The solution to Homework #11 carried out the memory mapping subtask of code generation, and a part of the code shaping subtask. It builds a “shaped tree” that represents only the executable part of the program. All operator identification has been done, and nodes have been introduced wherever explicit type conversion of a value is required. Answer the numbered questions given below, and complete the code shaping subtask by producing a sequence of abstract operations, expressed as tuples in the style of Figure 13.1 of the text, that define the VAX actions implementing the source program.

Directory “556/hw12drv” contains supporting material for this assignment:

- **minilax.ala**: An ALADIN specification for the attribution to specify jump cascades (see Figure 10.6 of the text), sequencing (discussed below) and selection of the proper code for multiplication (see the solution to Homework #3).
- **optout**: The result of an analysis of minilax.c.
- **visit.c**: A recursive C routine mechanically derived from optout to walk the shaped tree according to the specified visit sequence. This routine simply walks the tree; all relevant attribute computation must be added to it.
- **graph.h, graph.c**: Tuple output routines to verify your implementation.

Each tuple is output as a side effect of evaluating a function, and therefore the order in which these “computation graph” functions is invoked determines the sequence of operations. The *pred* and *done* attributes are included to partially fix the order in which the computation graph generation functions are invoked. This is the only way in which an ALADIN specification can specify sequencing — by expressing it as functional dependence.

1) Carefully explain the computation order specified by rule A_2_5, and why that order is desirable. Verify that the order of tuple generation routine invocations is correct in file “optout”.

2) No computation order is fix ed by *pred* and *done* in rules A_4_3_5 and A_4_3_7. Which order was chosen by GAG and incorporated into the visit sequences?

3) Define attribution to determine a computation order for rules A_4_3_5 and A_4_3_7 that minimizes the number of registers used (see Section 10.2.1 of the text).

4) Show that the attribution you define in (3) cannot be carried out during the visit sequence described by optout.c. Explain how you will avoid this problem in your
implementation.

5) Show that it is impossible to produce the tuples in the proper order using a program generated by GAG from an ALADIN specification, unless you retain the list of tuples in memory until a complete expression subtree has been attributed. (Hint: Remember that GAG requires a partitioned attribute grammar, and think about the properties of such grammars.) How can you avoid retaining this list when coding by hand?

The purposes of this assignment are to familiarize you with construction of a computation graph from a shaped tree. It is worth 20 points, and is due at the beginning of the lecture on April 15.