CSCI 3155: Homework Assignment 3

Spring 2012: Due Monday, February 27, 2012

Like last time, find a partner. You will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment. You are welcome to talk about these questions in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Recall the evaluation guideline from the course syllabus. Make sure that your file compiles and runs. A program that does not compile will not be graded.

Submission Instructions. To submit, upload to the moodle exactly two files named as follows:

- Homework3-YourIdentiKey.pdf with your answers to the written questions (scanned, clearly legible handwritten write-ups are acceptable)
- Homework3-YourIdentiKey.scala with your answers to the coding exercises

Replace YourIdentiKey with your IdentiKey. To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Getting Started. Download the code template Homework3.scala from the assignment page. Be sure to also write your name, your partner, and collaborators there.

1. Feedback. Complete the survey on the linked from the moodle after completing this assignment. Any non-empty answer will receive full credit.

2. Syntax: MYSTERY. In this exercise, we look the language MYSTERY used by the PL-Detective tool. The premise behind MYSTERY is that its syntax is fixed but its semantics can be changed (i.e., aspects of its semantics are a mystery). You can play with MYSTERY’s changeable semantics here:

   http://csci3155.cs.colorado.edu/pl-detective/hw/pldwelcome.htm

MYSTERY uses a syntax that is likely foreign to you, so this question asks you to figure out how write some simple MYSTERY programs based solely on the grammar specification and experimenting with the compiler.
Some aspects of MYSTERY’s semantics are fixed. The underlying computational model behind MYSTERY is imperative. Variables are the main state element, which are mutable. The keywords name constructs that are familiar from other programming languages.

From the syntax of MYSTERY, we see that a MYSTERY program is a block. A MYSTERY program starts by executing this block.

MYSTERY is designed to be as small as possible, yet large enough to illustrate many of the concepts that we will cover in this class. Thus, MYSTERY is missing many features that you may be used to in your favorite programming language. On the other hand, you will find later this semester that MYSTERY has some features that are more powerful than features in your favorite programming language.

For this exercise, write the following MYSTERY programs:

i. Write a program that prints out 1 through 100 (inclusive). You should test your program using this form:

   http://csci3155.cs.colorado.edu/pl-detective/hw/pldsyntax.htm

   You will receive a “User id” for the PL-Detective by e-mail. Use your account name as the “User id”, enter your code in the “Input” space, and click “Submit”. This will compile and run your program and present any errors or output in a new browser window. You may use as many attempts as you wish. As an additional challenge to yourself, try to do this in as few attempts as you can. Be sure to brag about how many tries it took in your submission!

ii. Write a program that has two procedures, sum and main. The procedure sum takes two integers as arguments and returns their sum. The procedure main calls sum with two arguments and prints out the result. You should use the same form to test your program:

   http://csci3155.cs.colorado.edu/pl-detective/hw/pldsyntax.htm

3. **Induction: Equivalence of Evaluation.** Consider the following simple arithmetic expression language represented in Scala:

   ```scala
   sealed abstract class Expr
   case class Value(i: Int) extends Expr
   case class Neg(sub: Expr) extends Expr
   case class Plus(left: Expr, right: Expr) extends Expr
   ```

   We can define an evaluator for this language called `aeval` as follows:

   ```scala
   def aeval(e: Expr): Int = e match {
   case Value(i)       => i
   case Neg(e)         => -aeval(e)
   case Plus(e1, e2)   => aeval(e1) + aeval(e2)
   }
   ```

   Alternatively, we can also define a function `redex` that performs local simplification (i.e., local reduction) and a `visit` function that applies a transformation to an expression bottom-up (like the exercise in Homework 2):
def redex(e: Expr): Expr = e match {
  case Neg(Value(i)) => Value(-i)
  case Plus(Value(i1), Value(i2)) => Value(i1 + i2)
  case _ => e
}

def visit(v: Expr => Expr, e: Expr): Expr = e match {
  case Value(i) => v(Value(i))
  case Neg(e) => v(Neg(visit(v, e)))
  case Plus(e1, e2) => v(Plus(visit(v, e1), visit(v, e2)))
}

We can see that visit(redex, e) should behave quite similarly to aeval(e) for any Expr value e. In fact, visit(redex, e) is also an evaluator. To make this precise, we can prove the following theorem.

**Theorem 1.** For all Expr values e and Int values i, if 
\[
\text{aeval}(e) \rightarrow^* i
\]
then 
\[
\text{visit}\text{\(\text{redex}, e\)} \rightarrow^* \text{Value}(i).
\]

For this exercise, prove Theorem 1. Be sure to state clearly the method of induction, the induction variable, and where you apply the induction hypothesis.

You may assume that aeval(e) terminates for any Expr value e. In other words, you may assume the following lemma:

**Lemma 1** (Termination and Determinism of aeval).

1. **Termination.** For all Expr values e,
\[
\text{aeval}(e) \rightarrow^* i
\]
for some Int value i.

2. **Determinism.** For all Expr values e and all Int values i and i', if 
\[
\text{aeval}(e) \rightarrow^* i \quad \text{and} \quad \text{aeval}(e) \rightarrow^* i',
\]
then i = i'.

Note that the other direction of Theorem 1 also holds, though you do not need show it for this exercise.

4. **Higher-Order Functions: Evaluator with Binding.** Suppose that we extend our arithmetic expression language with a let-binding construct:

   case class Let(bind: Expr, body: Int => Expr) extends Expr
to represent something like

\[
\text{let } x = e_{\text{bind}} \text{ in } e_{\text{body}}
\]

where \(x\) is bound to the value of \(e_{\text{bind}}\) in expression \(e_{\text{body}}\). Our representation does not actually introduce variables in our expression language. Instead, we represent \(e_{\text{body}}\) as a function that given an integer value, yields an expression (i.e., something like an expression with a hole to be filled in).

(a) Extend the \text{aeval} function from the previous question with a case to evaluate Lets. Make sure your test case uses Let and that the body of the Let uses the binding in some way.

(b) Compare and contrast this \text{aeval} function over this arithmetic language \text{Expr} and the \text{eval} function over the Boolean formulas \text{Formula} from Homework 2. Why does \text{eval} take an \text{Assignment}? Why don't we need something analogous for \text{aeval}? Explain.

5. Using Higher-Order Functions: Magic Squares. Recall the sat function that you wrote in Homework 2. This part will enable you to use your SAT solver from Homework 2 to do something fun: solve a puzzle.

Your SAT solver likely not very efficient; given a formula with \(n\) free variables, it probably always runs in \(O(2^n)\) time. That's fine for that homework assignment, and that's the worst case bound, but there has been significant progress in recent years in developing SAT solvers that can test for satisfiability much more efficiently in the "common case." One of the major applications of SAT solvers is in reasoning about the kinds of arithmetic hardware does, which is all carried out by bit twiddling.

Note that you don't actually need your solver working to complete these functions (i.e., you never need to call it in these functions). Here, we write functions to encode a puzzle into SAT. Once you have a puzzle encoded, you can try it out with your (inefficient) SAT solver for fun!

**Binary Arithmetic with SAT.** A \text{BitsFormula} is a list of boolean formulae representing the bits of a machine word, with the low order bit at the head of the list:

\[
\text{type } \text{BitsFormula} = \text{List}[\text{Formula}]
\]

For example, we would represent the number 13 as follows:

\[
\text{val } x = \text{List}(\text{B}(\text{true}), \text{B}(\text{false}), \text{B}(\text{true}), \text{B}(\text{true}))
\]

What is interesting about this type is that we can represent, for example, an unknown 4-bit number as

\[
\text{val } y = \text{List}(\text{Var}("a"), \text{Var}("b"), \text{Var}("c"), \text{Var}("d"))
\]

Below when we refer to \(x\) and \(y\), we will mean these two \text{BitsFormula}s. For this problem, we will only work with non-negative numbers.

We interpret the empty list \text{Nil} as 0 and allow padding of \text{B}(\text{false})s at the end. For example, \text{List}(\text{B}(\text{false})) and \text{List}(\text{B}(\text{false}), \text{B}(\text{false})) are also 0. Similarly, \text{List}(\text{B}(\text{true}), \text{B}(\text{false}), \text{B}(\text{true}), \text{B}(\text{false}))
is also 13.

**Conversions.** We have provided some conversion functions that allow to go between bit vectors and integers.

- **def intOfVec(v: BitsFormula): Int.** This function takes a BitsFormula composed solely of B(true)s and B(false)s and returns the integer equivalent. For example, `intOfVec(List(B(false), B(true)))` returns 2. This function does not handle the case when some element of the bit vector is neither B(true) nor B(false).

- **def vecOfInt(n: Int): BitsFormula.** This function takes a non-negative integer and returns the corresponding BitsFormula. For example, `vecOfInt(13)` returns \( x \) above.

- **def subst(a: Assignment, v: BitsFormula): BitsFormula.** This function reduces the \( v \) argument to a BitsFormula of all B(true)s and B(false)s by replacing the variables in \( v \) according to the Assignment \( a \) and then evaluating each bit. For example,

  ```scala
  subst(Map("x" -> true,"y" -> false),
       List(Var "x", B(true), And(Var "y", B(true)))))
  ```

  returns `List(B(true), B(true), B(false))`. We assume that all free variables in the input \( v \) are given in \( a \).

**Requirements and Hints.** Unless otherwise specified in this question, do not implement any functions by direct recursion. Instead use library methods from List and Tuple2. You might have been clamoring to use library methods in the last assignment. Now’s your chance!

For your convenience, we have provided some wrappers around the Scala library. You should be able to implement all of the exercises (concisely) using these wrappers:

```scala
def listMap[A,B](l: List[A])(f: A => B): List[B]
def foldLeft[A,B](l: List[A])(z: B)(f: (B,A) => B): B
def listPairMap[A1,A2,B](p: (List[A1], List[A2]))(f: (A1,A2) => B): List[B]
```

If your curious about the wrappers, a note about method calls in Scala. We have not talked specifically about objects, but for the moment, it is sufficient to rely on your prior background on objects. All values in Scala are actually objects (i.e., a record bundling data and operations). Methods are invoked on a receiver object using the same syntax as in Java, such as \( o.m() \) that calls method \( m \) on the receiver object \( o \).

Like in previous homework assignments, each function can be written quite concisely. If you are writing pages for a function, then you might reconsider your design.

(a) **Warm Up.** Write three functions that return the bit-wise not, and, and or, respectively:

  ```scala
  def bitnot(v: BitsFormula): BitsFormula
  def bitand(v1: BitsFormula, v2: BitsFormula): BitsFormula
  def bitor(v1: BitsFormula, v2: BitsFormula): BitsFormula
  ```

  Each of these functions should take 1 line of code using the library methods.
(b) **Binary Arithmetic.** Implement the following functions that manipulate BitFormulas to create Formulas that capture a property of interest.

i. **def isZero(v: BitsFormula): Formula.** This function returns a boolean formula representing whether the input BitsFormula represents zero. For example, if y is the four-bit number defined above, isZero(y) could return

\[
\text{And(Not(Var("a")), And(Not(Var("b")), And(Not(Var("c")), Not(Var("d")))))}
\]

This is really cool. Once we have this, we can ask a question like sat(isZero(y)), and the sat function will tell us, yes, there are choices for Var("a"), Var("b"), Var("c"), and Var("d") that make the formula true. When you're testing your code, you can use the functions subst and intOfVec above to look at your answers more easily.

ii. **def eqBitsFormula(v1: BitsFormula, v2: BitsFormula): Formula.** This function returns a formula representing whether the two bit vectors are equal. For example,

\[
\text{eqBitsFormula(List(Var("a")), List(B(true))})
\]

could return

\[
\text{Or(And(Var("a"), B(true)), And(Not(Var("a")), Not(B(true))))}
\]

In other words, the two vectors are equal if either both a and B(true) are true, or if both are false. Now we can ask questions like,

\[
\text{sat(eqBitsFormula(y, bitand(x, y))},
\]

that is, is there a 4-bit number such that if you bitwise and it with x, you get itself? You can assume for this problem that the two bit vectors have the same length. To implement this function, we ask that you write a helper function

\[
\text{def eqBit(f1: Formula, f2: Formula): Formula}
\]

that returns a formula that represents if the two bits are equal.

iii. **def add(v1: BitsFormula, v2: BitsFormula): BitsFormula.** This function returns a new BitsFormula representing the sum of the two vectors. Since there may be a carry out of the last bit, the resulting vector may have one more element than the input vectors. Your implementation for add should work for two bit vectors with different lengths. However, you may want to start by writing a version first that assumes two bit vectors with the same length. This function is the one exception. You can use direct recursion for this function, and hint: you will probably want to write a recursive helper function that has an extra parameter for the carry bit. We have also provided some other helper functions in the template.

(c) **Magic Squares.** A magic square is an n-by-n grid of the numbers 1 through \(n^2\) such that every number appears exactly once and the sum of each row, column, and diagonal is the same. For example, here is a 3x3 magic square:
In this part, you will write a function that enables you to solve a magic squares problem. In particular, what we will do is “reduce” the problem of finding a solution to the magic squares problem to a boolean formula. We can then use the sat function to actually find a solution.

To simplify things, we will consider only 3x3 squares. Here are the functions you should write:

i. **def disjoint(l: List[BitsFormula]): Formula**. This function takes a list of BitsFormulas and returns a formula representing whether all the BitsFormulas are different from each other. You may assume that all the BitsFormulas in the list have the same length. To help you write this function, we ask that you write a helper function

   **def disjointOne(v: BitsFormula, l: List[BitsFormula]): Formula**

   that returns a formula representing whether v is different from all the bit vectors in l.

ii. **def isDigit(v: BitsFormula): Formula**. The input is a BitsFormula of length 4 (i.e., exactly 4 boolean formulae). This function returns a formula that is true if and only if the BitsFormula is greater than or equal to 1 and less than or equal to 9.


   This function takes exactly nine BitsFormulas and returns a formula representing whether the tuple is a magic square. The tuple

   \[(v1, v2, v3, v4, v5, v6, v7, v8, v9)\]

   represents the following square:

   \[
   \begin{array}{ccc}
   v1 & v2 & v3 \\
   v4 & v5 & v6 \\
   v7 & v8 & v9 \\
   \end{array}
   \]

   You can assume that the given BitsFormulas consist of exactly four boolean formulae each (i.e., they are four-bit numbers). Hint: you will want to call isDigit and disjoint in your implementation of isMagic.

   You can try out your isMagic function by seeing if it can decide that the square above is magic. You can also try replacing one or two entries by variables, and then seeing if your code correctly finds the right numbers to fill in. However, you’ll have to wait a long, long time if you try making a lot of the entries in the magic square variables, since your sat function probably takes exponential time.
(d) **Testing.** Write unit tests in the same way as in previous homework assignments. Templates for the tests are given in the homework skeleton. Recall that a test case is a Boolean expression that evaluates to `true` to indicate success.

**Acknowledgements.** This project on Magic Squares is based on a similar one designed by Jeff Foster.