Simple Register Machine (Homework 1)

CSE 4102 Project 01, Spring Semester, 2018

Bryan Arnold

2/3/2018

Section: 001

Instructor: Jeffrey A. Meunier

Introduction

For this assignment, we created an interpreter for a very small imperative, which is a lower level language compared to other ones, such as object-oriented. This language will run on a two-register machine and will be simulated by a record data structure and several functions. Most of the code was provided to us by professor Meunier, but we implemented a few things here and there. The final product is a program that is entered into a single SML file. This will be used as the basis for future assignments.

Output

For the output for this project, I found it suitable to only show the output for the final run of the program outlined in section 6.7, as well as each function implementation and their returns. This shows the return types are correct for each function as well as every function when used works.

Note: I didn't test getA and getB, simply because they are useless for this assignment, as professor Meunier indicated in the assignment.

First, I'll show each function's implementation and return:

```
- type MachineState = {a:int, b:int};
type MachineState = {a:int, b:int}
-
- fun newMachine (a, b) : MachineState = {a=a, b=b};
val newMachine = fn : int * int -> MachineState
-
- fun getA (m : MachineState) = #a m;
val getA = fn : MachineState -> int
-
- fun getB (m : MachineState) = #b m;
```

```
val getB = fn : MachineState -> int
-
- fun setA (m : MachineState, newA) = newMachine (newA, #b m);
val setA = fn : MachineState * int -> MachineState
-
- fun setB (m : MachineState, newB) = newMachine (#a m, newB);
val setB = fn : MachineState * int -> MachineState
-
- datatype Instruction = SetA of int | SetB of int | Add | Sub | Disp;
datatype Instruction = Add | Disp | SetA of int | SetB of int | Sub
-
- fun i2s (SetA a) = "SetA " ^ Int.toString(a)
=  | i2s (SetB b) = "SetB " ^ Int.toString(b)
=  | i2s Add = "Add"
=  | i2s Sub = "Sub"
=  | i2s Disp = "Disp"
=  ;
val i2s = fn : Instruction -> string
-
- [SetA 10, SetB 2, Add, SetB 4, Sub, Disp];
val it = [SetA 10,SetB 2,Add,SetB 4,Sub,Disp] : Instruction list
-
- fun eval (m : MachineState, SetA a) : MachineState =
=       setA(m, a)
=   | eval (m : MachineState, SetB b) : MachineState =
=     setB(m, b)
=   | eval (m : MachineState, Add ) : MachineState =
=       setA(m, (#a m) + (#b m))
```

```
=   | eval (m : MachineState, Sub ) : MachineState =
=       setA(m, (#a m) - (#b m))
=   | eval (m : MachineState, Disp) : MachineState =
=   (print (Int.toString (#a m) ^ "\n");
=    m)
=  ;
val eval = fn : MachineState * Instruction -> MachineState
-
- fun run (m : MachineState, [] : Instruction list) = m
=   | run (m : MachineState, prog : Instruction list) =
=       let val instr = (hd prog)(* next instruc in prog list *)
=     val instrs = (tl prog) (* rest of prog list *)
=     val _ = print (i2s instr ^ "\n");
=     val m1 = eval (m, instr);
=    in
=   run (m1, (tl prog)) (* recursive call *)
=    end;
val run = fn : MachineState * Instruction list -> MachineState
```

Now, the final run of the program in section 6.7:

```
- val m = newMachine (0, 0);
val m = {a=0,b=0} : MachineState
-
- val prog = [SetA 10, SetB 2, Add, SetB 4, Sub, Disp];
val prog = [SetA 10,SetB 2,Add,SetB 4,Sub,Disp] : Instruction list
-
- run (m, prog);
```

SetA 10

SetB 2

Add

SetB 4

Sub

Disp

8

val it = {a=8,b=4} : MachineState

6.7 shows each instruction being ran at least once correctly, utilizes the recursive run function correctly, the eval function correctly, the instruction to string function correctly, the set functions correctly, while using the machine types correctly and instruction data structure correctly as well. So, this output was all I found suitable to be included in the report. The Machine state for m remained the same as well:

- m;

val it = {a=0,b=0} : MachineState

Source Code

```
(* Simple Register Machine (Homework 1) *)

(* CSE 4102 Project 01, Spring Semester, 2018 *)

(* Bryan Arnold *)

(* 2/3/2018 *)

(*Section: 001 *)

(* Instructor: Jeffrey A. Meunier *)


(* Type definition for MachineStates. A MachineState type *)

(* keeps track of two values that are associated with it, *)

(* integer a and integer b *);

type MachineState = {a:int, b:int};
```

```
(* Constructor for the MachineState type. Use *)
(* this to create a MachineType type *)
fun newMachine (a, b) : MachineState = {a=a, b=b};

(* getA and getB functions *)
(* These functions get the values of a and b *)
(* from the provided MachineState. Use this if *)
(* you want to get the values of a or b in a given *)
(* MachineState. *)
fun getA (m : MachineState) = #a m;
fun getB (m : MachineState) = #b m;

(* setA and setB functions *)
(* These functions set the values of a and b *)
(* from the provided MachineState by creating *)
(* a new state with the desired new value, and the other *)
(* old value. Use these functions to change the values *)
(* either in a or b in a given MachineState. *)
fun setA (m : MachineState, newA) = newMachine (newA, #b m);
fun setB (m : MachineState, newB) = newMachine (#a m, newB);

(* Datatype definition for the Instruction data type. *)
(* This type is used to define what instructions are permissible *)
(* and what can be used by the program. *)
datatype Instruction = SetA of int | SetB of int | Add | Sub | Disp;

(*i2s function takes an instruction *)
```

```
(* and turns it into it's string form. *)
(* Use this when you need to turn an instruction *)
(* into a string to be printed *)
fun i2s (SetA a) = "SetA " ^ Int.toString(a)
 | i2s (SetB b) = "SetB " ^ Int.toString(b)
 | i2s Add = "Add"
 | i2s Sub = "Sub"
 | i2s Disp = "Disp"
 ;

(* Default instruction list for the given problem *)
[SetA 10, SetB 2, Add, SetB 4, Sub, Disp];

(* eval function takes a state and an instruction *)
(* in the given format, and computes what was *)
(* desired by the function. Use this to evaluate *)
(* an instruction by replacing a MachineStates' value *)
(* for a or b, add b into a, subtract b from a, or display *)
(* the value of a *);
fun eval (m : MachineState, SetA a) : MachineState =
      setA(m, a)
  | eval (m : MachineState, SetB b) : MachineState =
            setB(m, b)
  | eval (m : MachineState, Add ) : MachineState =
      setA(m, (#a m) + (#b m))
  | eval (m : MachineState, Sub ) : MachineState =
      setA(m, (#a m) - (#b m))
  | eval (m : MachineState, Disp) : MachineState =
```

```
            (print (Int.toString (#a m) ^ "\n");
             m)
;

(*This function iterates through an instruction list *)
(* recursively, prints the current instruction being *)
(* looked at, call the eval function on it, then repeats *)
(* until the end of the list is reached. Use this to *)
(* evaluate multiple instructions in an instruction list. *)
fun run (m : MachineState, [] : Instruction list) = m
  | run (m : MachineState, prog : Instruction list) =
      let val instr = (hd prog)                    (* next instruc in prog list *)
                  val instrs = (tl prog)           (* rest of prog list *)
                  val _ = print (i2s instr ^ "\n");
                  val m1 = eval (m, instr);
            in
                  run (m1, (tl prog))                    (* recursive call *)
            end;
```