CS 538

Midterm Exam

Thursday, November 5, 1998

11:00 AM— 1:00 PM

Instructions

Answer any *four* questions. (If you answer more, only the first four will count.) Each question is worth 25 points. 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. (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 of a programming language that uses static typing. Give an example of a programming language that uses dynamic typing.

   (c) (9 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.

2. (a) (15 points)

   Write a Scheme function `(partition pivot list)` where `pivot` is an integer value and `list` is a list of integers. `partition` partitions `list` into two sublists. The first sublist contains all the values in `list` that are less than or equal to `pivot`; the second sublist contains all the values in `list` that are greater than `pivot`. For example, `(partition 5 '(1 3 5 9 11 0)) = ((1 3 5 0) (9 11))`.

   (b) (10 points)

   Explain how you could add parallelism to `partition` using the `pcall` or `future` constructs (or both). Explain why each `pcall` or `future` you added will allow significant parallel evaluation. (**Do not** add `pcall` or `future` operations where only trivial improvements will result).

3. (a) (10 points)
   Write a Scheme function `(insert-all val list)` where `val` is any scalar (atomic) value and `list` is a list of scalar (atomic) values. `insert-all` places `val` in all possible positions in `list`. That is, if `list` contains n values, then `insert-all` creates a list of n+1 sublists. Each sublist is `list`, with `val` inserted into a distinct position. For example `(insert-all '@ '(1 2 3)) = ((1 2 3 @) (1 2 @ 3) (1 @ 2 3) (@ 1 2 3))`. `(insert-all val ()) = ((val))`.

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

4. (a) (10 points)
   Write a Scheme function `(factor list)` where `list` is a list of two or more values. `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 `(factor '(a b c d e f)) = (a f (b c d e))`.

   (b) (15 points)
   Write a Scheme function `(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,
   `(middle ())` returns `()`,
   `(middle '(a))` returns `(a)`,
   `(middle '(a b)` returns `(a b)`,
   `(middle '(a b c))` returns `(a (b) c)`,
   `(middle '(a b c d))` returns `(a (b c) d)`,
   `(middle '(a b c d e))` returns `(a (b (c) d) e)`.
   You may use `factor` in your implementation of `middle` if you wish.

5. 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 key-

word 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 `(lambda L (body))` defines a function that takes a variable number of parameters, all bound to the list `L`.)

6. (a) (10 points)

   Recall that `pcall` instructs Scheme to evaluate the arguments of a call in parallel. Give an example of a call which may not be optimized using `pcall` (because parallel evaluation of the arguments may produce incorrect results).

   (b) (15 points)

   Even if the arguments to a call have side-effects, if may be safe to evaluate them in parallel (using `pcall`). Give an example of such a call. If side-effects alone are insufficient to invalidate use of `pcall`, what criteria should we use to decide when parallel evaluation of arguments to a call using `pcall` is safe and correct?