CS401 Optimization Final report .
I started off the term having just finished the CS321-322 Compilers sequence working with the mini compiler. The task at hand was to learn about global optimizations, by extending the MIL optimizer to optimize across block boundaries.

I began by reading the optimizations chapter of Mogensen's Basics of Compiler Design. I took the information from this and designed and implemented global constant propagation. The high level overview of global constant propagation is to create a lattice structure for each block parameter, and for every parameter that has a small number of constant values for it, replace the block call with a derived block using the constant value for that parameter.
Later I constructed the data flow algorithms for Available and Anticipated Expressions, two of the four data flow passes needed for lazy code motion.

Details of the Global Constant Propagation To retrieve the information about callers to a block, I created a data flow path that runs through the code objects for each block, returning a linked list of block calls to the current block.

```
/** getBlockCall
 *  @param id The id of the Block which you want block calls to.
 *    @return a list of BlockCall objects which call id
 *
 *  If the tail of this object is a Block call,
 *  compare if it calls the passed in id
 *  calls getBlockCall on the following code c.
 *  if the tail calls block id, cons the tail
 *  to the list returned from c.getBlockCall

 */
public BlockCalls getBlockCall(String id)
case Code { return null; }
case Bind {
    BlockCall thisCall = null;
    BlockCall bc = t.isBlockCall();
    if (bc instanceof BlockCall) {
        if (bc.callsBlock(id)) {
            thisCall = bc;
        }
    }
    BlockCalls calls = c.getBlockCall(id);
    if (thisCall != null)
        calls = new BlockCalls(thisCall, calls);

    return calls;
}
```

```
case Done {
    BlockCall thisCall = null;
    BlockCall bc = t.isBlockCall();
    if (bc != null)
    {
        if (bc.callsBlock(id)) {
            thisCall = bc;
        }
    }
    BlockCalls calls = null;
    if (thisCall != null)
        calls = new BlockCalls(thisCall, calls);

    return calls;
}
case Match {
    BlockCalls block_calls = null;
    if (def!=null) {
        if (def.callsBlock(id)) {
            block_calls = new BlockCalls(def, block_calls);
        }
    }

    for (int i=0; i<alts.length; i++) {
        BlockCall alt_blockCall = alts[i].getBlockCall(id);
        if (alt_blockCall != null
        && alt_blockCall.callsBlock(id))
        {
            block_calls = new BlockCalls(alt_blockCall, block_calls);
        }
      }
    return block_calls;
}

public BlockCall getBlockCall(String id)
case TAlt {
    if (bc != null && bc.callsBlock(id))
        return bc;
    return null;
}
```

Once a list is retrieved, there are two cases to look at, if the calls are recursive or not. For the recursive case, there are two types of potential modifications of the formals. They are both handled by the checkArguments method in the Block class. First, there is the possibility of block calling itself with different parameters than its formals list. The getBlock call method is used on the code

field, and on each BlockCall returned if the argument is a Constant, it is saved to the lattice structure, if it is a different variable than the lattice value for that argument is set to NAC. Once the return calls are checked, a second pass is run to see if the incoming argument is given a new value with a call to Bind. Currently, if Bind is called on an incoming parameter the argument is set to NAC. In the non-recursive case the block call arguments are checked, if a parameter is constant it is added to the lattice for that parameter, if the set of constants is too large the lattice gets set to the NAC value, otherwise the lattice is a set of constants used for that parameter.

```
private  Atom [] checkArguments()
   case Block {
       Atom [] arguments = new Atom[formals.length];
        for (int i = 0; i < formals.length; ++i)
            arguments[i] = formals[i];
        BlockCalls bc = code.getBlockCall(id);
        while (bc != null) {
            for (int i = 0; i < arguments.length; ++i)
            {
                Atom a = bc.head.args[i];
                if (!arguments[i].sameAtom(a)) {
                    if (a.isConst() != null) {
                        arguments[i] = a;
                    }
                    else {
                        arguments[i] = NAC.obj;
                    }
                }
            }
            bc = bc.next;
        }
        arguments = code.checkformals(arguments);
       for (int i = 0; i < formals.length; ++i) {
           if (arguments[i] == null) break;
           if (arguments[i] != NAC.obj) {
               if (arguments[i].isConst() != null
               && !arguments[i].sameAtom(formals[i])) {
                   arguments[i] = NAC.obj;
               }
           }
           else {
               debug.Log.println("Argument " + i + " is modified");
           }
       }
       return arguments;
   }
```

Once the lattice information is complete any constants found are used to create a derived block with that parameter replaced by the constant, with a hardcoded limit of 3 levels of derivation.
The derived block is created using the following code segment.

```
// Current method, copy the entire block, and apply AtomSubst
b.code = code.copy();
b.formals = nfs;
b.replacedVar = knownArgs[j][k];
b.code.replaceCalls(id, j, formals[j], b);
b.code = b.code.apply(new AtomSubst(formals[j], knownArgs[j][k], null));
```

Results from Global Constant Propogation The resulting MIL program after a being run through this algorithm simplifies the resulting code and removes unnecessary comparisons that can be determined by the current local optimizer. The first example of the improvements is the following test program:

```
void mini_main() {
  int   N = 10;
  int[] a = new int[N];

  // Compute some fibonnacci numbers:
  a[0] = 0;
  a[1] = 1;
  int   i = 2;
  while (i<N) {
    a[i] = a[i-1] + a[i-2];
    i = i + 1;
  }

  // Print them out:
  i = 0;
  while (i<N) {
    print a[i];
    i = i + 1;
  }
}
```

This program results in the following optimized code after being run through the local optimizer. The number of code blocks in not reduced, primarily because the n parameter, although unchanged is not know by the blocks.

```
not recursive
b4() =
  return _
--------------------------------------
recursive
b3(N, i, a) =
  t2 <- mul((i, 4))
  t3 <- add((a, t2))
  t4 <- load((t3))
  _ <- print((t4))
  i <- add((i, 1))
  t35 <- lt((i, N))
  case t35 of
    True() -> b3(N, i, a)
    False() -> b4()
--------------------------------------
not recursive
b8(N, a) =
  t29 <- lt((0, N))
  case t29 of
    True() -> b3(N, 0, a)
    False() -> b4()
--------------------------------------
recursive
b7(N, i, a) =
  t7 <- mul((i, 4))
  t8 <- add((a, t7))
  t12 <- add((t8, -4))
  t13 <- load((t12))
  t17 <- add((t8, -8))
  t18 <- load((t17))
  t19 <- add((t13, t18))
  _ <- store((t8, t19))
  i <- add((i, 1))
  t36 <- lt((i, N))
  case t36 of
    True() -> b7(N, i, a)
    False() -> b8(N, a)
--------------------------------------
not recursive
b0() =
  a <- newarray((40))
  _ <- store((a, 0))
  t22 <- add((a, 4))
  _ <- store((t22, 1))
  b7(10, 2, a)
```

After being run through the global constant propagation algorithm, as well as running the local optimizer after each global constant propagation pass, the unnecessary comparison of 10 to 0 in b8 is completely removed, as well as no longer needing to pass the n parameter throughout the program.

```
--------------------------------------
not recursive
b4() =
  return _
--------------------------------------
recursive
b11(i, a) =
  t48 <- mul((i, 4))
  t49 <- add((a, t48))
  t50 <- load((t49))
  _ <- print((t50))
  t52 <- add((i, 1))
  t53 <- lt((t52, 10))
  case t53 of
    True() -> b11(t52, a)
    False() -> b4()
--------------------------------------
recursive
b9(i, a) =
  t37 <- mul((i, 4))
  t38 <- add((a, t37))
  t39 <- add((t38, -4))
  t40 <- load((t39))
  t41 <- add((t38, -8))
  t42 <- load((t41))
  t43 <- add((t40, t42))
  _ <- store((t38, t43))
  t45 <- add((i, 1))
  t46 <- lt((t45, 10))
  case t46 of
    True() -> b9(t45, a)
    False() -> b11(0, a)
--------------------------------------
not recursive
b0() =
  a <- newarray((40))
  _ <- store((a, 0))
  t22 <- add((a, 4))
  _ <- store((t22, 1))
  b9(2, a)
```

Reflections of the Global Constant Propagation Algorithm Looking back at the code for Global Constant Propagation, and thinking about it after completing the Available and Anticipated Expressions, if I were to rework the code, a better approach would have been taking the information about all parameters together; if (x,y) are always constants paired together, derive a block with both of them replaced together, which would remove the case of creating 4 blocks if the two pairs of constants used were (0,1) and (1,0). As it is currently implemented, each of these cases would be specialized into b(x/0) b(x/1) b(y/0) b(y/1).
An alternative approach to this optimization could possibly be achieved during the construction of the abstract syntax tree, by keeping a count of assignments to each Id object, if the number of assignments is one at the end of this construction this information could possibly be used during MIL code generation and replacing the Id with its only assigned value.

Dataflow Analysis Available expressions was the first dataflow analysis that I wrote, using the algorithm presented in the "Dragon Book". This algorithm uses intersection of the 'OUT' sets of the preceding blocks to form the 'IN' set of a block, the IN set is passed through each code segment, facts are removed from this set by the variables that are assigned to by a Bind object, and facts are added to by Bind and Done objects. Learning about the transfer functions, gen/kill functions, and meets functions really gave me a better perspective into how compilers arrive at the information necessary to perform optimizations that cross block boundaries, prior to reading about and implementing this algorithm, I was able to look at a piece of IL / ASM code and point out optimizations, but unable to generalize it in a algorithmic way.

Much of the work on the dataflow analysis algorithms was in choosing the best representation of the IN/OUT sets. In my readings, a specific choice was not presented and it was left as an exercise for the user. Professor Jones explained to me how in many compilers a bitfield is used to represent them by assigning every expression to a particular bit, which would have required a mapping function that always maps the same expression to the same bit. Instead of this manner, I chose instead to use a linked list for the set representation. Each node in the linked list is a G_Fact object, denoting GlobalFact, with each G_Fact object holding a Tail object and a linked list of atoms that the value of the expression is stored in.
My Familiarity with linked lists was beneficial in writing the meets functions, but did not preclude bugs. I had mistakenly added elements that were not in the intersection twice to the results for the union meets. Additionally instead of adding a new tail to the end of the list in the Gen function, I mistakenly replaced the last node with the new node. The pass through the data is listed below, the flow of the algorithm begins at the block object, and to compute if there have been changes the new outset is compared with the old outset.

```
public int Calculate_Avail_Expr()
```

```
case Block {
    boolean union = true;
    avail_Next_Out = code.outset(avail_In_Set, id);
    int oldlen = G_Facts.length(avail_Out_Set);
    if ((oldlen != G_Facts.length(avail_Next_Out) )
        ||
      (oldlen != G_Facts.length(G_Facts.meets(avail_Next_Out, avail_Out_Set, !union)))
      ) {
       return 1;
    }
    return 0;
}
```

The code objects within the block create the outset from the inset using the method outset, using the generate and kill set operations.

```
public G_Facts outset(G_Facts ins, String id)
    case Bind {
        G_Facts outs = null;
        G_Fact d = null;
        if (t.isPure()) {
            d = new G_Fact(t, new Atoms(v, null));
        }

        if (ins == null) {
            if (d != null) {
                outs = new G_Facts(d, null);
            }
        }
        else {
            outs = t.addIns(ins);
            outs  = G_Facts.meets(outs, ins, true);
            outs = outs.kill(v);
            if (d != null) {
                outs = outs.gen(d);
            }
        }
        if (c == null) {
            debug.Log.println("unlinked bind call found!?!");
            return outs;
        }
        return c.outset(outs, id);
        }
    case Done {
            G_Facts outs = t.addIns(ins);
            return G_Facts.meets(outs, ins, true);
        }
```

```
case Match {
        Sets outs = null;
        if (def != null) {
            G_Facts defOuts = def.addIns(ins);
            if (defOuts != null) {
                outs = new Sets(new Set(defOuts), outs);
            }
        }
        for (int i=0; i<alts.length; i++) {
            G_Facts altsOuts = alts[i].addIns(ins);
            if (altsOuts != null) {
                outs = new Sets(new Set(altsOuts), outs);
            }

        }
        return ins;
        }
```

Reflections of Available Expressions Work on the Available Expressions began as I was beginning to be quite comfortable with the original MIL objects and the local optimizations. I realize after the fact that I would have been better able to find bugs in the linked list and set code if I had written them either with unit tests or another type of correctness verifications, several hours of debugging would have been saved by writing the tests first.
I am satisfied with the clarity of the final code, but a potential improvement would be to replace the set of tail objects with a specific bitfield assigned to each possible expression in the program.

Anticipated Expressions was the final dataflow pass that I wrote. It consisted of a recursive descent algorithm that takes the intersection of the IN sets from successor blocks and adds to it and removes from in much in the same was as Available Expressions, but flowing through the block in reverse. It reuses many of the set operations that were implemented for Available expressions, and presented the challenge of understanding where it is appropriate to check if the IN sets have been changed since the last pass, as opposed to the forward data flow analysis that Available Expressions used, rather than computing the set before running through each of the blocks, it is needed to compute them after running the data flow through the blocks. After discovering this it seems natural that a reverse flow would be opposite the forwards analysis, but this was not apparent when I initially wrote the algorithm.

Challenges: Working with a Monadic Intermediary Language presented me with the challenge of becoming familiar with the concept of Monads, and the way in which the Monad Laws relate to the typical applications of compiler optimizations. It was challenging to navigate through the code initially, but I found that working with the Sweet code directly allowed me to concentrate only on the code flow, and not be distracted by jumping from file to file modifying or adding new functions. The use of a new temporary for each operation as well as the listing of needed parameters on each block presented the challenge of understanding how to achieve an intersection from the OUT sets of two blocks to accurately be able to test the available and anticipated expressions functions.

Overall I feel that I have learned quite a bit about how algorithmically compilers can perform optimizations across block boundaries.
The next task will be to clear up the remaining bug of renaming the variables used in the sets for after they enter (for a forward analysis) or leave (reverse) the start of a block. Once that is complete, the Postponable expressions pass should be a straightforward implementation, by writing the code for set difference, and applying the same pattern of a forward dataflow analysis as the available expressions. Finishing up with the Used Expressions pass with a similar approach to the reverse flow of Anticipated Expressions.