Ben Reichert
Homework #1
CS322

Question 1

I found that working backwards by implementing features in the target language and then in the source language actually made things easier at times. The Mult class was fairly straightforward as it was a binary operator, similar to the Plus or Minus classes. I just copied the Plus class, and changed the operand from '+' to '*'.

```
class Mult extends IExpr {
    ...
    int eval(Memory mem){
      return l.eval(mem) * r.eval(mem);
    }

    Code compileTo(Reg reg, Code next){
      Reg tmp = new Reg();
      return l.compileTo(tmp,
          r.compileTo(reg,
         new Op(reg, tmp, '*', reg,next)));
    }
    String show() {
      return "(" + l.show() + " * " + r.show() + ")";
    }
  }
```

The only change to the Target language was adding another case to the Op class:

```
case '*' : r.set(x.get() * y.get());
                break;
```

For the Not class, I added a case in the Op class Target code with the "!" char, and the appropriate boolean not expression. Since Not is a unary operator, I could not use the code from the Plus or Mult classes. Instead, I opted to do the Target code first so I could simplify the Src code. The show() method is pretty boring, and the constructor.

Here is the relevant code:

```
class Not extends BExpr{
  ...
  boolean eval(Memory mem){
    return !exp.eval(mem);
  }

  String show(){
      { return "not(" + exp.show() + ")";}
  }

  Code compileTo(Reg reg, Code next){
    return exp.compileTo(reg,
           new NotT(reg, next));
  }
}
```

Since I already wrote the NotT Target.java code, the Source langauge is simple. The eval method simply evaluates the boolean expression, and returns the negated version.

```
class NotT extends Code {
  ...
  void print(){
    System.out.println("   " + reg + " <- ![" + reg + "]");
    next.print();
  }

  Block run(Memory mem){
    reg.setBool(!reg.getBool());
    return next.run(mem);
  }
}
```

The Target code run method is similar to the eval method in Src in that it negates the value. Instead of using the set() and get() methods of register objects, we use the boolean wrapper methods as we are handling boolean data, but storing it internally as integer values.

Even was a little trickier to think about, but still easy to understand once implemented. For the eval() method, we evaluate the expression coming in. From this, we get an integer value, and mod by 2. Then we do the == comparison to get a boolean value and return.

```
class Even extends BExpr {
...
    boolean eval(Memory mem){
      return exp.eval(mem)%2 == 0;
    }
    String show() {
      return "even("+ exp.show() + ")";
    }
   Code compileTo(Reg reg, Code next){
     return exp.compileTo(reg, new EvenT(reg,next));
    }
  }
```

As you can see in the Source code, we compile to a new EvenT in Target.java. In which, we get the integer value in the register, mod by two, and through the comparison operator return the boolean value. We then set the boolean result back into the register.

```
class EvenT extends Code {
  …
  void print(){
    System.out.println("  " + register + " <- EV[" + register +"]");
    next.print();
  }

  Block run(Memory mem){
    boolean result = false;
    if(register.get()%2 == 0){
      result = true;
    }
    register.setBool(result);
    return next.run(mem);
  }

}
```

Halve is very similar to Even except instead of doing %2, we divide by two. Since this is integer division, we get remainders, but ignore them. So 7/2 would be 3 remainder 1, or just 3.

```
class Halve extends IExpr{
   …
    int eval(Memory mem){
      return exp.eval(mem)/2;
    }
    String show(){
      return "halve("+ exp.show() + ")";
    }
    Code compileTo(Reg reg, Code next){
      return exp.compileTo(reg, new HalveT(reg,next));
    }
  }
```

The Halve target code does the same thing as the Source code, setting the register to it's original value, divided in half.

```
class HalveT extends Code{
  …
  void print(){
    System.out.println("  " + reg + " <- HV[" + reg + "]");
    next.print();
  }

   Block run(Memory mem){
    reg.set(reg.get()/2);
    return next.run(mem);
  }
}
```

Less than or Equal to (LTE and LTET) is similar to the Mult class as it is a binary operation. The eval() method is straightforward as it just does a boolean comparison of the two evaluated sides.

```
class LTE extends BExpr {
...
    boolean eval(Memory mem){
      return (l.eval(mem) <= r.eval(mem));
    }
    String show(){
      return "(" + l.show() + " <= " + r.show() + ")";
    }
    Code compileTo(Reg reg, Code next){
      Reg tmp = new Reg();
      return l.compileTo(tmp,
             r.compileTo(reg,
             new LTET(reg, tmp, reg, next)));
    }
  }
```

The Target code run() method sets takes the two sides as integer operands, does a boolean <= comparison and then sets that value in the register.

```
class LTET extends Code{
 ...
  void print(){
    System.out.println(" " + r + " <- " + x + " <= " + y);
    next.print();
  }

  Block run(Memory mem){
    r.setBool(x.get() <= y.get());
    return next.run(mem);
  }
}
```

Testing

Throughout all the 5 methods implemented I would write test programs that would test that method before moving onto the next. The only logical thing to do when testing the Not, Even, and LTE operators was to shove them into the boolean conditional in the while loop of the (edited) Main.java file. This was in part due to the ease of testing since the while was already constructed and wrapping the conditional with another function is trivial. While it's not pretty, it was a way to ensure the tested output from the compiler and interpreter was the same. I got the expected output and compiler output to match for every test I did (except when I wrote bag bugs!). Doing some example runs on paper also confirmed that this testing program was working as it should be.

Here is that test file:

```
class Main {
  public static void main(String[] args) {
    Stmt s
     //if 2 is even print 4, else print 123456
     = new Seq(new If(new Even(new Int(4)),
           new Print(new Int(4)), new Print(new Int(123456))),
       new Seq(new If(new Not(new Even(new Int(4))),
           new Print(new Int(4)), new Print(new Int(123456))),
       new Seq(new Assign("t", new Int(1)),
       new Seq(new Assign("i", new Int(1)),
       new Seq(new While(new Not(new LTE(new Halve(new Int(10)), new Var("i"))),
                         new Seq(new Assign("t", new Mult(new Var("t"), new Var("i"))),
                                 new Assign("i", new Plus(new Var("i"), new Int(1))))),
       new Print(new Var("t")))))));
       ...
  }
}
```

And the expected output:

```
Complete program is:
    if (even(4)) {
      print 4;
    } else {
      print 123456;
    }
    if (not(even(4))) {
      print 4;
    } else {
      print 123456;
    }
    t = 1;
    i = 1;
    while (not((halve(10) <= i))) {
      t = (t * i);
      i = (i + 1);
    }
    print t;
Running on an empty memory:
Output: 4
Output: 123456
Output: 24

…..

L9:
  r12 <- 4
  r12 <- EV[r12]
  r12 -> L7, L8

Running on an empty memory:
Output: 4
Output: 123456
Output: 24
```

Question 2

I've scanned in my AST drawing for the program, which will be an entire page of this finalized document. Attached at end of document.

Here is the java code that constructs the tree in code.
Question2.java:

```
class Problem2 {
  public static void main(String[] args) {
    Stmt s
    = new Seq(new Assign("t", new Int(0)),
      new Seq(new Assign("x", new Int(6)),
      new Seq(new Assign("y", new Int(7)),
      new Seq(new While(new LT(new Int(0), new Var("x")),
        new Seq(new If(new Not(new Even(new Var("x"))),
              new Assign("t", new Plus(new Var("t"), new Var("y"))),
              new Assign("t", new Var("t"))),
          new Seq(new Assign("y", new Plus(new Var("y"), new Var("y"))),
              new Assign("x",new Halve(new Var("x")))))),
       new Print(new Var("t"))))));
  ...
  }
}
```

Running the java program:

```
Complete program is:
    t = 0;
    x = 6;
    y = 7;
    while ((0 < x)) {
      if (not(even(x))) {
        t = (t + y);
      } else {
        t = t;
      }
      y = (y + y);
      x = halve(x);
    }
    print t;
Running on an empty memory:
Output: 42
...
```

Of course the answer is 42! Both the interpreter and compiler output the same result as expected. When I stepped through the application on paper I realized that we are adding 14 to 28 to t and then printing, so 42! This was verified through the output of the ASM and Source interpreted code. The only difference between the code we were tasked to build and mine is

that it has extra parentheses due to how I print the code as it is run (the parentheses made it easier to debug while writing tests, etc).

Question 3

Output Pseudo-ASM from Question 2 program:

```
Entry point is at L6
L0:
  r9 <- 0
  r1 <- [x]
  r1 <- r9<r1
  r1 -> L4, L5

L1:
  r4 <- [y]
  r3 <- [y]
  r3 <- r4+r3
  [y] <- r3
  r2 <- [x]
  r2 <- HV[r2]
  [x] <- r2
  goto L0

L2:
  r7 <- [t]
  r6 <- [y]
  r6 <- r7+r6
  [t] <- r6
  goto L1

L3:
  r8 <- [t]
  [t] <- r8
  goto L1

L4:
  r5 <- [x]
  r5 <- EV[r5]
  r5 <- ![r5]
  r5 -> L2, L3

L5:
  r0 <- [t]
  print r0
  stop

L6:
  r12 <- 0
  [t] <- r12
  r11 <- 6
  [x] <- r11
  r10 <- 7
  [y] <- r10
  goto L0

Running on an empty memory:
Output: 42
```

The first optimization that I saw that could be applied to the Target code was that the Assign method used a register that wasn't needed. For example:

```
L6:
  r12 <- 0
  [t] <- r12
```

That register r12 isn't needed. We could simply reduce that instruction set to [t] <- 0 and skip the register all together.

The second optimization that I noticed was that this code does an awful lot of nothing:

```
L3:
  r8 <- [t]
  [t] <- r8
  goto L1
```

This is the Assignment of t=t; in the source language. Through analysis, we could pick out this operations by tracking if the registers are used again. If they are not, and a value is assigned to them, then they can be removed without consequence. In this case, we could replace all goto L3 operations with goto L1 to bypass these set of operations that don't do anything.

The third optimization that I could do would be that halving, and even could be further optimized through bitshifting instead of traditional division operators. For example, the halve operation divides by 2 on integers. Since we can shift the bits right, we will fill the left-most digit with a 0, and the right-most will get dropped. Since the right-most is the 1 placeholder we drop the remainder if it was a 1, and get the result of the original integer divided by two. All of this with only 1 clock cycle as compared to dozens when doing traditional division. With the even method, we can just check if the last bit is a 1, and if so return false. If the 1 placeholder is a 0, then the number is even and we can return true. This reduces the overhead of dividing by 2 and checking if the remainder is 1 or 0.

PROBLEM 2 AST