Objectives to be covered on the final by topic

1. Semantic Analysis (Cooper, Ch. 4)
   (a) abstract syntax tree (AST)
      i. know what an abstract syntax tree is
      ii. know how it is different from a parse tree
      iii. given a program in a language with known syntax and semantics, be able to create a reasonable AST for it
   (b) symbol table
      i. know what a symbol table is
      ii. know that it maps symbolic identifiers to information about that identifier (e.g. type)
      iii. be able to construct a symbol table for a TL07 program
   (c) type checking
      i. know what types are and that they serve as a conservative approximation of different values variables and expressions can take at runtime
      ii. know how types can be checked for a simple language using a post-order traversal of an AST

2. Code Generation (Cooper Ch. 7)
   (a) 3-address code
      i. know what 3-address code is, and why it is called that
      ii. be able to understand and write programs in ILOC
      iii. be able to translate programs from a high-level language like TL07 into ILOC
      iv. be able to simulate the execution of ILOC programs
   (b) virtual registers
      i. know what virtual registers are and why they are used
      ii. given a program in a familiar language (C++ or Java), be able to what values can be stored in virtual registers
   (c) tree-walk code generation
      i. be able to explain how a tree walk code generator works
      ii. be able to recognize the code for a tree walk code generator
      iii. given a TL07 program in AST, be able to manually translate it into ILOC
      iv. given the semantics for a simple programming language, be able to write a tree-walk code generator that produces ILOC code

3. Program Analysis (Cooper Ch. 9)
   (a) control flow graphs and dominator trees
      i. know what a control flow graph (CFG) is and what the nodes and edges represent
      ii. be able to identify the successors and predecessors of a given node
      iii. know what a path through the control flow graph is, and what it signifies
      iv. given a program in 3-address code, be able to draw its control flow graph (with or without the nodes labelled by instructions)
v. be able to recognize loops in control flow graphs
vi. be able to identify, if statements in control flow graphs
vii. know what the dominator relationship signifies
viii. given a simple control flow graph, be able to identify a node’s successors, predecessors, dominators, proper dominators, and immediate dominator
ix. given a simple control flow graph, be able to derive its dominator tree
x. given a control flow graph, be able write a program that has that control flow graph
xi. given a dominator tree, be able to identify a node’s dominators, proper dominators, and immediate dominator

(b) iterative dataflow analysis
i. know what iterative dataflow analysis is
ii. given a program along with control and transfer functions describing an iterative dataflow analysis, be able to manually perform the analysis
iii. be able to recognize and give examples of common analyses: (live variables, available expressions, very busy expressions, and constant propagation)

(c) static single assignment form (SSA)
i. know what static single assignment form is
ii. understand how having SSA’s single assignment property makes dataflow problems and optimizations simpler (avoids fixed-point algorithms, has less redundancy)
iii. know what $\phi$-functions are:
   • $\phi$ is the Greek letter phi
   • why $\phi$-functions are needed
   • that they go in “join nodes”
   • that each operand corresponds to a given CFG-predecessor of the join node
   • that all the phi-functions in a given block ‘execute’ simultaneously
iv. know (informally) what a dominance frontier is, and how it relates to $\phi$-function placement
v. given a program in ILOC, be able to rewrite it in static single assignment form
vi. given a program in SSA form, be able to rewrite it into non-SSA 3-address code (assuming no mutually-dependent phi-functions)
vii. given a simple program in SSA form, be able to simulate its execution

4. Machine-Independent Optimizations (Cooper Ch 8, 10)

(a) general
i. know that optimizations are transformations (or algorithms for transformations) that preserve semantics but hopefully improve performance
ii. know that optimizations usually aim to improve execution time but may also aim to improve performance with respect to other characteristics, such as reduce memory usage or power consumption
iii. know the general categories of machine-independent optimizations found in Cooper’s taxonomy (Eliminate Useless/Unreachable Code, Code Motion, Specialize, Enable Other Optimizations, Eliminate Redundancy):
   • be able to describe how the optimization in each category
   • be able to give an example of one optimization in each category
   • be able to recognize which category the optimizations listed in Figure 10.1 belong to
iv. understand why analysis needs to be conservative

(b) constant propagation (folding)
i. know what constant propagation is
ii. understand how constant propagation might speed a program up
iii. given a program, be able to carry out constant propagation and constant folding by hand
iv. know how it could be done with an iterative dataflow analysis

(c) useless code elimination
i. know what useless code elimination is
ii. know what the difference between useless and unreachable code elimination is
iii. know that both can be called dead-code elimination
iv. understand how useless code elimination might speed a program up
v. given a program, be able to carry out useless code elimination by hand

(d) global redundancy elimination
i. know what redundancy elimination is
ii. understand how redundancy elimination might speed a program up
iii. understand how global redundancy elimination splits the analysis between local analysis in basic blocks, and a global iterative data flow analysis
iv. be able to carry out global redundancy elimination by hand

(e) SSA-based value numbering
i. know that SSA-based value numbering is a type of redundancy elimination
ii. understand how SSA-based value numbering works
iii. be able to carry out SSA-based value numbering by hand

5. Machine-Dependent Optimizations(cooper, Ch 11, 12, 13)

(a) instruction selection
i. know that instruction selection chooses the specific machine operations to carryout
ii. be able to describe, at a high-level, how tree walk, peephole, and tree pattern matching work
iii. be able to describe the major phases of a modern systematic peephole optimizer (expand, simplify, match)
iv. know what the window is and be able to describe its role in peephole optimization
v. given a description of the patterns to match and replace, be able to manually perform a peephole optimization

(b) instruction scheduling
i. understand how instruction scheduling is important to mask latencies and make fuller use of hardware resources
ii. understand that scheduling is most important on VLIW-type processors, but can have an impact even on CISC processors with out-of-order execution
iii. given a specification of operation latencies and constraints on how many instructions can be issued per cycle, be able to manually schedule instructions using list scheduling

(c) register allocation, general
i. know what register allocation (in the broad sense) is,
ii. be able to explain the difference between the sub-problems of register assignment and register allocation (in the narrow sense)
iii. know that both can be NP-complete depending on the specific constraints, but allocation (spilling) is harder than assignment (coloring)
iv. be able to explain the difference between bottom-up and top-down register allocation
v. know what a live range is
vi. know what live range splitting is
vii. know what live range spilling is
viii. know what coalescing is
(d) Chaitin-Briggs bottom-up graph coloring
   i. be able to manually partition the definitions and uses of a variable (or virtual register) into minimally-sized live ranges
   ii. given a program in ILOC, be able draw an interference graph (assuming each virtual register represents a single live range)
   iii. know what the ‘colors’ in the interference graph represent
   iv. know what kinds of spill cost metrics can be used and to incorporate into the register allocation problem
   v. given an interference graph, be able to manually simulate the Chaitin-Briggs coloring algorithm
   vi. understand the trade-off involved in reserving or not reserving spill registers

6. PL History / Functional Programming (Scott Ch. 10)
   (a) History
      i. know that the Turing Machine, Lambda Calculus, and general purpose programming languages can all compute the same set of functions
      ii. know that four major programming language paradigms are: procedural, object-oriented, functional, and logical.
      iii. know that functional programming languages get their inspiration from lambda calculus
      iv. recognize Fortran, Algol, C, and Pascal as procedural imperative languages
      v. recognize Lisp, Scheme, ML, and Haskell as functional programming languages
      vi. recognize Simula ’67, SmallTalk, C++, Java, and C# as object oriented programming languages
      vii. know that Lisp is a dynamic language whereas ML has statically inferred types
   (b) Functional Programming Concepts
      i. know that functional programming avoids assignment and looping
      ii. know that functional programming relies heaving on recursion and parameter passing
      iii. know features commonly found in functional programming languages:
         A. first-class function values and higher-order functions
         B. extensive polymorphism
         C. list types and operators
         D. recursion
         E. structured function returns
         F. constructors (aggregates) for structured objects
         G. garbage collection

bold denotes most important objective, expected to be satisfied
italics denotes objectives, expected of ‘A’ students
normal font denotes tertiary objectives that are less important but also desirable