**CS 301** Spring 2016
**Project Phase 4**
April 15 – April 29

# IC **Compiler**
# Data-Flow Analysis & Optimization

## Plan

The last phase is here! In this assignment, you will extend your compiler to support a general data-flow analysis framework and several optimizations implemented in this framework. It is also time to overload some favorite acronyms from the beginning of the course. DFA now means Data-Flow Analysis and CFG means Control-Flow Graph. We will get you a good head start on design in meetings Tuesday April 19.

## Implementation Details

This section describes high-level implementation tasks. Some concrete code structure is described in a following section.

**Control-flow Graph.** Build a control-flow graph representation of the TAC for each method. Design a package that builds a CFG for a method's `TACList`. I suggest a simple representation where each CFG node is a single instruction. I provide a few classes. You will mainly need to write a transformation from `TACList` to `ControlFlowGraph` (and the reverse).

**Data-Flow Analysis Framework.** Implement a data-flow analysis framework with a general data-flow engine for forward and backward analyses. Then, implement indiviudal data-flow analyses using this engine. Each analysis simply describes its lattice and transfer functions. The engine should use the iterative algorithm we explored to solve the given data-flow equations on a CFG and provide access to the resulting IN and OUT values on each basic block. A later section in this document describes suggested code structure.

**Warm-Ups.** Use the provided *Reachable Analysis* to:

- Convert the provided *Unreachable Code Elimination* optimization to print a warning when unreachable code is encountered.
- Report an error if the end of a non-void method is reachable without first executing a `return` statement. (See the complementary *Unreachable Code Elimination* optimization for reference.)

**Analyses.** Implement the following analyses using your framework:

1. *Live Variables Analysis*: compute the variables that may be live at each program point.
2. *Reaching Copies*: compute the definitions generated by *Copy* instructions that always reach each program point.
3. At least one of:
   - *Constant folding analysis*: determine variables whose values are constant at each program point.
   - *Available expressions*: compute the expressions available at each program point. (If you implement this analysis, please see the note at the end of this document about case classes and equality.)

**Optimizations.** Use the results of the analyses you implemented to perform the following optimizations:

1. *Dead Code Elimination (DCE)*: Removes code that updates variables whose values are not used in any executions. This optimization will use the results from the live variable analysis.
2. *Copy Propagation (CP)*: Use the results from the reaching copies analysis to perform copy propagation.
3. Depending on which analyses you implemented, at least one of:
   - *Constant Folding (CF)*: Uses the results from constant folding analysis and replaces each constant expression with the computed constant. (The Dragon book describes this optimization in detail.)
   - *Common Subexpression Elimination (CSE)*: Reuses expressions already computed in the program. Here you will use the information computed with the Available Expressions analysis. If an expression is available before an instruction that re-computes that expression, replace the computation by the variable holding the result of that expression. If there is no such variable, create a temporary variable to store the result of the expression at the site where the expression is initially computed.

     Common subexpression elimination should also remove redundant run-time checks such as array bounds checks or null pointer checks. (Earlier, we may cast redundant null-check elimination as a distinct analysis and optimization. This works, but CSE can consider null checks or array bounds checks as "expressions" that can be removed if always "available" and provide a general way to eliminate some such checks.)

     Note that your CSE implementation will find only syntactically equivalent expressions.

To improve the effectiveness of your optimizations, try composing them in different orders and repeating the composed analyses. Each of these individual optimizations often creates opportunities for the others. (In the TINY compiler, we even computed a fixed point over CF, CP, and DCE. You could think or experiment to determine whether this is reasonable here.)

**Command Line Invocation.** In addition to all of the options from the previous assignments, your compiler should support the following command-line options:

- Option `-dce` for dead code elimination;
- Option `-cfo` for constant folding;
- Option `-cse` for common subexpression elimination;
- Option `-cpp` for copy propagaion;
- Option `-opt` to perform all optimizations once.
- Option `-printDFA` to print the data-flow facts computed for each program point.

The compiler will perform the optimizations in the order they occur on the command line. The above arguments may appear multiple times in the same command line — in that case the compiler will execute them multiple times, *in the specified order*. The compiler should only perform the analyses necessary for the optimizations requested.

When the `-printIR` also occurs on the command line, you must print the TAC before or after optimizations, depending on where it occurs in the command line. For instance, with: `-cfo -printIR -dce`, you must print the TAC after the compiler performs constant folding, but before it removes dead code.

Your compiler must also print out the computed data-flow information when supplied with the command line option `-printDFA`. Specifically, the compiler should print the data-flow information at each point in the program for each analysis implemented. *Make sure your output is readable.* Each data-flow fact must clearly indicate the program statement it references, and whether it represents the information before or after the statement.

If you have not already, you may find it cleanest to convert your command line option handling to something like the following (or a nicer functional equivalent using *fold left*):

```
  var file: Option[String] = None
  var printAST = false
  var printSymTab = false
  var printTAC = false
  for (a <- args) {
    a match {
      case "-d" => Util.debug = true
      case "-printAST" => printAST = true
      case "-printSymTab" => printSymTab = true
      case "-printIR" => printTAC = true
      // ... more flags ...
      case x => file match {
        case None => file = Some(x)
        case Some(_) =>
          println("Unrecognized option or second file: " + x)
          System.exit(2)
      }
    }
  }
```

## Code Structure

You should extend your code base with three additional packages:

- `cfg`: Code to represent and build a CFG for a `TACList`.
- `dfa`: Code for the general data-flow framework, as well as the specific instances necessary for this assignment and any supporting classes.
- `opt`: Code to implement the optimizations listed above.

I provide a few optional starter files to help with organization. Feel free to use none, any, or all.

**Basic Blocks and Control Flow Graphs.** These two classes in `ic.cfg` represent one basic block and a control flow graph. Each basic block is restricted to one instruction, which is sufficient for this assignment. (Larger basic blocks avoid some computation and space overhead and facilitates more sophisticated instruction selection schemes that operate on the block level, but they are not necessary for the analyses we are performing.) When constructing the CFG, don't forget to include dummy nodes for `enter` and `exit`. (These can hold a `TACComment` instruction, a `TACNoOp` instruction, etc.). You should not need to modify these two classes, although you are free to do so if you wish.

In addition to `toString`, the `ControlFlowGraph` supports generating dot files to show the control flow graph with the `dotToFile` method. If you generate `a.dot`, the following commands will convert it to a PDF showing the graph: `dot -Tpdf < a.dot > a.pdf`

**Data Flow Analysis.** This section proposes an object-oriented representation as a convenient way to package the pieces of the general data-flow analysis and specific instances. A functional approach works just as well (a function that takes one argument for each of the abstract members below, solves, and returns a representation of the IN/OUT values).

A general class for solving data-flow instances in `ic.dfa` will be the superclass of all analysis instances:

```
abstract class DataFlowAnalysis[T](val cfg: ControlFlowGraph) {
    def solve(): Unit = { ... }
    def in(b: BasicBlock): T = { ... }
    def out(b: BasicBlock): T = { ... }
```

```
    // return true iff the analysis is a forward analysis
    def isForward(): Boolean

    // initial value for out[enter] or in[exit], depending on direction.
    def boundary(): T

    // Top value in the lattice of T elements.
    def top(): T

    // Return the meet of t1 and t2 in the lattice.
    def meet(t1: T, t2: T): T

    // Return true if t1 and t2 are equivalent.
    def equals(t1: T, t2: T): Boolean

    // Return the result of applying the transfer function for instr to t.
    def transfer(instr: TInstr, t: T): T
}
```

This class is parameterized by `T`, the type of value contained in the lattice. The `solve` method computes the solution for the CFG passed into the constructor. After calling `solve`, the `in` and `out` methods can be used to access the data-flow facts for each basic block. To use the framework, extend this class with a new class (*e.g.*, `LiveVariableAnalysis`) that defines the six abstract methods describing the lattice, transfer functions, meet operator, boundary value, and direction of the analysis.

The starter code contains `ReachableAnalysis`, a simple example analysis that determines which TAC instructions are unreachable (because there are return statements on all paths leading to them). Use this analysis to help debug your code or get an idea of how to structure other analyses. You can often implement the transfer functions using a large pattern match.

`DataFlowAnalysis.scala` contains several debugging statements that can be enabled with the command line flag `-d`. Comment them out or replace them as you see fit.

**Make reasonable design choices** about how to represent data-flow facts. Feel free to use `scala.collection` classes wherever possible. *Clarity of design and ease of implementation should be the primary motivation for any initial design choice.*

**Optimization.** As with the data-flow analysis, this section proposes an object-oriented representation – a functional representation works at least as well.

This class in `opt` is the superclass for all optimizations:

```
abstract class Optimization {
    // apply the optimization to each method in p.
    def optimize(p: Program): Unit = { ... }

    // apply the optimization to the method md.
    def optimize(md: MethodDecl): Unit
}
```

To create an optimization, create a subclass of `Optimization` and define `optimize(md)` to compute data-flow information and perform the optimization on `md`. That method should replace the method's TAC list with the optimized version. You can either provide methods in your `TACList` class to modify an existing list, or you can construct a new list to replace the old one. For reference, I provide a simple `UnreachableCodeElimination` optimization that uses the `ReachableAnalysis` to eliminate unreachable code.

The `Optimization` methods return `Unit`, but you can change them to return a `Boolean` indicating whether the TAC changed. While not required, this allows you to provide a `-iter` command-line option that iteratively applies all optimizations over and over again until no additional changes happen (or a fixed upper bound on the number of iterations is reached).

## Schedule

**Friday, April 22:** Your compiler should generate the Control Flow Graph for a `TACList` and support the `-printDFA` option for at least one data-flow analysis. (Reaching Copies and Live Variables may be the most straight forward, followed by Constant Folding, and then Available Expressions.) Include a brief status update in the writeup directory that indicates which test programs to use to verify the correctness of your analyses.

**Friday, April 29:** Full optimizer due. Your compiler should support the command line options listed above for the optimizations you have implemented.

Include a brief writeup in the writeup directory to describe any important details about the data-flow and optimization passes, a summary of your testing methodology, and any known bugs.

In your write up, demonstrate your analyses and how your compiler performs optimizations on several small representative IC programs. (That is, show me a few small IC programs, and both their unoptimized and optimized TAC.) Do you see any performance improvement? There may be other limitations of your back end preventing the optimizations from making a huge difference. What factors may be limiting how efficient the code is in this regard?

## Extensions

*Many* more optimizations are possible in your framework. If you want to try others, consider Partial Redundancy Elimination, Loop Invariant Code Motion, or any other analysis from our discussions. These would all make excellent additions.

## Scala Case Classes, Equality, and Hashing.

Keep in mind that, by default, Scala uses structural equality for == tests on case classes. That is, two distinct objects of a case class will be considered equal if all of their fields are equal. Thus, the following prints true, even though x and y are distinct objects.

```
case class TACNot(dst: TACOperand, src: TACOperand) extends TACInstr { }

val x = TACNot(t1, t2)
val y = TACNot(t1, t2)

println(x == y) // prints true.
```

That means that if you are storing TAC instructions in a set, searching for them in a list, or creating a map where the domain is TAC instructions, you may not be able to distinguish between two occurences of the same instruction. So, if the TAC instruction for "t1 = !t2" appears in multiple places in a TAC list, the standard Scala set. map, and list implementations will consider them all equal to each other.

This will potentially be an issue for analyses like CSE, for which you may choose to construct sets of TAC instructions that compute the same expression. In such a case, you will likely want to use reference equality so that you can keep track of different occurences of the same instruction. To switch to reference equality, add the following to your TACInstr base class:

```
abstract class TACInstr {
  override def equals(o: Any) = {
    o.isinstanceOf[AnyRef] && this eq o.asInstanceOf[AnyRef]
  }
}
```

The function eq is a pointer equality test. With this extension, the above test "x == y" yields false.

Also, objects considered equal *must always have the same hash code.* Whenever you add your own .equals method, you must ensure that hashCode is properly defined. For example, the following class ignores the y field in the equals method:

```
case class Point(x: Int, y: Int) {
  override def equals(o: Any) = {
    o.isinstanceOf[Point] && (x == o.asInstanceOf[Point].x)
  }
}

val x = Point(1,2)
val y = Point(1,3)
val set = new HashSet(x,y)
```

The set set will contain both points, however, even though the are considered equal, because they hash to different values. So, you must add a hashCode method to reflect the notion of equality you are using:

```
case class Point(x: Int, y: Int) {
  override def equals(o: Any) = {
    o.isinstanceOf[Point] && (x == o.asInstanceOf[Point].x)
  }
  override def hashCode() = { x }
}

val x = Point(1,2)
val y = Point(1,3)
val set = new HashSet(x,y)
```