1. Language Evaluation. The C language makes use of pointers. Discuss briefly the benefits and costs of pointers in C.

2. Scoping and parameter passing. What is the output of the following program, assuming lexical scoping and

(a) pass by value?
(b) pass by reference?
(c) pass by value-result?
(d) pass by name

3. Syntax and Semantics. Identify the context-free, context-sensitive, and semantic errors in the following C code.

```c
main()
{
    int a, b,
    i = 2;
    a = i * b;
}
```


(a) Using the grammar given below, construct a parse tree for the string

\[ 7 + 3 \times 4/6 - 1. \]

(b) What is the value of the expression above?

(c) Modify the grammar to make multiplication and division right-associative and of lower precedence than addition and subtraction.

(d) Using the modified grammar, construct a parse tree for the string

\[ 7 + 3 \times 4/6 - 1. \]

(e) What is the value of the expression now?
5. Abstraction. Build a complete algebraic specification (operations, variables, and axioms) for the following data type:

**Linear Lists with Predecessors.** Lists are often manipulated with functions *head* (first item in the list) and *tail* (rest of the list). Then head(tail(L)) is a reference to the item that succeeds, or follows, the item pointed to by L. Sometimes it makes sense to talk about the *preceding* item in a list.

A *pred-list* is empty, or it consists of

- a data item of type T (first item pointed to);
- a pred-list that contains the succeeding items in the list (tail); and
- a pointer to the item preceding x (pred).

(Note: such a list is easily implemented as a doubly-linked list.)

In the example below, M points to a pred-list. Item 7 is the first item, N points to the tail of the list, and L points to the pred of the list.

The operations that need to be performed are the following:

(a) **create** creates pointer to an empty pred-list
(b) **isempty** takes a pointer to a pred-list as an argument and returns true if the pred-list is empty and false otherwise.
(c) **first-item** takes a pointer to a pred-list as an argument and returns the first item in the list. Invoking first-item on an empty list causes an error.
(d) **tail** takes a pointer to a pred-list as an argument and returns the tail of a pred-list.
(e) **pred** takes a pointer to a pred-list as an argument and returns a pointer to the pred-list whose first item is the predecessor of L. In the above example, pred(M) = L.
(f) **add-pred** takes a pointer to a pred-list L and an item x as arguments and inserts x immediately in front of L. The (unchanged) pointer to L is returned. For example, add-pred(M, 4) results in

(g) **add-tail** takes a pointer to a pred-list L and an item x as arguments and inserts x as the head of the tail of L. The (unchanged) pointer to L is returned. For example, add-tail(M, 3) results in
6. Attribute grammars. A description of an attribute grammar for a subset of Wren is given below. Using the 
attribute grammar, construct and decorate a parse tree for the expression

\[ a + b \ast 5 \]

Attribute grammars can be used to define the semantics of a language in terms of another (simpler) language. 
This application of attribute grammars is called translational semantics. A sequence of assembly-like instructions 
is associated with each non-terminal symbol, eventually producing a translation of the entire program 
into assembly code. The translation will consist of a list of assembly-language instructions.

The assembly language corresponds to a very simple computer with one accumulator and memory words that 
hold integers. Instructions have zero or one operand; a second operand, if needed, is assumed to be in the 
accumulator. The opcodes are LOAD, STORE, ADD, SUB, MULT, DIV. The operator LOAD puts variables 
into the accumulator and STORE moves the contents of the accumulator into a variable.

The BNF, assembly language, attributes, attribute grammar, and auxiliary functions are given below.