Homework 4 — assigned Monday 26 March — due Thursday 5 April

4.1 An arithmetic expression translator: writing recursive functions over algebraic datatypes (40pts) [C; K.1.1; K.2.3; K.2.4; K.2.7]

In this exercise we use the same SML data type as in homework exercise 3.2. The goal is to translate an expression, which may have free variables, into a C program that evaluates the expression. If the expression has free variables, the C program queries them from `stdin`. For instance, the expression `Add (Var "x", Var "a")` should be translated into the C program

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int a;
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _result_;

  printf ("a?\n");
  scanf ("%d", &a);
  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = a;
  _t_3_ = _t_1_ + _t_2_;
  _result_ = _t_3_;

  printf ("result=%d\n", _result_);
  return 0;
}
```

The C program queries the user for the values of the free variables of the expression in alphabetical (more precisely, lexicographical) order.

Any subexpression that can be evaluated independent of the values of free variables must be evaluated by the translator, not by the C program. This process is known as *constant folding*. To make this more precise: if and only if the arguments to an operator do not depend on any free variables, we consider that the result of the operator does not depend on any free variables. (This means that we do not use algebraic knowledge to simplify, e.g., $x - x$ to 0.)

If in the course of constant folding we encounter an attempt to divide by zero, we raise an exception.

The C program uses temporary variables $\mathtt{t}_n$, one for each node of the expression tree that could not be immediately evaluated by the translator; the right-hand side of each assignment statement is either one of the free variables or a C binary arithmetic operator (`+`,`-`,`*`,`/`) where each operand is either a temporary or

a constant. Finally, the temporary corresponding to the root of the expression tree is assigned to `result`, and that is printed.

Write a function `translate: expr -> string` that converts an expression into a string containing the appropriate C program.

For ease of testing, we impose additional specifications. The generated C code must obey the following evaluation order: to evaluate an expression $e_1 \circ e_2$, first evaluate e_1, then evaluate e_2, and lastly apply the operator, $\circ$. The generated C code must use the same white space and punctuation as in the examples (which are separately provided as C files).

Additional examples

- `Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"}`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _result_;

  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = 5 + _t_1_;
  _t_3_ = _t_2_ + _t_2_;
  _result_ = _t_3_;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- `Let {body=Add (Num 3,Num 6),value=Add (Num 5,Var "x"),var="x"}`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int x;
  int _result_;

  printf ("x?\n");
  scanf ("%d", &x);
```

```
  _result_ = 9;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- `Add (Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"},`
  `Add (Var "x",Var "a"))`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int a;
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _t_4_;
  int _t_5_;
  int _t_6_;
  int _t_7_;
  int _result_;

  printf ("a?\n");
  scanf ("%d", &a);
  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = 5 + _t_1_;
  _t_3_ = _t_2_ + _t_2_;
  _t_4_ = x;
  _t_5_ = a;
  _t_6_ = _t_4_ + _t_5_;
  _t_7_ = _t_3_ + _t_6_;
  _result_ = _t_7_;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- `Mul`

```
    (Add
       (Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"},
        Add (Var "x",Var "a")),
     Sub
```

```
(Let {body=Add (Var "x",Var "x"),value=Add (Num 5,Var "x"),var="x"},
 Div (Add (Var "x",Var "a"),Add (Var "x",Var "a"))))
```

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int a;
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _t_4_;
  int _t_5_;
  int _t_6_;
  int _t_7_;
  int _t_8_;
  int _t_9_;
  int _t_10_;
  int _t_11_;
  int _t_12_;
  int _t_13_;
  int _t_14_;
  int _t_15_;
  int _t_16_;
  int _t_17_;
  int _t_18_;
  int _t_19_;
  int _result_;

  printf ("a?\n");
  scanf ("%d", &a);
  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = 5 + _t_1_;
  _t_3_ = _t_2_ + _t_2_;
  _t_4_ = x;
  _t_5_ = a;
  _t_6_ = _t_4_ + _t_5_;
  _t_7_ = _t_3_ + _t_6_;
  _t_8_ = x;
  _t_9_ = 5 + _t_8_;
  _t_10_ = _t_9_ + _t_9_;
  _t_11_ = x;
```

```
  _t_12_ = a;
  _t_13_ = _t_11_ + _t_12_;
  _t_14_ = x;
  _t_15_ = a;
  _t_16_ = _t_14_ + _t_15_;
  _t_17_ = _t_13_ / _t_16_;
  _t_18_ = _t_10_ - _t_17_;
  _t_19_ = _t_7_ * _t_18_;
  _result_ = _t_19_;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- Let
  {body=Let {body=Add (Var "x",Var "a"),value=Num 3,var="a"},value=Num 6,
   var="x"}

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int _result_;


  _result_ = 9;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- Let
  {body=Div (Var "x",Var "x"),
   value=Let
           {body=Let {body=Add (Var "x",Var "a"),value=Num 3,var="a"},
            value=Num 6,var="x"},var="x"}

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int _result_;


  _result_ = 1;

  printf ("result=%d\n", _result_);
  return 0;
```

```
}
```

- `Sub (Var "x",Var "x")`

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int x;
  int _t_1_;
  int _t_2_;
  int _t_3_;
  int _result_;

  printf ("x?\n");
  scanf ("%d", &x);

  _t_1_ = x;
  _t_2_ = x;
  _t_3_ = _t_1_ - _t_2_;
  _result_ = _t_3_;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- `Let`

```
    {body=Let {body=Div (Var "x",Var "y"),value=Add (Var "x",Num 9),var="y"},
     value=Let
             {body=Div (Var "x",Var "x"),
              value=Let
                      {body=Let
                              {body=Add (Var "x",Var "a"),value=Num 3,var="a"},
                       value=Num 6,var="x"},var="x"},var="x"}
```

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int _result_;


  _result_ = 0;

  printf ("result=%d\n", _result_);
  return 0;
}
```

- `Let`

```
{body=Let {body=Div (Var "z",Var "y"),value=Add (Var "x",Num 9),var="y"},
 value=Let
        {body=Div (Var "x",Var "x"),
         value=Let
                {body=Let
                        {body=Add (Var "x",Var "a"),value=Num 3,var="a"},
                 value=Num 6,var="x"},var="x"},var="x"}
```

```
#include <stdio.h>
int main (int argc, char ** argv)
{
  int z;
  int _t_1_;
  int _t_2_;
  int _result_;

  printf ("z?\n");
  scanf ("%d", &z);

  _t_1_ = z;
  _t_2_ = _t_1_ / 10;
  _result_ = _t_2_;

  printf ("result=%d\n", _result_);
  return 0;
}
```

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

4.2.1 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.

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.

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

4.2.2 The emulator (interpreter) (36pts)

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
*)
```

Place the code inside a structure `ListStackMachine`. (In the example above, the code was not yet placed in a structure.)

4.2.3 Analysis and discussion (24pts)

Having implemented the interpreter, answer the following questions.

1. (3pts) Manually execute the code below on the stack machine. (The idea is that you should be able to do this entirely by hand, by reference to the machine definition given in the table. Feel free, however, to implement a modified, tracing, version of the machine that will do this automatically; if you do so, take the time to examine the output and understand what is going on.) 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. (3pts) Write stack machine code to create and print the list `Cons(111,Cons(222,Nil))`.

3. (3pts) 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. (3pts) 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. (3pts) 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. (3pts) 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. (6pts) 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

Submission instructions: see course mailing list.

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.