Data Structures and Functional Programming	**Problem Set 3**
CS 3110, Fall 2013	**Due at 11:59 PM, Thursday, October 3**
Version: 1	**Last Modified: September 22, 2013**

This assignment has three parts. You should write your solution to each part in various separate files: For part 1, `part1-proof.pdf` and `induction.ml`. For part 2, `ralist.ml`. For part 3, `eval.ml` and `repl.ml`. The entirety of the problem set should then be zipped and submitted as `ps3.zip` as instructed at the end of the problem set. To help get you started, we have provided a stub file for each part on CMS. You should download and edit these files. Once you have finished, submit your solution using CMS, linked from the course website.

Instructions

Compile Errors

All code you submit must compile. **Programs that do not compile will be heavily penalized.** If your submission does not compile, we will notify you immediately. You will have 48 hours after the submission date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming

We will be using an automatic grading script, so it is **crucial** that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions are **treated as compile errors** and you will have to submit a patch.

Code Style

Finally, please pay attention to style. Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct may still lose points. Take the extra time to think out the problems and find the most elegant solutions before coding them up. Good programming style is important for all assignments throughout the semester.

Late Assignments

Please carefully review the course website's policy on late assignments, as **all** assignments handed in after the deadline will be considered late. Verify on CMS that you have submitted the correct version, **before** the deadline. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

Part One: Induction and Recursion

In this part, you will write a function and then prove its correctness using induction.

The Towers of Hanoi

The Towers of Hanoi puzzle is a perennial example of inductive logic. The set up of the puzzle is as follows: There are 3 rods arranged in a row. n disks are stacked on the first rod, with the largest disk on the bottom and the smallest on top. The goal is to move all n disks from the first rod to the third rod without breaking the following rules:

1. Only one disk may be moved at a time.
2. Only the top-most disk on a stack may be moved.
3. No disk can ever be moved on top of a smaller disk.

We provide you with a custom data type `peg` with three values, representing the three rods, and a type `move`, representing a move from one peg to another:

```
type peg = A | B | C
type move = peg * peg
```

The move `(x,y)` indicates that the top disk should be taken from stack x and placed on top of stack y. Note that not all moves are valid: depending on the state of the stacks, a move might violate the rules of the game.

Exercise 1: Implementation

Write a function

```
towers : int -> move list
```

that takes in an integer representing the number of disks beginning on peg A, and returns a list of `moves`. Your moves, when executed from head to tail, should abide by the rules of the game, and should end with all the disks stacked on peg C.

We have provided you with a graphical representation of your solutions, which you can view by opening a command line window and navigating to the directory containing `induction.ml` (where you should implement your OCaml solutions to Part 1) and `display_solution.sh`. Type into the command-line "chmod 777 display_solution.sh" to make the display script executable, and then type "./display_solution.sh". Our script will run your OCaml function and use your moves to solve the puzzle. If your function works, all the disks should end up in order on the third peg.

Exercise 2: Proof of correctness

Write an inductive proof of the correctness of your towers function. You must prove that your function correctly solves a game with n disks for $n > 0$.

We have provided an example proof `listproof.pdf` with the problem set release. Your proof should follow the format of that example proof.

Your proof must be submitted in `.pdf` format, in the file `part1-proof.pdf`. You are free to create your file using whatever software you prefer, but we recommend using LaTeX. We have provided the LaTeXsource for `listproof.pdf` that you may use as a template if you wish.

Part 2: Random Access Lists

One of the deficiencies in the purely functional representation for the List datatype is that we do not have the random access to elements that we enjoy when using arrays. In this exercise we will build a functional data structure that provides fast random access to a list-like data structure.

The motivating idea behind this data structure is that a naïve implementation of lists closely mirrors a unary implementation of the natural numbers:

```
type     nat = Zero | Succ of      nat
type 'a list = Nil  | Cons of 'a * 'a list
```

Taking the tail of a list is analogous to subtracting one from a number:

```
let pred = function
  | Zero        -> failwith "Error: pred"
  | Succ n      -> n

let tail = function
  | Nil         -> failwith "Error: tl"
  | Cons (x,xs) -> xs
```

while appending two lists is analogous to adding two numbers:

```
let rec add = function
  | Zero,m    -> m
  | Succ n, m -> Succ (add (n,m))

let rec append = function
  | [],ys          -> ys
  | Cons(x,xs) ,ys -> Cons(x,(append (xs,ys)))
```

This analogy suggests that we can think of a list as a natural number that carries some extra information. Container abstractions designed with this analogy in mind are called *numerical representations*.

We will be designing a numerical representation for lists that is based on a binary representation of numbers. In the binary representation of a natural number, a one in the kth position stands for 2^k. You can think of this as representing 2^k applications of the `Succ` constructor. Similarly, in the binary representation of a random access list, a "one" in the kth position contains 2^k elements; in a sense it represents 2^k applications of the `Cons` constructor.

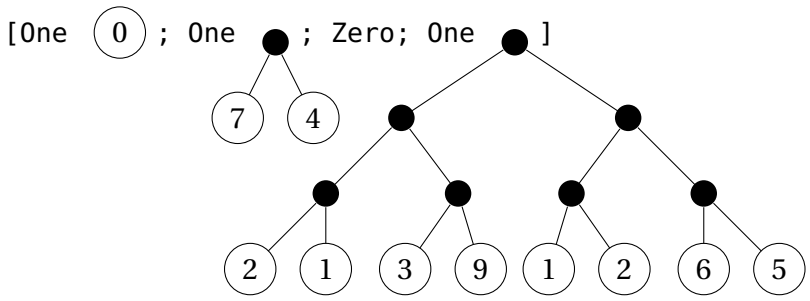
Of course, the "ones" in the random access list must actually store the 2^k entries in the list. Since there are exactly 2^k entries corresponding to the kth digit, we can store the entries in a completely balanced binary tree. We will use the following representation:

```
type 'a tree   = Leaf of 'a   | Node of int * 'a tree * 'a tree
type 'a tbit   = Zero         | One of 'a tree
type 'a ralist = 'a tbit list
```

The `int` stored in each `Node` stores the number of leaves of the subtree rooted at that node.

For example, suppose that we wish to store the list `[0; 7; 4; 2; 1; 3; 9; 1; 2; 6; 5]`. This list contains 10 elements and the binary representation of `10` is `0b1011`, so the `ralist` representation should take the form `[One _; One _; Zero; One _]`[1]. The digit in the zeroth place corresponds to the first 2^0 elements (`[0]`). The bit in the first place corresponds to the 2^1 elements `[7; 4]`. The bit in the second place is `Zero`, so it contributes no elements to the total; the digit in the third place corresponds to the 2^3 elements `[2; 1; 3; 9; 1; 2; 6; 5]`.

We can draw this representation as follows:



Exercise 3: Ralist traversal and modification

Your first task will be to implement various traversal and modification functions for working with the `tree` and `ralist` data structures. Implement the following functions in the file `ralist.ml`.

```
(* Returns the size of a given tree *)
size_tree : 'a tree -> int

(* Combines two trees by inserting them as the left and right
 * subtrees of a common root
 *)
merge : 'a tree -> 'a tree -> 'a tree

(* look_tree k t returns the kth leaf of t.
 *
 * fails if t has fewer than k nodes
 *)
look_tree : int -> 'a tree -> 'a
```

[1] note that in our representation, we put the least significant bits at the head, so the list looks backwards when compared with the number.

```
(* update_tree k x t returns a new tree t' that has the value of the kth leaf
 * node replaced by x.
 *
 * fails if t has fewer than k nodes
 *)
update_tree : int -> 'a -> 'a tree -> 'a tree

(* increment v l should produce a new ralist l' that contains v followed by
 * the elements of l.
 *)
increment : 'a -> 'a ralist -> 'a ralist

(* decrement l should produce the leftmost value of l paired with a new ralist
 * that contains all of the other values of l.
 *
 * fails if l is empty.
 *)
decrement : 'a ralist -> 'a * 'a ralist
```

These functions should establish and maintain the representation invariants outlined above.

Exercise 4: Ralist implementation

With these helper functions in hand, implement the following list functions in `ralist.ml`:

```
empty     : 'a ralist
is_empty  : 'a ralist -> bool
cons      : 'a -> 'a ralist -> 'a ralist
head      : 'a ralist -> 'a
tail      : 'a ralist -> 'a ralist

to_list   : 'a ralist -> 'a list
from_list : 'a list   -> 'a ralist

lookup    : int -> 'a ralist -> 'a
update    : int -> 'a -> 'a ralist -> 'a ralist
```

Most of these functions are analogous to `list` constructors or `List` module functions:

- `empty` is analogous to `[]`
- `is_empty` is analogous to `((=) [])`
- `cons` is analogous to `::`
- `head` is analogous to `List.hd`
- `tail` is analogous to `List.tl`
- `lookup` is analogous to `List.nth`

The remaining functions are straightforward:

- `to_list` should convert an `ralist` to a standard `list`.
- `from_list` should convert a standard `list` to an `ralist`.
- `update n x l` should produce a new `ralist` that is the same as `l` except that the `n`th entry should be `x`. `update` should fail if `n` is larger than the length of `l`.

To Submit

Submit the file `ralist.ml` with each of the functions implemented as described above

Part Three: A Scheme Interpreter

For this problem, you will implement an interpreter for a simplified version of the Scheme language. Scheme is a functional language whose semantics is very much like Ocaml. Evaluation follows the environment model presented in class very closely.

We have provided a lexical analyzer and parser that can take a string input, parse it as a Scheme program, and output an Abstract Syntax Tree (AST) representing the code. The input program is converted to a tree of type `Ast.expr`, defined in the `Ast` module. Unlike OCaml, there is not a lot of static typechecking done, so you will find the syntax of Scheme quite a bit less restrictive than OCaml (not necessarily a good thing). Scheme uses dynamic scoping as well as some runtime type-checking for certain expressions. Your job will be to write functions for evaluating the AST that is returned from the parser. You will also have to check for various runtime type errors (such as applying an if-then-else to a non-Booolean), and to raise a runtime type error if the types of the expressions do not make sense in a given context.

A few Scheme programs are provided in the file `etc/input.txt`. Here is a sample run with our solution code.

```
zardoz > (load "etc/input.txt")
>> <fun>
>> <fun>
>> <fun>
>> <fun>
>> <fun>
zardoz > (fact 4)

>> 24
zardoz > (length (list 0 0 0))

>> 3
zardoz > (rev (list 1 2 3 4))

>> (4 3 2 1)
zardoz > (cons 1 2)

>> (1 . 2)
zardoz > (fib 10)
```

```
>> (55 34 21 13 8 5 3 2 1 1 0)
zardoz > ((lambda (x y) (- x y)) 3110 2110)

>> 1000
zardoz >
```

The Scheme Language

Values

The language has six types of values:

1. **Integers**, which correspond to the OCaml type `int`
2. **Strings**, which correspond to the OCaml type `string`
3. **Bools**, which correspond to the OCaml type `bool`
4. **Closures**, which represent functions and their environments.
5. `cons` pairs
6. `nil`.

There is also a seventh type `Undef` that is used internally for creating recursive definitions and is not available to the Scheme programmer. The following code, taken from `heap.ml`, declares the type **`value`**. When an expression is evaluated successfully, the resulting value is an entity of this type.

```
type value = Int of int | Str of string | Bool of bool
           | Closure of expr * env
           | Cons of value * value | Nil
           | Undef
```

Expressions

The lexical analyzer and parser convert an input program formatted in plain text to an AST, an entity of type `Ast.expr`, which can then be evaluated. The parser recognizes the following patterns:

- **Integers** (`Ast.Int_e of int`) consist of sequences of one or more numeric characters. In our version of Scheme, this excludes negative integers, so that 42 and 007 are considered valid integers, whereas -1 is not
- **Strings** (`Ast.Str_e of string`) consist of sequences of zero or more characters enclosed in double quotes, e.g. "Hello". Certain strings can be escaped by the backslash character:

Character	Escaped
\	\\
”	\”
<newline>	\n
<tab>	\t

Type	Example values
int	0,1,2,-1,-2
bool	true, false
string	"Hello world!","xyzzy"

- **Bools** (`Ast.Bool_e of bool`) are specified by the strings `#t` for true or `#f` for false
- **Identifiers** (ids) (`Ast.Id_e of Ast.id`) consist of a single lower- or upper-case letter followed by zero or more lower- or upper-case letters, numeric digits, underscores, and single-quote characters.

In the following structures, the parentheses must be present:

- `(lambda x e)` represents a function that takes a single value `v` as its input and evaluates `e` with `x` bound to `v`. This is represented as the type

```
Ast.Fun_e of Ast.id list * Ast.expr
```

- `(lambda (x1 x2 x3 ... xn) e)` represents a function that takes n values $v_1, \ldots, v_n$ as its input and evaluates `e` with `xi` bound to `vi` for all i. This is represented as the type

```
Ast.Fun_e of Ast.id list * Ast.expr
```

- `(define x e)` evaluates the expression `e` and binds it to the id `x` at the top level. It is represented in the AST as

```
Ast.Def_e of Ast.id list * Ast.expr
```

  There is also a `definerec` version for creating recursive values (e.g. (definerec x (cons 1 x))).

- `(define (f x1 ... xn) e)` defines a function that takes n values `(v1 ... vn)` as its input and evaluates the expression `e`. The evaluation function should return a closure containing the AST representation of the function and the current environment. This is then bound to `f` at the top level. It is represented in the AST as

```
Ast.Def_e of Ast.id list * Ast.expr
```

  There is also a `definerec` version for creating recursive functions.

- `(if e0 e1 e2)` first evaluates `e0`. If the result is not a boolean, then the interpreter should raise a runtime exception. If the result is true, it should evaluate and return the result of `e1`. If the result is false, it should evaluate and return the result of `e2`. It is represented in the AST as

```
Ast.If_e of Ast.expr * Ast.expr * Ast.expr
```

- `(BO e1 e2)` represents one of the following binary operations:

BO	AST Name	Meaning
+	AST.Plus	Integer addition
-	Ast.Minus	Integer subtraction
*	Ast.Mult	Integer multiplication
/	Ast.Div	Integer division
%	AST.Mod	Integer modulus
=	Ast.Eq	Comparison (equality)
!=	Ast.Neq	Comparison (inequality)
<	Ast.Lt	Comparison (less than)
<=	AST.Leq	Comparison (less than or equal to)
>	Ast.Gt	Comparison (greater than)
>=	Ast.Geq	Comparison (greater than or equal to)
&	AST.And	Logical AND
|	Ast.Or	Logical OR

A binary operation is represented by the following AST type:

```
Ast.Binop_e of op * expr * expr
```

- `(UO e1)` represents one of the following unary operations:

UO	AST Name	Meaning
-	AST.Not	Logical complement
car	Ast.Car	Take the first element of a cons pair
cdr	Ast.Cdr	Take the second element of a cons pair
null	Ast.Null	Return true if the argument is nil, otherwise return false
load	Ast.load	Read a file (specified by a string-valued argument) from disk

A unary operation is represented by the following AST type:

```
Ast.Unop_e of op * expr
```

The minus sign is the only operator that can be used as both a unary and a binary operator.

Lists

The binary operator cons is used to create a pair. The expression `(cons e1 e2)`, when evaluated, should recursively evaluate `e1` and `e2`, then create a **`value`** of the form `Cons (v1, v2)` from the resulting values. There are corresponding projections `car` and `cdr` that extract the first and second components of a pair, respectively; thus

```
(car (cons 1 2)) = 1
(cdr (cons 1 2)) = 2
```

The names car and cdr are historical; they stand for *contents of the address register* and *contents of the decrement register*, respectively, and refer to hardware components of the IBM 704 computer on which LISP was originally implemented in the late 1950s.

The special entity `nil` is used with `cons` to form lists. Lists in Scheme are sequences of objects combined using cons and terminated by `nil`. For example, the list

```
(cons 1 (cons 2 (cons 3 nil)))
```

is the Scheme equivalent of the OCaml list `[1;2;3]`. The object `nil` serves as the empty list, and is analogous to the OCaml `[]`. The AST representation of `nil` is `Ast.Nil_e`.

There is a shorthand way to create lists in Scheme. Instead of typing

```
(cons 1 (cons 2 (cons 3 nil)))
```

one can type

```
(list 1 2 3)
```

and the resulting value is the same list. The output form of a list, i.e. the string that should be printed by your interpreter when you type either of the above expressions at the prompt, is `(1 2 3)`.

The display form of the value `(cons 1 2)` is `(1 . 2)`. Thus if you type `(cons 1 2)` at your Scheme interpreter, it should respond with `(1 . 2)`. Note, however, that if the second component is `nil` instead of 2, it should respond with `(1)`, because in that case it is a list of one element 1.

More generally, if you have any sequence of objects combined with cons, but not terminated with `nil`, then the output form puts a dot before the last element. Thus

```
(cons 1 (cons 2 3)) = (1 2 . 3)
(cons 1 (cons 2 nil)) = (1 2)
```

and

```
(cons (cons 1 2) (cons 3 4)) = ((1 . 2) 3 . 4)
(cons (cons 1 2) (cons 3 nil)) = ((1 . 2) 3)
```

Lazy Evaluation

Lazy Evaluation is an alternative to the eager evaluation semantics we implemented in our interpreter. We will simulate lazy evaluation within our interpreter with two new keywords: `delay`

and `force`. `delay` should suspend a computation, creating a "thunk" that is not evaluated until it is passed into a `force` expression. Forcing a thunk evaluates the delayed expression in the environment in which it was delayed. Note that `force` should not do anything on expressions that are not thunks; you may find using a reserved variable (one whose identifier begins with an underscore) helpful for this. Note that you are not allowed to change any of the interpreter's types or fundamental structure: You may only add two new handler functions and their entries in the dispatch table.

Semantic Overview

Operations in Scheme are all represented in prefix notation and must be enclosed in parentheses. For instance, to add 5 and 7, we would type `(+ 5 7)`. There is no notion of precedence and all compound expressions must be enclosed in parentheses. For instance, the OCaml expression

```
5 + (2 * 7)
```

would be written

```
(+ (5 (* 2 7)))
```

in Scheme. Comparisons may be performed on `Int`, `String`, and `Bool` types, while arithmetic operators are valid only on `Int`s, and boolean operations are valid only on `Bool`s.

Variables can be bound to values in several ways. Top-level global variables are bound using the `define` keyword. Local variables are bound using the **`let`** keyword. The expression

```
(let x 5 (* 2 x))
```

is equivalent to the OCaml

```
let x = 5 in 2*x.
```

Your Tasks

All code you submit must adhere to the specifications defined in the respective `mli` files. You must implement functions in the following files:

1. `eval.ml`: This module is responsible for evaluating Scheme code that has been converted to an AST. Complete the function `eval`. This should take in an `Ast.expr` and a `Heap.env` and return a `Heap.value`.

2. `heap.ml`: This module contains data types for scheme values, as well as types for keeping track of variable bindings. You can familiarize yourself with all functions outlined in heap.mli. This will include a function `value_to_string`. For `Str` values, simply output the appropriate string, and for `Int` and `Bool` values simply use OCaml's string casting methods. Closures should output as `<fun>` and `nil` as `()`.

3. `repl.ml`: This module contains functions for parsing input code and starting evaluation. First, complete the function `eval_one`. This should handle any top level operations (declarations and loads) that can't exist elsewhere in a program. Any declarations made in a loaded file should be included in the scope of the remainder of the calling program. Next, implement the function `eval_list`, which should be able to evaluate a sequence of expressions read in from the interpreter. There are helper functions for reading and writing from the interpreter provided for you as well.

   You may add function definitions to any `.mli` files, however you should be sure not to change any existing definitions.

Building Your Code

We have provided a `Makefile` that you can use to build your project. This will output a file `interpreter.exe` which you can execute to launch the command line interface. We have also inluded shell scripts `build.sh` and `clean.sh`, which build and clean your project, respectively.

What to Submit

Submit the following files: `eval.ml`, `repl.ml`, `heap.ml`, `ast.ml`, `util.ml`, `eval.mli`, `repl.mli`, `heap.mli`, `ast.mli`, and `util.mli`.

Submitting to CMS

To submit, zip the entire directory into a file `ps3.zip` containing the contents of the sub-directories `part1/`, `part2/` and `part3/`. In the root directory of the release run:

```
> zip -r ps3 .
```

and submit the resulting `ps3.zip` to CMS.