Programming Language Concepts

CSCI-344 — **Programming 4**

Term 20161 — **September 30, 2016**

Standard ML Programming

Due: October 13, 2016

1 Introduction

In this programming assignment, you will implement a number of functions in Standard ML in order to gain familiarity with the language and to practice functional programming.

Download `prog04.sml` and `prog04_tests.sml`. The former is a template for your submission and also includes a number of supporting functions. The latter is a test suite for the assignment.

2 Description

Complete the following exercises. Each one is to define one or more Standard ML functions. See Requirements and Submissions for important restrictions.

A. (10pts) Write functions `unzip` of type `('a * 'b) list -> 'a list * 'b list` and `zip` of type `'a list * 'b list -> ('a * 'b) list`. The function `unzip` takes a list of pairs and produces a pair of lists, while the function `zip` takes a pair of lists and produces a list of pairs; in each case, the order of elements is preserved. When `zip` is applied to lists of unequal length, the excess elements from the tail of the longer one are ignored. For example, `unzip [(1,2),(3,4),(5,6)]` should return `([1,3,5],[2,4,6])` and `zip ([1,3,5],[2,4])` should return `[(1,2),(3,4)]`. The `unzip` and `zip` functions satisfy the following algebraic laws:

- ⋆ `unzip-zip` law: `zip (unzip xys)` = `xys`
- ⋆ `zip-unzip` law: `unzip (zip (xs, ys))` = `(xs, ys)` if `length xs` = `length ys`

B. (5pts) Write a function `compound` of type `int -> ('a -> 'a) -> 'a -> 'a`. The application `compound n f`, for non-negative `n`, should return a function that, when applied to an argument (call it `x`), returns `f` applied `n` times to `n`. The idea of `compound n f` is that it returns `fn x => f (f (f (f ... (f x)...)))`, with `n` copies of `f`. For example, `compound 10 (fn x => x + 1) 10` should return 20, `compound 5 (fn x => x + 1) 10` should return 15, `compound 0 (fn x => x + 1) 10` should return 10, `compound 4 (fn x => x + x) 2` should return 32, and `compound 0 (fn x => x + x) 2` should return 2. Note: A function `f` applied 0 times to `x` is equivalent to `x`. Hint: You saw this function in μScheme in Chapter 2 of *Programming Languages: Build, Prove, and Compare.*

C. (5pts) Write a function `exp` of type `int -> int -> int`. The application `exp b e`, for non-negative `e`, should return b^e. For example, `exp 3 2` should return `9` and `exp 2 4` should return `16`. Note: `exp` must be implemented with `compound` and must not be recursive. Note: It will be convenient to define $0^0 = 1$.

D. (5pts) Write a function `existsUnique` of type `('a -> bool) -> 'a list -> bool`. The function `existsUnique` should return `true` if the input function returns `true` on exactly one element of the input list and should return `false` if the input function returns `true` on zero elements or more than one element of the input list. For example, `existsUnique (fn x => x mod 2 = 1) []` should return `false`, `existsUnique (fn x => x mod 2 = 1) [1]` should return `true`, `existsUnique (fn x => x mod 2 = 1) [1,2]` should return `true`, `existsUnique (fn x => x mod 2 = 1) [1,2,3]` should return `false`, and `existsUnique (fn x => x mod 2 = 1) [2,4,5,6,8]` should return `true`.

E. (5pts) Write a function `allAlt` of type `('a -> bool) -> 'a list -> bool`. The function `allAlt` should return `true` if the input function returns `true` on the elements in odd positions of the input list (first element, third element, fifth element, ...) and returns `false` on the elements in even positions of the input list (second element, fourth element, sixth element, ...). For example, `allAlt (fn x => x mod 2 = 1) []` should return `true`, `allAlt (fn x => x mod 2 = 1) [1]` should return `true`, `allAlt (fn x => x mod 2 = 1) [1,2,3,4,5,6,7,8,9]` should return `true`, and `allAlt (fn x => x mod 2 = 1) [1,2,3,4,6,7,8,9]` should return `false`.

F. (15pts) The μScheme interpreter in Standard ML implements environments using lists (see chunk 353 from Chapter 5 of *Programming Languages: Build, Prove, and Compare*):

```
type name = string
fun nameCompare (n1: name, n2: name) : order = String.compare (n1, n2)
fun nameEqual (n1: name, n2: name) : bool =
   case nameCompare (n1, n2) of EQUAL => true | _ => false
exception NotFound of name

(* list representation for environments *)
type 'a lenv = (name * 'a) list
val lenvEmpty = []
fun lenvFind (name, rho) =
   case rho of
      [] => raise NotFound name
    | (n, d)::tail =>
        if nameEqual (name, n) then d else lenvFind (name, tail)
fun lenvBind (name, data, rho) = (name, data) :: rho
```

We might call this the *list representation* for environments.

Now consider a *binary search tree representation* for environments:

```
datatype 'a btree = Leaf | Node of 'a btree * 'a * 'a btree
type 'a tenv = (name * 'a) btree
```

We maintain the invariant that a `'a tenv` is a binary search tree, ordered by the `name` component of the `name * 'a` elements of the tree. Furthermore, we maintain the invariant that all `name * 'a` elements of the tree have distinct `name` components.

- (5pts) Write a value `tenvEmpty` of type `'a tenv` that represents the empty environment (i.e., the environment that does not bind any names).
- (5pts) Write a function `tenvFind` of type `name * 'a tenv -> 'a` such that `tenvFind (name, rho)` returns the data associated with the name `name` in the environment `rho`. If the name `name` is not bound in the environment `rho`, then `tenvFind (name, rho)` should raise the exception `NotFound`. Note: `tenvFind` must be implemented so as to efficiently traverse the appropriate portion of the binary search tree.
- (5pts) Write a function `tenvBind` of type `name * 'a * 'a tenv -> 'a tenv` such that `tenvBind (name, data, rho)` returns the environment `rho` extended by a binding of the name `name` to the data `data`. Note: `tenvBind` must be implemented so as to efficiently traverse the appropriate portion of the binary search tree. Note: `tenvBind` must maintain the invariant that all `name * 'a` elements of the tree have distinct `name` components.

Hint: Use the provided `nameCompare` of type `name * name -> order` to compare names.

Hint: You may use the provided `btreeInsert` and `btreeLookup` functions, either directly in your implementation of `tenvFind` and `tenvBind` or indirectly as inspiration for your implementation of `tenvFind` and `tenvBind`.

G. (15pts) Now consider a *function representation* for environments:

```
type 'a fenv = name -> 'a
```

- (5pts) Write a value `fenvEmpty` of type `'a fenv` that represents the empty environment (i.e., the environment that does not bind any names).
- (5pts) Write a function `fenvFind` of type `name * 'a fenv -> 'a` such that `fenvFind (name, rho)` returns the data associated with the name `name` in the environment `rho`. If the name `name` is not bound in the environment `rho`, then `fenvFind (name, rho)` should raise the exception `NotFound`.
- (5pts) Write a function `fenvBind` of type `name * 'a * 'a fenv -> 'a fenv` such that `fenvBind (name, data, rho)` returns the environment `rho` extended by a binding of the name `name` to the data `data`.

Hint: This problem is similar in spirit to Problem H from Programming 03: Scheme Programming.

H. (40pts) In class, we noted that list append (`append` in Scheme and `@` in Standard ML) must copy the first list and, therefore, is a linear time operation. Algorithms that make extensive use of list append may suffer (in running time).

An *append* list is a (simple) implementation of the list abstract data type that makes construction cheap ($O(1)$), but makes destruction expensive ($O(n)$). In particular, appending two append lists is a constant time operation, but splitting a list into a head and a tail is a linear time operation. It is useful for algorithms that construct large lists in an irregular fashion and need only convert to a (Scheme or Standard ML) list at the end.

The `'a alistNN` and `'a alist` types are defined as follows:

```
datatype 'a alistNN = Sing of 'a | Append of 'a alistNN * 'a alistNN
datatype 'a alist = Nil | NonNil of 'a alistNN
```

The `'a alistNN` type represents the "non-nil" append lists, while the `'a alist` type represents arbitrary (nil or non-nil) append lists.

(a) (5pts) Write a function `alistAppend` of type `'a alist -> 'a alist -> 'a alist` that appends to append lists. The application `alistAppend xs ys` should run in $O(1)$ time.

(b) (5pts) Write a function `alistCons` of type `'a * 'a alist -> 'a alist` that adds a new element to the front of an append list. The application `alistCons (x,xs)` should run in $O(1)$ time.

(c) (5pts) Write a function `alistSnoc` of type `'a list * 'a -> 'a alist` that adds a new element to the rear of an append list. (*snoc* is "backwards" *cons*.) The application `alistSnoc (xs, x)` should run in $O(1)$ time.

(d) (5pts) Write a function `alistUnsnoc` of type `'a alist -> ('a list * 'a) option` that splits an append list into its last element and the remaining elements (as an append list). The application `alistUnsnoc xs` should run in $O(n)$ time.

(e) (5pts) Write a function `alistMap` of type `('a -> 'b) -> 'a alist -> 'b alist` that performs a *map* on an append list. The function `alistMap f xs` should run in $O(n * t)$ time (where t is the running time of the function f).
Note: Repeatedly calling `alistUncons` or `alistUnsnoc` will not achieve the required running time.

(f) (5pts) Write a function `alistFilter` of type `('a -> bool) -> 'a alist -> 'a alist` that performs a *filter* on an append list. The function `alistMap f xs` should run in $O(n * t)$ time (where t is the running time of the function f).
Note: Repeatedly calling `alistUncons` or `alistUnsnoc` will not achieve the required running time.

(g) (5pts) Write a function `alistFoldl` of type `('a * 'b -> 'b) -> 'b -> 'a alist -> 'b` that performs a *left-fold* on an append list. The function `alistFoldl f b xs` should run in $O(n * t)$ time (where t is the running time of the function f).
Note: Repeatedly calling `alistUncons` or `alistUnsnoc` will not achieve the required running time.

(h) (5pts) Write a function `alistToList` of type `'a alist -> 'a list` that converts an append list to a (Standard ML) list. The function `alistToList xs` should run in $O(n)$ time.
Note: `alistToList` must be implemented with either `alistFoldl` or `alistFoldr` and must not be recursive.

I. (15 pts; bonus 10 pts) A propositional-logic formula may be represented by the following Standard ML datatype:

```
datatype fmla =
   F_Var of string
 | F_Not of fmla
 | F_And of fmla * fmla
 | F_Or of fmla * fmla
```

The `F_Var` constructor represents a propositional variable (identified by a string); the `F_Not` constructor represents the logical negation of a sub-formula; the `F_And` and `F_Or` constructors represent the logical conjunction and disjunction of two sub-formulas.

(a) (5pts) Write a function `fmlaSize` of type `fmla -> int` that returns the size of a propositional-logic formula. A propositional variable has size 1; logical negation has size 1 plus the size of its sub-formula; logical conjunction and disjunction have size 1 plus the sizes of their sub-formulas.

(b) (5pts) Write a function `fmlaVarsOf` of type `fmla -> string list` that returns a list of all variables in a propositional-logic formula. The result list should not contain duplicates.

(c) (5pts) Write a function `fmlaEval` of type `fmla * bool env -> bool` that evaluates a propositional-logic formula. The second component of the argument (of type `bool env`) assigns a truth value to variables; assume that all variables that appear in the propositional-logic formula are bound in the environment.

(d) (bonus 10pts) Write a function `fmlaTautology` of type `fmla -> bool` that determines whether or not a propositional-logic formula is a tautology. (A propositional-logic formula is a tautology if *all* assignments of truth values to variables causes the propositional-logic formula to evaluate to true.)

3 Requirements and Submission

The purpose of this assignment is to practice programming the Standard ML language, not to practice searching the Standard ML libraries. Therefore, if there is a function from the Standard Basis Library that "solves" a particular problem, then you are asked to nonetheless define the function "from scratch" and without looking at the definition of the function from the Standard Basis Library.

Helper functions may be defined at the top level. You may also define local functions using `local` or `let`.

Always use pattern matching to inspect and deconstruct values. The use of `null`, `hd`, `tl`, `#1`, `#2`, etc., in any exercise will result in zero credit for that exercise.

Your submission must be a valid Standard ML program. In particular, it must pass the following test:

```
$ cat prog04.sml | /usr/local/pub/mtf/plc/bin/mosml -P full -quietdec
```

without any error messages. If your submission produces error messages (e.g., syntax errors or type errors), then your submission will not be tested and will result in zero credit for the assignment.

Submit `prog04.sml` to the `Programming 04` Dropbox on MyCourses by the due date.

Document History

September 30, 2016
Original version

A Interpreter

A reference Standard ML interpreter (MoscowML) is available on the CS Department Linux systems (e.g., `glados.cs.rit.edu` and `queeg.cs.rit.edu` and ICLs 1 and 2) at:

```
/usr/local/pub/mtf/plc/bin/mosml
```

Use the reference interpreter to check your code.

A.1 Interactive mode

Simply executing

```
$ /usr/local/pub/mtf/plc/bin/mosml -P full
```

will run the interpreter interactively, but without line editing.

Executing

```
$ rlwrap /usr/local/pub/mtf/plc/bin/mosml -P full
```

or

```
$ ledit /usr/local/pub/mtf/plc/bin/mosml -P full
```

will run the interpreter interactively with line editing. (See the manual pages for `rlwrap` and `ledit` for more details.)

A.2 Batch mode

Executing

```
$ cat prog04.sml | /usr/local/pub/mtf/plc/bin/mosml -P full
```

will run the interpreter on the contents of the file `prog04.sml`, but with prompts printed.

Executing

```
$ cat prog04.sml | /usr/local/pub/mtf/plc/bin/mosml -P full -quietdec
```

will run the interpreter on the contents of the file `prog04.sml` without prompts printed.

Executing

```
$ cat prog04.sml prog04_tests.sml | /usr/local/pub/mtf/plc/bin/mosml -P full -quietdec
```

will run the interpreter on the contents of the files `prog04.sml` and `prog04_tests.sml` without prompts printed.