B1. Compilers (25 points) Answer all questions.

1. (10 points) Build the LR(0) DFA for the following grammar:

   S -> E $
   E -> id
   E -> id (E)
   E -> E + id

2. (3 points) Is this an LR(0) grammar? Give evidence.
3. (3 points) Is this an SLR grammar? Give evidence.
4. (3 points) Is this an LR(1) grammar? Give evidence.
5. (3 points) Use a simple example to show the effect left recursion in a grammar can have on LL(1) parsing. Explain your example and the effect. What can be done to the grammar to eliminate the problem?
6. (3 points) Use a simple example (hopefully the same one for ease) to show the effect left recursion in a grammar can have on LR(1) parsing. Explain your example and the effect. What can be done to the grammar to eliminate the problem?

B2. Compilers (25 points) Answer all questions.

1. (6 points) Describe the actions of the type checker/inferencer for the following type rule:

$$\frac{\begin{array}{l} O,M \vdash e0 : T0 \\ O,M \vdash e1 : T1 \\ \ldots \\ O,M \vdash en : Tn \\ T0 \leq T \\ M(T,f) = (T1',\ldots,Tn',Tn+1') \\ Ti \leq Ti' \quad 1 \leq i \leq n \end{array}}{O,M \vdash e0 \ @\ T.f(e1,\ldots,en) : Tn+1'}$$

In this rule, the notation O denotes the symbol table environment, M denotes the method table, e denotes an expression, and T a type.

2. (6 points) Describe how code is generated for a dynamic dispatch. Describe how code is generated for a static dispatch. Discuss the advantages to replacing a dynamic dispatch by a static dispatch.

3. Assume that you are given a compiler for a statically scoped, object-oriented language with single inheritance and method overriding. Assume that the compiler is constructed with the standard separate phases for lexical analysis, parsing, semantic analysis, intermediate code generation, symbol table management, and code generation. The storage allocation mechanism is a stack of activation records and heap allocation when needed.

(a) (4 points) Describe how inheritance and method overriding affect the symbol table management.

(b) You have been asked to modify the compiler in several ways. For each modification listed below, (i) State the phases that would need to be modified. (ii) Describe briefly how the phases would have to be modified to incorporate the modification.

- (3 points) Add a CASE (or switch) statement, when the if-then and if-then-else statements already exist.
- (3 points) Change the symbol for exponentiation from "^" to "**", keeping the same semantics.
- (3 points) Add runtime checking of array reference out of bounds before every access to an array.

B3. Compilers (25 points) Answer all questions.

1. Efficient utilization of registers is important in generating good code. A common method for register allocation is to formulate the problem as a *graph coloring* problem. The concept of *live variables* is used to build an interference graph that represents the interferences between symbolic registers.

   a. (6 points) Compute the set of live variables after each statement in the following code segment. Show the set of variables that are live *after* the statement, beside each statement. The notation load[v+c] denotes loading the value at address of variable v + some constant c.

   ```
   live in: k, j
   g = load [j+8]
   h = k – 2
   f = g * h
   e = load [j+12]
   m = load [j+16]
   b = load [f]
   c = e + 8
   d = c
   k = m + 4
   j = b
   live out: d, k, j
   ```

   b. (6 points) Draw the corresponding interference graph for the code segment.

   c. (8 points) Briefly describe the role of colors, graph coloring, and the overall standard (Chaitin-style) algorithm for register allocation based on graph coloring. Be sure to explain when spill code is determined to be needed, when it is inserted, and how this process fits into the overall algorithm.

2. (5 points) Briefly discuss the relative costs and tradeoffs of the reference counting and mark-sweep garbage collection methods. Be sure to justify why you think there have been so many more garbage collection methods developed since these two methods.

B4. Compilers (25 points) Answer all questions.

A *dominator* of a node *b* in a control flow graph is any node *a* such that all paths from the *entry* node to *b* pass through *a*. By definition, a node dominates itself.

1. (7 points) Construct a table that shows the dominators for each node in the control flow graph below. Be sure to include the entry and exit nodes in your table.
2. (6 points) The set of dominators of each node n of a control flow graph can be computed by formulating the computation as a data flow problem. Write a data flow equation for computing the dominator relation over a control flow graph.
3. (5 points) State and justify whether the problem is *forward* or *backward*, what the *transfer function* for a node is, and what the *confluence operator* is.
4. (3 points) Describe one use for dominator information in optimizing compilers. Be sure to demonstrate how the dominator information is used for this purpose.
5. (4 points) Justify or argue against the claim "There is no one best standard order for the optimization phases of an optimizing compiler."