Homework 4 — ML programming [A; C; E; K] — assigned 9 March — due 30 March

In all exercises except 4.3, you must use higher-order functions, including built-in ones from the Standard ML Basis Library, as well as those you define for your own data types, in preference to pointwise recursive function definitions. In exercise 4.3, use your aesthetic judgment.

In this homework, you may use the ML module language to structure your code. If you do, each structure and signature should be in a separate file (e.g., `expressions.sig` **and** `expressions.sml`**), and all the files for one exercise should be grouped into a common subdirectory; for instance** `hw4.1/expressions.sig`**. You should use the SML/NJ top level only to load all the modules with a sequence of** `use` **invocations and then execute your tests; and you must not use** `open` **anywhere.**

4.1 An arithmetic expression evaluator: writing recursive functions over algebraic datatypes (10pts) [C; K.1.1; K.2.3; K.2.4; K.2.7]

We use the following data type declaration to introduce a language of simple arithmetic expressions, with variables and binding:

```
datatype expr = Num of int
              | Var of string
              | Let of {var: string, value: expr, body: expr}
              | Add of expr * expr
              | Sub of expr * expr
              | Mul of expr * expr
              | Div of expr * expr
type env = string -> int
exception Unbound of string
val emptyEnv: env = fn s => raise (Unbound s)
fun extendEnv oldEnv s n s' = if s' = s then n else oldEnv s'
exception ExprDivByZero
```

Write a function `evalInEnv`, with type `env -> expr -> int`, which computes the arithmetic value of an expression (which may have free variables) in a given environment (a mapping from variables to `int` values).

Then you can define:

```
fun eval e = evalInEnv emptyEnv e
```

so that `eval` evaluates closed expressions.

Note: the code presented in class for this problem was written in the pointwise style.

4.2 Boolean formulae: writing recursive functions over algebraic datatypes (30pts) [A.1; C; K.1.1; K.2.3; K.2.4]

We can use the following declaration to introduce a language of Boolean formulae:

```
datatype expr = Const of bool
              | Var of int
              | And of expr list
              | Or of expr list
              | Not of expr
```

For instance, the Boolean formula $(\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2)$ is represented by the ML term `And [Or [Not (Var 1), Var 2, Var 3], Or [Var 1, Not (Var 2)]]`.

4.2.1 Simple Boolean evaluator (10pts)

Write a function `eval`, with type `env -> expr -> bool`, which computes the Boolean value of a formula.

The type `env` is the type of environments; an environment is simply an assignment of Boolean values to variables x_i. Choose your own ML representation for the type `env`.

4.2.2 More on Boolean formulae: satisfiability checker (10pts)

Write a function `satisfiable: expr -> bool`, which determines if the given formula is satisfiable, i.e., true for some assignment of Boolean values to the variables that appear in the formula. Note: efficiency is not a concern.

4.2.3 More on Boolean formulae: tautology checker (10pts)

Write a function `tautology: expr -> bool`, which determines if the given formula is a tautology, i.e., true for *all* possible assignments of Boolean values to the variables that appear in the formula.

4.3 Interpreter (60pts) [A.7; C; E; K.1.1; K.2.2; K.2.3]

4.3.1 General description

Consider a simple stack machine for integer list manipulation. The stack machine has a program *p*, a program counter *pc*, a stack *s* that may hold integers and lists, and a stack top pointer *sp*. The instructions of the machine and their effect on the stack are described by the following table:

In the table, *i* stands for an integer, *a* for an address, within the program, *r* for an address within the program, *t* for a list, *v* for an arbitrary value, and *s* for the remainder of the stack (0 or more values).

An instruction with an argument (`CST`, `GOTO`, `IFNZRO`, `CALL`, `LISTCASE`) takes up two positions in the program.

Instruction	Stack before		Stack after	Effect
`CST` *i*	*s*	⇒	*i s*	Push integer constant i
`ADD`	$i_2\ i_1\ s$	⇒	$(i_1+i_2)\ s$	Add integers
`SUB`	$i_2\ i_1\ s$	⇒	$(i_1-i_2)\ s$	Subtract integers
`DUP`	*v s*	⇒	*v v s*	Duplicate
`SWAP`	$v_2\ v_1\ s$	⇒	$v_1\ v_2\ s$	Swap
`POP`	*v s*	⇒	*s*	Pop
`GOTO` *a*	*s*	⇒	*s*	Jump to *a*
`IFNZRO` *a*	*i s*	⇒	*s*	Jump to *a* if $i \neq 0$
`CALL` *a*	*v s*	⇒	*v r s*	Call function at *a*, pushing return address *r*
`RET`	*v r s*	⇒	*v s*	Return: jump to *r*
`MKNIL`	*s*	⇒	`Nil` *s*	Push `Nil` (the empty list)
`MKCONS`	*t i s*	⇒	`Cons`(*i*,*t*) *s*	Push cons node
`LISTCASE` *a*	`Nil` *s*	⇒	*s*	If `Nil`, do not jump
`LISTCASE` *a*	`Cons`(*i*,*t*) *s*	⇒	*t i s*	If `Cons`, unpack components and jump to *a*
`PRINT`	*v s*	⇒	*v s*	Print *v* and keep it
`STOP`	*s*	⇒	_	Halt the machine

Under any circumstances not covered by the table, the machine will crash. For instance, if the instruction to be executed is an `ADD` but the top stack element is a list rather than an integer, the machine will crash.

4.3.2 The emulator (interpreter) (20pts)

Given the following elements of a list stack machine interpreter written in ML, complete the rest of the code.

```
datatype bytecode = B_CST
                  | B_ADD
                  | B_SUB
                  | B_DUP
                  | B_SWAP
                  | B_POP
                  | B_GOTO
                  | B_IFNZRO
                  | B_CALL
                  | B_RET
                  | B_MKNIL
                  | B_MKCONS
                  | B_LISTCASE
                  | B_PRINT
                  | B_STOP
                  | B_INT of int
                  | B_ADDR of int
```

```
type program = bytecode Vector.vector

datatype list' = Nil
               | Cons of int * list'

datatype object = (**** part A of your code here ****)

type stack = (**** part B of your code here ****)

fun listToString Nil = "Nil"
  | listToString (Cons (i, l)) =
      "Cons(" ^ Int.toString i ^ ", " ^ listToString l ^ ")"

fun objectToString (Integer i) = Int.toString i
  | objectToString (List l) = listToString l

fun execcode (p: program) (s: stack, pc: int)
  : stack * int =
   let
      fun step (s: stack, pc: int): (stack * int) option =
         case Vector.sub (p, pc) of
            B_CST => (case Vector.sub (p, pc+1) of B_INT i =>
                         SOME (Integer i :: s, pc+2))
          | B_ADD => (case s of (Integer i2)::(Integer i1)::s' =>
                         SOME (Integer (i1+i2) :: s', pc+1))


 (**** part C (the main part) of your code here ****)


      fun loop spc =
         let
            val next = step spc
         in
            case next of
               NONE => spc
             | SOME spc' => loop spc'
         end
   in
      loop (s, pc)
   end

fun exec p = execcode p ([], 0)

(* an example: *)
```

```
val program = Vector.fromList [B_CST, B_INT 1, B_CST,
                               B_INT 1, B_SUB, B_MKNIL,
                               B_MKCONS, B_PRINT, B_STOP]
val xxx = exec program;

(* under SML/NJ, the following is printed:
Cons(0, Nil)
val program =
  #[B_CST,B_INT 1,B_CST,B_INT 1,B_SUB,B_MKNIL,B_MKCONS,B_PRINT,B_STOP]
  : bytecode vector
val xxx = ([List (Cons (#,#))],8) : stack * int
```

4.3.3 Analysis and discussion (40pts)

Having implemented the interpreter, answer the following questions.

1. (5pts) Manually execute the code below on the stack machine. Show the stack contents after every instruction and show what is printed on the console:

   ```
   0: CST 100
   2: CST 11
   4: ADD
   5: MKNIL
   6: MKCONS
   7: PRINT
   8: STOP
   ```

2. (5pts) Write stack machine code to create and print the list `Cons(111,Cons(222,Nil))`.

3. (5pts) The instructions `CALL` and `RET` can be used to implement simple functions. What does the following stack machine program do, assuming that the integer 42 is on the stack top when execution is started at instruction address 0?

   ```
   0: CALL 4
   2: PRINT
   3: STOP
   4: DUP
   5: DUP
   6: MKNIL
   7: MKCONS
   8: MKCONS
   9: MKCONS
  10: RET
   ```

   Show the contents of the stack after each instruction, in the order in which the instructions are executed.

4. (5pts) Write a stack machine program that, given that integer $n \geq 0$ is on the stack top, builds and prints an n-element list of the form `Cons(`n`,Cons(`$n-1$`,...,Cons(1,Nil),...))`. Provide both an iterative and a recursive solution.

5. (5pts) The instruction `LISTCASE` can be used to test whether the list on the stack top is `Nil` or a `Cons`. If the stack top element is `Nil`, execution simply continues with the next instruction. If the stack top element is `Cons(`i`,`t`)`, then the integer i and the list tail t are unpacked on the stack and execution continues at address a.

   Consider the following stack machine code fragment:

   ```
   21: LISTCASE 27
   23: CST 999
   25: GOTO 28
   27: POP
   28: PRINT
   ```

   Assume that the list `Cons(111,Cons(222,Nil))` is on the stack top when the function at address 21 is called (by `CALL 21`). What is on the stack top, and is therefore printed, when control reaches instruction 28? What is printed if the empty list is on the stack top when `CALL 21` is executed?

6. (5pts) Write a stack machine function that, given that a list is on the stack top, computes the sum of the list elements. Provide both an iterative and a recursive solution.

7. (10pts) Write an ML function

   ```
   mklist: int -> bytecode Vector.vector
   ```

   which, for a given integer $n \geq 0$ generates stack machine code which, if executed starting with an initial program counter of 0 and an empty initial stack, will build and print a list of n `Cons` nodes of the form:

   `Cons(`$2n$`, Cons(`$2n-2$`, ..., Cons(2, Nil) ...))`

How to turn in

Turn in your code by running

~jackmp/cs451TA/handin your-file

on a regular UNM CS machine. You should use whatever filename is appropriate in place of your-file.

Include the following statement with your submission, signed and dated:
I pledge my honor that in the preparation of this assignment I have complied with the University of New Mexico Board of Regents' Policy Manual.