Instructions

Answer question #1 and any three others. (If you answer more, only the first four will count.) Point values are as indicated. Please try to make your answers neat and coherent. Remember, if we can’t read it, it’s wrong. Partial credit will be given, so try to put something down for each question (a blank answer always gets 0 points!).

1. (1 point)
   Delaying the computation of an expression until it is needed is called:
   (a) Lazy evaluation
   (b) Lax evaluation
   (c) Looney evaluation
   (d) Lopsided evaluation

2. (a) (25 points)
   Recall that the subsets function, called as (subsets L) can produce a very large output list, even for rather small input lists. In particular, if input list L has n elements, subsets will compute an output list whose length is $2^n$.

   Given the large output that might be computed, it might be useful to produce a version of subsets, called lazy-subsets, that produces a lazy list as its output. Recall that the empty list is represented as the pair (#f . #f). A list of length one, (a ), is represented as (a . #f).

   A list of length two or more is represented by the pair (a . fct) where a is the first value in the list (the car of the list) and fct is a suspension function (of no arguments). When called, fct will compute the remainder of the list, in lazy list form.

   Give a Scheme implementation of (lazy-subsets L). In keeping with the intent of lazy lists, your implementation should compute list members incrementally, as suspension functions are called. It should not compute the whole result in advance, and then “dribble” values out, one by one.

   (b) (8 points)
   An alternative to returning a lazy list is to make subsets a co-routine. Then it could compute one subset, return it and suspend itself. When resumed, it could compute one more subset and suspend again, etc. What are the advantages of using the co-routine approach? What are the disadvantages?
3. (a) (20 points)
   Write a Scheme function \((\text{partition } \text{pivot } \text{list})\) where \(\text{pivot}\) is an integer and \(\text{list}\) is a list of integers. \(\text{partition}\) partitions \(\text{list}\) into three sublists. The first sublist contains all the values in \(\text{list}\) that are less than \(\text{pivot}\); the second sublist contains all the values in \(\text{list}\) that are equal to \(\text{pivot}\); the third sublist contains all the values in \(\text{list}\) that are greater than \(\text{pivot}\). For example, \((\text{partition 5 '}(1 3 5 9 11 0))\) returns the list \('((1 3 0) (5) (9 11))'\).

(b) (13 points)
   Explain how you could add parallelism to \(\text{partition}\) using the \(\text{pcall}\) or \(\text{future}\) constructs (or both). Explain why each \(\text{pcall}\) or \(\text{future}\) you added will allow significant parallel evaluation. (\textbf{Do not} add \(\text{pcall}\) or \(\text{future}\) operations where only trivial improvements will result).

4. (a) (18 points)
   Write a Scheme function \((\text{insert-all } \text{val } \text{list})\) where \(\text{val}\) is any scalar (atomic) value and \(\text{list}\) is a list of scalar (atomic) values. \(\text{insert-all}\) places \(\text{val}\) in all possible positions in \(\text{list}\). That is, if \(\text{list}\) contains \(n\) values, then \(\text{insert-all}\) creates a list of \(n+1\) sublists. Each sublist is \(\text{list}\), with \(\text{val}\) inserted into a distinct position. For example the call \((\text{insert-all '@ '} '(1 2 3))\) = \('((1 2 3 @) (1 2 @ 3) (1 @ 2 3) (@ 1 2 3))'\).

(b) (15 points)
   Write a Scheme function \((\text{perm } \text{L})\) that returns a list of sublists, each of which is a permutation of \(\text{L}\). For example, \((\text{perm '}(a b c))\) might return \('((a b c) (b a c) (b c a) (a c b) (c a b) (c b a))'\). The order in which permutations of \(\text{L}\) appear is unimportant, but if \(\text{L}\) contains \(n\) values, then \((\text{perm } \text{L})\) should return a list containing the \(n!\) permutations of \(\text{L}\). \((\text{perm } (\))\) should return \(()\). You may use \(\text{insert-all}\) in your implementation of \(\text{perm}\) if you wish.

5. (a) (13 points)
   Explain what \(\text{call/cc}\) (also known as \(\text{call-with-current-continuation}\)) does.

(b) (20 points)
   One of the advantages of using \(\text{call/cc}\) is that it is possible to return to a previous computation and re-execute it, possibly with different values or parameters. For example, consider the following piece of Scheme code, possibly nested deeply within a program:

   \[
   \begin{align*}
   \text{(let ( (option 'fast) )}
   \text{(if (equal? option 'fast)}
   \text{ (computation1)}
   \text{ (computation2)}
   \text{ ) )}
   \end{align*}
   \]

   This fragment first sets \(\text{option}\) to \(\text{fast}\), computing its answer using a call to \(\text{computation1}\), which is fast and usually accurate. The caller can then look at the answer returned and determine whether it is satisfactory. If it is not, he might wish to redo the computation using the \text{slow} option, executing \(\text{computation2}\). But how do we get back to the original execution context to try the other option? \(\text{call/cc}\) can allow us to do this.

   Modify the code fragment shown above so that when the \text{fast} option is initially tried, the user gets the result of \(\text{computation1}\) \textbf{plus} a continuation, \(k\). If \(k\) is called with the parameter \text{slow}, the code fragment is reactivated and this time \(\text{computation2}\) is chosen. All your changes should be local to the code shown above so that this restructuring could be used in any program that contains this fragment.
6. (a) (8 points)
   Explain the difference between the **scope** of an identifier and the **lifetime** of a data value.

(b) (8 points)
Explain the difference between *static* typing and *dynamic* typing. Give an example that illustrates the difference between static and dynamic typing.

(c) (8 points)
Explain the difference between overloading and polymorphism. Give an example of a programming language that supports overloading. Give an example of a programming language that allows polymorphism. What are the advantages and disadvantages of each?

(d) (9 points)
Explain the difference between virtual methods and ordinary methods in a language like C++ or Java. Why are virtual methods needed in these languages?