $f$ | `(/` $f$`)` |
| bu $f$ $x$ | `BinaryToUnary(`$f$`,`$x$`)` | `(bu` $f$ $x$`)` |
| $f_1 \circ f_2$ | `Compose(`$f_1$`,` $f_2$`)` | `(o` $f_1$ $f_2$`)` |
| $[f_1, \ldots, f_n]$ | `FunSeq[`$f_1$`; ...;` $f_n$`]` | `(`$f_1$ `...` $f_n$`)` |

Figure 11: S-expression notation for MINIFP functional forms. $i$ denotes an integer, $x$ denotes an object, and $f$ denotes a functional form.

can be expressed in s-expressions as follows:

| | |
|---|---|
| IP | `(o (/ +) (o (a x) trans))` |
| MM | `(o (a (a (o (/ +) (o (a x) trans)))) (o (a distl) (o distr (1 (o trans 2)))))` |

Because it is common to create a function by composing a sequence of functions, the MINIFP s-expression syntax allows a composition to take any number of functional forms. So

`(o` $f_1$ $f_2$ ... $f_{n-1}$ $f_n$`)`

is an abbreviation for

`(o` $f_1$ `(o` $f_2$ ... `(o` $f_{n-1}$ $f_n$`)` ...`))`

For example, here are alternative s-expression notations for IP and MM:

| | |
|---|---|
| IP | `(o (/ +) (a x) trans)` |
| MM | `(o (a (a (o (/ +) (a x) trans))) (a distl) distr (1 (o trans 2)))` |

Your task in this part is to flesh out the following functions in the file `MiniFP.ml` to implement the s-expression syntax described above:

```
objToSexp : MiniFP.obj -> Sexp.sexp
funFormToSexp : MiniFP.funForm -> Sexp.sexp
sexpToObj : Sexp.sexp -> MiniFP.obj
sexpToFunForm : Sexp.sexp -> MiniFP.funForm
```

The first two functions unparse objects and functional forms into s-expressions, respectively. The second two functions parse s-expressions into objects and functional forms, respectively. For example:

```
# objToSexp (Seq[Seq[Int 1; Int 2]; Seq[Int 3; Int 4; Int 5]; Int 6]);;
- : Sexp.sexp =
Sexp.Seq
 [Sexp.Seq [Sexp.Int 1; Sexp.Int 2];
  Sexp.Seq [Sexp.Int 3; Sexp.Int 4; Sexp.Int 5]; Sexp.Int 6]
```

```
# funFormToSexp (Compose(Map(BinaryToUnary(Add,Int 1)), FunSeq[Id; (Const (Int 17))]));;
- : Sexp.sexp =
Sexp.Seq
 [Sexp.Sym "o";
  Sexp.Seq [Sexp.Sym "a"; Sexp.Seq [Sexp.Sym "bu"; Sexp.Sym "+"; Sexp.Int 1]];
  Sexp.Seq [Sexp.Sym "id"; Sexp.Seq [Sexp.Sym "$"; Sexp.Int 17]]]

# sexpToObj (Sexp.Seq [Sexp.Seq [Sexp.Int 1; Sexp.Int 2];
                       Sexp.Seq [Sexp.Int 3; Sexp.Int 4; Sexp.Int 5]; Sexp.Int 6]);;
- : MiniFP.obj = Seq [Seq [Int 1; Int 2]; Seq [Int 3; Int 4; Int 5]; Int 6]

# sexpToFunForm (Sexp.Seq
 [Sexp.Sym "o";
  Sexp.Seq [Sexp.Sym "a"; Sexp.Seq [Sexp.Sym "bu"; Sexp.Sym "+"; Sexp.Int 1]];
  Sexp.Seq [Sexp.Sym "id"; Sexp.Seq [Sexp.Sym "$"; Sexp.Int 17]]]);;
- : MiniFP.funForm =
  Compose (Map (BinaryToUnary (Add, Int 1)), FunSeq [Id; Const (Int 17)])
```

The file `MiniFP.ml` also alread contains the following additional function definitions for converting between strings and data types for MiniFP objects and functional forms:

```
let objToString obj = Sexp.sexpToString (objToSexp obj)
let funFormToString ff = Sexp.sexpToString (funFormToSexp ff)
let stringToObj s = sexpToObj (Sexp.stringToSexp s)
let stringToFunForm s = sexpToFunForm (Sexp.stringToSexp s)
```

For example:

```
# objToString (Seq[Seq[Int 1; Int 2]; Seq[Int 3; Int 4; Int 5]; Int 6]);;
- : string = "((1 2) (3 4 5) 6)"

# funFormToString (Compose(Map(BinaryToUnary(Add,Int 1)), FunSeq[Id; (Const (Int 17))]));;
- : string = "(o (a (bu + 1)) (id ($ 17)))"

# stringToObj "((1 2) (3 4 5) 6)";;
- : MiniFP.obj = Seq [Seq [Int 1; Int 2]; Seq [Int 3; Int 4; Int 5]; Int 6]

# stringToFunForm "(o (a (bu + 1)) (id ($ 17)))";;
- : MiniFP.funForm =
Compose (Map (BinaryToUnary (Add, Int 1)), FunSeq [Id; Const (Int 17)])

# stringToFunForm "(o id distr trans (a (a (bu + 1))))";;
- : MiniFP.funForm =
Compose (Id,
 Compose (Distr, Compose (Trans, Map (Map (BinaryToUnary (Add, Int 1))))))
```

*Notes:*

- Because the `obj` data type defines the `Int` and `Seq` constructors, you will have to use the explicitly qualified `Sexp.Int` and `Sexp.Seq` constructors for the `sexp` data type.
- You will need to translate a composition of two or more functions into nested composition pairs.
- You can test both your s-expression parsing/unparsing and your interpreter by evaluating the expresion `test()`. This function, which is defined in the file `MiniFPTest.ml`, tests the interpreter on a suite of test cases, each of which is a triple of strings having the form

  (*function-form-string*, *object-string*, *result-string*)

The tester first parses *function-form-string* and *object-string* into a function form $f$ and object $x$, respectively. It then applies $f$ to $x$, and converts the resulting object into a string. If this string matches *result-string*, the test passes by displaying `OK!`. Otherwise, the test fails by displaying `***ERROR***` and showing the difference between the expected result and the actual result. If the application of $f$ to $x$ raises an exception, the tester converts the exception message to an error string; this makes it possible for the tester to verify that the interpreter handles error situations correctly.

- Your implementaiton should be able to pass all the test cases in `MiniFPTest.ml`. You are encouraged to add additional test cases.

*Individual Problem Header Page*
*Please make this the first page of your hardcopy submission of individual problems.*

# CS251 Problem Set 5 Individual Problems
## Due ??

Name:

Date & Time Submitted:

*By signing below, I attest that I have followed the policy for individual problems set forth in the Course Information handout. In particular, I have not consulted with any person except Lyn about these problems and I have not consulted any materials from previous semesters of CS251.*

Signature:

*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 design future problem sets. I will fill out the* **Score** *column when grading you problem set.*

| **Part** | **Time** | **Score** |
|---|---|---|
| General Reading | | |
| Problem 1a [20] | | |
| Problem 1b [20] | | |
| **Total** | | |

*Group Problem Header Page*
*Please make this the first page of your hardcopy submission for group problems.*

# CS251 Problem Set 5 Group Problems
## Due ??

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with*):

*By signing below, I/we attest that I/we have followed the collaboration policy as specified in the Course Information handout.*
Signature(s):

*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 2a [5] | | |
| Problem 2b [55] | | |
| Problem 2c [20] | | |
| **Total** | | |