1. (1 point)
   In Scheme the \texttt{list?} predicate tests if a value is a:
   (a) vector
   (b) string
   (c) rational
   (d) list

2. (a) (13 points)
   Most programming languages, including C, C++ and Java, support the notion of nested visibility. That is, within a series of nested namescopes, the nearest (innermost) definition of an identifier is always used, even if the identifier is also defined in enclosing scopes.

   What are the advantages of supporting nested visibility in a language definition? Are there any disadvantages? If so, how could such disadvantages be remedied?

   (b) (20 points)
   In implementing nested visibility, the nearest definition of an identifier is always used, even if that definition leads to type errors. Thus if we see an array reference \texttt{a[1]}, we use the nearest definition of \texttt{a}, even if that definition specifies that \texttt{a} is something other than an array.

   What if we modify the nested scoping rule so that the nearest definition that is valid is used? Thus we “skip” declarations that can’t be right, and look for a definition that makes sense in the context in which the identifier is used. Are there any drawbacks to this modified definition of nested scoping? Does it ever lead to any ambiguities?
3. (a) (20 points)
In Scheme association lists are simply lists of key/value pairs. For example,

```
((a 10) (b 20) (c 30))
```

To access data, a lookup function like `assoc` matches a key against the car of each pair. If a
match is found, the matching pair is returned; otherwise `#f` is returned.
If we are using association lists in Multilisp, the linear structure of the list does not allow the
effective use of `pcall` or `future` to speed lookup in long lists.
Suggest a new representation for association lists that allows Multilisp to use `pcall` or
`future` effectively in lookups. Note that keys in association lists need not have `<` defined for
them, so binary searches or binary search trees won’t always work. Your new representation
may use ordinary lists for small association lists, but for large ones, a more complex structure
will be needed.
Show the code for `(lookup key new-assoc-list)` which will act like `assoc`, finding a
pair whose key values matches `key`, or returning `#f` if no matching pair is found. Your imple-
mentation should use `pcall` or `future` to speed lookup in sufficiently long association lists.

(b) (13 points)
Explain how you would implement `(enter pair new-assoc-list)` where `pair` is the
usual key/value pair and `new-assoc-list` is your new representation of association lists.
Your implementation need not use `pcall` or `future` (though it may if you wish), but it should
preserve the structure needed to make lookup efficient. You may assume that a pair with the
same key value as `pair` is not already in `new-assoc-list`.

4. (a) (13 points)
Explain what `call/cc` (also known as `call-with-current-continuation`) does.

(b) (20 points)
Consider the following Scheme code:

```
(define (top v)
  (middle (cons v (call/cc (lambda (k) k)))))
)
(define (middle w)
  (bottom (cons w (call/cc (lambda (ko) ko)))))
)
(define (bottom x)
  (if (number? (cdr (car x)))
    (if (number? (cdr x))
      x
      ((cdr x) 2))
    ((cdr (car x)) 3))
)
```

What value does the call `(top 1)` return? Explain why.
5. (33 points)
Most programming languages, including Scheme, pass parameters **positionally**. That is, the first value after the function is the first parameter, the next value is the second parameter, etc. For long parameter lists this approach is tedious and error-prone. It is easy to forget the exact order in which parameters must go. Moreover, default values for non-critical parameters are difficult to provide.

An alternative to positional parameters is **keyword parameters**. Now each parameter value is labeled with the name of the parameter it represents. The order in which parameters are passed is unimportant, and unnamed parameters may assume default values. For example, assume \( F \) is a Scheme function that has five parameters, \( a \) to \( e \). In keyword form \( (F ' (a 10) ' (b 20)) \) and \( (F ' (b 20) ' (a 10)) \) would both have the same effect, setting \( a \) to 10 and \( b \) to 20, with the remaining parameters assuming default values.

Assume we have function \( F \), with parameters \( a \) to \( e \), already coded in standard positional form. Explain how to modify \( F \) so that it now takes its parameters in keyword form. (Note that \( \text{(lambda L (body))} \) defines a function that takes a variable number of parameters, all bound to the list \( L \).)

6. (a) (10 points)
Write a Scheme function \( \text{(factor list)} \) where \( list \) is a list of two or more values. \( \text{factor} \) returns a list containing three values: the head of \( list \), the very last element of \( list \), and a list contains all the values of \( list \) with the very first and very last elements removed.
For example \( \text{(factor ' (a b c d e f))} = (a f (b c d e)) \).

(b) (23 points)
Write a Scheme function \( \text{(middle L)} \) that groups the elements of list \( L \) into sublists, starting at the middle. That is, the middle one or two elements of \( L \) are grouped into a sublist. Then the two elements immediately adjacent to the middle elements are grouped into a sublist. Then the next two adjacent elements are grouped with the middle list, etc., until all elements of \( L \) are included.
For example,
\( \text{(middle ())} \) returns ()
\( \text{(middle ' (a))} \) returns (a)
\( \text{(middle ' (a b))} \) returns (a b)
\( \text{(middle ' (a b c))} \) returns (a (b c))
\( \text{(middle ' (a b c d))} \) returns (a (b c) d)
\( \text{(middle ' (a b c d e))} \) returns (a (b (c d) e))
You may use \( \text{factor} \) in your implementation of \( \text{middle} \) if you wish.